

Inside Microsoft® SQL Server™ 2005: The Storage Engine

By Kalen Delaney - (Solid Quality Learning)



Publisher: **Microsoft Press**

Pub Date: **October 11, 2006**

Print ISBN-10: **0-7356-2105-5**

Print ISBN-13: **978-0-7356-2105-3**

Pages: **464**

[Table of Contents](#) | [Index](#)

Overview

This practical, hands-on book offers deep, thorough coverage of the internals of architecture and resource management in SQL Server 2005, focusing on the Storage Engine. The book features extensive code samples and table examples.

Inside Microsoft® SQL Server™ 2005: The Storage Engine

By

Kalen Delaney - (Solid Quality Learning)



.....

Publisher: **Microsoft Press**

Pub Date: **October 11, 2006**

Print ISBN-10: **0-7356-2105-5**

Print ISBN-13: **978-0-7356-2105-3**

Pages: **464**

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Foreword](#)

[Acknowledgments](#)

[Introduction](#)

[Chapter 1.](#)

[Installing and Upgrading to SQL Server 2005](#)

[SQL Server 2005 Prerequisites](#)

[Pre-Installation Decisions](#)

[Getting Ready to Install](#)

[To Migrate or Upgrade?](#)

[Selecting Components](#)

Summary

Chapter 2.

SQL Server 2005 Architecture

Components of the SQL Server Engine

Memory

Final Words

Chapter 3.

SQL Server 2005 Configuration

Using SQL Server Configuration Manager

System Configuration

Final Words

Chapter 4.

Databases and Database Files

System Databases

Sample Databases

Database Files

Creating a Database

Expanding or Shrinking a Database

Using Database Filegroups

Altering a Database

Databases Under the Hood

Setting Database Options

Database Snapshots

The tempdb Database

Database Security
Moving or Copying a Database
Compatibility Levels
Summary
Chapter 5.

Logging and Recovery

Transaction Log Basics
Changes in Log Size
Reading the Log
Backing Up and Restoring a Database
Summary

Chapter 6.

Tables

System Objects
Creating Tables
User-Defined Data Types
IDENTITY Property
Internal Storage
Constraints
Altering a Table
Summary

Chapter 7.

Index Internals and Management

Index Organization

[Creating an Index](#)
[The Structure of Index Pages](#)
[Index Space Requirements](#)
[Special Indexes](#)
[Table and Index Partitioning](#)
[Data Modification Internals](#)
[Managing Indexes](#)
[Using Indexes](#)
[Summary](#)

[Chapter 8.](#)

[Locking and Concurrency](#)

[Concurrency Models](#)
[Transaction Processing](#)
[Locking](#)
[Lock Compatibility](#)
[Internal Locking Architecture](#)
[Bound Connections](#)
[Row-Level Locking vs. Page-Level Locking](#)
[Row Versioning](#)
[Other Features That Use Row Versioning](#)
[Controlling Locking](#)
[Summary](#)

[About the Author](#)

[Additional Resources for Developers](#)

[Visual Basic 2005](#)
[Visual C# 2005](#)
[Web Development](#)

[Data Access](#)
[SQL Server 2005](#)
[Other Visual Studio 2005 Topics](#)
[Other Developer Topics](#)
[More Great Developer Resources](#)
[Developer Step by Step](#)
[Developer Reference](#)
[Advanced Topics](#)
[Index](#)

Copyright

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

Copyright © 2007 by Kalen Delaney

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN-10: 0-7356-2105-5

Library of Congress Control Number 2006932080

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 1 0 9 8 7 6

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.

Microsoft, Active Directory, MS-DOS, Outlook, Visual SourceSafe, Windows, Windows NT, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Ben Ryan

Project Editor: Denise Bankaitis

Technical Editor: Andrew J. Kelly

Copy Editor: Ina Chang

Indexers: Lee Ross and Tony Ross Body Part No. X11-97531

Dedication

For Dan, forever

Foreword

This is the seminal architectural description of SQL Server's database engine. Kalen Delaney's previous book, describing SQL Server 2000, is on my desk and the desks of most SQL Server experts used whenever one wants a concise description of how something works or the design rational for some aspect of the system. Indeed, most SQL Server architects and developers use Kalen's book as a reference. So, we have been pestering Kalen with the question: "When can I get the SQL Server 2005 book?"

Having just finished reading the galleys, I understand why the book took so long. This is not a revision of the previous book. So much has changed and there are so many new features that Kalen had to write a brand new book. She worked closely with the architects to understand the new designs, and then she had to explain the designs in her concise and precise style. The result is well worth the wait. The book is encyclopedic. I watched the development of SQL Server 2005 from the inside, sometimes participating in design discussions; still, it is a very large system so there are many aspects that I did not understand. In reading the galleys, I learned a lot. I will be regularly re-reading sections in the years to come to refresh my understanding.

This book explains how the system works and gives you a deep understanding of what the designers intended for example diving deep into page formats, explaining allocation strategies, explaining the difference between temp tables and table variables, explaining how snapshot isolation works, explaining recovery options, explaining catalog views, explaining isolation levels and how they are implemented, and so on. The presentation is deep, concise, and yet very accessible and understandable. Along the way, the book gives sage advice on application design, database design, physical data layout, and performance tuning.

To put this book in perspective, SQL Server Books Online describes *what* exists; many books describe *how to* use those features; this book describes the *design principles* behind the database engine features and *how they work*. When you understand the principles and mechanisms, you can often infer what will work well and what will be problematic. So I heartily recommend this book to anyone who wants to design, implement, or manage systems involving SQL Server 2005.

Jim Gray

Technical Fellow, Microsoft Research

1999 ACM Turing Award Winner

Acknowledgments

As always, a work like this is not an individual effort, although my name is the only one on the cover. I could not have written it at all without information, help, and encouragement from many people.

First on my list is you, the reader. Thank you, thank you, thank 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. 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.

Thanks to Ron Soukup for writing the first edition of *Inside SQL Server* and for giving me his "seal of approval" for taking on the subsequent editions. Thanks to my former editor at *SQL Server Professional Journal*, Karen Watterson, for opening so many doors for me, including passing on my name to the wonderful people at Microsoft Press. Thanks to Karen Forster and all my editors at *SQL Server Magazine*, who support my ongoing writing habit.

As usual, the SQL Server development team at Microsoft has been awesome. Although David Campbell was not directly involved in much of my research this time, I always knew he was there in spirit, and he always had an encouraging word when I saw him. He arranged for Lubor Kollar to be my project liaison inside Microsoft. Lubor not only helped me track down the right expert for every topic I needed to research, he also arranged for the internal review of every one of the chapters in this volume, so that instead of just the one external reviewer arranged by Microsoft Press, there were always at least two internal Microsoft reviewers for every chapter. Thank you, Lubor. One expert's name kept coming up again and again when I asked Lubor who I should talk to, and that was Sunil Agarwal. Sunil seemed to know everything about almost everything,

and if he didn't know some small detail, he knew who did, and could get on the phone and get the answer for me. Sunil also reviewed every single chapter, providing even more insights and suggestions. Sunil, this book is yours almost as much as it is mine; I couldn't have done it without you.

Paul Randal's contributions were enormous as well, especially in the areas dealing with the actual physical storage structures. I have fond memories of hours in his office, enjoying his whiteboard artistry as he drilled down into details of files, partitions, allocation units, and the mysterious hobts. Slava Oks, right down the hall from Paul, also opened his door and his mailbox to me during the grueling process of writing [Chapter 2](#), *SQL Server 2005 Architecture*. He made the details of memory management actually understandable, and I only hope I have done half as good a job of explaining the details as he did. Any mistakes are my fault entirely, although there would be a lot more of them if not for the careful reading and further explanations provided by Eric Christensen.

Mirek Sztajno, Sameer Tejani, and Stefano Stefani met with me and responded to my (sometimes seemingly endless) e-mails. Cliff Dibble reviewed the chapter on tables and provided so much additional information about the metadata that reading his review was an education in itself. If I just cut and pasted his insightful comments and suggestions together, I would have an entire magazine article. Don Vilen helped me revise the big architecture diagram in [Chapter 2](#) to reflect the changes in SQL Server 2005. Stephen Jiang, Michael Raheem, Conor Cunningham, Ron Dar Ziv, Eric Hanson, SriKumar Rangarajan, Ryan Stonecipher, Steve Schmidt, Peter Byrne, Jonathan Morrison, Kevin Farlee, Jun Fang, Wey Guy, Ajay Kalhan, Boris Baryshnikov, Wei Xiao, Santtu Voutilainen, and Tengiz Kharatishvili also offered valuable technical insights and information when responding to my e-mails or to questions from Sunil. I hope you all know how much I appreciated every piece of information I received.

I am also indebted to Bob Ward, Cindy Gross, Bob Dorr, Keith Elmore and Ken Henderson of the SQL Server Product Support team, not just for answering occasional questions, but for making so much information about SQL Server available through whitepapers, 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 in putting together the SQL Server documentation in the Books Online.

While writing this edition, I almost felt like a part of the SQL Server team myself. This was due not only to the generosity of the developers but also, in no small part, to Leona Lowry for finding me office space in the same building as most of the team. Thank you, Leona. The welcome you gave me was much appreciated.

I also would like to extend a very special thank you to Jim Gray, who has written the foreword for all the editions of *Inside SQL Server*. Jim is my idol and my hero, and also an incredibly nice guy. It still goes straight to my heart every time he says nice things about my writing, and it means the world to me that he has agreed once again to write my foreword.

My editors at Microsoft Press deserve thanks also. Ben Ryan, my acquisitions editor and friend, got the project off the ground and kept it flying, and overcame many obstacles along the way. Kristine Haugseth, the initial project editor, made sure the chapters started coming, and Denise Bankaitis, the final project editor, made sure the chapters kept coming all the way to the end. My external technical editor, Andy Kelly, made sure those chapters turned out well. But, of course, Ben, Kristine, Denise, and Andy couldn't do it all by themselves, so I'd like to thank the rest of the editorial team, including manuscript editor Ina Chang. I know you worked endless hours to help make this book a reality. I would also like to let my agent, Claudette Moore, know how much I appreciate all that she had to put up with from me to get the contract for this book arranged.

As a relatively independent trainer and writer, I don't have coworkers to chat with every day in the office. But I've actually got something even better. As a SQL Server MVP, I work with dozens of other SQL Server professionals to provide online support on the public newsgroups in the SQL Server forums. We share a private newsgroup with Microsoft Support Engineers and members of the development team, who serve as our interface with Microsoft, and through this newsgroup as well as various conferences, I have gotten to know many wonderful people in this group personally. I would like to extend my heartfelt thanks to all of the SQL Server MVPs, but most especially Tibor Karaszi and Ron Talmage, who participate in the private newsgroup, for the brilliant exchanges and challenging ideas discussed, and for all the support and encouragement given to me. This includes, of course, our former Microsoft MVP lead Steve Dybing, and our current lead Ben Miller. I'd like to say that being a part of the SQL Server MVP team has been one of the greatest honors and privileges of my professional life.

One particular MVP is due special thanks and that is Andy Kelly, who served as my external technical editor. Andy also is a member of another group I need to thank my colleagues and partners at Solid Quality Learning. In late 2002, five of my colleagues and I, all of whom were very experienced SQL Server professionals, decided to form a company dedicated to providing the most advanced, high-quality SQL Server training and consulting in the world. Now the company has grown to over 50 SQL Server professionals, based in many countries around the world. I would especially like to thank Fernando Guerrero for the uncounted hours of work he has performed as CEO to keep the company growing and viable. I would also like to thank Itzik Ben-Gan, an MVP and partner in Solid Quality Learning, for keeping me excited about Transact-SQL and writing the two accompanying volumes on T-SQL for this *Inside SQL Server 2005* series.

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 share with me. Much of what I have learned has been inspired by questions from my curious students. Some of my students, such as Lara Rubbelke, have become friends, 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 life after over 21 years of marriage. My daughter, Melissa, is a role model for me, and provided the final motivation I needed for finishing this book, as we raced to see whether I would finish it before she finished her doctoral dissertation in Linguistics at University of Edinburgh. My three sons, Brendan, Rickey, and Connor are now for the most part all grown, and are all generous, loving, and compassionate young men, still deciding what course their lives will follow for the immediate future. Brendan and Rickey are both starting university studies this fall. Brendan is intending to go into medicine, and hopes to continue his terrific fiction writing on the side. Rickey, my most computer-savvy son, has decided to follow his other passion and major in theater arts, while doing his own game programming on the side. And Connor is writing songs for the guitar, making movies, and acting in plays as he starts his high school journey. My boys have been a source of much needed affection, and they all know when what I really need is just a good laugh, and they all can find ways to amuse me. I feel truly blessed to have them in my life.

Kalen Delaney, 2006

Introduction

For me, the most wonderful thing about writing a book and having it published is getting feedback from you, the readers. Of course, it's also one of the worst things when I get feedback from readers who are frustrated that their favorite topics were not included. However, by this third time around, I think I can accept the fact that this book cannot be all things to all people, as much as I might want it to be. Microsoft SQL Server 2005 is such a huge, complex product that not even with a new multi-volume format can we cover every feature. My hope is that you'll look at the cup as half full instead of half empty and appreciate the volumes of *Inside Microsoft SQL Server 2005* for what they *do* include. As for the topics that aren't included, I hope you'll find the information you need in other sources.

The focus of this series, as the name *Inside* implies, is on the core SQL Server engine in particular, the query processor and the storage engine. This series doesn't talk about 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 I had to draw the line somewhere or else we would need 10 volumes in the series and have no hope of finishing it before the release of the next version of the product!

A History of *Inside Microsoft SQL Server*

The first edition of *Inside Microsoft SQL Server*, written for version 6.5, did attempt to cover almost all features of the product. Back then, the product was much smaller. Also, few other SQL Server

books were available, so the original author, Ron Soukup, couldn't just refer his readers to other sources for information on certain topics. Even so, some topics were not covered in the first edition, including replication and security. Ron also did not cover any details of backing up or restoring a SQL Server database, and he didn't really discuss the use and management of the transaction log.

I took over the book for SQL Server 7.0, and I completely rewrote many of the sections describing the internals of the storage engine because the entire storage engine had changed. The structure of pages, index organization, and management of locking resources were all completely different in version 7.0.

Inside Microsoft SQL Server 7.0 discussed transactions, stored procedures, and triggers all in one chapter. For the SQL Server 2000 edition, with the new feature of user-defined functions and new trigger capabilities, I split these topics into two chapters. In the 7.0 edition, query processing and tuning were covered in one huge chapter; the SQL Server 2000 edition separated these topics into two chapters, one dealing with the internals of query processing and how the SQL Server optimizer works and the other providing guidance on how to write better-performing queries. *Inside Microsoft SQL Server 2000* also included many details about the workings of the transaction log, as well as an in-depth discussion of how the log is used during backup and restore operations.

Series Structure

Early in the planning stages for *Inside Microsoft SQL Server 2005*, I realized that it would be impossible to cover everything I wanted to cover in a single volume. My original thought was to have one volume on the storage engine components and the actual data management and a second volume on using the Transact-SQL (T-SQL) language and optimizing queries. I soon realized that this second topic itself was too big for a single volume, partly because

SQL Server 2005 has so many new Transact-SQL features. Adequate coverage of all the new programming constructs would require a volume of its own, so at that point I invited T-SQL guru Itzik Ben-Gan to write a volume on Transact-SQL in SQL Server 2005. Itzik is an extremely prolific writer, and he had over 500 pages written before I completed the planning for my storage engine volume. At that point, he realized that the Transact-SQL language itself was too big for a single volume and that his work would need two volumes to cover everything we felt was necessary. So *Inside Microsoft SQL Server 2005* is a work in four volumes.

Although one goal of ours was to minimize the amount of overlap between volumes so that readers of the complete series would not have to deal with duplicate content, we also realized that not everyone would start with the same volume. Itzik and I have different approaches to describing SQL Server query processing, index use, and tuning, so when those topics are covered in more than one volume, that duplication is actually a bonus.

Inside Microsoft SQL Server 2005: T-SQL Querying

The T-SQL querying volume describes the basic constructs of the Transact-SQL query language and presents a thorough discussion of logical and physical query processing. It also introduces a methodology for query tuning. Itzik provides a detailed discussion of the use and behavior of all the new T-SQL query constructs, including CTEs, the PIVOT and UNPIVOT operators, and ranking functions. He covers enhancements to the TOP clause and provides examples of many new and useful ways to incorporate aggregation into your queries. New capabilities of data modification operations (INSERT, UPDATE, and DELETE) are also described in depth.

Inside Microsoft SQL Server 2005: T-SQL Programming

The T-SQL programming volume focuses on the programmability features of the T-SQL language and covers the planning and use of transactions, stored procedures, functions, and triggers in your SQL Server applications. Itzik compares set-based and cursor programming techniques and describes how to determine which technique is appropriate, and he covers CLR versus relational programming, again describing which model is appropriate for which activities. The book covers the use of temporary objects and explores the new error-handling functionality in SQL Server 2005. Itzik discusses issues with working with various datatypes, including XML data and user-defined CLR datatypes. Finally, there is a chapter on SQL Server Service Broker, which allows controlled asynchronous processing in database applications.

Inside Microsoft SQL Server 2005: The Storage Engine

The volume you are holding covers the SQL Server 2005 storage engine. I started working on this volume by taking the chapters from *Inside Microsoft SQL Server 2000* that dealt with storage issues and then determining which new features were appropriate to cover. I soon realized that some reorganization was necessary, and I ended up with a full chapter on the architecture of the SQL Server 2005 engine and a whole chapter on the transaction log. As in all previous editions, I go into great depth on the actual physical storage of both data and indexes in the data files, and I describe the way that the file space is allocated and managed. Undocumented trace flags and DBCC commands are introduced where appropriate

to illustrate certain features and to allow you to confirm your understanding of SQL Server's behavior.

New features in SQL Server 2005 are pointed out as I discuss them; here are some of the most important new features covered in detail in this volume. Note that other new features are mentioned but that not all are covered in depth.

- SQL Server 2005 metadata views, including compatibility views, catalog views, and dynamic management views (and functions)
- Database snapshots
- User/schema separation
- Storage of large data objects, including row-overflow data and *varchar* (MAX) data
- Storage of partitioned tables and indexes
- Online index building and rebuilding
- Snapshot isolation and row-level versioning

Inside Microsoft SQL Server 2005: Query Tuning and Optimization

The final volume in the series will be based on real-world observations of the way SQL Server 2005 performs and will explain how to get the most out of the product in real applications. This will

be a multi-author work, with each author covering the areas of his or her greatest expertise. Covered topics will include:

- Methodology for determining where tuning is needed
- How the optimizer determines the ideal query plans
- Monitoring SQL Server 2005 with SQL Server Profiler
- Plan caching and reuse
- Forcing query plans
- Best practices for partitioning and indexed views
- Choosing the best indexes
- Tips and tricks for both retrieval queries and data modifications

Examples and Scripts

Many of the features and behaviors described in this volume are illustrated using T-SQL code. Some of the code is just a few lines long, but other examples require very complex coding, including multi-way joins of some of the dynamic management views, all of which have impossibly long and difficult-to-type names.

All code samples longer than a couple of lines are available for download from the companion Web site at www.InsideSQLServer.com/companion.

Topics Not Covered

As I mentioned, even in four volumes, certain features and aspects of the product cannot be covered. Also keep in mind that the books in the series are not intended to be how-to books for database administrators or for database application programmers. They are intended to explain how SQL Server works behind the scenes, so you will have a solid foundation on which to build and troubleshoot your applications and will understand why SQL Server behaves the way it does.

In addition to business intelligence (Analysis Services, Integration Services, and Reporting Services) and high availability (replication, database mirroring, log shipping, and clustering), other topics that are beyond the scope of this book include:

- Notification Services
- XML indexes
- Full-text search
- Client programming interfaces

Caveats and Disclaimers

To illustrate some of the behaviors of SQL Server, this volume discusses some undocumented features of the product or undocumented objects, such as internal tables. Some of these are potentially "discoverable" on your own, usually by looking at the definition of the supported functions, procedures, or views. In those cases, I am simply saving you time by providing information that you could have eventually discovered on your own. Another category of undocumented features is undocumented DBCC commands or trace flags, which I introduce only for the purpose of allowing deeper analysis or more thorough observation of certain product behavior.

For the most part, these are not discoverable unless someone tells you about them. Please keep in mind that undocumented means unsupported. This means that if you have additional questions about an undocumented feature that I describe, you cannot call Customer Support Services at Microsoft and expect the representative on the phone to answer your questions. There is also no guarantee that an undocumented feature will continue to behave in the same way in the next version of SQL Server. In some cases, undocumented features can change behavior in a service pack, and Microsoft will not be obligated to tell you about this change in a readme file or a Knowledge Base article. Throughout this volume, I will let you know when I refer to features or tools that are undocumented, and in some cases, I will also reiterate that Microsoft provides no support for them. However, consider this a global caveat for all such undocumented features.

How to Get Support

Every effort has been made to ensure the accuracy of this book's content. If you run into problems, you can refer to one of the following sources.

Companion Web Site

Despite my best intentions, as well as review by members of the SQL Server team at Microsoft, this book is not perfect, as no book is. Updates and corrections will be posted on the companion Web site at www.InsideSQLServer.com/companion. In addition, if you find anything that you think is incorrect, feel free to use the feedback form on that site to inform me of the problem.

Microsoft Learning

Microsoft provides corrections for books at the following Web address:

<http://www.microsoft.com/learning/support>

To connect directly with the Microsoft Learning Knowledge Base and enter a query regarding an issue you have encountered, you can go to <http://www.microsoft.com/learning/support/search.asp>.

In addition to sending feedback to the author, you can send comments or questions to Microsoft by using either of the following methods:

Postal Mail:

Microsoft Learning
Attn: *Inside Microsoft SQL Server 2005* Editor
One Microsoft Way
Redmond, WA 98052-6399

E-mail:

mspinput@microsoft.com

Please note that product support is not offered through the preceding addresses. For SQL Server support, go to www.microsoft.com/sql. You can also call Standard Support at 425-635-7011 weekdays between 6 A.M. and 6 P.M. Pacific time, or you can search Microsoft's Support Online at www.support.microsoft.com/support.

I hope you find value in this book even if I haven't covered every single SQL Server topic that you're interested in. You can let me know what you'd like to learn more about, and I can perhaps refer you to other books or white papers. Or maybe I'll write an article for *SQL Server Magazine*. You can contact me via my Web site at www.InsideSQLServer.com.

Chapter 1. Installing and Upgrading to SQL Server 2005

In this chapter:

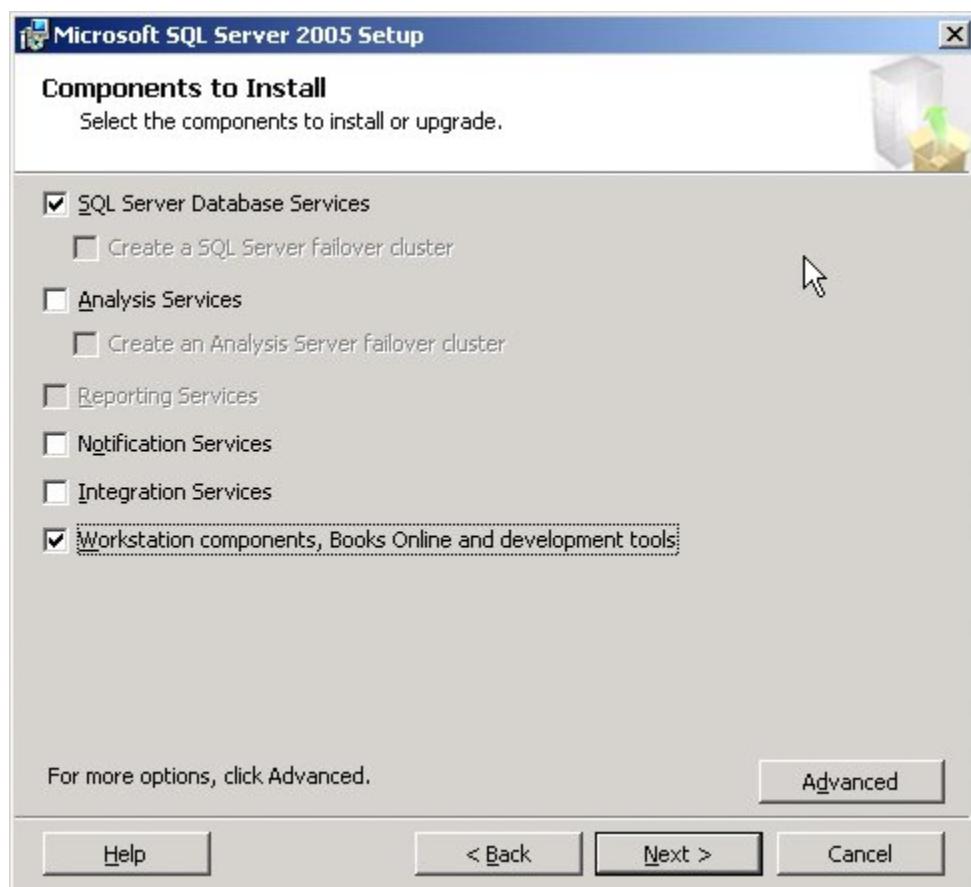
<u>SQL Server 2005 Prerequisites</u>	<u>2</u>
<u>Pre-Installation Decisions</u>	<u>7</u>
<u>Getting Ready to Install</u>	<u>18</u>
<u>To Migrate or Upgrade?</u>	<u>20</u>
<u>Selecting Components</u>	<u>25</u>
<u>Summary</u>	<u>28</u>

Microsoft SQL Server is Microsoft's premiere database management system, and SQL Server 2005 is the most powerful and feature-rich version yet. In addition to the core database engine that allows you to store and retrieve large volumes of relational data and the world-class query optimizer that determines the fastest way to process your queries and access your data, dozens of other

components increase the usability of your data and make your data and applications more available and more scalable. As you can imagine, no one book could cover all of these features in depth. In fact, the entire *Inside SQL Server 2005* series of books covers only the most important features of the core database engine.

For a first, high-level view of the components in SQL Server 2005, take a look at [Figure 1-1](#). This dialog box, which appears during product installation, allows you to choose which components of SQL Server 2005 to install. You can also decide which, if any, of the client tools and connectivity components to install.

Figure 1-1. List of components to install



In this chapter, we'll first look at the prerequisites for installing or upgrading to SQL Server 2005. If you are using SQL Server 7.0 or SQL Server 2000, you can either upgrade or install a new SQL Server 2005 instance and migrate your data. I'll discuss the differences between these two methods and describe the benefits and drawbacks of each. I'll also briefly describe each feature and component that you can install and mention some of the tools you can use to inspect or modify the decisions you make.

As I'll discuss in more detail later in the chapter, SQL Server Management Studio replaces two tools from earlier versions: Query

Analyzer, through which you could type in your SQL queries for execution, and SQL Server Enterprise Manager, which provided a graphical user interface (GUI) for carrying out certain tasks. When you use Management Studio, if you want to run a SQL query, you must open a query window; to perform actions using the GUI, you use a component of Management Studio called Object Explorer. I won't cover every component of Management Studio in detail; for more information, see this book's online companion content for a white paper by Ron Talmage that describes the SQL Server 2005 tools suite.

SQL Server 2005 Prerequisites

The installation CD or downloaded setup files include complete instructions for installing SQL Server 2005. When you first put the disc into your CD drive or execute the xxx file, you'll see the screen shown in [Figure 1-2](#). The first option allows you to open a document containing detailed lists of the software and hardware requirements for SQL Server 2005. The online companion content contains a detailed upgrade guide called *SQL Server 2005 Upgrade Handbook*. The handbook is also available at the Microsoft SQL Server Web site: <http://www.microsoft.com/technet/prodtechnol/sql/2005/sqlupgrd.mspx>. This chapter points out some crucial issues to be aware of when you set up your environment; however, you should not consider it a replacement for the upgrade guide or the setup documentation in SQL Server 2005 Books Online.

Figure 1-2. The initial splash screen when installing SQL Server 2005

[[View full size image](#)]



You can find the complete Books Online on your installation disk at
\\Servers\\Setup\\help\\1033\\Setupsq19.chm or
\\Tools\\Setup\\help\\1033\\Setupsq19.chm, where 1033 can be replaced by another language ID if you are not installing the English version of SQL Server. For example, French is 1036, German is 1031, and Japanese is 1041. You can find the complete list of language ID values at www.microsoft.com/globaldev/reference/lcid-all.mspx.

The complete SQL Server 2005 documentation is also available online at <http://msdn2.microsoft.com/en-us/library/ms130214.aspx>.

SQL Server 2005 Editions

SQL Server 2005 is available in multiple editions. The edition you install will depend on the features you want and the hardware resources you want to take advantage of. The software and the hardware requirements will also vary by edition. If you have the installation CD already, you know the edition and its requirements. If you haven't yet procured the software, take note that there are editions for Pentium-compatible 32-bit processors as well as for Intel Itanium architecture processors (indicated in the requirements document as IA64) and AMD architecture/Intel Extended Systems architecture processors (indicated in the requirements document as X64).

SQL Server 2005 has either 6 or 10 editions, depending on whether you think that SQL Server running on a 32-bit platform is always a different edition from one running on a 64-bit platform. (In the following list of the six editions, four are available for either 32-bit or 64-bit.) Most editions are available on their own CD.

The following information is from SQL Server Books Online:

- **SQL Server 2005 Enterprise Edition (32-bit and 64-bit)** The most comprehensive edition of SQL Server. Enterprise Edition is

ideal for the largest organizations and the most complex requirements.

- **SQL Server 2005 Evaluation Edition (32-bit and 64-bit)** Supports the same feature set as SQL Server 2005 Enterprise Edition but is supported on some of the operating systems that Enterprise Edition is not supported on. For example, the 32-bit Evaluation Edition is supported on Microsoft Windows 2000 Professional Edition (with Service Pack 4) and Windows XP (with Service Pack 2), whereas Enterprise Edition requires Windows 2000 Advanced Server or Windows Server 2003 Enterprise or later. A 120-day Evaluation Edition for the 32-bit platform is available for download from
<http://www.microsoft.com/sql/downloads/2005/default.mspx>.
- **SQL Server 2005 Standard Edition (32-bit and 64-bit)** The data management and analysis platform for small- and medium-sized organizations, where high availability is not the most important aspect of the database system. Some of the differences between Standard Edition and Enterprise Edition include the number of instances supported (the maximum number of nodes in a cluster). Standard Edition also does not support some of the high-availability features, such as online and parallel index operations and hot-add memory. Standard Edition is now supported on some non-server-grade operating systems such as Windows 2000 Professional Service Pack (SP) 4 and Windows XP Professional SP2.
- **SQL Server 2005 Workgroup Edition (32-bit only)** The data management solution for small organizations that need a database with no limits on size or number of users. Workgroup Edition can serve as a front-end Web server or a server for departmental or branch office operations. This edition includes the core database features of SQL Server.

- **SQL Server 2005 Developer Edition (32-bit and 64-bit)**
Includes all of the functionality of SQL Server 2005 Enterprise Edition but has different licensing. Developer Edition can be used only as a development and test system, not as a production server. Developer Edition runs on all the same platforms as Evaluation Edition.
- **SQL Server 2005 Express Edition (32-bit only)** A free, easy-to-use, and simple-to-manage database that can be redistributed (subject to licensing agreement). It functions as the client database as well as a basic server database. Some of the main limitations of Express Edition are that it does not come with the rich suite of client tools and the SQL Agent job scheduler.

The features in Enterprise Edition (as well as in Developer Edition and Evaluation Edition) that are not in Standard Edition generally relate to hardware support and high-availability features. When I discuss such features, I'll let you know. For full details on what is in each edition, see the Books Online topic "SQL Server 2005 Database Engine Features."

Software Requirements

The installation documentation is very clear about which editions of SQL Server are supported on which operating systems. Keep in mind that the limitations are not necessarily due to particular SQL Server features not being able to run on particular platforms; rather, they might be based on what kind of operations you are expected to do with the database. For example, as mentioned earlier, the Evaluation and Developer editions have all the features of Enterprise Edition, but they can run on lower-end operating systems. An organization that deploys Enterprise Edition, on the other hand, is expected to run an enterprise-level application with many concurrent users and large volumes of data. Such large-scale applications are best served by running on an operating system and hardware platform that can support these larger

numbers of users and much larger volumes of data. As you read the installation documentation, pay attention to which service packs are required for the various operating systems. For example, all editions of SQL Server 2005 will run on Windows 2000, but only if SP4 has been applied.

For most editions, if you have the appropriate operating system, you don't need to install additional software before you run the setup program, Setup.exe. You do need to have the Microsoft .NET Framework 2.0 installed before you install SQL Server, but Setup.exe will install it for you for all editions except Express Edition. (For Express Edition, you must download the .NET Framework 2.0 from Microsoft's Web site.) You might also need to install additional software before installing some of the components. For example, to install and run Reporting Services, you must have Microsoft Internet Information Services (IIS) 5.0 or later installed.

Hardware Requirements

After it confirms that you have all the prerequisite components installed, Setup.exe will verify that the computer where SQL Server will be installed meets all of the other requirements for a successful installation. A new component of Setup.exe called the System Configuration Checker (SCC) scans the computer and checks for conditions that prevent a successful installation. Some of these conditions relate to the availability of needed services, and others relate to minimum hardware specifications. If the conditions aren't met, SCC will give a warning or block the installation of SQL Server. For example, it will block setup if the requirement for processor type is not met. However, if the minimum or recommended processor speed is not met, or if the minimum or recommended RAM check is not met, SCC will issue a warning but will not stop the setup process.

You must meet additional hardware and software prerequisites to install SQL Server failover clustering. Microsoft Cluster Server (MSCS)

is required on at least one node of your server cluster, and MSCS is supported only if it is installed on a compatible hardware configuration.

Again, I strongly suggest that you thoroughly read the upgrade guide before installing SQL Server 2005. This guide is useful information even if you are not upgrading but installing your first SQL Server. Keep in mind that the hardware requirements are only the minimum SQL Server needs to run. If you want SQL Server to run well and fast, you'll probably want to exceed the minimum, by as much as your budget will allow, in many cases. The following sections discuss some of the most critical issues to be aware of when planning your hardware environment.

Processors

All SQL Server 2005 32-bit editions need at least a Pentium III processor, with a minimum speed of 500 MHz and a recommended speed of at least 1 GHz. For 64-bit editions, the minimum is slightly higher, but the recommended speed is still at least 1 GHz. For any real database operations, you'll probably want to far exceed this minimum, so the difference between 32-bit and 64-bit is really moot. The maximum number of processors depends on the edition you are using. Enterprise Edition, both 32-bit and 64-bit, supports the maximum number of processors supported by the operating system.

You'll probably also want to investigate hyperthreading and the new dual-core CPUs, through which SQL Server sees each of your processors as two processors. In both cases, you'll need only a single license per CPU chip even though SQL Server will see twice as many CPUs, which can result in greatly increased performance in some cases.

Disks

The disk space requirements given in the installation documentation and during setup are only for the SQL Server components, the initial size of system databases, and the initial size of the sample databases and sample code (if you choose to install them). No space requirement is shown for your own databases because only you can estimate how much space you'll need. As we look at file structures and table storage later in this book, you'll get some ideas about how to realistically estimate the space you will need for your data.

The installation requirements do not specify any specialized kind of file system or even having more than one drive, either logical or physical. For a minimum installation, all you need is your C drive. As we look at SQL Server files and filegroups later, I'll talk about why you might want to spread out your data as well as separate it from your transaction log. We will look at the tempdb database as a special case, and I'll tell you why you might want to manage your tempdb disk space separately from other system databases. You can easily change the location of tempdb after installation, so I'll save that discussion for later.

You should try to have the best disk system you can afford. For most SQL Server installations, you'll want a disk system that supports redundancy to provide for higher availability in case of failure.

Redundancy (or fault tolerance) is usually provided by some type of RAID system. RAID-10, or a stripe of mirrors, is the ultimate choice for performance and recoverability. RAID-10 is not really a separate type of RAID, but rather a combination of RAID-1 (mirroring) and RAID0 (striping). A series of disks are first mirrored to provide the fault-tolerance features and increased read performance of RAID-1 and then striped to provide the performance advantages of RAID-0. Performance for both writes and reads is excellent. RAID-01 is the opposite in which a series of disks are striped in two RAID-0 configurations and then mirrored together.

Both RAID-10 and RAID-01 refer to combining the capabilities of RAID0 (striping) with RAID-1 (mirroring) for greater speed and fault tolerance without the overhead of computing a parity value, which is

used in RAID 5. However, the terminology is not standard. Some sources use the terms *RAID-10* and *RAID-01* interchangeably, so it is important to understand which physical configuration you end up with. Although the performance is comparable between the two, the stripe of mirrors is vastly more fault tolerant than a mirror of stripes.

Memory

The hardware requirements specify a minimum of 512 MB of memory for all editions of SQL Server 2005 except Express, with a recommended value of at least 1 GB. For your production systems, obviously you'll want much more memory than that. I'll discuss memory use in detail when I talk about SQL Server architecture in [Chapter 2](#). The most important detail you should keep in mind before installation is that a 32-bit platform can directly address only 4 GB of memory, no matter how much memory is installed in the machine. Of that 4 GB, applications such as SQL Server can normally use only 2 GB, but there is an operating system flag that allows 3 GB of memory to be directly addressed by a user application. Additional memory beyond that must be accessed indirectly usually through the /PAE flag in the operating system and the AWE configuration options in SQL Server. A true 64-bit system can directly address far more than 4 GB of memory, so you do not have to worry about special flags or options if you want to use up to the maximum amount of memory that your edition of SQL Server supports.

Pre-Installation Decisions

Most of the choices you make during the installation process can be changed later, although some are harder to change than others. In this section, I'll discuss three of the most important decisions you'll need to make during the installation, and I strongly suggest that you think about each of them before running SQL Server Setup.

Security and the User Context

To install SQL Server on a machine, you need not be an administrator of the domain, but you must have administrator privileges on the machine. Users can install most of the SQL Server client utilities without administrator privileges.

Before you set up your system, give some thought to the user context in which SQL Server and SQL Server Agent will run. A new SQL Server installation can set up the SQL Server engine to run in the context of the special system (LocalSystem) account. This account is typically used to run services, but it has no privileges on other machines, nor can it be granted such privileges. The Local System account option is provided for backward compatibility only. The Local System account has permissions that SQL Server Agent does not require. Avoid running SQL Server Agent as the Local System account whenever possible. It is recommended to have the SQL Server service and the SQL Server Agent service run in the context of a domain account. This allows SQL Server to more easily perform tasks that require an external security context, such as backing up to another machine or using replication. If you don't have access to a domain account and set up SQL Server to run in the context of the Local System account, you should be aware that this account is automatically added to the SQL Server sysadmin role by

default. This will still give SQL Server sufficient privileges to run on the local machine but may not be as secure as using a domain account with appropriate privileges.

If you're not going to use the LocalSystem account, it's a good idea to change the account under which SQL Server will run to a user account that you've created just for this purpose (rather than use an actual local or domain account of a real user). The account must be in the local Administrators group if you're installing SQL Server on Windows 2000 or Windows 2003. You can create this account before you begin installing SQL Server, or you can change the account under which SQL Server runs at a later time. Changing the accounts for the SQL Server services should never be done directly via the Service or through the Services Control Panel. It is very important to use the SQL Server Configuration Manager utility to make all changes to the services associated with SQL Server. This utility ensures that all changes to the account name or password are properly dealt with across all the components involved. By making the change directly in the control panel, you can prevent the services from starting or operating properly. When you choose or create a user account for running SQL Server, be sure to configure the account so the password never expires; if the password expires, SQL Server won't start until the information for the service is updated.

By default, the installation program chooses to use the same account for both the SQL Server and the SQL Server Agent services. SQL Server Agent needs a domain-level security context to connect to other computers more often than does SQL Server. For example, if you plan to publish data for replication, SQL Server Agent needs a security context to connect to the subscribing machines. If you specify a domain account but the domain controller cannot validate the account (because the domain controller is temporarily unavailable, for example), go ahead and install using the LocalSystem account and change it later using the SQL Server Configuration Manager (available through the Start menu).

During installation, you are given the option to set up SQL Server to allow connections using Windows Authentication Mode only or using Mixed Mode security. If you select Windows Authentication Only, SQL Server will allow access only if the connection already has been authenticated by Windows and has a valid Security ID (SID). The Windows-authenticated SID determines the level of access the connection will have in SQL Server. With Windows Authentication, you cannot choose to connect under a different SQL Server logon name than the one indicated by your Windows logon.

If you choose Mixed Mode security during installation, each client will specify when a connection is made, whether it is connecting using Windows Authentication or SQL Server Authentication. If the client requests SQL Server Authentication, a SQL Server login name and password must be supplied. During installation, if you choose Mixed Mode authentication, it means that you can log on using the SQL Server system administrator (sa) login; you are then asked to specify a password for that all-powerful login name. Unlike in SQL Server 2000, a password is required for the sa login and it cannot be left blank. Be sure to pick a password you'll remember because, by design, there is no way to read a password (it is stored in encrypted form) it can only be changed to something else. A strong password is required one that contains at least six characters and includes a combination of uppercase and lowercase letters, digits, and special characters.

For example, to use a query window to change the password to `1ceCre@m`, you would issue the following command:

```
ALTER LOGIN sa WITH PASSWORD = '1ceCre@m'
```

Using Object Explorer, you can change the password from the SQL Server Security folder. Open the Logins folder under Security, and

then double-click the entry for sa. A dialog box will open that allows you to change the password.

Warning



If you choose Windows Authentication Only during installation and later change it to Mixed Authentication, you will not be asked for a password for the sa account. You will automatically have a blank password. You must remember to use one of the techniques described earlier to give the sa account a secure password.

Note that the way the actual password is stored is different in SQL Server 2005 than it was in SQL Server 2000. Passwords are now always case-sensitive regardless of the server collation.

Characters and Collation

Another decision you should consider before installation is the collation for your SQL Server, which includes both the character set and sort order you want SQL Server to use for your system databases. Although user databases can be installed with different collation than the system databases, and individual character columns can have their own collation, it can make database management more complex if you have to keep track of multiple collations. You should also consider the collation you choose during

installation to be the permanent collation for the system databases because it can be a very complex operation to change on an existing SQL Server.

Most non-Unicode characters are stored in SQL Server as a single byte (8 bits), which means that 256 (2^8) different characters can be represented. But the combined total of characters in all the world's languages is more than 256. So if you don't want to store all your data as Unicode, which takes twice as much storage space, you must choose a character set that contains all the characters (referred to as the *repertoire*) that you need to work with. For installations in the Western Hemisphere and Western Europe, the ISO character set (also often referred to as Windows Characters, ISO 8859-1, Latin-1, or ANSI) is the default and is compatible with the character set used by all versions of Windows in those regions. (Technically, there is a slight difference between the Windows character set and ISO 8859-1.) If you choose ISO, you might want to skip the rest of this section on character sets, but you should still familiarize yourself with sort order issues.

You should also ensure that all your client workstations use a character set that is consistent with the characters used by your installation of SQL Server. If, for example, the character ¥, the symbol for Japanese yen, is entered by a client application using the standard Windows character set, internally SQL Server stores a byte value (also known as a *code point*) of 165 (0xA5). If an MS-DOS-based application using code page 437 retrieves that value, that application will display the character Ñ. (MS-DOS uses the term *code pages* to mean character sets; you can think of the terms as interchangeable.) In both cases, the internal value of 165 is stored, but the Windows character set and the MS-DOS code page 437 render it differently. You must consider whether the ordering of characters is what is semantically expected (as discussed later in the "[Sort Orders](#)" section) and whether the character will be rendered on the application monitor (or other output device) as expected.

SQL Server provides services in the OLE DB provider and the ODBC driver that use Windows services to perform character set conversions. Conversions cannot always be exact, however, because by definition, each character set has a somewhat different repertoire of characters. For example, there is no exact match in code page 437 for the Windows character Ó, so the conversion must give a close, but different, character.

Windows 2000, Windows 2003, and SQL Server 2005 also support 2-byte characters that allow representation of virtually every character used in any language. This is known as Unicode, and it provides many benefits with some costs. The principal cost is that 2 bytes instead of 1 are needed to store a character. SQL Server allows storage of Unicode data by using the datatypes *nchar*, *nvarchar*, and *ntext*. I'll discuss these in more detail when I talk about datatypes in [Chapter 6](#). The use of Unicode characters for certain data does not affect the character set chosen for non-Unicode data, which is stored using the datatypes *char*, *varchar*, and *text*.

The ASCII Character Set

The first 128 characters are the same for the character sets ISO, code page 437, and code page 850. These 128 characters make up the ASCII character set. (Standard ASCII is only a 7-bit character set; ASCII was simple and efficient to use in telecommunications because the character and a "stop bit" for synchronization could all be expressed in a single byte.) If your application uses ASCII characters but not the so-called *extended characters* (typically, characters with diacritical marks, such as à, Å, and ä) that differentiate the upper 128 characters among these three character sets, it probably doesn't matter which character set you choose. In this situation (only), it doesn't

matter whether you choose one of these three character sets or use different character sets on your client and server machines because the rendering and sorting of every important character uses the same byte value in all cases.

Versions of SQL Server before SQL Server 2000 did not support Unicode, but they did support double-byte character sets (DBCS). DBCS is a hybrid approach and is the most common way for applications to support Asian languages such as Japanese and Chinese. With DBCS encoding, some characters are 1 byte and others are 2 bytes. The first bit in the character indicates whether the character is a 1-byte or a 2-byte character. (In non-Unicode datatypes, each character is considered 1 byte for storage; in Unicode datatypes, every character is 2 bytes.) To store two DBCS characters, a field would need to be declared as *char(4)* instead of *char(2)*. SQL Server correctly parses and understands DBCS characters in its string functions. The DBCS characters are still available for backward compatibility, but for new applications that need more flexibility in character representation, you should consider using the Unicode datatypes exclusively.

Sort Orders

The specific character set used might not be important in many sites where most of the available character sets can represent the full range of English and Western European characters. However, in nearly every site, whether you realize it or not, the basics of sort order (which is more properly called *collating sequence*) is important. Sort order determines how characters compare and assign their values. It determines whether your SQL operations are case sensitive. For example, is an uppercase A considered identical

to a lowercase a? If your sort order is case sensitive, these two characters are not equivalent, and SQL Server will sort A before a because the byte value for A is less than that for a. If your sort order is case insensitive, whether A sorts before or after a is unpredictable unless you also specify a preference value. Uppercase preference can be specified only with a case-insensitive sort order; this means that although A and a are treated as equivalent for comparisons, A is sorted before a.

If your data will include extended characters, you must also decide whether your data will be accent insensitive. For our purposes, accents refer to any diacritical marks. An accent-insensitive sort order treats a and ä as equivalent, for example.

For certain character sets, you must also specify two other options. For double-byte character sets, you can specify width insensitivity, which means that equivalent characters represented in either 1 or 2 bytes are treated the same. For a Japanese sort order, you can specify kana insensitivity, which means that katakana characters are always unequal to hiragana characters. If you use only ASCII characters and no extended characters, you should simply decide whether you need case sensitivity and choose your collation settings accordingly.

Sort order affects not only the ordering of a result set but also which rows of data qualify for that result set. If a query's search condition is

WHERE name='SMITH'

the case sensitivity determines whether a row with the name *Smith* qualifies. Character matching, string functions, and aggregate functions (MIN, MAX, COUNT (DISTINCT), GROUP BY, UNION,

CUBE, LIKE, and ORDER BY) all behave differently with character data depending on the sort order specified.

Sort Order Semantics

You can think of characters as having primary, secondary, and tertiary sort values, which are different for different sort order choices. A character's primary sort value is used to distinguish it from other characters, without regard to case and diacritical marks. It is essentially an optimization: if two characters do not have the same primary sort value, they cannot be considered identical for either comparison or sorting and there is no reason to look further. The secondary sort value distinguishes two characters that share the same primary value. If the characters share the same primary and secondary values (for example, A = a), they are treated as identical for comparisons. The tertiary value allows the characters to compare identically but sort differently. Hence, based on A = a, *apple* and *Apple* are considered equal. However, *Apple* sorts before *apple* when a sort order with uppercase preference is used. If there is no uppercase preference, whether *Apple* or *apple* sorts first is simply a random event based on the order in which the data is encountered when retrieved.

For example, a sort order specification of "case insensitive, accent sensitive, uppercase preference" defines all A-like values as having the same primary sort value. A and a not only have the same primary sort value, but they also have the same secondary sort value because they are defined as equal (case insensitive). The character à has the same primary value as A and a but is not declared equal to them (accent sensitive), so à has a different secondary value. And although A and a have the same primary and secondary sort values, with an uppercase preference sort order, each has a different tertiary sort value, which allows it to compare identically but sort differently. If the sort order doesn't have

uppercase preference, *A* and *a* will have the same primary, secondary, and tertiary sort values.

Some sort order choices allow for accent insensitivity. This means that extended characters with diacritics are defined with primary and secondary values equivalent to those without. If you want a search of *name* ='*Jose*' to find both *Jose* and *José*, you should choose accent insensitivity. Such a sort order defines all *E*-like characters as equal:

E=e=è=É=é=ê=ë

Note



When you decide on the case sensitivity for your SQL Server databases, you should be aware that it applies to object names as well as user to data because object names (the metadata) are stored in tables just like user data. So if you have a case-insensitive sort order, the table *CustomerList* is seen as identical to a table called *CUSTOMERLIST* and to one called *customerlist*. If your server is case sensitive and you have a table called *CustomerList*, SQL Server will not find the table if you refer to it as *customerlist*.

Binary Sorting

As an alternative to specifying case or accent sensitivity, you can choose a binary sorting option, in which characters are sorted based on their internal byte representation. If you look at a chart for the character set, you see the characters ordered by this numeric value. In a binary sort, the characters are sorted according to their position by value, just as they are in the chart. Hence, by definition, a binary sort is always case sensitive and accent sensitive and every character has a unique byte value.

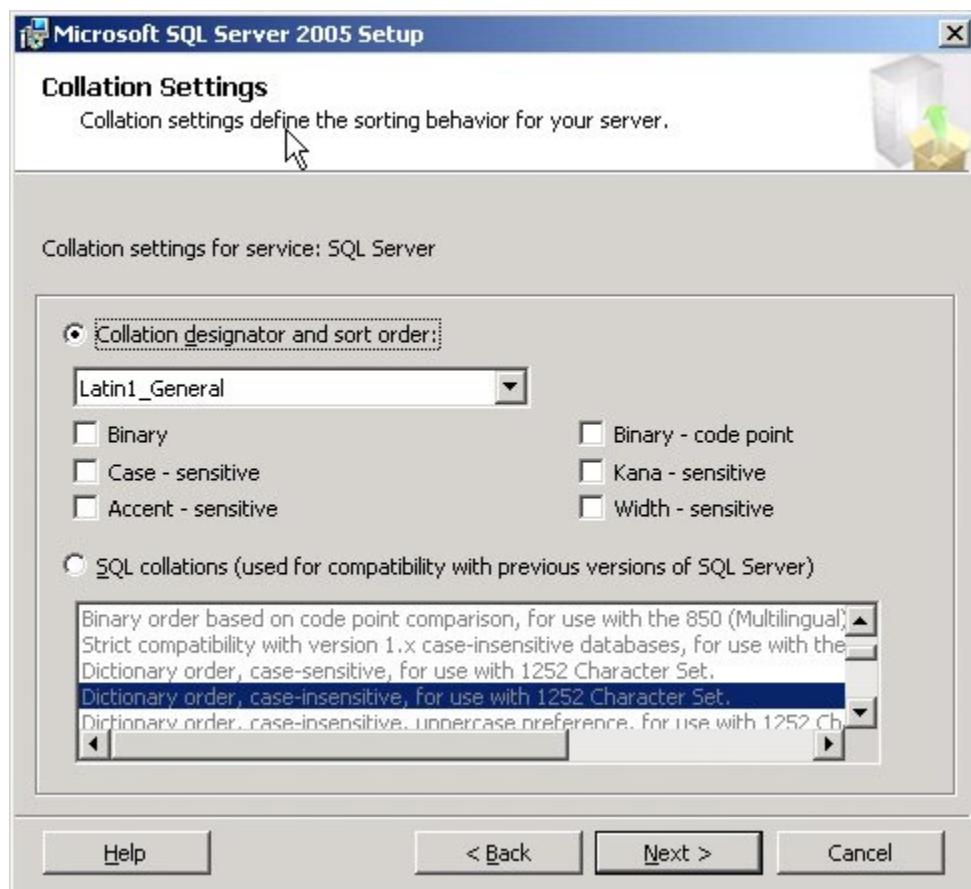
Binary sorting is the fastest sorting option because all that is required internally is a simple byte-by-byte comparison of the values. But if you use extended characters, binary sorting is not always semantically desirable. Characters that are A-like, such as Ä, ä, Å, and å, all sort after Z because the extended character A's are in the top 128 characters and Z is a standard ASCII character in the lower 128. If you deal with only ASCII characters (or otherwise don't care about the sort order of extended characters), you want case sensitivity, and you don't care about "dictionary" sorting, binary sorting is an ideal choice. In case-sensitive dictionary sorting, the letters *abcxyzABCXYZ* sort as *AaBbCcXxYyZz*. In binary sorting, all uppercase letters appear before any lowercase letters for example, *ABCXYZabcxyz*.

Specifying a Collation

During the installation of SQL Server 2005, you are asked to specify a default collation for your server. This is the collation that your system databases will use, and any new database you create will use this collation by default. In [Chapter 4](#), I'll describe how you can create a database with a different collation than the one chosen during installation. Two types of collation are available; the documentation calls them Windows Collation and SQL Collation.

[Figure 1-3](#) shows the installation dialog box in which you choose your collation. The two choices can be described as Windows Locale Collation Designator and SQL Collation. The dialog box doesn't use the term Windows Local Collation Designator; as you can see, it is described only as Collation Designator.

Figure 1-3. Two installation choices for collation and sort order



SQL Collation is intended for backward compatibility and basically addresses all three collation issues in a single description string: the code page for non-Unicode data, the sort order for non-Unicode data, and the sort order for Unicode data. In [Figure 1-3](#), you can see that a SQL Collation is highlighted in the lower window, even though I selected a collation from the top window by using the option button. The highlighted SQL Collation is actually the default for SQL Server 2005 installation on a U.S. English SQL Server, and the description of the SQL Collation is "Dictionary Order, Case-Insensitive, For Use With 1252 Character Set." The code page for non-Unicode data is 1252, the sort order for non-Unicode data is case insensitive, and the sort order for Unicode is Dictionary Order. The actual name of this collation is

SQL_Latin1_General_CP1_CI_AS, and you can use this name when referring to this collation by using SQL statements, as you'll see in later chapters. The prefix *SQL* indicates that this is a SQL Collation as opposed to a Windows Collation. The *Latin1_General* part indicates that the Unicode sort order corresponds to dictionary order, *CP1* indicates the default code page 1252, and *CI_AS* indicates case insensitive, accent sensitive. Other collation names use the full code page number; only 1252 has the special abbreviation *CP1*. For example,

SQL_Scandinavian_CP850_CS_AS is the SQL Collation using Scandinavian sort order for Unicode data, code page 850 for non-Unicode data, and a case-sensitive, accent-sensitive sort order.

The SQL Collations are intended only for backward compatibility with previous SQL Server versions. There is a fixed list of possible values for SQL Collations, and not every combination of sort orders, Unicode collations, and code pages is possible. Some combinations don't make any sense, and others are omitted just to keep the list of possible values manageable.

As an alternative to a SQL Collation for compatibility with a previous installation, you can define a collation based on a Windows locale. In Windows Collation, the Unicode data types always have the same sorting behavior as non-Unicode data types. A Windows Collation is the default for non-English SQL Server installations, with the specific default value for the collation taken from your Windows operating system configuration. You can choose to always sort by byte values for a binary sort, or you can choose case sensitivity and accent sensitivity.

Performance Considerations

Binary sorting uses significantly fewer CPU instructions than sort orders with defined sort values. So binary sorting is ideal if it fulfills your semantic needs. However, you won't pay a noticeable penalty for using either a simple case-insensitive sort order (for example, Dictionary Order, Case Insensitive) or a case-sensitive choice that offers better support than binary sorting for extended characters. Most large sorts in SQL Server tend to be I/O-bound, not CPU-bound, so the fact that fewer CPU instructions are used by binary sorting doesn't typically translate to a significant performance difference. When you sort a small amount of data that is not I/O-bound, the difference is minimal; the sort will be fast in both cases.

A more significant performance difference results if you choose a sort order of Case Insensitive, Uppercase Preference. Recall that this choice considers all values as equal for comparisons, which also includes indexing. Characters retain a unique tertiary sort order, however, so they might be treated differently by an ORDER BY clause. Uppercase Preference, therefore, can often require an additional sort operation in queries.

Consider a query that specifies `WHERE last_name >= 'Jackson'` `ORDER BY last_name`. If an index exists on the `last_name` field, the query optimizer will likely use it to find rows whose `last_name` value

is greater than or equal to *Jackson*. If your application has no need to use Uppercase Preference, the optimizer knows that because SQL Server will be retrieving records based on the order of *last_name*, there will be no need to physically sort the rows because they will be found in the index in the proper order and will be extracted in that order already. *Jackson*, *jackson*, and *JACKSON* are all qualifying rows. All are indexed and treated identically, and they simply appear in the result set in the order in which they were encountered. If Uppercase Preference is required for sorting, a subsequent sort of the qualifying rows is required to differentiate the rows even though the index can still be used for selecting qualifying rows. If many qualifying rows are present, the performance difference between the two cases (one needs an additional sort operation) can be dramatic. This doesn't mean you shouldn't choose Uppercase Preference. If your application requires those semantics, performance might well be a secondary concern. But you should be aware of the trade-off and decide which is most important to you.

Installing Multiple Instances of SQL Server

The last pre-installation decision I will discuss here is whether to install a default SQL Server instance or a named instance. SQL Server 2005 allows you to install multiple instances of SQL Server on a single computer. One instance of SQL Server can be referred to as the *default* instance, and all others must have names and are referred to as *named instances*. Each instance is installed separately and has its own default collation, its own system administrator password, and its own set of databases and valid users. If you are upgrading from SQL Server version 7.0, the upgraded instance is a default instance.

The ability to install multiple instances on a single machine provides benefits in a few different areas. First is the area of application

hosting. One site can provide machine resources and administration services for multiple companies. These might be small companies that don't have the time or resources (human or machine) to manage and maintain their own SQL servers. They might want to hire another company to "host" their SQL Server private or public applications. Each company needs full administrative privileges over its own data. However, if each company's data were stored in a separate database on a single SQL server, any SQL Server system administrator (sa) would be able to access any of the databases. Installing each company's data on a separate SQL Server instance allows each company to have full sa privileges for its own data but no access to any other company's data. In fact, each company might be totally unaware that other companies are using the same application server.

A second area in which multiple instances provide great benefits is server consolidation. Instead of having 10 separate machines to run 10 separate applications within a company, all the applications can be combined on one machine. With separate instances, each application can still have its own administrator and its own set of users and permissions. For the company, this means fewer boxes to administer and fewer operating system licenses required. For example, one Windows 2003 Advanced Server with 10 SQL Server instances costs less than 10 separate server boxes.

An added benefit to running multiple instances on one big computer is that you are more likely to take care of a more powerful computer, which will lead to greater reliability. With the kind of investment required to buy a machine capable of running 10 SQL Server 2005 instances, you'll probably want to set it up in its own climate-controlled data center instead of next to someone's desk. You'll meticulously maintain both the hardware and the software and let only trained engineers get near it. This means that reliability and availability will be greatly enhanced.

The third area in which multiple instances are beneficial is testing and support. Because the separate instances can be different versions of SQL Server or even the same version with different service packs installed, you can use one box for reproducing problem reports or testing bug fixes. You can verify which versions the problems can be reproduced on and which versions they can't. A support center can similarly use multiple instances to make sure that it can use exactly the same version that the customer has installed.

Installing Named Instances of SQL Server

When you install multiple SQL Server instances on a single machine, only one instance is the "default" instance. This is the SQL Server that is accessed by supplying the machine name as the name of the SQL Server. For example, to use SQL Server Management Studio to access the default SQL Server on my machine called TENAR, I simply type *TENAR* in the SQL Server text box in the Connect To SQL Server dialog box. Any other instances on the same machine will have an additional name, which must be specified along with the machine name. I can install a second SQL Server instance with its own name for example, *INSTANCE1*. Then I must enter *TENAR\INSTANCE1* in the initial text box to connect with Management Studio.

Note



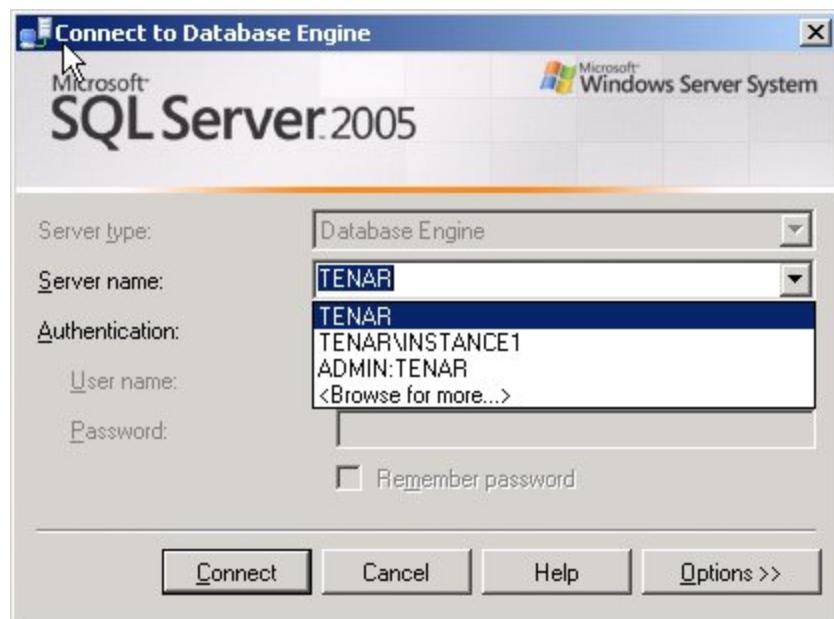
The instance name is not the same as the service name for the instance. And to make matters even more interesting, there is also a "display name" for the service, used in GUI tools such as the Services page in the Windows Computer

Manager. The service name of the default instance is always MSSQLSERVER with a display name of SQL Server (MSSQLServer). The service name of a named instance is MSSQL\$INSTANCE_NAME with a display name of SQL Server (INSTANCE_NAME). Service names are used when starting, stopping, or configuring the service through the operating system.

To make a connection to a SQL Server instance, you need to use the instance name. The name of the default instance is always the same as the name of the machine, and the name of the instance used when connecting to a named instance is MACHINE_NAME\INSTANCE_NAME.

[Figure 1-4](#) shows the connection box with the names of the two instances on my machine called TENAR. The default instance is just called TENAR, and my named instance, which I called INSTANCE1, must be specified as TENAR\INSTANCE1.

Figure 1-4. Choosing which instance of SQL Server to connect to



Your default instance can be SQL Server 7.0, SQL Server 2000, or SQL Server 2005. Named instances can be either SQL Server 2000 or SQL Server 2005. Microsoft specifies that you can have up to 50 instances of SQL Server 2005 Enterprise Edition or Developer Edition on a single machine, and up to 16 instances of other editions or versions. But this is only the supported limit. Nothing is hardcoded into the system to prevent you from installing additional instances as long as you have the resources on the machine to support them.

Each instance has its own separate directory for storing the server executables, and each instance's data files can be placed wherever you choose. During installation, you can specify the desired path for the program files and the data files. Each instance also has its own SQL Server Agent service. The service names for the default instance do not change they are MSSQLServer and

SQLServerAgent. For an instance named INSTANCE1, the services are named MSSQL\$ INSTANCE1 and SQLAGENT\$ INSTANCE1.

However, only one installation of the tools is used for all instances of the same version of SQL Server. That is, you will have only one set of tools to use with all your SQL Server 2005 instances and one to use for all your SQL Server 2000 instances.

Some of the services are instance-aware, and you'll get a separate service for each instance for each of the following:

- SQL Server Agent
- Analysis Server
- Report Server
- Full-Text Search

Instance-unaware services are shared among all installed SQL Server instances; they are not associated with a specific instance, are installed only once, and cannot be installed side by side.

Instance-unaware services in SQL Server 2005 include:

- Notification Services
- Integration Services
- SQL Server Browser
- SQL Server Active Directory Helper
- SQL Writer

Getting Ready to Install

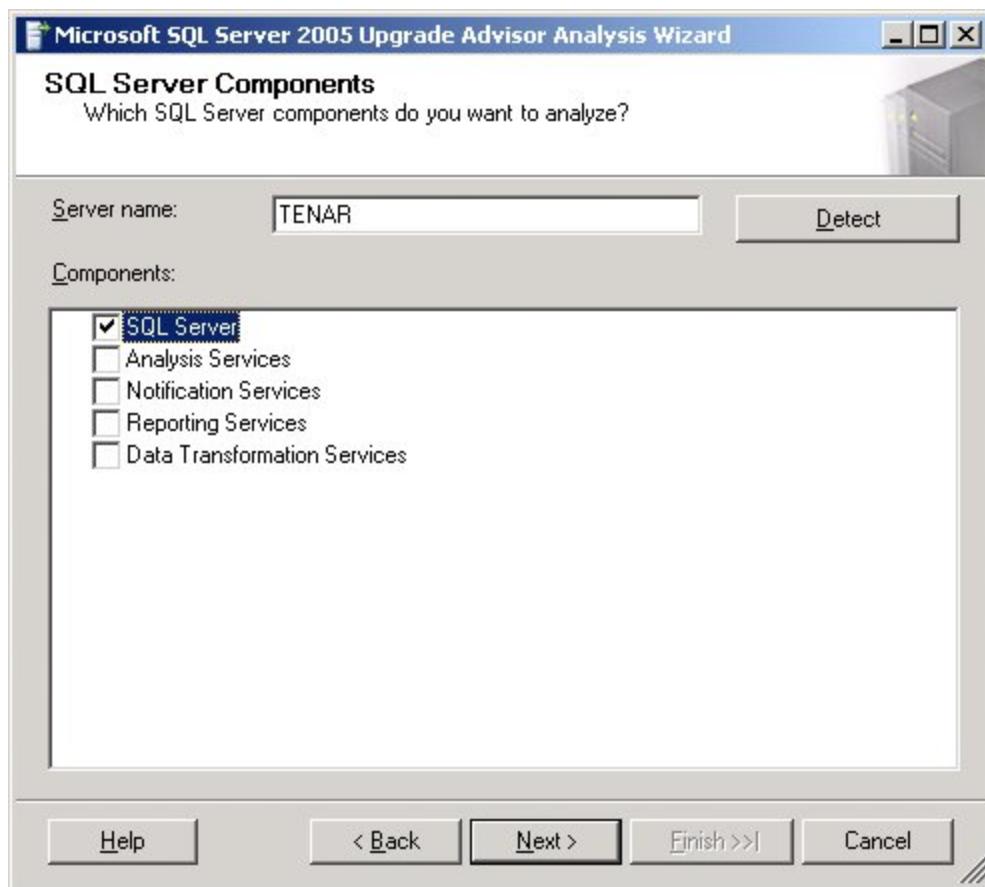
If you look again at the opening screen for SQL Server 2005 setup, you'll see that you have two other pre-installation choices in addition to the hardware and software requirements document. First, you can look at release notes that contain late-breaking or other important information to supplement the Books Online documentation. You should read this entire document before installing SQL Server 2005. In addition to late-breaking information, the release notes might contain information about bugs in the product or the documentation that may be fixed in a subsequent release. For example, the release notes for one of the CTP releases reported that the documentation for the dynamic management view `sys.dm_db_session_space_usage` was incorrect and reported the wrong names for the last two columns of output. They also report a product bug: the `partition_number` column in the `sys.dm_db_index_operational_stats` dynamic management function returns incorrect values.

SQL Server 2005 Upgrade Advisor

The final pre-installation option on the opening setup screen is useful only if you have existing SQL Server 7.0 or SQL Server 2000 databases that you will be upgrading to SQL Server 2005. The Upgrade Advisor is a tool for database developers and administrators to analyze SQL Server 7.0 and SQL Server 2000 servers before upgrading to SQL Server 2005. The Upgrade Advisor allows you to analyze your existing database components. It then provides reports that identify changes you must make due to deprecated features and new behaviors in SQL Server 2005.

The link on the opening screen installs the Upgrade Advisor, which you can then run as often as you need it. (An alternative way to install it is by launching SQLUASetup.msi in the \Servers\Redist\Upgrade Advisor subdirectory on the installation CD or from the downloaded setup files.) Each time you run the Upgrade Advisor, you can analyze one or more databases in a single instance of SQL Server 7.0 or SQL Server 2000. The initial dialog box of the Upgrade Advisor is shown in [Figure 1-5](#). You indicate the name of the server you want to analyze and select which components to analyze: database server, analysis services, notification services, reporting services, or data transformation services.

Figure 1-5. Choosing which components to analyze prior to upgrading



The Upgrade Advisor contains rules for items that must be fixed before the upgrade, rules for items that must be fixed after the upgrade, and rules to detect and report items that can be fixed at any time. Items that must be fixed before the upgrade are deemed "upgrade blockers" they will prevent the setup process from successfully installing SQL Server 2005. There are actually only five upgrade blockers for the SQL Server engine:

- **There is a database with an ID value of 32767.** SQL Server 2005 reserves this database ID value for a special hidden

database called the resource database, which I'll cover in [Chapter 4](#). A database with this ID must be detached (or dropped) before upgrading to SQL Server 2005. You can reattach the database either before or after doing the upgrade, but if you do it before, you'll need to verify that it didn't get the same database ID value.

- **Duplicate Security Identifiers (SIDs) exist.** If your SQL Server 7.0 or SQL Server 2000 database has two login IDs with the same SID, an upgrade to SQL Server 2005 will fail. You must remove all but one of the duplicates.
- **Login names match a fixed server role.** If a SQL Server 7.0 or SQL Server 2000 login name is the same as one of the fixed server roles, an upgrade to SQL Server 2005 will fail. You must change the login name.
- **A database has a user name "sys".** All SQL Server 2005 databases include a schema and user "sys" that is used for special purposes. If your database already has a regular user "sys", you must change the name before upgrading. If the user is not renamed, the database will be in a suspect state after the upgrade process and will be unavailable until the database is brought online.
- **A duplicate index name for a table or view exists.** If two indexes with the same name exist on the same object (table or view), the upgrade will fail. You must use `sp_rename` to rename one of the indexes.

To Migrate or Upgrade?

Obviously, the Upgrade Advisor is useful only if you have existing SQL Server databases. But if you have existing databases, you should first consider whether it is better to upgrade or to install a new instance for SQL Server 2005 and then migrate the existing data to the new SQL Server. Each choice has its advantages. With either choice, you'll run the same Setup.exe application; during the setup process, you will indicate whether you want to install a new SQL Server or upgrade an existing SQL Server.

Migrating

To migrate a database from an existing SQL Server instance (called the *source*) to SQL Server 2005 (the *destination*), you'll need to choose the option to install a new SQL Server, and as a follow-up, you will move your objects and data to the new server. All the following methods migrate one database at a time from the source instance to the destination instance, so in general, I'll be talking about migrating a single database rather than migrating an instance. To migrate an instance, each database must be migrated. The benefits of migrating existing databases to SQL Server 2005 include the following:

- The migration option provides more granular control over the upgrade process.
- Having the new and old instances side by side helps with testing and verification.

- The source instance of SQL Server stays online during the entire setup process.
- You have greater flexibility in case of a failure during the setup operation.

The drawbacks of the migrate method include the following:

- Extra hardware will probably be required to support two side-by-side installations.
- You might have to modify your application code to point to the destination installation because it will have a new instance name.
- You can't migrate from a default instance to a default instance on the same machine because only one instance on any machine is considered the default. Even if the source database is no longer needed and is removed, there is no way to convert a named instance to a default instance.

For moving or copying your data to the second SQL Server instance, several tools are available.

- **Detach and attach** SQL Server allows you to detach a database from one SQL Server instance and attach it to another instance. If you want to keep the data available on the source instance, you can copy the database files after detaching the database and then attach the files to both the source and the destination SQL Server 2005 instances. Detach and attach are available through the graphical tools in SQL Server 2000 and later.

- **Backup and restore** The source databases can be backed up from the original SQL Server instance and restored to the new instance. No file copy operation is needed.
- **Copy Database Wizard** This wizard for copying databases is available through SQL Server Integration Services (SSIS), which is the successor to Data Transformation Services (DTS). The wizard allows you to choose which objects from the source database to copy, so you can migrate only a subset of your database if you want.
- **Manual migration** If you want the destination database to be based on the original version of the source database, perhaps before you make some schema changes, you can use a script that creates all the objects on the destination SQL Server 2005 database. You can copy data using bcp from files containing the original data (assuming you have these available).

Upgrading

Upgrading your source databases means that the setup program converts your database to SQL Server 2005 in place. The benefits of this direct upgrade include the following:

- For small systems, upgrading is faster and easier than migrating.
- No additional hardware is required to upgrade, as long as the existing hardware meets the requirements for SQL Server 2005.

- Your applications do not need any modifications regarding connections and can continue pointing to the same server name.

The drawbacks include the following:

- You have less control over what objects and data are included in the SQL Server database and which are not.
- The SQL Server instance will be offline during part of the upgrade process.
- In-place upgrade is not the recommended approach for all SQL Server components. For example, if you have Analysis Services cubes, it is recommended that they be migrated instead of upgraded to take best advantage of all the new SQL Server capabilities.

Upgrade Internals

If you choose to upgrade instead of migrate, keep in mind that it is an in-place operation, and one SQL Server instance is converted to a new version. The bits of the executable are replaced, and the data files are modified. During much of this process, the SQL Server engine will be unavailable, and once the process passes a certain point, the decision to upgrade is irrevocable and you cannot go back to the previous version.

The upgrade process goes through the following steps:

- **Verify prerequisites.** Your original instance is completely available during this step. If you have run the Upgrade Advisor and followed its recommendations, you shouldn't have any

missing prerequisites.

- **Check for upgrade blockers.** Again, your original instance is completely available during this step. If you have any upgrade blockers, the install process will fail at this point, but if you followed all the recommendations of the Upgrade Advisor, you shouldn't have any at this point.
- **Install SQL Server 2005 binaries in a new directory location.** The default location for the server-side executables is C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL. The default location for the client-side executables is C:\Program Files\Microsoft SQL Server\90.
- **Stop the execution of the previous version.** Obviously, the previous instance is not available at this point.
- **Point the service to the new executable.** This is an in-place upgrade, so the service name for the SQL server will stay the same. The registry is updated to map the location of the new executables to the existing service name.
- **Start the new SQL Server service in single-user mode.** Once the SQL Server service starts under the new name, it is considered to be at the point of no return. Even if the installation is canceled at this point, you cannot cleanly revert to your old version under the original service name.
- **Attach the resource database.** After the new service starts running, a new hidden system database is attached to the instance. I'll explain more about this special resource database in [Chapter 4](#).
- **Stop and restart SQL Server service.** Attaching the resource database is the only thing that needs to be done while in single-user mode. Once that's done, the SQL Server service can be

stopped and restarted in multi-user mode.

- **Start updating all databases.** The new SQL server is partially available at this point. Each database becomes available as it is upgraded. If the upgrade is from SQL Server 2000 to SQL Server 2005, this database update is a relatively fast operation. The primary change made to each database is the introduction of new metadata structures (as discussed in [chapters 4](#) and [6](#)). If you're upgrading from SQL Server 7.0 to SQL Server 2005, the changes are a little more invasive because they entail changes to the physical database files. SQL Server 2000 introduced two new kinds of allocation structures that are stored in the first few pages of each file (as well as every 512-K page thereafter at fixed locations), and these special pages are needed in SQL Server 2005 data files. If these specific pages are used in your SQL Server 7.0 data files for other purposes, such as storing table or index data, that data will have to be moved. I'll talk about these special allocation structures in [Chapter 4](#).
- **Execute replication and SQL Agent upgrade scripts.** Depending on the tasks you defined in your original SQL server, additional scripts might have to be run to prepare these tasks to run under SQL Server 2005. In particular, execution and SQL Agent tasks will need a bit of upgrading.
- **Uninstall old binaries.** The original executable and supporting files are no longer needed, so they are removed as a final step in the upgrade process. Your new instance of SQL Server 2005 is now fully available.

Post-Upgrade Operations

As mentioned earlier, if you choose to upgrade rather than migrate, you will not be able to test the new version while still running the previous version. Upgrading is basically an all-or-nothing operation. Once you upgrade, all your data management applications and operations will run on SQL Server 2005 and you will not be able to revert to the previous version unless you completely reinstall SQL Server 2000 and restore all your old data from backups.

There are several tasks you might need to perform immediately after upgrading to SQL Server 2005. Some tasks suggested by the Upgrade Advisor might be peripheral to the relational database engine therefore, I will not cover them here. (These tasks might include migrating DTS packages to SSIS, reconfiguring log shipping, and rebuilding Analysis Services cubes.) The final screen you'll see when installing SQL Server 2005 ([Figure 1-6](#)) suggests that you run the Surface Area Configuration Tool, which gives you the choice of configuring your services and connections or your SQL Server 2005 features. When using the Surface Area Configuration Tool to configure your services and connections, you can control which SQL Server-related services will start automatically when your machine starts and which will have to be started manually. You should also look at the configuration for features; you might be surprised which features are disabled by default. For security and safety reasons, some of the most widely advertised features of SQL Server 2005 will not be available unless you open the Surface Area Configuration Tool and enable them. These include CLR integration and Native XML Web Services. [Figure 1-7](#) shows the screen from the Surface Area Configuration Tool that lists all the features that must be enabled before they can be used.

Figure 1-6. The final setup screen showing the option to open the Surface Area Configuration Tool

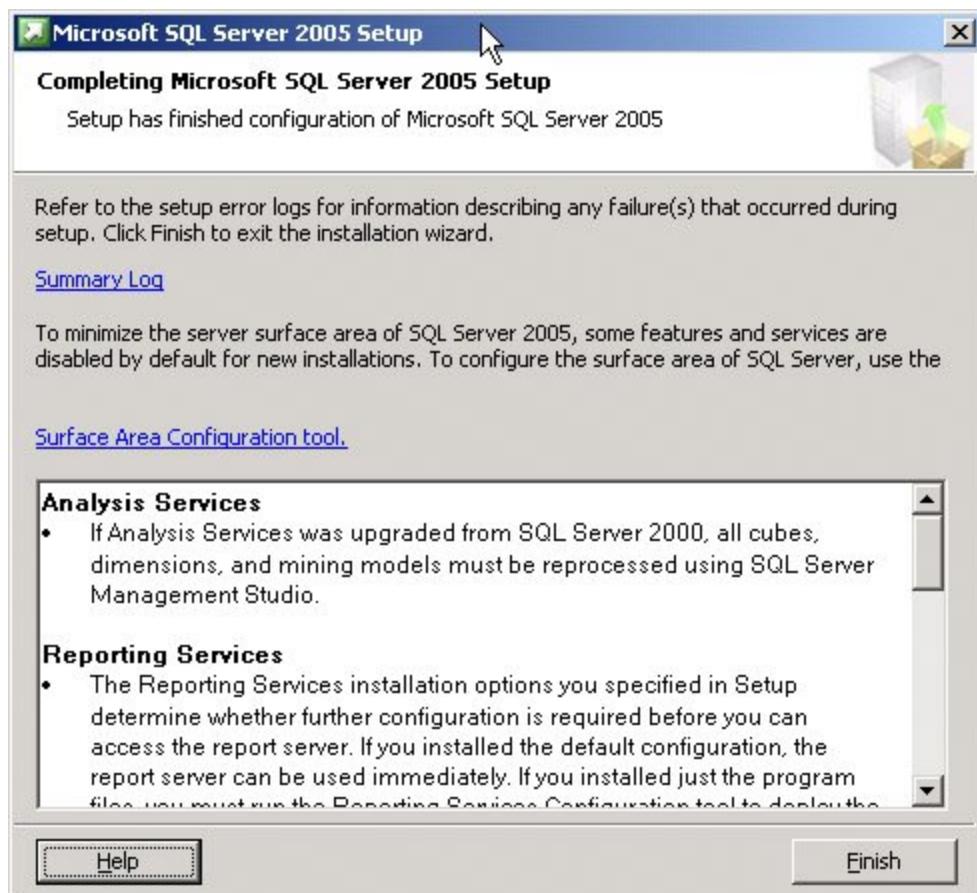
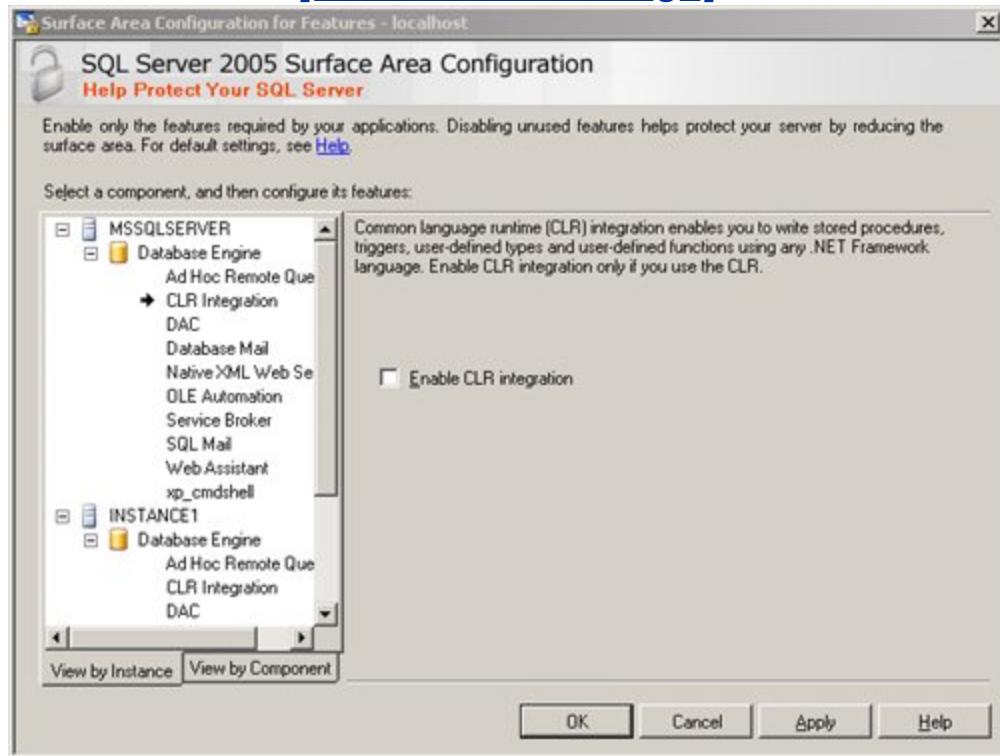


Figure 1-7. Using the Surface Area Configuration Tool to disable unused or unneeded features

[\[View full size image\]](#)



The most important post-installation tasks related to the core database engine include the following:

- **Update statistics.** The optimizer is constantly being improved with each SQL Server version and each service pack, and so are the techniques by which SQL Server accumulates and maintains the statistics that the optimizer bases its decisions

on. Although the SQL Server 2005 optimizer can use the statistics gathered by earlier versions, it comes up with the best query plans when the statistics have been gathered using the most up-to-date algorithms. I'll discuss statistics in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

- **Change database COMPATIBILITY level.** The database's compatibility level controls whether new reserved words will be recognized in unquoted identifier names and whether certain syntax constructs will be recognized. In an upgraded database, the database compatibility level is set to the level of the source server so that object names will continue to be recognized and the previously valid syntax will continue to work. However, if a database stays in an earlier compatibility level, some of the new functionality of the new SQL Server version will be unavailable. The new reserved words for SQL Server 2005 are PIVOT, UNPIVOT, REVERT, and TABLESAMPLE. If you have any objects using these words as object names, you should stay in your older compatibility level (70 or 80) until you have changed the names and modified the code that refers to these names. However, until you change the compatibility level to 90 (for SQL Server 2005), you will not be able to use any of these new reserved words for the purpose they were intended for in SQL Server 2005.

Among the most drastic behavior changes when you move from compatibility level 70 or 80 to 90 are that you can no longer use the operators *= and =* to specify outer joins and that you must use the OUTER JOIN keywords. (Inner and outer joins will be covered in detail in another volume in this series, *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.)

Another major change, if you have used optimizer hints in any of your code, is that the keyword WITH is now required in most cases with the use of table hints. In previous versions of SQL

Server, WITH was optional. Hints will be discussed in several places in this series of books, including [Chapter 8](#) of this volume, where I'll discuss the hints that control locking and isolation.

More than two dozen other new behaviors are introduced with compatibility level 90. See the entry for `sp_dbcmptlevel` in Books Online for the full list.

After upgrading to SQL Server 2005, you'll want to make sure that the new SQL Server service starts with no errors and start testing your applications. You'll want to monitor your system activity and compare the resource usage under the new version with baselines you recorded when running your applications under the previous version.

Selecting Components

The basic installation process is the same whether you are upgrading, migrating, or creating a new SQL Server 2005 instance. Regardless of what components you previously installed, when you run setup for SQL Server 2005, you'll need to choose which components you want to include with the installation. (See [Figure 1-1](#) for the list.) I am not running on a Windows cluster, so the option to install a SQL Server failover cluster is not available. Also, the Virtual PC where I am installing SQL Server 2005 does not have IIS installed, and that is a prerequisite for Reporting Services, so the option to install Reporting Services is also grayed out.

Following is the list of all the optional components you can select for installation or upgrade. Note that the database engine itself does not have to be selected. You can choose to simply install the client components, Analysis Services, or another combination of SQL Server 2005 components.

- SQL Server Database Services (Database Engine)
- Analysis Services
- Reporting Services
- Notification Services
- Integration Services
- Workstation components, Books Online, and development tools

Depending on which of these components you select, additional choices will be available on the subsequent screens.

SQL Server Database Services (Database Engine)

The database engine is the core service for storing, processing, and securing data. It provides controlled access and rapid transaction processing to meet the requirements of the most demanding data-consuming applications in your enterprise.

The core database engine is the main focus of this book. If you choose Advanced Options from the main component screen, you'll see that you can select some additional items to include as part of the database engine.

- **Replication** Replication is a set of technologies for copying and distributing data and database objects from one database to another and then synchronizing between databases to maintain consistency. By using replication, you can distribute data to different locations and to remote or mobile users by means of local and wide area networks, dial-up connections, wireless connections, and the Internet.
- **Full-text search** Full-text search lets you issue full-text queries against plain character-based data in SQL Server tables. Full-text queries can include words and phrases or multiple forms of a word or a phrase.
- **Other options** The database engine also provides additional support for sustaining high availability using failover clustering. If you are installing on a system that already has MSCS installed, you will see a dialog box for specifying the Virtual

SQL Server Name right after the dialog box that asks for your instance name. Although installing SQL Server 2005 on a cluster is straightforward, the internals of clustering and the issues involved with configuring your machine and managing your cluster are beyond the scope of this book.

Analysis Services

Analysis Services delivers online analytical processing (OLAP) and data mining functionality for business intelligence applications. Analysis Services supports OLAP by allowing you to design, create, and manage multidimensional structures that contain data aggregated from other data sources, such as relational databases. For data mining applications, Analysis Services enables you to design, create, and visualize data mining models. You can construct these mining models from other data sources by using a wide variety of industry-standard data mining algorithms.

Reporting Services

Reporting Services delivers enterprise, Web-enabled reporting functionality so you can create reports that draw content from a variety of data sources, publish reports in various formats, and centrally manage security and subscriptions.

Notification Services

Notification Services is an environment for developing and deploying applications that generate and send notifications. You can use Notification Services to generate and send timely, personalized

messages to thousands or millions of subscribers, and you can deliver the messages to a variety of devices.

Integration Services

Integration Services is a platform for building high-performance data integration solutions, including packages that provide extract, transform, and load (ETL) processing for data warehousing. SQL Server Integration Services in SQL Server 2005 replaces SQL Server 2000 Data Transformation Services (DTS).

Workstation Components, Books Online, and Development Tools

The Workstation Components, Books Online, and Development Tools selection offers many options. You'll definitely want to install the management tools and SQL Server Books Online. Most of the other optional components don't take up too much disk space, so for a test server, I suggest installing them all until you can determine whether you need them. You will also have a chance to choose which sample applications you want; you should choose based on the components you are installing and testing. For example, if you are not installing Analysis Services, you need not install the sample applications for Analysis Services. I also suggest installing the sample database called AdventureWorks, at least on a test server. Many of the examples in Books Online refer to this sample database. Note that there is also a sample database for Analysis Services called AdventureWorksDW. Do not use that one for the examples in this book. You'll also find that many of the articles written about SQL Server, as well as many books (including this one), use examples that include sample data from the AdventureWorks database.

If you're upgrading from SQL Server 7.0 or SQL Server 2000, one of the most important changes you'll notice about the client tools is that the two graphical tools from the earlier versions, SQL Enterprise Manager and SQL Query Analyzer, have been combined into a single tool called SQL Server Management Studio. This tool allows you to create, test, and analyze queries, and it also allows you to manage all the administrative tasks such as creating jobs and alerts, taking or scheduling backups, managing your metadata and scripts, and monitoring any number of SQL Server instances. One reason that the development and administrative tasks were combined into a single tool is that many people who work with SQL Server are responsible for both those jobs. SQL Server developers often oversee aspects of database management, and many database administrators have to develop, test, and tune queries. The Management Studio tool gives you one tool for all your direct access to SQL Server.

If you prefer a query tool with a much smaller footprint, either for testing queries or running commands to troubleshoot problems, you can use another client tool: SQLCMD. SQLCMD is a command-line tool that is intended to replace osql. It allows you to run pre-created scripts and capture the output, or work interactively, submitting single batches and then inspecting the results. SQLCMD is much richer than osql because it allows full parameterization of any pre-created scripts.

Both SQL Server Management Studio and SQLCMD provide an option to connect to SQL Server 2005 using a special reserved connection called Dedicated Administrator Connection, or DAC. This connection is guaranteed to be always available, even if misuse of server resources doesn't allow any other connections to SQL Server to be opened. I'll tell you more about DAC in the next chapter, when I talk about SQL Server architecture.

Summary

SQL Server 2005 offers unprecedented ease of installation for a database server of its caliber. But you should understand your choices in terms of hardware, security, and sort order before you go too far down the path of a major application rollout. SQL Server installation is also fast and reliable. You need very little custom configuration because it configures itself as needed.

The SQL Server installation process offers options for unattended and remote installation, for installing multiple instances of SQL Server on the same machine, for installing a Desktop version, and for client toolsonly installation. Utilities are provided for changing many of the options chosen during the initial installation, but to add new components, you must keep your original CD handy or copy its contents to a network share.

Chapter 2. SQL Server 2005 Architecture

In this chapter:

[Components of the SQL Server Engine](#) [29](#)

[Memory](#) [49](#)

[Final Words](#) [63](#)

Throughout this book, I'll drill down into the details of specific features of the Microsoft SQL Server Database Engine. In this chapter, you'll get a high-level view of the components of that engine and how they work together. My goal is to help you understand how the topics covered in subsequent chapters fit into the overall operations of the engine.

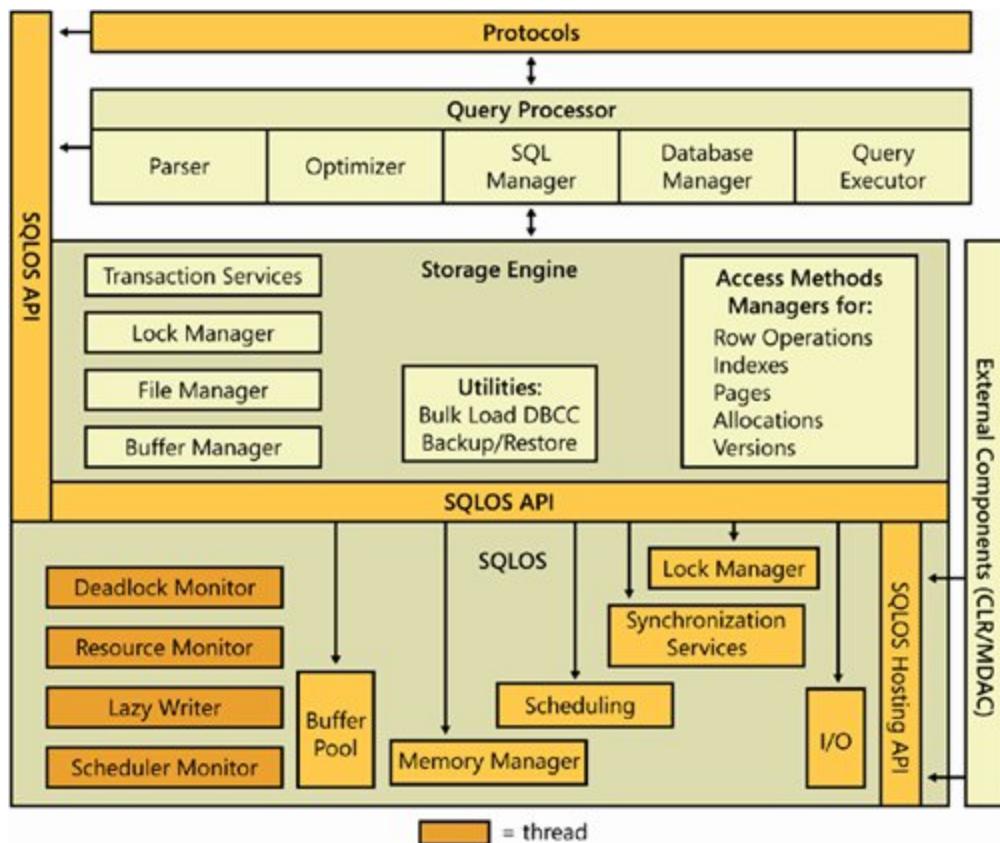
In this chapter, however, I'll dig deeper into one big area of the SQL Server engine that isn't covered later: the SQL operating system (SQLOS) and, in particular, the components related to memory management and scheduling.

Components of the SQL Server Engine

[Figure 2-1](#) shows the general architecture of SQL Server, which has four major components (three of whose subcomponents are listed): protocols, the relational engine (also called the Query Processor), the storage engine, and the SQLOS. Every batch submitted to SQL Server for execution, from any client application, must interact with these four components. (For simplicity, I've made some minor omissions and simplifications and ignored certain "helper" modules among the subcomponents.)

Figure 2-1. The major components of the SQL Server database engine

[[View full size image](#)]



The protocol layer receives the request and translates it into a form that the relational engine can work with, and it also takes the final results of any queries, status messages, or error messages and translates them into a form the client can understand before sending them back to the client. The relational engine layer accepts SQL batches and determines what to do with them. For Transact-SQL queries and programming constructs, it parses, compiles, and optimizes the request and oversees the process of executing the batch. As the batch is executed, if data is needed, a request for that

data is passed to the storage engine. The storage engine manages all data access, both through transaction-based commands and bulk operations such as backup, bulk insert, and certain DBCC (Database Consistency Checker) commands. The SQLOS layer handles activities that are normally considered to be operating system responsibilities, such as thread management (scheduling), synchronization primitives, deadlock detection, and memory management, including the buffer pool.

Observing Engine Behavior

SQL Server 2005 introduces a suite of new system objects that allow developers and database administrators to observe much more of the internals of SQL Server than before. These metadata objects are called dynamic management views (DMVs) and dynamic management functions (DMFs). You can access them as if they reside in the new sys schema, which exists in every SQL Server 2005 database, but they are not real objects. They are similar to the pseudo-tables used in SQL Server 2000 for observing the active processes (sysprocesses) or the contents of the plan cache (syscacheobjects). However, the pseudo-tables in SQL Server 2000 do not provide any tracking of detailed resource usage and are not always directly usable to detect resource problems or state changes. Some of the DMVs and DMFs do allow tracking of detailed resource history, and there are more than 80 such objects that you can directly query and join with SQL SELECT statements. The DMVs and DMFs expose changing server state information that might span multiple sessions, multiple transactions, and multiple user requests. These objects can be used for diagnostics, memory and process tuning, and monitoring across all sessions in the server.

The DMVs and DMFs aren't based on real tables stored in database files but are based on internal server structures, some of which I'll

discuss in this chapter. I'll discuss further details about the DMVs and DMFs in various places in this book, where the contents of one or more of the objects can illuminate the topics being discussed. The objects are separated into several categories based on the functional area of the information they expose. They are all in the sys schema and have a name that starts with *dm_*, followed by a code indicating the area of the server with which the object deals. The main categories I'll address are:

- ***dm_exec_**** Contains information directly or indirectly related to the execution of user code and associated connections. For example, *sys.dm_exec_sessions* returns one row per authenticated session on SQL Server. This object contains much of the same information that *sysprocesses* contains in SQL Server 2000 but has even more information about the operating environment of each sessions.
- ***dm_os_**** Contains low-level system information such as memory, locking, and scheduling. For example, *sys.dm_osSchedulers* is a DMV that returns one row per scheduler. It is primarily used to monitor the condition of a scheduler or to identify runaway tasks.
- ***dm_tran_**** Contains details about current transactions. For example, *sys.dm_tran_locks* returns information about currently active lock resources. Each row represents a currently active request to the lock management component for a lock that has been granted or is waiting to be granted. This object replaces the pseudo table *syslockinfo* in SQL Server 2000.
- ***dm_io_**** Keeps track of input/output activity on network and disks. For example, the function *sys.dm_io_virtual_file_stats* returns I/O statistics for data and log files. This object replaces the table-valued function *fn_virtualfilestats* in SQL Server 2000.

- ***dm_db_**** Contains details about databases and database objects such as indexes. For example, *sys.dm_db_index_physical_stats* is a function that returns size and fragmentation information for the data and indexes of the specified table or view. This function replaces *DBCC SHOWCONTIG* in SQL Server 2000.

SQL Server 2005 also has dynamic management objects for its functional components; these include objects for monitoring full-text search catalogs, service broker, replication, and the common language runtime (CLR).

Now let's look at the major SQL Server engine modules.

Protocols

When an application communicates with the SQL Server Database Engine, the application programming interfaces (APIs) exposed by the protocol layer formats the communication using a Microsoft-defined format called a *tabular data stream (TDS) packet*. There are Net-Libraries on both the server and client computers that encapsulate the TDS packet inside a standard communication protocol, such as TCP/IP or Named Pipes. On the server side of the communication, the Net-Libraries are part of the Database Engine, and that protocol layer is illustrated in [Figure 2-1](#). On the client side, the Net-Libraries are part of the SQL Native Client. The configuration of the client and the instance of SQL Server determine which protocol is used.

SQL Server can be configured to support multiple protocols simultaneously, coming from different clients. Each client connects to SQL Server with a single protocol. If the client program does not know which protocols SQL Server is listening on, you can configure the client to attempt multiple protocols sequentially. In [Chapter 3](#), I'll

discuss how you can configure your machine to use one or more of the available protocols. The following protocols are available:

- **Shared Memory** The simplest protocol to use, with no configurable settings. Clients using the Shared Memory protocol can connect only to a SQL Server instance running on the same computer, so this protocol is not useful for most database activity. Use this protocol for troubleshooting when you suspect that the other protocols are configured incorrectly. Clients using MDAC 2.8 or earlier cannot use the Shared Memory protocol. If such a connection is attempted, the client is switched to the Named Pipes protocol.
- **Named Pipes** A protocol developed for local area networks (LANs). A portion of memory is used by one process to pass information to another process, so that the output of one is the input of the other. The second process can be local (on the same computer as the first) or remote (on a networked computer).
- **TCP/IP** The most widely used protocol over the Internet. TCP/IP can communicate across interconnected networks of computers with diverse hardware architectures and operating systems. It includes standards for routing network traffic and offers advanced security features. Enabling SQL Server to use TCP/IP requires the most configuration effort, but most networked computers are already properly configured.
- **Virtual Interface Adapter (VIA)** A protocol that works with VIA hardware. This is a specialized protocol; configuration details are available from your hardware vendor.

Tabular Data Stream Endpoints

SQL Server 2005 also introduces a new concept for defining SQL Server connections: the connection is represented on the server end by a TDS endpoint. During setup, SQL Server creates an endpoint for each of the four Net-Library protocols supported by SQL Server, and if the protocol is enabled, all users have access to it. For disabled protocols, the endpoint still exists but cannot be used. An additional endpoint is created for the dedicated administrator connection (DAC), which can be used only by members of the sysadmin fixed server role. (I'll discuss the DAC in more detail shortly.)

The Relational Engine

As mentioned earlier, the relational engine is also called the query processor. It includes the components of SQL Server that determine exactly what your query needs to do and the best way to do it. By far the most complex component of the query processor, and maybe even of the entire SQL Server product, is the query optimizer, which determines the best execution plan for the queries in the batch. The optimizer is discussed in great detail in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*; in this section, I'll give you just a high-level overview of the optimizer, as well as of the other components of the query processor.

The relational engine also manages the execution of queries as it requests data from the storage engine and processes the results returned. Communication between the relational engine and the storage engine is generally in terms of OLE DB row sets. (*Row set* is the OLE DB term for a *result set*.) The storage engine comprises the components needed to actually access and modify data on disk.

The Command Parser

The command parser handles Transact-SQL language events sent to SQL Server. It checks for proper syntax and translates Transact-SQL commands into an internal format that can be operated on. This internal format is known as a *query tree*. If the parser doesn't recognize the syntax, a syntax error is immediately raised that identifies where the error occurred. However, non-syntax error messages cannot be explicit about the exact source line that caused the error. Because only the command parser can access the source of the statement, the statement is no longer available in source format when the command is actually executed.

The Query Optimizer

The query optimizer takes the query tree from the command parser and prepares it for execution. Statements that can't be optimized, such as flow-of-control and DDL commands, are compiled into an internal form. The statements that are optimizable are marked as such and then passed to the optimizer. The optimizer is mainly concerned with the DML statement *SELECT, INSERT, UPDATE, and DELETE*, which can be processed in more than one way, and it is the optimizer's job to determine which of the many possible ways is the best. It compiles an entire command batch, optimizes queries that are optimizable, and checks security. The query optimization and compilation result in an execution plan.

The first step in producing such a plan is to *normalize* each query, which potentially breaks down a single query into multiple, fine-grained queries. After the optimizer normalizes a query, it *optimizes* it, which means it determines a plan for executing that query. Query optimization is cost based; the optimizer chooses the plan that it determines would cost the least based on internal metrics that include estimated memory requirements, CPU utilization, and number of required I/Os. The optimizer considers the type of statement requested, checks the amount of data in the various

tables affected, looks at the indexes available for each table, and then looks at a sampling of the data values kept for each index or column referenced in the query. The sampling of the data values is called *distribution statistics*. (I'll discuss this topic further in [Chapter 7](#).) Based on the available information, the optimizer considers the various access methods and processing strategies it could use to resolve a query and chooses the most cost-effective plan.

The optimizer also uses pruning heuristics to ensure that optimizing a query doesn't take longer than it would take to simply choose a plan and execute it. The optimizer doesn't necessarily do exhaustive optimization. Some products consider every possible plan and then choose the most cost-effective one. The advantage of this exhaustive optimization is that the syntax chosen for a query will theoretically never cause a performance difference, no matter what syntax the user employed. But with a complex query, it could take much longer to estimate the cost of every conceivable plan than it would to accept a good plan, even if not the best one, and execute it.

After normalization and optimization are completed, the normalized tree produced by those processes is compiled into the execution plan, which is actually a data structure. Each command included in it specifies exactly which table will be affected, which indexes will be used (if any), which security checks must be made, and which criteria (such as equality to a specified value) must evaluate to TRUE for selection. This execution plan might be considerably more complex than is immediately apparent. In addition to the actual commands, the execution plan includes all the steps necessary to ensure that constraints are checked. Steps for calling a trigger are slightly different from those for verifying constraints. If a trigger is included for the action being taken, a call to the procedure that comprises the trigger is appended. If the trigger is an *instead-of* trigger, the call to the trigger's plan replaces the actual data modification command. For *after* triggers, the trigger's plan is branched to right after the plan for the modification statement that

fired the trigger, before that modification is committed. The specific steps for the trigger are not compiled into the execution plan, unlike those for constraint verification.

A simple request to insert one row into a table with multiple constraints can result in an execution plan that requires many other tables to also be accessed or expressions to be evaluated. In addition, the existence of a trigger can cause many additional steps to be executed. The step that carries out the actual INSERT statement might be just a small part of the total execution plan necessary to ensure that all actions and constraints associated with adding a row are carried out.

The SQL Manager

The SQL manager is responsible for everything related to managing stored procedures and their plans. It determines when a stored procedure needs recompilation, and it manages the caching of procedure plans so that other processes can reuse them.

The SQL manager also handles autoparameterization of queries. In SQL Server 2005, certain kinds of ad hoc queries are treated as if they were parameterized stored procedures, and query plans are generated and saved for them. SQL Server can save and reuse plans in several other ways, but in some situations using a saved plan might not be a good idea. For details on autoparameterization and reuse of plans, see *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

The Database Manager

The database manager handles access to the metadata needed for query compilation and optimization, making it clear that none of

these separate modules can be run completely separately from the others. The metadata is stored as data and is managed by the storage engine, but metadata elements such as the datatypes of columns and the available indexes on a table must be available during the query compilation and optimization phase, before actual query execution starts.

The Query Executor

The query executor runs the execution plan that the optimizer produced, acting as a dispatcher for all the commands in the execution plan. This module steps through each command of the execution plan until the batch is complete. Most of the commands require interaction with the storage engine to modify or retrieve data and to manage transactions and locking.

The Storage Engine

The SQL Server storage engine has traditionally been considered to include all the components involved with the actual processing of data in your database. SQL Server 2005 separates out some of these components into a module called the SQLOS, which I'll describe shortly. In fact, the SQL Server storage engine team at Microsoft actually encompasses three areas: access methods, transaction management, and the SQLOS. For the purposes of this book, I'll consider all the components that Microsoft does not consider part of the SQLOS to be part of the storage engine.

Access Methods

When SQL Server needs to locate data, it calls the access methods code. The access methods code sets up and requests scans of data pages and index pages and prepares the OLE DB row sets to return to the relational engine. Similarly when data is to be inserted, the access methods code can receive an OLE DB row set from the client. The access methods code contains components to open a table, retrieve qualified data, and update data. The access methods code doesn't actually retrieve the pages; it makes the request to the buffer manager, which ultimately serves up the page in its cache or reads it to cache from disk. When the scan starts, a look-ahead mechanism qualifies the rows or index entries on a page. The retrieving of rows that meet specified criteria is known as a *qualified retrieval*. The access methods code is employed not only for queries (selects) but also for qualified updates and deletes (for example, UPDATE with a WHERE clause) and for any data modification operations that need to modify index entries.

The Row and Index Operations

You can consider row and index operations to be components of the access methods code because they carry out the actual method of access. Each component is responsible for manipulating and maintaining its respective on-disk data structuresnamely, rows of data or B-tree indexes, respectively. They understand and manipulate information on data and index pages.

The row operations code retrieves, modifies, and performs operations on individual rows. It performs an operation within a row, such as "retrieve column 2" or "write this value to column 3." As a result of the work performed by the access methods code, as well as by the lock and transaction management components (discussed shortly), the row is found and appropriately locked as part of a transaction. After formatting or modifying a row in memory, the row operations code inserts or deletes a row. There are special

operations that the row operations code needs to handle if the data is a Large Object (LOB) datatypetext, image, or ntext or if the row is too large to fit on a single page and needs to be stored as overflow data. We'll look at the different types of page storage structures in [Chapter 6](#).

The index operations code maintains and supports searches on B-trees, which are used for SQL Server indexes. An index is structured as a tree, with a root page and intermediate-level and lower-level pages (or branches). A B-tree groups records that have similar index keys, thereby allowing fast access to data by searching on a key value. The B-tree's core feature is its ability to balance the index tree. (*B* stands for *balanced*.) Branches of the index tree are spliced together or split apart as necessary so the search for any given record always traverses the same number of levels and therefore requires the same number of page accesses.

Page Allocation Operations

The allocation operations code manages a collection of pages as named databases and keeps track of which pages in a database have already been used, for what purpose they have been used, and how much space is available on each page. Each database is a collection of 8-kilobyte (KB) disk pages that are spread across one or more physical files. (In [Chapter 4](#), you'll find more details about the physical organization of databases.)

SQL Server uses eight types of disk pages: data pages, LOB pages, index pages, Page Free Space (PFS) pages, Global Allocation Map and Shared Global Allocation Map (GAM and SGAM) pages, Index Allocation Map (IAM) pages, Bulk Changed Map (BCM) pages, and Differential Changed Map (DCM) pages. All user data is stored on data or LOB pages, and all index rows are stored on index pages. PFS pages keep track of which pages in a database are available to hold new data. Allocation pages (GAMs,

SGAMs, and IAMs) keep track of the other pages. They contain no database rows and are used only internally. Bulk Changed Map pages and Differential Changed Map pages are used to make backup and recovery more efficient. I'll explain these types of pages in [Chapter 6](#) and [Chapter 7](#).

Versioning Operations

Another type of data access new to SQL Server 2005 is access through the version store. Row versioning allows SQL Server to maintain older versions of changed rows. SQL Server's row versioning technology supports snapshot isolation as well as other features of SQL Server 2005, including online index builds and triggers, and it is the versioning operations code that maintains row versions for whatever purpose they are needed.

[Chapters 4, 6, and 7](#) deal extensively with the internal details of the structures that the access methods code works with: databases, tables, and indexes.

Transaction Services

A core feature of SQL Server is its ability to ensure that transactions are *atomic*; that is, all or nothing. In addition, transactions must be durable, which means that if a transaction has been committed, it must be recoverable by SQL Server no matter what even if a total system failure occurs 1 millisecond after the commit was acknowledged. There are actually four properties that transactions must adhere to, called the ACID properties: atomicity, consistency, isolation, and durability. I'll discuss all four of these properties in [Chapter 8](#), when I discuss transaction management and concurrency issues.

In SQL Server, if work is in progress and a system failure occurs before the transaction is committed, all the work is rolled back to the state that existed before the transaction began. Write-ahead logging makes it possible to always roll back work in progress or roll forward committed work that has not yet been applied to the data pages. Write-ahead logging ensures that the record of each transaction's changes are captured on disk in the transaction log before a transaction is acknowledged as committed and that the log records are always written to disk before the data pages where the changes were actually made are written. Writes to the transaction log are always synchronous; that is, SQL Server must wait for them to complete. Writes to the data pages can be asynchronous because all the effects can be reconstructed from the log if necessary. The transaction management component coordinates logging, recovery, and buffer management. These topics are discussed later in this book; at this point, we'll just look briefly at transactions themselves.

The transaction management component delineates the boundaries of statements that must be grouped together to form an operation. It handles transactions that cross databases within the same SQL Server instance, and it allows nested transaction sequences. (However, nested transactions simply execute in the context of the first-level transaction; no special action occurs when they are committed. And a rollback specified in a lower level of a nested transaction undoes the entire transaction.) For a distributed transaction to another SQL Server instance (or to any other resource manager), the transaction management component coordinates with the Microsoft Distributed Transaction Coordinator (MS DTC) service using operating system remote procedure calls. The transaction management component marks save points within a transaction at which work can be partially rolled back or undone.

The transaction management component also coordinates with the locking code regarding when locks can be released, based on the isolation level in effect. It also coordinates with the versioning code

to determine when old versions are no longer needed and can be removed from the version store. The isolation level in which your transaction runs determines how sensitive your application is to changes made by others and consequently how long your transaction must hold locks or maintain versioned data to protect against those changes.

SQL Server 2005 supports two concurrency models for guaranteeing the ACID properties of transactions: optimistic concurrency and pessimistic concurrency. Pessimistic concurrency guarantees correctness and consistency by locking data so that it cannot be changed; this is the concurrency model that every earlier version of SQL Server has used exclusively. SQL Server 2005 introduces optimistic concurrency, which provides consistent data by keeping older versions of rows with committed values in an area of tempdb called the version store. With optimistic concurrency, readers will not block writers and writers will not block readers, but writers will still block writers. The cost of these non-blocking reads and writes must be considered. To support optimistic concurrency, SQL Server needs to spend more time managing the version store. In addition, administrators will have to pay even closer attention to the tempdb database and the extra maintenance it requires.

Five isolation-level semantics are available in SQL Server 2005. Three of them support only pessimistic concurrency: Read Uncommitted (also called "dirty read"), Repeatable Read, and Serializable. Snapshot Isolation Level supports optimistic concurrency. The default isolation level, Read Committed, can support either optimistic or pessimistic concurrency, depending on a database setting.

The behavior of your transactions depends on the isolation level and the concurrency model you are working with. A complete understanding of isolation levels also requires an understanding of locking because the topics are so closely related. The next section

gives an overview of locking; you'll find more detailed information on isolation, transactions, and concurrency management in [Chapter 8](#).

Locking Operations

Locking is a crucial function of a multi-user database system such as SQL Server, even if you are operating primarily in the snapshot isolation level with optimistic concurrency. SQL Server lets you manage multiple users simultaneously and ensures that the transactions observe the properties of the chosen isolation level. Even though readers will not block writers and writers will not block readers in snapshot isolation, writers do acquire locks and can still block other writers, and if two writers try to change the same data concurrently, a conflict will occur that must be resolved. The locking code acquires and releases various types of locks, such as share locks for reading, exclusive locks for writing, intent locks taken at a higher granularity to signal a potential "plan" to perform some operation, and extent locks for space allocation. It manages compatibility between the lock types, resolves deadlocks, and escalates locks if needed. The locking code controls table, page, and row locks as well as system data locks.

Concurrency, with locks or row versions, is an important aspect of SQL Server. Many developers are keenly interested in it because of its potential effect on application performance. [Chapter 8](#) is devoted to the subject, so I won't go into it further here.

Other Operations

Also included in the storage engine are components for controlling utilities such as bulk load, DBCC commands, and backup and restore operations. There is a component to control sorting operations and one to physically manage the files and backup

devices on disk. These components are discussed in [Chapter 4](#). The log manager makes sure that log records are written in a manner to guarantee transaction durability and recoverability; I'll go into detail about the transaction log in [Chapter 5](#).

Finally, there is the buffer manager, a component that manages the distribution of pages within the buffer pool. I'll discuss the buffer pool in much more detail later in the chapter.

The SQLOS

Whether the components of the SQLOS layer are actually part of the storage engine depends on whom you ask. In addition, trying to figure out exactly which components are in the SQLOS layer can be rather like herding cats. I have seen several technical presentations on the topic at conferences and have exchanged e-mail and even spoken face to face with members of the product team, but the answers vary. The manager who said he was responsible for the SQLOS layer defined the SQLOS as everything he was responsible for, which is a rather circular definition. Earlier versions of SQL Server have a thin layer of interfaces between the storage engine and the actual operating system through which SQL Server makes calls to the OS for memory allocation, scheduler resources, thread and worker management, and synchronization objects. However, the services in SQL Server that needed to access these interfaces can be in any part of the engine. SQL Server requirements for managing memory, schedulers, synchronization objects, and so forth have become more complex. Rather than each part of the engine growing to support the increased functionality, all services in SQL Server that need this OS access have been grouped together into a single functional unit called the SQLOS. In general, the SQLOS is like an operating system inside SQL Server. It provides memory management, scheduling, IO management, a framework

for locking and transaction management, deadlock detection, general utilities for dumping, exception handling, and so on.

Another member of the product team described the SQLOS to me as a set of data structures and APIs that could potentially be needed by operations running at any layer of the engine. For example, consider various operations that require use of memory. SQL Server doesn't just need memory when it reads in data pages through the storage engine; it also needs memory to hold query plans developed in the query processor layer. [Figure 2-1](#) (shown earlier) depicts the SQLOS layer in several parts, but this is just a way of showing that many SQL Server components use SQLOS functionality.

The SQLOS, then, is a collection of structures and processes that handles many of the tasks you might think of as being operating system tasks. Defining them in SQL Server gives the Database Engine greater capacity to optimize these tasks for use by a powerful relational database system.

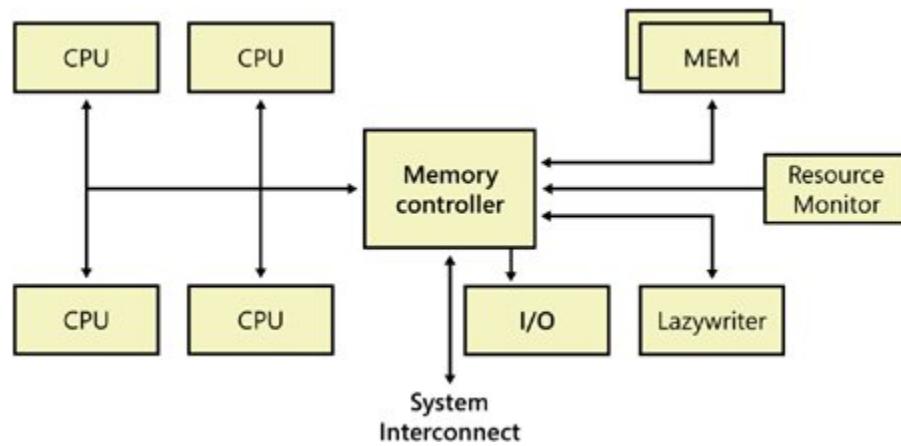
NUMA Architecture

SQL Server 2005 is Non-Uniform Memory Access (NUMA) aware, and both scheduling and memory management can take advantage of NUMA hardware by default. You can use some special configurations when you work with NUMA, so I'll provide some general background here before discussing scheduling and memory.

The main benefit of NUMA is scalability, which has definite limits when you use symmetric multiprocessing (SMP) architecture. With SMP, all memory access is posted to the same shared memory bus. This works fine for a relatively small number of CPUs, but problems appear when you have many CPUs competing for access to the shared memory bus. The trend in hardware has been to have more

than one system bus, each serving a small set of processors. NUMA limits the number of CPUs on any one memory bus. Each group of processors has its own memory and possibly its own I/O channels. However, each CPU can access memory associated with other groups in a coherent way, and I'll discuss this a bit more later in the chapter. Each group is called a *NUMA node* and the nodes are connected to each other by means of a high speed interconnection. The number of CPUs within a NUMA node depends on the hardware vendor. It is faster to access local memory than the memory associated with other NUMA nodes. This is the reason for the name *Non-Uniform Memory Access*. [Figure 2-2](#) shows a NUMA node with four CPUs.

Figure 2-2. A NUMA node with four CPUs



SQL Server 2005 allows you to subdivide one or more physical NUMA nodes into smaller NUMA nodes, referred to as *software NUMA* or *soft-NUMA*. You typically use *soft-NUMA* when you have many CPUs and do not have hardware NUMA because soft-NUMA allows only for the subdividing of CPUs but not memory. You can also use soft-NUMA to subdivide hardware NUMA nodes into groups of fewer CPUs than is provided by the hardware NUMA. Your soft-NUMA nodes can also be configured to listen on its own port.

Only the SQL Server scheduler and SQL Server Network Interface (SNI) are soft-NUMA aware. Memory nodes are created based on hardware NUMA and are therefore not affected by soft-NUMA.

TCP/IP, VIA, Named Pipes, and shared memory can take advantage of NUMA round-robin scheduling, but only TCP and VIA can affinitize to a specific set of NUMA nodes. See Books Online for how to use the SQL Server Configuration Manager to set a TCP/IP address and port to single or multiple nodes.

The Scheduler

Prior to SQL Server 7.0, scheduling was entirely dependent on the underlying Windows operating system. Although this meant that SQL Server could take advantage of the hard work done by the Windows engineers to enhance scalability and efficient processor use, there were definite limits. The Windows scheduler knew nothing about the needs of a relational database system, so it treated SQL Server worker threads the same as any other process running on the operating system. However, a high-performance system such as SQL Server functions best when the scheduler can meet its special needs. SQL Server 7.0 was designed to handle its own scheduling to gain a number of advantages, including the following:

- A private scheduler could support SQL Server tasks using fibers (newly available in Windows NT 4.0) as easily as it supported using threads.
- Context switching and switching into kernel mode could be avoided as much as possible.

The scheduler in SQL Server 7.0 and SQL Server 2000 was called the User Mode Scheduler (UMS) to reflect the fact that it ran primarily in user mode, as opposed to kernel mode. SQL Server 2005 calls its scheduler the SOS Scheduler and improves on UMS even more.

One major difference between the SQL Server scheduler, whether UMS or SOS, and the Windows scheduler is that the SQL Server scheduler runs as a cooperative rather than preemptive scheduler. This means it relies on the workers threads or fibers to voluntarily yield often enough so one process or thread doesn't have exclusive control of the system. The SQL Server product team has to make sure that its code runs efficiently and voluntarily yields the scheduler in appropriate places; the reward for this is much greater control and scalability than is possible with the Windows generic scheduler.

SQL Server Workers

You can think of the SQL Server scheduler as a logical CPU used by SQL Server workers. A worker can be either a thread or a fiber that is bound to a logical scheduler. If the Affinity Mask Configuration option is set, each scheduler is affinitized to a particular CPU. (I'll talk about configuration in the next chapter.) Thus, each worker is also associated with a single CPU. Each scheduler is assigned a worker limit based on the configured Max Worker Threads and the number of schedulers, and each scheduler is responsible for creating or destroying workers as needed. A

worker cannot move from one scheduler to another, but as workers are destroyed and created, it can appear as if workers are moving between schedulers.

Workers are created when the scheduler receives a request (a task to execute) and there are no idle workers. A worker can be destroyed if it has been idle for at least 15 minutes, or if SQL Server is under memory pressure. Each worker can use at least half a megabyte of memory on a 32-bit system and at least 2 gigabytes (GB) on a 64-bit system, so destroying multiple workers and freeing their memory can yield an immediate performance improvement on memory-starved systems. SQL Server actually handles the worker pool very efficiently, and you might be surprised to know that even on very large systems with hundreds or even thousands of users, the actual number of SQL Server workers might be much lower than the configured value for Max Worker Threads. In a moment, I'll tell you about some of the dynamic management objects that let you see how many workers you actually have, as well as scheduler and task information (discussed next).

SQL Server Schedulers

In SQL Server 2005, each actual CPU (whether hyperthreaded or physical) has a scheduler created for it when SQL Server starts up. This is true even if the affinity mask option has been configured so that SQL Server is set to not use all of the available physical CPUs. In SQL Server 2005, each scheduler is set to either ONLINE or OFFLINE based on the affinity mask settings, and the default is that all schedulers are ONLINE. Changing the affinity mask value can set a scheduler's status to OFFLINE, and you can do this without having to restart SQL Server. Note that when a scheduler is switched from ONLINE to OFFLINE due to a configuration change, any work already assigned to the scheduler is first completed and no new work is assigned.

SQL Server Tasks

The unit of work for a SQL Server worker is a *request*, or a *task*, which you can think of as being equivalent to a single batch sent from the client to the server. Once a request is received by SQL Server, it is bound to a worker, and that worker processes the entire request before handling any other request. This holds true even if the request is blocked for some reason, such as while it waits for a lock or for I/O to complete. The particular worker will not handle any new requests but will wait until the blocking condition is resolved and the request can be completed. Keep in mind that a SPID (session ID) is not the same as a task. A SPID is a connection or channel over which requests can be sent, but there is not always an active request on any particular SPID.

In SQL Server 2000, each SPID is assigned to a scheduler when the initial connection is made, and all requests sent over the same SPID are handled by the same scheduler. The assignment of a SPID to a scheduler is based on the number of SPIDs already assigned to the scheduler, with the new SPID assigned to the scheduler with the fewest users. Although this provides a rudimentary form of load balancing, it doesn't take into account SPIDs that are completely quiescent or that are doing enormous amounts of work, such as data loading.

In SQL Server 2005, a SPID is no longer bound to a particular scheduler. Each SPID has a preferred scheduler, which is the scheduler that most recently processed a request from the SPID. The SPID is initially assigned to the scheduler with the lowest load. (You can get some insight into the load on each scheduler by looking at the `load_factor` column in the DMV `dm_osSchedulers`.) However, when subsequent requests are sent from the same SPID, if another scheduler has a load factor that is less than a certain percentage of the average of all the scheduler's load factor, the new task is given to the scheduler with the smallest load factor. There is

a restriction that all tasks for one SPID must be processed by schedulers on the same NUMA node. The exception to this restriction is when a query is being executed as a parallel query across multiple CPUs. The optimizer can decide to use more CPUs that are available on the NUMA node processing the query, so other CPUs (and other schedulers) can be used.

Threads vs. Fibers

The User Mode Scheduler, as mentioned earlier, was designed to work with workers running on either threads or fibers. Windows fibers have less overhead associated with them than threads do, and multiple fibers can run on a single thread. You can configure SQL Server to run in fiber mode by setting the Lightweight Pooling option to 1. Although using less overhead and a "lightweight" mechanism sounds like a good idea, you should carefully evaluate the use of fibers.

Certain components of SQL Server don't work, or don't work well, when SQL Server runs in fiber mode. These components include SQLMail and SQLXML. Other components, such as heterogeneous and CLR queries, are not supported at all in fiber mode because they need certain thread-specific facilities provided by Windows. Although it is possible for SQL Server to switch to thread mode to process requests that need it, the overhead might be greater than the overhead of using threads exclusively. Fiber mode was actually intended just for special niche situations in which SQL Server reaches a limit in scalability due to spending too much time switching between threads contexts or switching between user mode and kernel mode. In most environments, the performance benefit gained by fibers is quite small compared to benefits you can get by tuning in other areas. If you're certain you have a situation that could benefit from fibers, be sure to do thorough testing before you set the option on a production server. In addition, you might

even want put in a call to Microsoft Customer Support Services just to be certain.

NUMA and Schedulers

With a NUMA configuration, every node has some subset of the machine's processors and the same number of schedulers. When a machine is configured for hardware NUMA, each node has the same number of processors, but for soft-NUMA that you configure yourself, you can assign different numbers of processors to each node. There will still be the same number of schedulers as processors. When SPIDs are first created, they are assigned to nodes on a round-robin basis. The scheduler monitor then assigns the SPID to the least loaded scheduler on that node. As mentioned earlier, if the spid is moved to another scheduler, it stays on the same node. A single processor or SMP machine will be treated as a machine with a single NUMA node. Just like on an SMP machine, there is no hard mapping between schedulers and a CPU with NUMA, so any scheduler on an individual node can run on any CPU on that node. However, if you have set the Affinity Mask Configuration option, each scheduler on each node will be fixed to run on a particular CPU.

Every NUMA node has its own lazywriter, which I'll talk about later. Every node also has its own Resource Monitor, which are managed by a hidden scheduler. The Resource Monitor has its own spid, which you can see by querying the *sys.dm_exec_requests* and *sys.dm_os_workers* DMVs, as shown here:

```
SELECT session_id,
       CONVERT (varchar(10), t1.status) AS status,
       CONVERT (varchar(20), t1.command) AS command,
       CONVERT (varchar(15), t2.state) AS
       worker_state
  FROM sys.dm_exec_requests AS t1 JOIN
```

```
sys.dm_os_workers AS t2  
ON t2.task_address = t1.task_address  
WHERE command = 'RESOURCE MONITOR'
```

Every node has its own Scheduler Monitor, which can run on any SPID and runs in a preemptive mode. The Scheduler Monitor checks the health of the other schedulers running on the node, and it is also responsible for sending messages to the schedulers to help them balance their workload. Every node also has its own I/O Completion Port (IOCP), which is the network listener.

Dynamic Affinity

In SQL Server 2005 (in all editions except SQLExpress), processor affinity can be controlled dynamically. When SQL Server starts up, all scheduler tasks are started on server startup, so there is one scheduler per CPU. If the affinity mask has been set, some of the schedulers are then marked as offline and no tasks are assigned to them.

When the affinity mask is changed to include additional CPUs, the new CPU is brought online. The Scheduler Monitor then notices an imbalance in workload and starts picking workers to move to the new CPU. When a CPU is brought offline by changing the affinity mask, the scheduler for that CPU continues to run active workers, but the scheduler itself is moved to one of the other CPUs that is still online. No new workers are given to this scheduler, which is now offline, and when all active workers have finished their tasks, the scheduler stops.

Binding Schedulers to CPUs

Remember that normally schedulers are not bound to CPUs in a strict one-to-one relationship, even though there is the same number of schedulers as CPUs. A scheduler is bound to a CPU only when the affinity mask is set. This is true even if you set the affinity mask to use all of the CPUs which is the default. For example, the default Affinity Mask Configuration value is 0, which means to use all CPUs, with no hard binding of scheduler to CPU. In fact, in some cases when there is a heavy load on the machine, Windows can run two schedulers on one CPU.

For an eight-processor machine, an affinity mask value of 3 (bit string 00000011) means that only CPUs 0 and 1 will be used and two schedulers will be bound to the two CPUs. If you set the affinity mask to 255 (bit string 11111111), all the CPUs will be used, just like with the default. However, with the affinity mask set, the eight CPUs will be bound to the eight schedulers.

In some situations, you might want to limit the number of CPUs available but not bind a particular scheduler to a single CPU for example, if you are using a multiple-CPU machine for server consolidation. Suppose you have a 64-processor machine on which you are running eight SQL Server instances and you want each instance to use eight of the processors. Each instance will have a different affinity mask that specifies a different subset of the 64 processors, so you might have affinity mask values 255 (0xFF), 65280 (0xFF00), 16711680 (0xFF0000), and 4278190080 (0xFF000000). Because the affinity mask is set, each instance will have hard binding of scheduler to CPU. If you want to limit the number of CPUs but still not constrain a particular scheduler to running on a specific CPU, you can start SQL Server with traceflag 8002. This lets you have CPUs mapped to an instance, but within the instance, schedulers are not bound to CPUs.

Observing Scheduler Internals

SQL Server 2005 has several dynamic management objects that provide information about schedulers, work, and tasks. These are primarily intended for use by Microsoft Customer Support Services, but you can use them to gain a greater appreciation for the information SQL Server keeps track of. Note that all these objects require a SQL Server 2005 permission called View Server State. By default, only an administrator has that permission, but it can be granted to others. For each of the objects, I will list some of the more useful or interesting columns and provide the description of the column taken from SQL Server 2005 Books Online. For the full list of columns, most of which are useful only to support personnel, you can refer to Books Online, but even then you'll find that some of the columns are listed as "for internal use only."

sys.dm_osSchedulers

This view returns one row per scheduler in SQL Server. Each scheduler is mapped to an individual processor in SQL Server. You can use this view to monitor the condition of a scheduler or to identify runaway tasks. Interesting columns include the following:

- **parent_node_id** The ID of the node that the scheduler belongs to, also known as the *parent node*. This represents a NUMA node.
- **scheduler_id** The ID of the scheduler. All schedulers that are used to run regular queries have IDs of less than 255. Those with IDs greater than or equal to 255, such as the dedicated administrator connection scheduler, are used internally by SQL Server.
- **cpu_id** The ID of the CPU with which this scheduler is associated. If SQL Server is configured to run with affinity, the

value is the ID of the CPU on which the scheduler is supposed to run. If the affinity mask has not been specified, the `cpu_id` will be 255.

- **is_online** If SQL Server is configured to use only some of the available processors on the server, this can mean that some schedulers are mapped to processors that are not in the affinity mask. If that is the case, this column returns 0. This means the scheduler is not being used to process queries or batches.
- **current_tasks_count** The number of current tasks associated with this scheduler, including the following. (When a task is completed, this count is decremented.)
 - Tasks that are waiting to be executed by a worker
 - Tasks that are currently running or waiting

- Completed tasks
- **runnable_tasks_count** The number of tasks waiting to run on the scheduler.
- **current_workers_count** The number of workers associated with this scheduler, including workers that are not assigned any task.
- **active_workers_count** The number of workers that have been assigned a task.
- **work_queue_count** The number of tasks waiting for a worker. If current_workers_count is greater than active_workers_count, this work queue count should be 0 and the work queue should not grow.
- **pending_disk_io_count** The number of pending I/Os. Each scheduler has a list of pending I/Os that are checked every time there is a context switch to determine whether they have been completed. The count is incremented when the request is inserted. It is decremented when the request is completed. This number does not indicate the state of the I/Os.
- **load_factor** The internal value that indicates the perceived load on this scheduler. This value is used to determine whether a new task should be put on this scheduler or another scheduler. It is useful for debugging purposes when schedulers appear to not be evenly loaded. In SQL Server 2000, a task is routed to a particular scheduler. In SQL Server 2005, the routing decision is based on the load on the scheduler. SQL Server 2005 also uses a load factor of nodes and schedulers to help determine the best location to acquire resources. When a task is added to the queued, the load factor increases. When a

task is completed, the load factor decreases. Using load factors helps the SQLOS balance the work load better.

sys.dm_os_workers

This view returns a row for every worker in the system. Interesting columns include the following:

- **is_preemptive** A value of 1 means that the worker is running with preemptive scheduling. Any worker running external code is run under preemptive scheduling.
- **is_fiber** A value of 1 means that the worker is running with lightweight pooling.

sys.dm_os_threads

This view returns a list of all SQLOS threads that are running under the SQL Server process. Interesting columns include the following:

- **started_by_sqlservr** Indicates the thread initiator. A 1 means that SQL Server started the thread and 0 means that another component, such as an extended procedure from within SQL Server, started the thread.
- **creation_time** The time when this thread was created.
- **stack_bytes_used** The number of bytes that are actively being used on the thread.

- **Affinity** The CPU mask on which this thread is supposed to be running. This depends on the value in the sp_configure "affinity mask."
- **Locale** The cached locale LCID for the thread.

sys.dm_os_tasks

This view returns one row for each task that is active in the instance of SQL Server. Interesting columns include the following:

- **task_state** The state of the task. The value can be one of the following:
 - PENDING: Waiting for a worker thread
 - RUNNABLE: Runnable but waiting to receive a quantum
 - RUNNING: Currently running on the scheduler
 - SUSPENDED: Has a worker but is waiting for an event
 - DONE: Completed
 - SPINLOOP: Processing a spinlock, as when waiting for a signal
- **context_switches_count** The number of scheduler context switches that this task has completed.

- **pending_io_count** The number of physical I/Os performed by this task.
- **pending_io_byte_count** The total byte count of I/Os performed by this task.
- **pending_io_byte_average** The average byte count of I/Os performed by this task.
- **scheduler_id** The ID of the parent scheduler. This is a handle to the scheduler information for this task.
- **session_id** The ID of the session associated with the task.

sys.dm_os_waiting_tasks

This view returns information about the queue of tasks that are waiting on some resource. Interesting columns include the following:

- **session_id** The ID of the session associated with the task.
- **exec_context_id** The ID of the execution context associated with the task.
- **wait_duration_ms** The total wait time for this wait type, in milliseconds. This time is inclusive of signal_wait_time.
- **wait_type** The name of the wait type.
- **resource_address** The address of the resource for which the task is waiting.

- **blocking_task_address** The task that is currently holding this resource.
- **blocking_session_id** The ID of the session of the blocking task.
- **blocking_exec_context_id** The ID of the execution context of the blocking task.
- **resource_description** The description of the resource that is being consumed.

The Dedicated Administrator Connection

Kalen, Use "extreme" or "unusual" rather than "pathological"?

Under pathological conditions such as a complete lack of available resources, it is possible for SQL Server to enter an abnormal state in which no further connections can be made to the SQL Server instance. In SQL Server 2000, this situation means that an administrator cannot get in to kill any troublesome connections or even begin to diagnose the possible cause of the problem. SQL Server 2005 introduces a special connection called the dedicated administrator connection (DAC) that is designed to be accessible even when no other access can be made.

Access via the DAC must be specially requested. You can also connect to the DAC using the command-line tool SQLCMD, by using the /A flag. This method of connection is recommended because it uses fewer resources than the graphical interface method but offers more functionality than other command-line tools, such as osql. Through SQL Server Management Studio, you can

specify that you want to connect using DAC by preceding the name of your SQL Server with **ADMIN:** in the Connection dialog box.

For example, to connect to the default SQL Server instance on my machine, TENAR, I would enter **ADMIN:TENAR**. To connect to a named instance called SQL2005 on the same machine, I would enter **ADMIN:TENAR\SQL2005**.

DAC is a special-purpose connection designed for diagnosing problems in SQL Server and possibly resolving them. It is not meant to be used as a regular user connection. Any attempt to connect using DAC when there is already an active DAC connection will result in an error. The message returned to the client will say only that the connection was rejected; it will not state explicitly that it was because there already was an active DAC. However, a message will be written to the error log indicating the attempt (and failure) to get a second DAC connection. You can check whether a DAC is in use by running the following query. If there is an active DAC, the query will return the SPID for the DAC; otherwise, it will return no rows.

```
SELECT t2.session_id
FROM sys.tcp_endpoints as t1 JOIN
sys.dm_exec_sessions as t2
    ON t1.endpoint_id = t2.endpoint_id
WHERE t1.name='Dedicated Admin Connection'
```

You should keep the following in mind about using the DAC:

- By default, the DAC is available only locally. However, an administrator can configure SQL Server to allow remote connection by using the configuration option called Remote Admin Connections.

- The user logon to connect via the DAC must be a member of SYSADMIN server role.
- There are only a few restrictions on the SQL statements that can be executed on the DAC. (For example, you cannot run BACKUP or RESTORE using the DAC.) However, it is recommended that you do not run any resource-intensive queries that might exacerbate the problem that led you to use the DAC. The DAC connection is created primarily for troubleshooting and diagnostic purposes. In general, you'll use the DAC for running queries against the dynamic management objects, some of which you've seen already and many more of which I'll discuss later in this book.
- A special thread is assigned to the DAC that allows it to execute the diagnostic functions or queries on a separate scheduler. This thread cannot be terminated. You can kill only the DAC session, if needed. The DAC scheduler always uses the scheduler_id value of 255, and this thread has the highest priority. There is no lazywriter thread for the DAC, but the DAC does have its own IOCP, a worker thread, and an idle thread.

You might not always be able to accomplish your intended tasks using the DAC. Suppose you have an idle connection that is holding on to a lock. If the connection has no active task, there is no thread associated with it, only a connection ID. Suppose further than many other processes are trying to get access to the locked resource, and that they are blocked. Those connections still have an incomplete task, so they will not release their worker. If 255 such processes (the default number of worker threads) try to get the same lock, all available workers might get used up and no more connections can be made to SQL Server. Because the DAC has its own scheduler, you can start it, and the expected solution would be to kill the connection that is holding the lock but not do any further processing

to release the lock. But if you try to use the DAC to kill the process holding the lock, the attempt will fail. SQL Server would need to give a worker to the task in order to kill it, and there are no workers available. The only solution is to kill several of the (blameless) blocked processes that still have workers associated with them.

Note



To conserve resources, SQL Server 2005 Express Edition does not support a DAC connection unless started with a trace flag 7806.

The DAC is not guaranteed to always be usable, but because it reserves memory and a private scheduler and is implemented as a separate node, a connection will probably be possible when you cannot connect in any other way.

Memory

Memory management is a huge topic, and to cover every detail would require a whole volume in itself. My goal in this section is twofold: first, to provide enough information about how SQL Server uses its memory resources so you can determine whether memory is being managed well on your system; and second, to describe the aspects of memory management that you have control over so you can understand when to exert that control.

By default, SQL Server 2005 manages its memory resources almost completely dynamically. When allocating memory, SQL Server must communicate constantly with the operating system, which is one of the reasons the SQLOS layer of the engine is so important.

The Buffer Pool and the Data Cache

The main memory component in SQL Server is the buffer pool. All memory not used by another memory component remains in the buffer pool to be used as a data cache for pages read in from the database files on disk. The buffer manager manages disk I/O functions for bringing data and index pages into the data cache so data can be shared among users. When other components require memory, they can request a buffer from the buffer pool. A buffer is a page in memory that's the same size as a data or index page. You can think of it as a page frame that can hold one page from a database. Most of the buffers taken from the buffer pool for other memory components go to other kinds of memory caches, the largest of which is typically the cache for procedure and query plans, which is usually called the *procedure cache*.

Occasionally, SQL Server must request contiguous memory in larger blocks than the 8-KB pages that the buffer pool can provide so memory must be allocated from outside the buffer pool. Use of large memory blocks is typically kept to a minimum, so direct calls to the operating system account for a small fraction of SQL Server memory usage.

Access to In-Memory Data Pages

Access to pages in the data cache must be fast. Even with real memory, it would be ridiculously inefficient to scan the whole data cache for a page when you have gigabytes of data. Pages in the data cache are therefore hashed for fast access. *Hashing* is a technique that uniformly maps a key via a hash function across a set of hash buckets. A *hash table* is a structure in memory that contains an array of pointers (implemented as a linked list) to the buffer pages. If all the pointers to buffer pages do not fit on a single hash page, a *linked list* chains to additional hash pages.

Given a dbid-fileno-pageno identifier (a combination of the database ID, file number, and page number), the hash function converts that key to the hash bucket that should be checked; in essence, the hash bucket serves as an index to the specific page needed. By using hashing, even when large amounts of memory are present, SQL Server can find a specific data page in cache with only a few memory reads. Similarly, it takes only a few memory reads for SQL Server to determine that a desired page is not in cache and that it must be read in from disk.

Note



Finding a data page might require that multiple buffers be accessed via the hash

buckets chain (linked list). The hash function attempts to uniformly distribute the dbid-fileno-pageno values throughout the available hash buckets. The number of hash buckets is set internally by SQL Server and depends on the total size of the buffer pool.

Managing Pages in the Data Cache

You can use a data page or an index page only if it exists in memory. Therefore, a buffer in the data cache must be available for the page to be read into. Keeping a supply of buffers available for immediate use is an important performance optimization. If a buffer isn't readily available, many memory pages might have to be searched simply to locate a buffer to free up for use as a workspace.

In SQL Server 2005, a single mechanism is responsible both for writing changed pages to disk and for marking as free those pages that have not been referenced for some time. SQL Server maintains a linked list of the addresses of free pages, and any worker needing a buffer page uses the first page of this list.

Every buffer in the data cache has a header that contains information about the last two times the page was referenced and some status information, including whether the page is dirty (has been changed since it was read in to disk). The reference information is used to implement the page replacement policy for the data cache pages, which uses an algorithm called LRU-K.^[1] This

algorithm is a great improvement over a strict LRU (Least Recently Used) replacement policy, which has no knowledge of how recently a page was used. It is also an improvement over an LFU (Least Frequently Used) policy involving reference counters because it requires far fewer adjustments by the engine and much less bookkeeping overhead. An LRU-K algorithm keeps track of the last K times a page was referenced and can differentiate between types of pages, such as index and data pages, with different levels of frequency. Its can actually simulate the effect of assigning pages to different buffer pools of specifically tuned sizes. SQL Server 2005 uses a K value of 2, so it keeps track of the two most recent accesses of each buffer page.

[1] The LRU-K algorithm was introduced by O'Neil, O'Neil, and Weikum, in the Proceedings of the ACM SIGMOD Conference, May 1993.

The data cache is periodically scanned from the start to the end. Because the buffer cache is all in memory, these scans are quick and require no I/O. During the scan, a value is associated with each buffer based on its usage history. When the value gets low enough, the dirty page indicator is checked. If the page is dirty, a write is scheduled to write the modifications to disk. Instances of SQL Server use a write-ahead log so the write of the dirty data page is blocked while the log page recording the modification is first written to disk. (I'll discuss logging in much more detail in [Chapter 5](#).) After the modified page has been flushed to disk, or if the page was not dirty to start with, the page is freed. The association between the buffer page and the data page it contains is removed, by removing information about the buffer from the hash table, and the buffer is put on the free list.

Using this algorithm, buffers holding pages that are considered more valuable remain in the active buffer pool while buffers holding pages not referenced often enough eventually return to the free buffer list. The instance of SQL Server determines internally the size

of the free buffer list, based on the size of the buffer cache. The size cannot be configured.

The work of scanning the buffer, writing dirty pages, and populating the free buffer list is primarily performed by the individual workers after they have scheduled an asynchronous read and before the read is completed. The worker gets the address of a section of the buffer pool containing 64 buffers from a central data structure in the SQL Server engine. Once the read has been initiated, the worker checks to see whether the free list is too small. (Note that this process has consumed one or more pages of the list for its own read.) If so, the worker searches for buffers to free, examining all 64 buffers, regardless of how many it actually finds to free in that group of 64. If a write must be performed for a dirty buffer in the scanned section, the write is also scheduled.

Each instance of SQL Server also has a lazywriter thread for each NUMA node that scans through the buffer cache associated with that node. The lazywriter thread sleeps for a specific interval of time, and when it wakes up, it examines the size of the free buffer list. If the list is below a certain threshold, which depends on the total size of the buffer pool, the lazywriter thread scans the buffer pool to repopulate the free list. As buffers are added to the free list, they are also written to disk if they are dirty.

When SQL Server uses memory dynamically, it must constantly be aware of the amount of free memory. The lazywriter for each node queries the system periodically to determine the amount of free physical memory available. The lazywriter expands or shrinks the data cache to keep the operating system's free physical memory at 5 megabytes (MB) plus or minus 200 KB to prevent paging. If the operating system has less than 5 MB free, the lazywriter releases memory to the operating system instead of adding it to the free list. If more than 5 MB of physical memory is free, the lazywriter reclaims memory to the buffer pool by adding it to the free list. The lazywriter reclaims memory to the buffer pool only when it

repopulates the free list; a server at rest does not grow its buffer pool.

SQL Server also releases memory to the operating system if it detects that too much paging is taking place. You can tell when SQL Server increases or decreases its total memory use by using the SQL Server Profiler to monitor the Server Memory Change event (in the Server category). An event is generated whenever memory in SQL Server increases or decreases by 1 MB or 5 percent of the maximum server memory, whichever is greater. You can look at the value of the data element called Event Sub Class to see whether the change was an increase or a decrease. An Event Sub Class value of 1 means a memory increase; a value of 2 means a memory decrease. I'll cover the SQL Server Profiler in more detail in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

Note



Prior to SQL Server 2005, you could mark tables so their pages were never put on the free list and were therefore kept in memory indefinitely. This process is called *pinning* a table. To pin and unpin, you used the *pintable* option of the *sp_tableoption* stored procedure. This command is still available in SQL Server 2005, but it has no effect. Therefore, if you used the *pintable* option in your SQL Server 2000 code, you don't have to immediately remove it. The SQL Server buffer management algorithm is good enough that you should never need pinning. There is no way in SQL Server 2005 to force a table's pages to stay in cache.

Checkpoints

The checkpoint process also scans the buffer cache periodically and writes any dirty data pages for a particular database to disk. The difference between the checkpoint process and the lazywriter (or the worker threads' management of pages) is that the checkpoint process never puts buffers on the free list. The purpose of the checkpoint process is only to ensure that pages written before a certain time are written to disk, so that the number of dirty pages in memory is always kept to a minimum, which in turn ensures that the length of time SQL Server requires for recovery of a database after a failure is kept to a minimum. In some cases, checkpoints may find few dirty pages to write to disk if most of the dirty pages have been written to disk by the workers or the lazywriters in the period between two checkpoints.

When a checkpoint occurs, SQL Server writes a checkpoint record to the transaction log, which lists all the transactions that are active. This allows the recovery process to build a table containing a list of all the potentially dirty pages. Checkpoints occur automatically at regular intervals but can also be requested manually.

Checkpoints are triggered when:

- A database owner explicitly issues a checkpoint command to perform a checkpoint in that database. In SQL Server 2005, you can run multiple checkpoints (in different databases) concurrently by using the CHECKPOINT command.

- The log is getting full (more than 70 percent of capacity) and the database is in SIMPLE recovery mode. (I'll tell you about recovery modes in [Chapter 3](#).) A checkpoint is triggered to truncate the transaction log and free up space. However, if no space can be freed up, perhaps because of a long-running transaction, no checkpoint occurs.
- A long recovery time is estimated. When recovery time is predicted to be longer than the Recovery Interval configuration option, a checkpoint is triggered. SQL Server 2005 uses a simple metric to predict recovery time because it can recover, or redo, in less time than it took the original operations to run. Thus, if checkpoints are taken at least as often as the recovery interval frequency, recovery completes within the interval. A recovery interval setting of 1 means that checkpoints occur at least every minute as long as transactions are being processed in the database. A minimum amount of work must be done for the automatic checkpoint to fire; this is currently 10 MB of log per minute. In this way, SQL Server doesn't waste time taking checkpoints on idle databases. A default recovery interval of 0 means that SQL Server chooses an appropriate value; for the current version, this is one minute.
- An orderly shutdown of SQL Server is requested, without the NOWAIT option. A checkpoint operation is then run in each database on the instance. An orderly shutdown occurs when you explicitly shut down SQL Server, unless you do so by using the SHUTDOWN WITH NOWAIT command. An orderly shutdown also occurs when the SQL Server service is stopped through Service Control Manager or the net stop command from an operating system prompt. You can also use the *sp_configure* Recovery Interval option to influence checkpointing frequency, balancing the time to recover vs. any impact on run-time performance. If you're interested in tracing how often checkpoints actually occur, you can start SQL Server

with trace flag 3502, which writes information to the SQL Server error log every time a checkpoint occurs.

The checkpoint process goes through the buffer pool, scanning the pages in a non-sequential order, and when it finds a dirty page, it looks to see whether any physically contiguous (on the disk) pages are also dirty so that it can do a large block write. But this means that it might, for example, write buffers 14, 200, 260, and 1000 when it sees that buffer 14 is dirty. (Those pages might have contiguous disk locations even though they're far apart in the buffer pool. In this case, the non-contiguous pages in the buffer pool can be written as a single operation called a *gather-write*.) The process continues to scan the buffer pool until it gets to page 1000. In some cases, an already written page could potentially be dirty again, and it might need to be written out to disk a second time.

The larger the buffer pool, the greater the chance that a buffer that has already been written will be dirty again before the checkpoint is done. To avoid this, SQL Server uses a bit associated with each buffer called a *generation number*. At the beginning of a checkpoint, all the bits are toggled to the same value, either all 0's or all 1's. As a checkpoint checks a page, it toggles the generation bit to the opposite value. When the checkpoint comes across a page whose bit has already been toggled, it doesn't write that page. Also, any new pages brought into cache during the checkpoint process get the new generation number so they won't be written during that checkpoint cycle. Any pages already written because they're in proximity to other pages (and are written together in a gather write) aren't written a second time.

Managing Memory in Other Caches

Buffer pool memory that isn't used for the data cache is used for other types of caches, primarily the procedure cache, which actually

holds plans for all types of queries, not just procedure plans. The page replacement policy, and the mechanism by which freeable pages are searched for, is quite a bit different than for the data cache.

SQL Server 2005 introduces a new common caching framework that is leveraged by all caches except the data cache. The framework consists of set of stores and the Resource Monitor. There are three types of stores: cache stores, user stores (which don't actually have anything to do with users), and object stores. The procedure cache is the main example of a cache store, and the metadata cache is the prime example of a user store. Both cache stores and user stores use the same LRU mechanism and the same costing algorithm to determine which pages can stay and which can be freed. Object stores, on the other hand, are just pools of memory blocks and don't require LRU or costing. One example of the use of an object store is the SQL Server Network Interface (SNI), which leverages the object store for pooling network buffers. For the rest of this section, my discussion of stores refers only to cache stores and user stores.

The LRU mechanism used by the stores is a straightforward variation of the clock algorithm, which SQL Server 2000 used for all its buffer management. You can imagine a clock hand sweeping through the store, looking at every entry; as it touches each entry, it decreases the cost. Once the cost of an entry reaches 0, the entry can be removed from the cache. The cost is reset whenever an entry is reused. With SQL Server 2000, the cost was based on a common formula for all caches in the store, taking into account the memory usage, the I/O, and the CPUs required to generate the entry initially. The cost is decremented using a formula that simply divides the current value by 2.

Memory management in the stores takes into account both global and local memory management policies. Global policies consider the total memory on the system and enable the running of the clock

algorithm across all the caches. Local policies involve looking at one store or cache in isolation and making sure it is not using a disproportionate amount of memory.

To satisfy global and local policies, the SQL Server stores implement two hands: external and internal. Each store has two clock hands, and you can observe these by examining the DMV `sys.dm_os_memory_cache_clock_hands`. This view contains one internal and one external clock hand for each cache store or user store. The external clock hands implement the global policy, and the internal clock hands implement the local policy. The Resource Monitor is in charge of moving the external hands whenever it notices memory pressure. There are many types of memory pressure, and it is beyond the scope of this book to go into all the details of detecting and troubleshoot memory problems. However, if you take a look at the DMV `sys.dm_os_memory_cache_clock_hands`, specifically at the `removed_last_round_count` column, you can look for a very large value (compared to other values). If you notice that value increasing dramatically, that is a strong indication of memory pressure. The companion content for this book contains a comprehensive white paper called "Troubleshooting Performance Problems in SQL Server 2005" that includes many details on tracking down and dealing with memory problems.

The internal clock moves whenever an individual cache needs to be trimmed. SQL Server attempts to keep each cache reasonably sized compared to other caches. The internal clock hands move only in response to activity. If a worker running a task that accesses a cache notices a high number of entries in the cache or notices that the size of the cache is greater than a certain percentage of memory, the internal clock hand for that cache starts up to free up memory for that cache.

The Memory Broker

Because memory is needed by so many components in SQL Server, and to make sure each component uses memory efficiently, Microsoft introduced a Memory Broker late in the development cycle for SQL Server 2005. The Memory Broker's job is to analyze the behavior of SQL Server with respect to memory consumption and to improve dynamic memory distribution. The Memory Broker is a centralized mechanism that dynamically distributes memory between the buffer pool, the query executor, the query optimizer, and all the various caches, and it attempts to adapt its distribution algorithm for different types of workloads. You can think of the Memory Broker as a control mechanism with a feedback loop. It monitors memory demand and consumption by component, and it uses the information it gathers to calculate the optimal memory distribution across all components. It can broadcast this information to the component, which then uses the information to adapt its memory usage. You can monitor Memory Broker behavior by querying the Memory Broker ring buffer:

```
SELECT * FROM sys.dm_os_ring_buffers  
WHERE ring_buffer_type =  
'RING_BUFFER_MEMORY_BROKER'
```

The ring buffer for the Memory Broker is updated only when the Memory Broker wants the behavior of a given component to change—that is, to grow, shrink, or remain stable (if it has previously been growing or shrinking).

Sizing Memory

When we talk about SQL Server memory, we're actually talking about more than just the buffer pool. SQL Server memory is actually organized into three sections, and the buffer pool is usually the

largest and most frequently used. The buffer pool is used as a set of 8-KB buffers, so any memory that is needed in chunks larger than 8 KB is managed separately. The DMV called `sys.dm_os_memory_clerks` has a column called `multi_pages_kb` that shows how much space is used by a memory component outside the buffer pool:

```
SELECT type, sum(multi_pages_kb)
FROM sys.dm_os_memory_clerks
WHERE multi_pages_kb != 0
GROUP BY type
```

If your SQL Server instance is configured to use Address Windowing Extensions (AWE) memory, that can be considered a third memory area. AWE is an API that allows a 32-bit application to access physical memory beyond the 32-bit address limit. Although AWE memory is measured as part of the buffer pool, it must be kept track of separately because only data cache pages can use AWE memory. None of the other memory components, such as the plan cache, can use AWE memory.

Note



If AWE is enabled, the only way to get information about SQL Server's actual memory consumption is by using SQL Server specific counters or DMVs inside the server; you won't get this information from OS-level performance counters.

Sizing the Buffer Pool

When SQL Server starts up, it computes the size of the virtual address space (VAS) of the SQL Server process. Each process running on Windows has its own VAS. The set of all virtual addresses available for process use constitutes the size of the VAS. The size of the VAS depends on the architecture (32- or 64-bit) and the operating system. VAS is just the set of all possible addresses; it might be much greater than the physical memory on the machine.

A 32-bit machine can directly address only 4 GB of memory, and by default, Windows itself reserves the top 2 GB of address space for its own use, which leaves only 2 GB as the maximum size of the VAS for any application, such as SQL Server. You can increase this by enabling a /3GB flag in the system's Boot.ini file, which allows applications to have a VAS of up to 3 GB. If your system has more than 3GB of RAM, the only way a 32-bit machine can get to it is by enabling AWE. One benefit in SQL Server 2005 of using AWE, is that memory pages allocated through the AWE mechanism are considered locked pages and can never be swapped out.

On a 64-bit platform, the AWE Enabled configuration option is present, but its setting is ignored. However, the Windows policy Lock Pages in Memory option is available, although it is disabled by default. This policy determines which accounts can make use of a Windows feature to keep data in physical memory, preventing the system from paging the data to virtual memory on disk. It is recommended that you enable this policy on a 64-bit system.

On 32-bit operating systems, you will have to enable Lock Pages in Memory policy when using AWE. It is recommended that you don't enable the Lock Pages in Memory policy if you are not using AWE.

Although SQL Server will ignore this option when AWE is not enabled, other processes on the system may be impacted.

Note



Memory management is much more straightforward on a 64-bit machine, both for SQL Server, which has so much more VAS to work with, and for an administrator, who doesn't have to worry about special operating system flags or even whether to enable AWE. Unless you are working only with very small databases and do not expect to need more than a couple of gigabytes of RAM, you should definitely consider running a 64-bit edition of SQL Server 2005.

[Table 2-1](#) shows the possible memory configurations for various editions of SQL Server 2005.

Table 2-1. SQL Server 2005 Memory Configurations

Configuration	VAS	Max Physical Memory	AWE/Locked Pages Support
Native 32-bit on 32-bit OS	2 GB	64 GB	AWE
with /3GB boot parameter	3 GB	16 GB	AWE
32-bit on x64 OS (WOW)	4 GB	64 GB	AWE
Native 64-bit on x64 OS	8 terabyte	1 terabyte	Locked Pages
Native 64-bit on IA64 OS	7 terabyte	1 terabyte	Locked Pages

In addition to the VAS size, SQL Server also calculates a value called Target Memory, which is the number of 8-KB pages it expects to be able to allocate. If the configuration option Max Server Memory has been set, Target Memory is the lesser of these two values. Target Memory is recomputed periodically, particularly when it gets a memory notification from Windows. A decrease in the number of target pages on a normally loaded server might indicate a response to external physical memory pressure. You can see the

number of target pages by using the Performance Monitor to examine the Target Server Pages counter in the *SQL Server: Memory Manager* object. There is also a DMV called `sys.dm_os_sys_info` that contains one row of general-purpose SQL Server configuration information, including the following columns:

- **physical_memory_in_bytes** The amount of physical memory available.
- **virtual_memory_in_bytes** The amount of virtual memory available to the process in user mode. You can use this value to determine whether SQL Server was started by using a 3-GB switch.
- **bpool_committed** The total number of buffers with pages that have associated memory. This does not include virtual memory.
- **bpool_commit_target** The optimum number of buffers in the buffer pool.
- **bpool_visible** Number of 8-KB buffers in the buffer pool that are directly accessible in the process virtual address space. When not using AWE, when the buffer pool has obtained its memory target (`bpool_committed = bpool_commit_target`), the value of `bpool_visible` equals the value of `bpool_committed`. When using AWE on a 32-bit version of SQL Server, `bpool_visible` represents the size of the AWE mapping window used to access physical memory allocated by the buffer pool. The size of this mapping window is bound by the process address space and, therefore, the visible amount will be smaller than the committed amount, and can be further reduced by internal components consuming memory for purposes other than database pages. If the value of `bpool_visible` is too low, you might receive out of memory errors.

Although the VAS is reserved, the physical memory up to the target amount is committed only when that memory is required for the current workload that the SQL Server instance is handling. The instance continues to acquire physical memory as needed to support the workload, based on the users connecting and the requests being processed. The SQL Server instance can continue to commit physical memory until it reaches its target or the operating system indicates that there is no more free memory. If SQL Server is notified by the operating system that there is a shortage of free memory, it frees up memory if it has more memory than the configured value for Min Server Memory. Note that SQL Server does not commit memory equal to Min Server Memory initially. It commits only what it needs and what the operating system can afford. The value for Min Server Memory comes into play only after the buffer pool size goes above that amount, and then SQL Server does not let memory go below that setting.

As other applications are started on a computer running an instance of SQL Server, they consume memory, and SQL Server might need to adjust its target memory. Normally, this should be the only situation in which target memory is less than commit memory, and it should stay that way only until memory can be released. The instance of SQL Server adjusts its memory consumption, if possible. If another application is stopped and more memory becomes available, the instance of SQL Server increases the value of its target memory, allowing the memory allocation to grow when needed.. SQL Server adjusts its target and releases physical memory only when there is pressure to do so. Thus, a server that is busy for a while can commit large amounts of memory that will not necessarily be released if the system becomes quiescent.

Note



There is no special handling of multiple SQL Server instances on the same

machine; there is no attempt to balance memory across all instances. They all compete for the same physical memory, so to make sure none of the instances becomes starved for physical memory, you should use the Min and Max Server Memory option on all SQL Server instances on a multiple-instance machine.

Observing Memory Internals

SQL Server 2005 includes several dynamic management objects that provide information about memory and the various caches. Like the dynamic management objects containing information about the schedulers, these objects are primarily intended for use by Customer Support Services to see what SQL Server is doing, but you can use them for the same purpose. To select from these objects, you must have the View Server State permission. Once again, I will list some of the more useful or interesting columns for each object; most of these descriptions are taken from SQL Server 2005 Books Online.

sys.dm_os_memory_clerks

This view returns one row per memory clerk that is currently active in the instance of SQL Server. You can think of a clerk as an accounting unit. Each store described earlier is a clerk, but some

clerks are not stores, such as those for the CLR and for full-text search. The following query returns a list of all the types of clerks:

```
SELECT DISTINCT type FROM sys.dm_os_memory_clerks
```

Interesting columns include the following:

- **single_pages_kb** The amount of single-page memory allocated, in kilobytes. This is the amount of memory allocated by using the single-page allocator of a memory node. This single-page allocator steals pages directly from the buffer pool.
- **multi_pages_kb** The amount of multiple-page memory allocated, in kilobytes. This is the amount of memory allocated by using the multiple-page allocator of the memory nodes. This memory is allocated outside the buffer pool and takes advantage of the virtual allocator of the memory nodes.
- **virtual_memory_reserved_kb** The amount of virtual memory reserved by a memory clerk. This is the amount of memory reserved directly by the component that uses this clerk. In most situations, only the buffer pool reserves virtual address space directly by using its memory clerk.
- **virtual_memory_committed_kb** The amount of memory committed by the clerk. The amount of committed memory should always be less than the amount of Reserved Memory.
- **awe_allocated_kb** The amount of memory allocated by the memory clerk by using AWE. In SQL Server, only buffer pool clerks (MEMORYCLERK_SQLBUFFERPOOL) use this mechanism, and only when AWE is enabled.

sys.dm_os_memory_cache_counters

This view returns a snapshot of the health of each cache of type userstore and cachestore. It provides run-time information about the cache entries allocated, their use, and the source of memory for the cache entries. Interesting columns include the following:

- **single_pages_kb** The amount of the single page memory allocated, in kilobytes. This is the amount of memory allocated by using the single-page allocator. This refers to the 8-KB pages that are taken directly from the buffer pool for this cache.
- **multi_pages_kb** The amount of multiple-page memory allocated, in kilobytes. This is the amount of memory allocated by using the multiple-page allocator of the memory node. This memory is allocated outside the buffer pool and takes advantage of the virtual allocator of the memory nodes.
- **multi_pages_in_use_kb** The amount of multiple-page memory being used, in kilobytes.
- **single_pages_in_use_kb** The amount of single-page memory being used, in kilobytes.
- **entries_count** The number of entries in the cache.
- **entries_in_use_count**: The number of entries in use in the cache.

sys.dm_os_memory_cache_hash_tables

This view returns a row for each active cache in the instance of SQL Server. This view can be joined to sys.dm_os_memory_cache_counters on the cache_address column. Interesting columns include the following:

- **buckets_count** The number of buckets in the hash table.
- **buckets_in_use_count** The number of buckets currently being used.
- **buckets_min_length** The minimum number of cache entries in a bucket.
- **buckets_max_length** The maximum number of cache entries in a bucket.
- **buckets_avg_length** The average number of cache entries in each bucket. If this number gets very large, it might indicate that the hashing algorithm is not ideal.
- **buckets_avg_scan_hit_length** The average number of examined entries in a bucket before the searched-for item was found. As above, a big number might indicate a less-than-optimal cache. You might consider running DBCC FREESYSTEMCACHE to remove all unused entries in the cache stores. You can get more details on this command in Books Online.

sys.dm_os_memory_cache_clock_hands

This DMV, discussed earlier, can be joined to the other cache DMVs using the cache_address column. Interesting columns include the

following:

- **clock_hand** The type of clock hand, either external or internal. Remember that there are two clock hands for every store.
- **clock_status** The status of the clock hand: suspended or running. A clock hand runs when a corresponding policy kicks in.
- **rounds_count** The number of rounds the clock hand has made. All the external clock hands should have the same (or close to the same) value in this column.
- **removed_all_rounds_count** The number of entries removed by the clock hand in all rounds.

Another tool for observing memory use is the command DBCC MEMORYSTATUS, which is greatly enhanced in SQL Server 2005. The book's companion content includes a Knowledge Base article that describes the output from the enhanced command.

NUMA and Memory

As mentioned earlier, one major reason for implementing NUMA is to handle large amounts of memory efficiently. As clock speed and the number of processors increase, it becomes increasingly difficult to reduce the memory latency required to use this additional processing power. Large L3 caches can help alleviate part of the problem, but this is only a limited solution. NUMA is the scalable solution of choice. SQL Server 2005 has been designed to take advantage of NUMA-based computers without requiring any application changes. Keep in mind that the NUMA memory nodes are completely dependent on the hardware NUMA configuration. If

you define your own soft-NUMA, as discussed earlier, you will not affect the number of NUMA memory nodes. So, for example, if you have an SMP computer with eight CPUs and you create four soft-NUMA nodes with two CPUs each, you will have only one MEMORY node serving all four NUMA nodes. Soft-NUMA does not provide memory to CPU affinity. However, there is a network I/O thread and a lazywriter thread for each NUMA node, either hard or soft.

The principle reason for using soft-NUMA is to reduce I/O and lazywriter bottlenecks on computers with many CPUs and no hardware NUMA. For instance, on a computer with eight CPUs and no hardware NUMA, you have one I/O thread and one lazywriter thread that could be a bottleneck. Configuring four soft-NUMA nodes provides four I/O threads and four lazywriter threads, which could definitely help performance.

If you have multiple NUMA memory nodes, SQL Server divides the total target memory evenly among all the nodes. So if you have 10 GB of physical memory and four NUMA nodes and SQL Server determines a 10-GB target memory value, all nodes will eventually allocate and use 2.5 GB of memory as if it were their own. In fact, if one of the nodes has less memory than another, it must use memory from other one to reach its 2.5 GB. This memory is called *foreign memory*. Foreign memory is considered local, so if SQL Server has readjusted its target memory and each node needs to release some, no attempt will be made to free up foreign pages first. In addition, if SQL Server has been configured to run on a subset of the available NUMA nodes, the target memory will *not* automatically be limited to the memory on those nodes. You must set the Max Server Memory value to limit the amount of memory.

In general, the NUMA nodes function largely independently of each other, but that is not always the case. For example, if a worker running on a node N1 needs to access a database page that is already in node N2's memory, it does so by accessing N2's memory,

which is called non-local memory. Note that non-local is not the same as foreign memory.

Read-Ahead

SQL Server supports a mechanism called read-ahead whereby the need for data and index pages can be anticipated and pages can be brought into the buffer pool before they're actually needed. This performance optimization allows large amounts of data to be processed effectively. Read-ahead is managed completely internally, and no configuration adjustments are necessary.

There are two kinds of read-ahead: one for table scans on heaps and one for index ranges. For table scans, the table's allocation structures are consulted to read the table in disk order. Up to 32 extents ($32 * 8 \text{ pages/extent} * 8192 \text{ bytes/page} = 2 \text{ MB}$) of read-ahead may be outstanding at a time. Four extents (32 pages) at a time are read with a single 256-KB scatter read. If the table is spread across multiple files in a file group, SQL Server will attempt to distribute the read-ahead activity across the files evenly.

For index ranges, the scan uses level one of the index structure (the level immediately above the leaf) to determine which pages to read ahead. When the index scan starts, read-ahead is invoked on the initial descent of the index to minimize the number of reads performed. For instance, for a scan of *WHERE state = 'WA'*, read-ahead searches the index for *key = 'WA'*, and it can tell from the level-one nodes how many pages must be examined to satisfy the scan. If the anticipated number of pages is small, all the pages are requested by the initial read-ahead; if the pages are non-contiguous, they're fetched in scatter reads. If the range contains a large number of pages, the initial read-ahead is performed and thereafter every time another 16 pages are consumed by the scan, the index is consulted to read in another 16 pages. This has several interesting effects:

- Small ranges can be processed in a single read at the data page level whenever the index is contiguous.
- The scan range (for example, `state = 'WA'`) can be used to prevent reading ahead of pages that won't be used because this information is available in the index.
- Read-ahead is not slowed by having to follow page linkages at the data page level. (Read-ahead can be done on both clustered indexes and nonclustered indexes.)

As you can see, memory management in SQL Server is a huge topic, and I've provided you with only a basic understanding of how SQL Server uses memory. This information should give you a start in interpreting the wealth of information valuable through the DMVs and troubleshooting. The companion content includes a white paper that offers many more troubleshooting ideas and scenarios.

Final Words

In this chapter, we've looked at the general workings of the SQL Server engine, including the key modules and functional areas that make up the engine. We've also looked at the interaction between SQL Server and the operating system. By necessity, I've made some simplifications throughout the chapter, but the information should provide some insight into the roles and responsibilities of the major components in SQL Server and the interrelationships among components.

Chapter 3. SQL Server 2005 Configuration

In this chapter:

[Using SQL Server Configuration Manager](#) [65](#)

[System Configuration](#) [67](#)

[Final Words](#) [85](#)

In this chapter, we'll take a look at the options for controlling how Microsoft SQL Server 2005 behaves. One main method of controlling the behavior of the SQL Server engine is to adjust configuration option settings, but you can configure that behavior in a few other ways as well. We'll look at using SQL Server Configuration Manager to control network protocols, and we'll examine the configuration options for controlling SQL Server server-wide settings.

Using SQL Server Configuration Manager

Configuration Manager is a tool for managing the services associated with SQL Server, configuring the network protocols used by SQL Server, and managing the network connectivity configuration from SQL Server client computers. It is installed as part of SQL Server. Configuration Manager is available from the Start menu by right-clicking the registered server in SQL Server Management Studio, or you can add it to any other Microsoft Management Console display. Configuration Manager combines the functionality of three SQL Server 2000 tools: Server Network Utility, Client Network Utility, and Service Manager.

Configuring Network Protocols

A specific protocol must be enabled on both the client and server for the client to connect and communicate with the server. SQL Server can listen for requests on all enabled protocols at once. The underlying operating system network protocols (such as Transmission Control Protocol/Internet Protocol [TCP/IP]) should already be installed on the client and the server. Network protocols are typically installed during Microsoft Windows setup; they are not part of SQL Server setup. A SQL Server Net-Library will not work unless its corresponding network protocol is installed on both the client and the server.

On the client computer, the SQL Native Client must be installed and configured to use a network protocol enabled on the server; this is usually done during SQL Server Tools setup. The SQL Native Client is new in SQL Server 2005; it is a stand-alone data access API used for both Object Linking and Embedding Database (OLE-DB) and Open Database Connectivity (ODBC). If the SQL Native Client

is available, any network protocol can be configured for use with a particular SQL Server client. You can use SQL Server Configuration Manager to enable a single protocol or to enable multiple protocols and specify an order in which they should be attempted. If the Shared Memory Protocol setting is enabled, that protocol is always tried first, but, as mentioned in [Chapter 2](#), it is available for communication only when the client and the server are on the same machine.

The following query returns the protocol used for the current connection, using the DMV `sys.dm_exec_connections`:

```
SELECT net_transport  
FROM sys.dm_exec_connections  
WHERE session_id = @@SPID;
```

Default Network Configuration

The network protocols that can be used to communicate with SQL Server 2005 from another computer are not all enabled for SQL Server during installation. To connect from a particular client computer, you might need to enable the desired protocol. The shared memory protocol is enabled by default on all installations, but it can be used to connect to the SQL Server Database Engine only from a client application on the same computer.

TCP/IP connectivity to SQL Server 2005 is disabled for new installations of the Developer, Evaluation, and SQL Express editions. OLE-DB applications connecting with MDAC 2.8 cannot connect to the default instance on a local server using ".", "(local)", or (<blank>) as the server name. To resolve this, supply the server name or enable TCP/IP on the server. Connections to local named

instances are not affected, nor are connections using the SQL Native Client. Installations in which a previous installation of SQL Server is present might not be affected.

[Table 3-1](#) describes the default network configuration settings.

Table 3-1. SQL Server 2005 Default Network Configuration Settings

SQL Server Edition	Type of Installation	Shared Memory	TCP/IP	Named Pipes	VIA
Enterprise	New	Enabled	Enabled	Disabled (available only locally)	Disabled
Developer	New	Enabled	Disabled	Disabled (available only locally)	Disabled
Standard	New	Enabled	Enabled	Disabled (available only locally)	Disabled
Workgroup	New	Enabled	Enabled	Disabled (available only locally)	Disabled

SQL Server Edition	Type of Installation	Shared Memory	TCP/IP	Named Pipes	VIA
Evaluation	New	Enabled	Disabled	Disabled (available only locally)	Disabled
SQL Server Express	New	Enabled	Disabled	Disabled (available only locally)	Disabled
All Editions	Upgrade or side-by-side installation	Enabled Settings preserved from the previous installation	Settings preserved from the previous installation	Settings preserved from the previous installation	Disabled

Managing Services

You can use Configuration Manager to start, pause, resume, or stop SQL Server-related services. The services available will depend on the specific components of SQL Server you have installed, but you should always include the SQL Server service itself and the SQL Server Agent service. Other services might include the SQL Server Full-Text Search service and SQL Server Integration Services (SSIS). You can also use Configuration Manager to view the current properties of the services, such as whether the service is set to start

automatically. Configuration Manager is the preferred tool for changing service properties, rather than using Windows service management tools. When you use a SQL Server tool such as Configuration Manager to change the account used by either the SQL Server or SQL Server Agent service, the SQL Server tool automatically makes additional configurations such as setting permissions in the Windows Registry so that the new account can read the SQL Server settings. Passwords changed using Configuration Manager take effect immediately without your needing to restart the service.

System Configuration

You can configure the SQL Server engine in several ways and through a variety of interfaces. One approach involves using a system-stored procedure called *sp_configure* and supplying a value for a particular configuration option. Some configuration options can be set at the operating system level, and most of these can be controlled using the Surface Area Configuration Tool (discussed in [Chapter 1](#)). In addition, some server behaviors can be controlled using operating system control interfaces; others, such as trace flags, can be set as startup parameters of the SQL Server engine executable, Sqlservr.exe.

Next we'll look at most of the configuration options for controlling the behavior of SQL Server 2005.

Task Management

As you saw in [Chapter 2](#), the operating system schedules all threads in the system for execution. Each thread of every process has a priority, and Windows executes the next available thread with the highest priority. By default, the operating system gives active applications a higher priority, but this priority setting is not appropriate for a server application running in the background, such as SQL Server 2005. To remedy this situation, the SQL Server installation program modifies the priority setting to eliminate favoring of foreground applications.

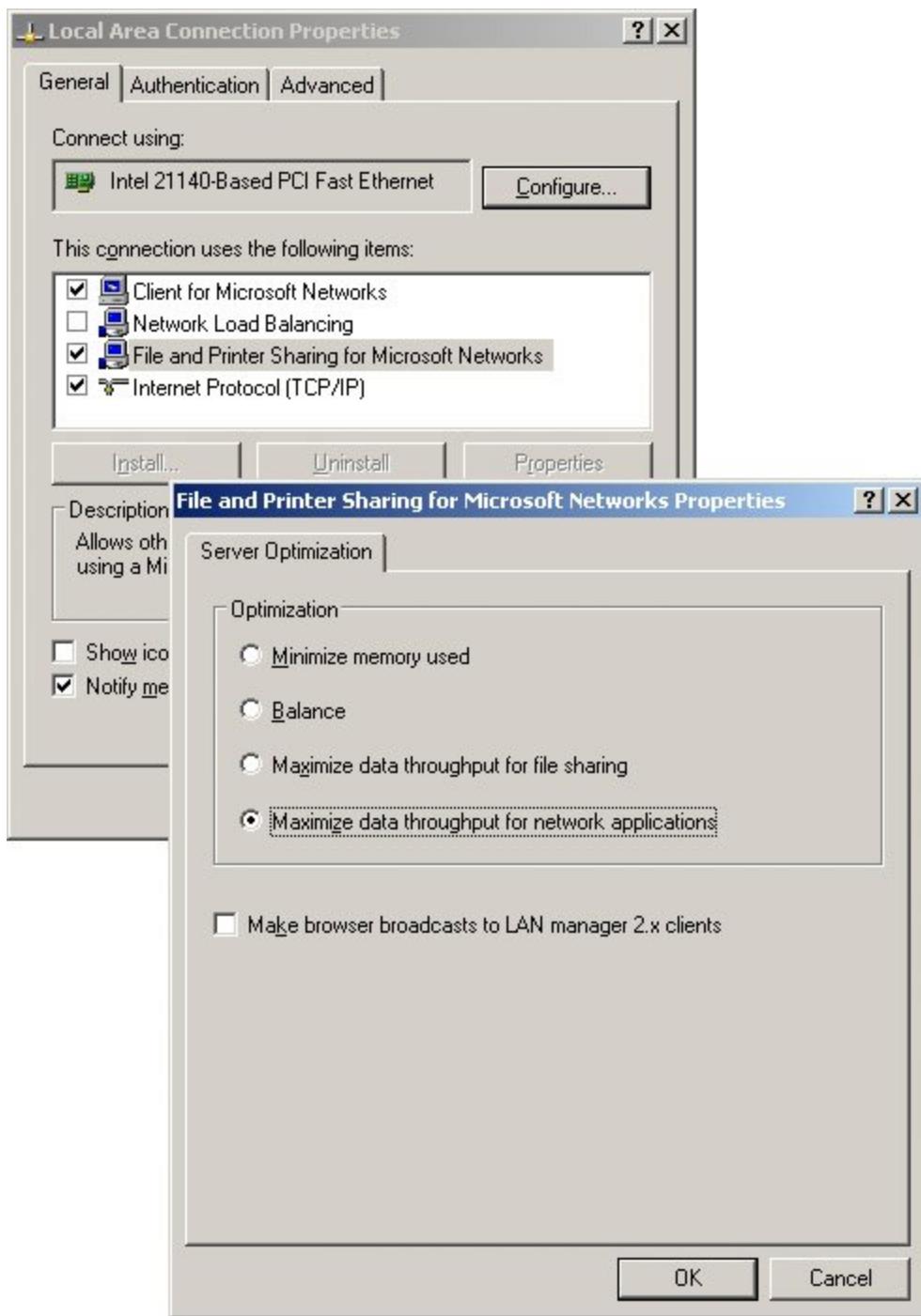
It's not a bad idea to periodically double-check this priority setting in case someone has set it back. From Control Panel in Windows, double-click the System icon to open the System Properties dialog box. On the Advanced tab, click the Performance Settings button.

The first set of option buttons are for specifying how to allocate processor resources, and you can adjust for the best performance of either programs or background services. Select Background Services so that all programs (both background and foreground) receive equal processor resources. If you plan to connect to SQL Server 2005 from a local client (one that is running on the same computer as the server), you can improve processing time by using this setting.

Resource Allocation

A computer running Windows Server for file and print services will need the most memory for file caching. If it is also running SQL Server, it makes sense to have more memory available for SQL Server. In Windows 2000 and Windows 2003, you can change the properties for file and print services by right-clicking My Network Places and choosing Properties. Then right-click Local Area Connection, and again choose Properties. Select File And Printer Sharing For Microsoft Networks, and then click the Properties button. The dialog box shown in [Figure 3-1](#), which is almost identical to the one you get in Windows 2000, appears.

Figure 3-1. Server optimization for network applications



The Maximize Data Throughput For Network Applications option is best for running SQL Server. You need high network throughput with minimal memory devoted to file caching so that more memory is available to SQL Server. When this option is set, network applications such as SQL Server have priority over the file cache for access to memory. Although you might expect that the Minimize Memory Used option would help, it minimizes the memory available for some internal network settings that are needed to support a lot of SQL Server users.

System Paging File Location

If possible, you should place the operating system paging file on a different drive than the files used by SQL Server. This is vital if your system will be paging. However, an even better approach is to add memory or change the SQL Server memory configuration to effectively eliminate paging. In general, SQL Server is designed to minimize paging, so if your memory configuration values are appropriate for the amount of physical memory on the system, so little page-file activity will occur that the file's location will be irrelevant.

Nonessential Services

You should disable any services you don't need. In Windows 2000 and 2003, you can right-click My Computer and choose Manage. Expand the Applications And Services node in the Computer Management tool, and click Services. In the right-hand pane, you will see a list of all the services available on the operating system. You can change a service's startup property by right-clicking its name and choosing Properties. Unnecessary services add overhead to the system and use resources that could otherwise go

to SQL Server. No unnecessary services should be marked for automatic startup. Avoid using a server that's running SQL Server as a domain controller, the group's file or print server, the Web server, or the Dynamic Host Configuration Protocol (DHCP) server. You should also consider disabling the Alerter, ClipBook, Computer Browser, Messenger, Network Dynamic Data Exchange (DDE), and Task Scheduler services, which are enabled by default but are not needed by SQL Server.

Network Protocols

You should run only the network protocols you actually need for connectivity. (I discussed the various protocols in [Chapter 2](#).) You can use the SQL Server Configuration Manager to disable unneeded services, as described earlier in this chapter.

Compatibility with Earlier Versions of SQL Server

A default instance of SQL Server 2005 can listen on the same network addresses as earlier versions of SQL Server, including SQL Server 2000 and SQL Server 7.0. Applications using earlier versions of the client tools can also connect to a default instance with no change. However, named instances of SQL Server 2005 listen on dynamic ports, and client computers using earlier versions of the client tools or components must be set up to connect to these addresses. You can do this by creating a server alias for use by the client. More information about setting up aliases is available in SQL Server 2005 Books Online.

Trace Flags

Books Online lists fewer than a dozen trace flags that are fully supported. You can think of trace flags as special switches that you can turn on or off to change the behavior of SQL Server. There are actually many dozens, if not hundreds, of trace flags. However, most were created for the SQL Server development team's internal testing of the product and were never intended for use by anybody outside Microsoft.

You can set trace flags on or off by using the *DBCC TRACEON* or *DBCC TRACEOFF* command or by specifying them on the command line when you start SQL Server using *Sqlservr.exe*. You can also use the SQL Server Configuration Manager to enable one or more trace flags every time the SQL Server service is started. (You can read about how to do that in Books Online.) Trace flags enabled with *DBCC TRACEON* are valid only for a single connection unless you specified an additional parameter of *1* when calling *DBCC TRACEON*, in which case they will be active for all connections, even ones opened before you ran *DBCC TRACEON*. Trace flags enabled as part of starting the SQL Server service are enabled for all sessions.

A few of the trace flags are particularly relevant to topics covered in this book, and I will discuss particular ones when I describe topics that they are related to. Because trace flags change the way SQL Server behaves, they can actually cause trouble if used inappropriately. Trace flags are not harmless features that you can experiment with just to see what happens, especially not on a production system. Using them effectively requires a thorough understanding of SQL Server default behavior (so that you know exactly what you'll be changing) and extensive testing to determine that your system really will benefit from the use of the trace flag.

SQL Server Configuration Settings

If you choose to have SQL Server automatically configure your system, it will dynamically adjust the most important configuration options for you. It's best to accept the default configuration values unless you have a good reason to change them. A poorly configured system can destroy performance. For example, a system with an incorrectly configured memory setting can break an application.

In certain cases, tweaking the settings rather than letting SQL Server dynamically adjust them might lead to a tiny performance improvement, but your time is probably better spent on application and database design, indexing, query tuning, and other such activities, which I'll talk about later in this book. You might see only a 5 percent improvement in performance by moving from a reasonable configuration to an ideal configuration, but a badly configured system can kill your application's performance.

SQL Server 2005 has only 14 configuration options that are not considered "advanced," and none of these directly affects performance. To see all the configuration options, you must change the value of the Show Advanced Options setting:

```
EXEC sp_configure 'show advanced options', 1  
GO
```

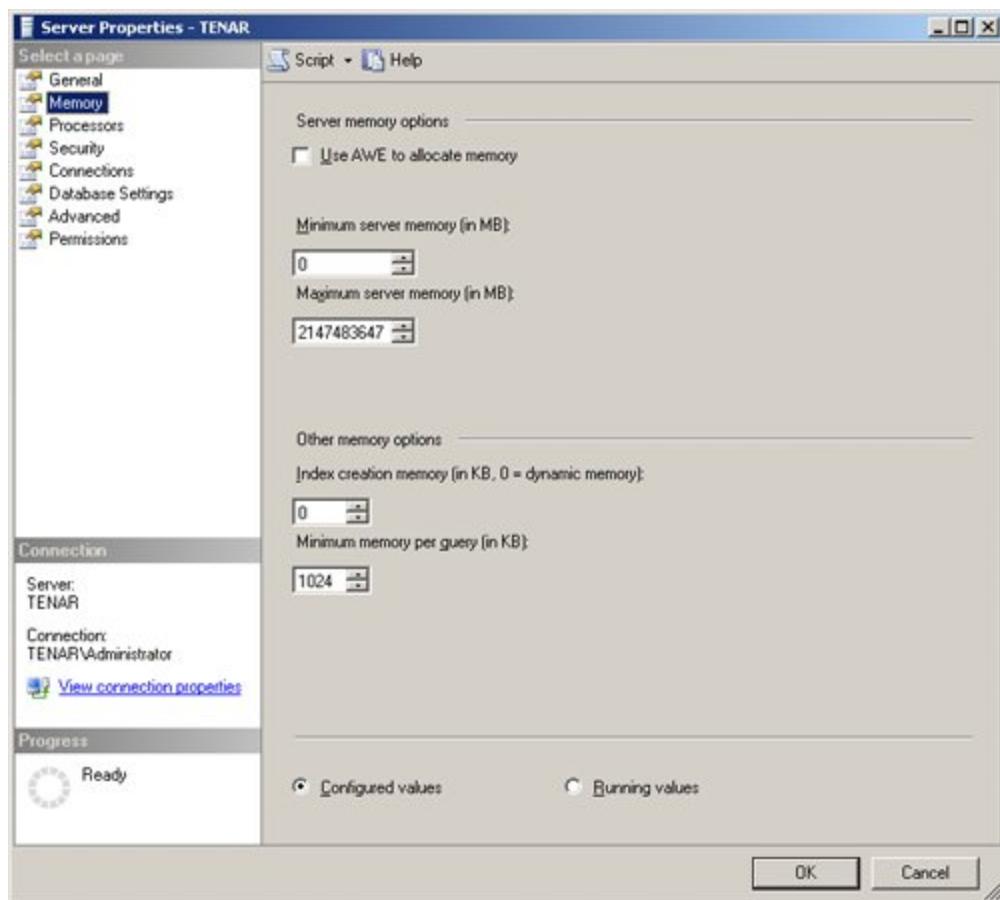
```
RECONFIGURE  
GO
```

You should change configuration options only when you have a clear reason for doing so, and you should closely monitor the effects of each change to determine whether the change improved or degraded performance. Always make and monitor changes one at a time.

The server-wide options discussed here are set via the *sp_configure* system stored procedure. Many of them can also be set from the Server Properties dialog box in the Object Explorer window of SQL Server Management Studio, but there is no single dialog box from which all configuration settings can be seen or changed. Most of the options that you can change from the Server Properties dialog box are controlled from one of the property pages that you reach by right-clicking your server. You can see the list of property pages in [Figure 3-2](#).

Figure 3-2. List of server property pages in SQL Server Management Studio

[[View full size image](#)]



If you use the `sp_configure` stored procedure, no changes take effect until the `RECONFIGURE` command runs. In some cases, you might have to specify `RECONFIGURE WITH OVERRIDE` if you are changing an option to a value outside its recommended range. Dynamic changes take effect immediately upon reconfiguration, but others do not take effect until the server is restarted. If an option's `run_value` and `config_value` as displayed by `sp_configure` are

different, you must restart the server for the *config_value* to take effect.

We won't look at every configuration option hereonly the most interesting ones or ones that are related to SQL Server performance. In most cases, we'll discuss options that you should *not* change. Some of these are resource settings that relate to performance only in that they consume memory (for example, Locks). But if they are configured too high, they can rob a system of memory and degrade performance. I'll group the *sp_configure* settings by functionality. Keep in mind that SQL Server sets almost all of these options automatically, and your applications will work well without you ever looking at these options.

Memory Options

Prior to SQL Server 7.0, you had to manually configure all sorts of memory options, including how much total memory SQL Server should use and how much of that memory should be used for specific purposes. Fortunately, this is no longer the case. In the preceding section, you saw how SQL Server uses memory, including how it allocates memory for different uses and when it reads data from or writes data to disk. However, we did not discuss how to control how much memory SQL Server actually uses for these purposes.

Min Server Memory and Max Server Memory

By default, SQL Server adjusts the total amount of the memory resources it will use. However, you can use the Min Server Memory and Max Server Memory configuration options to take manual control. The default setting for Min Server Memory is 0 megabytes (MB), and the default setting for Max Server Memory is

2147483647. If you use the *sp_configure* stored procedure to change both of these options to the same value, you basically take full control and tell SQL Server to use a fixed memory size. The absolute maximum of 2147483647 MB previously mentioned is actually the largest value that can be stored in the integer field of the underlying system table. It is not related to the actual resources of your system. Both of these options were discussed in [Chapter 2](#) in the section on memory, so I won't repeat that discussion here.

Set Working Set Size

The configuration option Set Working Set Size is a legacy setting from earlier versions, including SQL Server 2000. This setting is ignored in SQL Server 2005, even though you will not receive an error message when you try to use this value.

AWE Enabled

This option enables the use of the Address Windowing Extensions (AWE) API to support large memory sizes. With AWE enabled, SQL Server 2005 can use as much memory as the operating system version of Enterprise, Developer, or Standard edition allows. (The exception is when you are running 64-bit SQL Server 2005, in which case maximum memory is limited to 32 terabytes on Developer or Standard edition.) Instances of SQL Server 2005 running on Windows 2000 do not dynamically manage the size of the address space when you enable AWE memory. Instead, when it starts up, SQL Server commits all of the memory it can up to the amount available on the machine. When running on Windows 2003, SQL Server reserves only a small portion of AWE-mapped memory. As additional AWE-mapped memory is required, the operating system dynamically allocates it to SQL Server. Similarly, if fewer resources

are required, SQL Server can return AWE-mapped memory to the operating system for use by other processes or applications.

Use of AWE, in either Windows 2000 or Windows 2003, will lock the pages in memory so that they cannot be written to the paging file. Windows has to swap out other applications if additional physical memory is needed, so the performance of those applications might suffer. You should therefore set a value for Max Server Memory when you have also enabled AWE.

If you're running multiple instances of SQL Server 2000 on the same computer and each instance uses AWE memory, you must pay close attention to certain configuration issues. SQL Server Books Online documents offer the following guidelines:

- Make sure each instance has a Max Server Memory setting.
- The sum of the Max Server Memory values for all of the instances should be less than the amount of physical memory in the computer.

If the sum of the settings exceeds the physical memory on the computer, some of the instances either will not start or will have less memory than is specified in Max Server Memory. For example, suppose that a computer has 48 gigabytes (GB) of physical random access memory (RAM) and three instances of SQL Server 2000 running on it and that Max Server Memory is set to 16 GB for each instance. If you stop and restart all three instances:

- The first instance will start with the full 16 GB of memory.
- The second instance will start, but with slightly less than 16 GB of memory (up to 128 MB less).

- The third instance will start in dynamic memory mode and will have 128 MB or less memory available to it.

User Connections

SQL Server 2005 dynamically adjusts the number of simultaneous connections to the server if the User Connections configuration setting is left at its default of 0. Even if you set this value to a different number, SQL Server does not actually allocate the full amount of memory needed for each user connection until a user actually connects. When SQL Server starts up, it allocates an array of pointers with as many entries as the configured value for User Connections. If you must use this option, do not set the value too high because each connection takes approximately 28 KB of overhead regardless of whether the connection is being used. However, you also don't want to set it too low, because if you exceed the maximum number of user connections, you receive an error message and cannot connect until another connection becomes available. (The exception is the DAC connection, which can be used.) Keep in mind that the User Connection value is not the same as the number of users; one user, through one application, can open multiple connections to SQL Server. Ideally, you should let SQL Server dynamically adjust the value of the User Connections option.

Locks

The Locks configuration option sets the number of available locks (of all types). The default is 0, which means that when SQL Server starts, the lock manager acquires sufficient memory for an initial pool of 2500 lock structures per node. When the lock pool is exhausted, additional memory is acquired from the buffer pool. If

more memory is required for the lock pool than is available in the buffer pool, and more computer memory is available (the target memory has not been reached), SQL Server generally allocates memory dynamically to satisfy the request for locks. The dynamic lock pool cannot acquire more than 60 percent of the memory allocated to the buffer pool. Once the 60 percent limit has been reached, further requests for locks will generate an error.

If you set the Locks option to something other than 0, you override the ability of SQL Server to allocate lock resources dynamically. In that case, the lock manager cannot allocate more locks than the value specified in Locks. Because each lock consumes memory (96 bytes per lock), increasing this value can require increasing the amount of memory dedicated to the server.

The Locks option also affects when SQL Server will consider lock escalation. When Locks is set to 0, SQL Server considers lock escalation when the memory used by the current lock structures reaches 40 percent of the buffer pool. When Locks is not set to 0, SQL Server considers lock escalation when the number of locks reaches 40 percent of the value specified for Locks. Locking and lock escalation are discussed in detail in [Chapter 8](#).

Scheduling Options

As described earlier, SQL Server 2005 has a special algorithm for scheduling user processes using the SQLOS, which manages one scheduler per processor and makes sure that only one process can run on a scheduler at any given time. The SQLOS manages assignment of user connections to workers to keep the number of users per central processing unit (CPU) as balanced as possible. Five configuration options affect the behavior of the scheduler: Lightweight Pooling, Affinity Mask, Affinity64 Mask, Priority Boost, and Max Worker Threads.

Affinity Mask and Affinity64 Mask

From an operating system point of view, the ability of Windows to move process threads among different processors is efficient, but this activity can reduce SQL Server performance because each processor cache is reloaded with data repeatedly. By setting an affinity mask option, you can allow SQL Server to assign processors to specific threads and thus improve performance under heavy load conditions by eliminating processor reloads and reducing thread migration and context switching across processors. Setting an affinity mask to a *non-0* value not only controls the binding of schedulers to processors, but it also allows you to limit which processors will be used for executing SQL Server requests.

The value of an affinity mask is a 4-byte integer, and each integer controls one processor. If you set a bit representing a processor to 1, that processor is mapped to a specific scheduler. The 4-byte affinity mask can support up to 32 processors. For example, to configure SQL Server to use processors 0 through 5 on an eight-way box, you would set the affinity mask to 63, which is equivalent to a bit string of 00111111. To enable processors 8 through 11 on a 16-way box, you would set the affinity mask to 3840, or 0000111100000000. You might want to do this on a machine supporting multiple instances, for example. You would set the affinity mask of each instance to use a different set of processors on the computer.

To cover more than 32 CPUs, you configure a 4-byte affinity mask for the first 32 CPUs and up to a 4-byte Affinity64 mask for the remaining CPUs. Note that affinity support for servers with 33 to 64 processors is available only on 64-bit operating systems.

You can configure the affinity mask to use all the available CPUs. For an eight-way machine, an Affinity Mask setting of 255 means that all CPUs will be enabled. This is not exactly the same as a

setting of 0 because with the *non-zero* value, the schedulers are bound to a specific CPU, and with the 0 value, they are not.

Lightweight Pooling

By default, SQL Server operates in thread mode, which means that the workers processing SQL Server requests are threads. As I described earlier, SQL Server also lets user connections run in fiber mode. Fibers are less expensive to manage than threads. The Lightweight Pooling option can have a value of 0 or 1; 1 means that SQL Server should run in fiber mode. Using fibers may yield a minor performance advantage, particularly when you have 8 or more CPUs and all of the available CPUs are operating at or near 100 percent. However, the tradeoff is that certain operations, such as running queries on linked servers or executing extended stored procedures, must run in thread mode and therefore need to switch from fiber to thread. The cost of switching from fiber to thread mode for those connections can be noticeable and in some cases will offset any benefit of operating in fiber mode.

If you're running in an environment with a large number of CPUs, all operating at 100 percent capacity, and if System Monitor shows a lot of context switching, setting Lightweight Pooling to 1 might yield some performance benefit.

Priority Boost

If the Priority Boost setting is enabled, SQL Server runs at a higher scheduling priority. The result is that the priority of every thread in the server process is set to a priority of 13 in Windows 2000 and Windows 2003. Most processes run at the normal priority, which is 7. The net effect is that if the server is under load and is getting

close to maxing out the CPU, these normal priority processes will be effectively starved.

The default Priority Boost setting is 0, which means that SQL Server runs at normal priority whether or not you're running it on a single-processor machine. There are probably very few sites or applications for which setting this option makes much difference, but if your machine is totally dedicated to running SQL Server, you might want to enable this option (set it to 1) to see for yourself. It can potentially offer a performance advantage on a heavily loaded, dedicated system. As with most of the configuration options, you should use it with care. Raising the priority too high might affect the core operating system and network operations, resulting in problems shutting down SQL Server or running other operating system tasks on the server.

Max Worker Threads

SQL Server uses the operating system's thread services by keeping a pool of workers (threads or fibers) that take requests off the queue. It attempts to evenly divide the worker threads among the SQLoS schedulers so that the number of threads available to each scheduler is the Max Worker Threads setting divided by the number of CPUs. With 100 or fewer users, there are usually as many worker threads as active users (not just connected users who are idle). With more users, it often makes sense to have fewer worker threads than active users. Although some user requests have to wait for a worker thread to become available, total throughput increases because less context switching occurs.

The Max Worker Threads default value of 0 means that the number of workers will be configured by SQL Server, based on the number of processors and machine architecture. For example, for a SQL Server running on a four-way 32-bit machine, the default is 256 workers. This does not mean that 256 workers are created on

startup. It means that if a connection is waiting to be serviced and no worker is available, a new worker is created if the total is currently below 256. If this setting is configured to 256 and the highest number of simultaneously executing commands is, say, 125, the actual number of workers will not exceed 125. It might be even less than that because SQL Server destroys and trims away workers that are no longer being used. You should probably leave this setting alone if your system is handling 100 or fewer simultaneous connections. In that case, the worker thread pool will not be greater than 100.

[Table 3-2](#) lists the default number of workers given your machine architecture and number of processors. (Note that Microsoft recommends 1024 as the maximum for 32-bit operating systems.)

Table 3-2. Default Max Worker Threads Settings

CPU	32-Bit Computer	64-Bit Computer
Up to 4 processors	256	512
8 processors	288	576
16 processors	352	704

CPU	32-Bit Computer	64-Bit Computer
32 processors	480	960

Even systems that handle 4000 or more connected users run fine with the default setting. When thousands of users are simultaneously connected, the actual worker pool is usually well below the Max Worker Threads value set by SQL Server because from the perspective of the database, most connections are idle even if the user is doing plenty of work on the client.

Disk I/O Options

No options are available for controlling SQL Server's disk read behavior. All the tuning options to control read-ahead in previous versions of SQL Server are now handled completely internally. One option is available to control disk write behavior. This option controls how frequently the checkpoint process writes to disk.

Recovery Interval

The Recovery Interval option can be automatically configured. SQL Server setup sets it to 0, which means autoconfiguration. In SQL Server 2005, this means that the recovery time should be less than 1 minute. This option lets the database administrator control the checkpoint frequency by specifying the maximum number of

minutes that recovery should take, per database. SQL Server estimates how many data modifications it can roll forward in that recovery time interval. SQL Server then inspects the log of each database (every minute, if the recovery interval is set to the default of 0) and issues a checkpoint for each database that has made at least that many data modification operations since the last checkpoint. For databases with relatively small transaction logs, SQL Server issues a checkpoint when the log becomes 70 percent full, if that is less than the estimated number.

The Recovery Interval option does not affect the time it takes to undo long-running transactions. For example, if a long-running transaction takes two hours to perform updates before the server becomes disabled, the actual recovery takes considerably longer than the Recovery Interval value.

The frequency of checkpoints in each database depends on the amount of data modifications made, not on a time-based measure. So a database that is used primarily for read operations will not have many checkpoints issued.

As discussed earlier, most writing to disk doesn't actually happen during checkpoint operations. Checkpoints are just a way to guarantee that all dirty pages not written by other mechanisms will still be written to the disk in a timely manner. For this reason, you should keep the Recovery Interval value set at 0 (self-configuring).

Affinity I/O Mask and Affinity64 I/O Mask

These two options control affinity of a processor for I/O operations and work in much the same way as the two options for controlling processing affinity for workers. Setting a bit for a processor in either of these bit masks means that the corresponding processor will be used *only* for I/O operations. You will probably never need to set this option. However, if you do decide to use it, perhaps just for testing

purposes, you should use it in conjunction with the Affinity Mask or Affinity64 Mask option and make sure the bits set do not overlap. You should thus have a setting of 0 for both Affinity I/O Mask and Affinity Mask for a CPU, 1 for Affinity I/O Mask option and 0 for Affinity Mask, or 0 for Affinity I/O Mask and 1 for Affinity Mask.

Query Processing Options

SQL Server has several options for controlling the resources available for processing queries. As with all the other tuning options, your best bet is to leave the default values unless thorough testing indicates that a change might help.

Min Memory Per Query

When a query requires additional memory resources, the number of pages it gets is determined partly by the Min Memory Per Query option. This option is relevant for sort operations that you specifically request using an ORDER BY clause, and it also applies to internal memory needed by merge-join operations and by hash-join and hash-grouping operations. This configuration option allows you to specify a minimum amount of memory (in kilobytes) that any of these operations should be granted before they are executed. Sort, merge, and hash operations receive memory in a very dynamic fashion, so you rarely need to adjust this value. In fact, on larger machines, your sort and hash queries typically get much more than the Min Memory Per Query setting, so you shouldn't restrict yourself unnecessarily. If you need to do a lot of hashing or sorting, however, and you have few users or a lot of available memory, you might improve performance by adjusting this value. On smaller machines, setting this value too high can cause virtual memory to page, which hurts server performance.

Query Wait

The Query Wait option controls how long a query that needs additional memory will wait if that memory is not available. A setting of 1 means that the query waits 25 times the estimated execution time of the query, but it will always wait at least 25 seconds with this setting. A value of 0 or more specifies the number of seconds that a query will wait. If the wait time is exceeded, SQL Server generates error 8645:

`Server: Msg 8645, Level 17, State 1, Line 1
A time out occurred while waiting for memory
resources to execute the query. Re-run the query.`

Even though memory is allocated dynamically, SQL Server can still run out of memory if the memory resources on the machine are exhausted. If your queries time out with error 8645, you can try increasing the paging file size or even add more physical memory. You can also try tuning the query by creating more useful indexes so that hash or merge operations aren't needed. Keep in mind that this option affects only queries that have to wait for memory needed by hash and merge operations. Queries that have to wait for other reasons are not affected.

Blocked Process Threshold

This new option in SQL Server 2005 allows an administrator to request a notification when a user task has been blocked for more than the configured number of seconds. When Blocked Process Threshold is set to 0, no notification is given. You can set any value up to 86,400 seconds. When the deadlock monitor detects a task that has been waiting longer than the configured value, an internal

event is generated. You can choose to be notified of this event in one of two ways. You can use SQL Trace to create a trace and capture events of type Blocked process report, which you can find in the Errors and Warnings category on the Events Select screen in SQL Server Profiler. As long as a resource stays blocked on a deadlock-detectable resource, the event will be raised every time the deadlock monitor checks for a deadlock. An XML string will be captured in the Text Data column of the trace that describes the blocked resource and the resource being waited on. More information about deadlock detection is in [Chapter 8](#).

Alternatively, you can use event notifications to send information about events to a service broker service. Event notifications can provide a programming alternative to defining a trace, and they can be used to respond to many of the same events that SQL Trace can capture. Event notifications, which execute asynchronously, can be used to perform an action inside an instance of SQL Server 2005 in response to events with very little consumption of memory resources. Because event notifications execute asynchronously, these actions do not consume any resources defined by the immediate transaction.

Index Create Memory

The Min Memory Per Query option applies only to sorting and hashing used during query execution; it does not apply to the sorting that takes place during index creation. Another option, Index Create Memory, lets you allocate a specific amount of memory for index creation. Its value is also specified in kilobytes.

Query Governor Cost Limit

You can use the Query Governor Cost Limit option to specify the maximum number of seconds that a query can run. If you specify a *non-zero, nonnegative* value, SQL Server disallows execution of any query that has an estimated cost exceeding that value. Specifying 0 (the default) for this option turns off the query governor, and all queries are allowed to run without any time limit.

Max Degree Of Parallelism and Cost Threshold For Parallelism

SQL Server 2005 lets you run certain kinds of complex queries simultaneously on two or more processors. The queries must lend themselves to being executed in sections. Here's an example:

```
SELECT AVG(charge_amt), category  
FROM charge  
GROUP BY category
```

If the charge table has 100,000 rows and there are 10 different values for *category*, SQL Server can split the rows into groups and have only a subset of the groups processed on each processor. For example, with a four-CPU machine, categories 1 through 3 can be averaged on the first processor, categories 4 through 6 can be averaged on the second processor, categories 7 and 8 can be averaged on the third, and categories 9 and 10 can be averaged on the fourth. Each processor can come up with averages for only its groups, and the separate averages are brought together for the final result.

During optimization, the optimizer always finds the cheapest possible serial plan before considering parallelism. If this serial plan costs less than the configured value for the Cost Threshold For

Parallelism option, no parallel plan is generated. Cost Threshold For Parallelism refers to the cost of the query in seconds; the default value is 5. If the cheapest serial plan costs more than this configured threshold, a parallel plan is produced based on some assumptions about how many processors and how much memory will actually be available at run time. This parallel plan cost is compared with the serial plan cost, and the cheaper one is chosen. The other plan is discarded.

A parallel query execution plan can use more than one thread; a serial execution plan, which is used by a nonparallel query, uses only a single thread. The actual number of threads used by a parallel query is determined at query plan execution initialization and is called the degree of parallelism (DOP). The decision is based on many factors, including the Affinity Mask setting, the Max Degree Of Parallelism setting, and the available threads when the query starts executing.

You can observe when SQL Server is executing a query in parallel by querying the DMV `sys.dm_os_tasks`. A query that is running on multiple CPUs will have one row for each thread.

SELECT

```
task_address,  
task_state,  
context_switches_count,  
pending_io_count,  
pending_io_byte_count,  
pending_io_byte_average,  
scheduler_id,  
session_id,  
exec_context_id,  
request_id,  
worker_address,  
host_address
```

```
FROM sys.dm_os_tasks  
ORDER BY session_id, request_id;
```

Be careful when you use the Max Degree Of Parallelism and Cost Threshold For Parallelism options; they have server-wide impact.

There are other configuration options that I will not mention, most of which deal with aspects of SQL Server that are beyond the scope of this book. These include options for configuring remote queries, replication, SQL Agent, C2 auditing, and full-text search. There is a Boolean option to disallow use of the CLR in programming SQL Server objects; it is off (0) by default. The SQL Server 2000 option Allow Updates still exists but has no effect in SQL Server 2005. A few of the configuration options deal with programming issues, and these will be discussed in *Inside SQL Server: TSQL Programming*. These options include options for dealing with recursive and nested triggers, cursors, and accessing objects across databases.

Some of the configuration options take effect immediately after you set them, and they issue the *RECONFIGURE* (or in some cases, *RECONFIGURE WITH OVERRIDE*) statement. Others require that you restart your SQL Server instance.

[Table 3-3](#) lists all available configuration options, the possible settings, and the default values. The configuration options are marked with letter codes as follows:

- A = An advanced option. It should be changed only by an experienced database administrator, and it requires that you set the Show Advanced option to 1.
- RR = An option that requires a restart of the Database Engine.

- SC = Self-configuring option.

Table 3-3. SQL Server 2005 Configuration Options

Configuration Option	Minimum Value	Maximum Value	Default
Ad Hoc Distributed Queries (A)	0	1	0
Affinity I/O Mask (A, RR)	2147483648	2147483647	0
Affinity64 I/O Mask (A, available only on 64-bit version of SQL Server)	2147483648	2147483647	0
Affinity Mask (A)	2147483648	2147483647	0
Affinity64 Mask (A, available only on 64-bit version of SQL Server)	2147483648	2147483647	0
Agent XPs (A)	0	1	0

Configuration Option	Minimum Value	Maximum Value	Default
Allow Updates (Obsolete)	0	1	0
Awe Enabled (A, RR)	0	1	0
Blocked Process Threshold (A)	0	86400	0
C2 Audit Mode (A, RR)	0	1	0
CLR Enabled	0	1	0
Cost Threshold For Parallelism (A)	0	32767	5
Cross Db Ownership Chaining	0	1	0
Cursor Threshold (A)	1	2147483647	1
Database Mail XPs (A)	0	1	0

Configuration Option	Minimum Value	Maximum Value	Default
Default Full-Text Language (A)	0	2147483647	1033
Default Language	0	9999	0
Default Trace Enabled (A)	0	1	1
Disallow Results From Triggers (A)	0	1	0
Fill Factor (A, RR)	0	100	0
Ft Crawl Bandwidth (Max), (A)	0	32767	100
FT Crawl Bandwidth (min), (A)	0	32767	0
FT Notify Bandwidth (max), (A)	0	32767	100
FT Notify Bandwidth (min), (A)	0	32767	0

Configuration Option	Minimum Value	Maximum Value	Default
Index Create Memory (A, SC)	704	2147483647	0
In-Doubt Xact Resolution (A)	0	2	0
Lightweight Pooling (A, RR)	0	1	0
Locks (A, RR, SC)	5000	2147483647	0
Max Degree Of Parallelism (A)	0	64	0
Max Full-Text Crawl Range (A)	0	256	4
Max Server Memory (A, SC)	16	2147483647	2147483647
Max Text Repl Size	0	2147483647	65536

Configuration Option	Minimum Value	Maximum Value	Default
Max Worker Threads (A, RR)	128	32767 (1024 is the maximum recommended for 32-bit operating systems.)	0 auto-configures the number of worker threads based on the number of processors using the formula (256 + [number of processors * 4] * 8).
Media Retention (A, RR)	0	365	0
Min Memory Per Query (A)	512	2147483647	1024
Min Server Memory (A, SC)	0	2147483647	8
Nested Triggers	0	1	1

Configuration Option	Minimum Value	Maximum Value	Default
Network Packet Size (A)	512	32767	4096
Ole Automation Procedures (A)	0	1	0
Open Objects (A, RR, obsolete)	0	2147483647	0
PH_Timeout (A)	1	3600	60
Precompute Rank (A)	0	1	0
Priority Boost (A, RR)	0	1	0
Query Governor Cost Limit (A)	0	2147483647	0
Query Wait (A)	1	2147483647	1
Recovery Interval (A, SC)	0	32767	0

Configuration Option	Minimum Value	Maximum Value	Default
Remote Access (RR)	0	1	1
Remote Admin Connections	0	1	0
Remote Login Timeout	0	2147483647	20
Remote Proc Trans	0	1	0
Remote Query Timeout	0	2147483647	600
Scan For Startup Procs (A, RR)	0	1	0
Server Trigger Recursion	0	1	1
Set Working Set Size (A, RR, obsolete)	0	1	0
Show Advanced Options	0	1	0

Configuration Option	Minimum Value	Maximum Value	Default
SMO and DMO XPs (A)	0	1	1
SQL Mail XPs (A)	0	1	0
Transform Noise Words (A)	0	1	0
Two Digit Year Cutoff (A)	1753	9999	2049
User Connections (A, RR, SC)	0	32767	0
User Instance Timeout (A, appears only in SQL Server 2005 Express Edition)	5	65535	10
User Instances Enabled (A, appears only in SQL Server 2005 Express Edition)	0	1	0

Configuration Option	Minimum Value	Maximum Value	Default
User options	0	32767	0
Web Assistant Procedures (A)	0	1	0
xp_cmdshell (A)	0	1	0

You can determine the specific value for each configuration option by using the following statement.

```
SELECT * FROM sys.configurations
ORDER BY name ;
```

The Default Trace

One final option that doesn't seem to fit into any of the other categories is a new option in SQL Server 2005 called Default Trace Enabled. I mention it because the default value is 1, which means that as soon as SQL Server starts, it runs a server-side trace, capturing a predetermined set of information into a predetermined location. None of the properties of this default trace can be changed; the only thing you can do is turn it off.

You can compare the default trace to the blackbox trace in SQL Server 2000, but the blackbox trace takes a few steps to create, and it takes even more steps to have it start automatically when your SQL Server starts. This trace is so lightweight that you might find little reason to ever disable it. If you're not familiar with SQL Server tracing, you'll probably need to spend some time reading about traces; the GUI for creating them is called the SQL Server Profiler.

The default trace output file is stored in the same directory in which you installed your SQL Server, in the \log subdirectory. So if you've installed your SQL Server in the default location, the captured trace information will be in the file C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\LOG\log.trc. Every time you stop and restart SQL Server, a new trace file is created with a sequential numerical suffix, so the second trace file would be Log_01.trc, followed by Log_02.trc, and so on. If all the trace log files are removed or renamed, the next log file starts at Log.trc again.

You can open the trace files created through the default trace mechanism by using the SQL Server Profiler, just as you can any other trace file, or you can copy to a table by using the system function *fn_trace_gettable* and view the current contents of the trace while the trace is still running. As with any server-side trace that writes to a file, the writing is done in 128-KB blocks. Thus, on a very low-use SQL Server instance, it might look like nothing is being written to the file for quite some time. You need 128 KB of data for any writes to the physical file to occur. In addition, when the SQL Server service is stopped, whatever events have accumulated for this trace will be written out to the file.

Unlike the blackbox trace in SQL Server 2000, which captures every single batch completely and can quickly get huge, the default trace in SQL Server 2005 captures only a small set of events that were deemed likely to cause stability problems or performance degradation of SQL Server. The events captured are the following. (Please see the documentation on monitoring and tuning SQL

Server for performance for a description of what each event captures.)

- *ServiceControl*
- *Login*
- *Object*
- *Hash*
- *Sort Warnings*
- *Missing Column Statistics*
- *Missing Join Predicate*
- *Server Memory*
- *Data File Auto Grow*
- *Log File Auto Grow*
- *Data File Auto Shrink*
- *Log File Auto Shrink*
- *Audit Statement GD*
- *Audit Object GDR*

- *Audit Add/Drop Login*
- *Audit Login GDR*
- *Audit Login Change Property*
- *Audit Add Login to Server Role*
- *Audit Add DB User*
- *Audit Add Member to DB*
- *Audit Add/Drop Role*
- *Audit Backup/Restore*
- *Audit DBCC*

Not only can you not change anything about the files saved or their locations, you can't add or remove events, the data captured along with the events, or the filters that might be applied to the events. If you want something slightly different than the default trace, you can disable the predefined trace and create your own with whatever events, data, and filters you choose. Of course, you must then make sure the trace starts automatically. This is not impossible to do, but I suggest that you leave the default trace on, in addition to whatever other traces you need, so that you know that at least some information about the activities taking place on SQL Server is being captured.

Final Words

This chapter covered the primary tools for changing the behavior of SQL Server. The primary means of changing the behavior is by using configuration options, so we looked at the options that can have the biggest impact on SQL Server behavior, in particular its performance. To really know when changing the behavior is a good idea, it's important that you understand how and why SQL Server works the way it does. My hope is that this chapter has laid the groundwork for you to make informed decisions about configuration changes.

Chapter 4. Databases and Database Files

In this chapter:

<u>System Databases</u>	<u>88</u>
<u>Sample Databases</u>	<u>90</u>
<u>Database Files</u>	<u>92</u>
<u>Creating a Database</u>	<u>94</u>
<u>Expanding or Shrinking a Database</u>	<u>97</u>
<u>Using Database Filegroups</u>	<u>101</u>
<u>Altering a Database</u>	<u>104</u>
<u>Databases Under the Hood</u>	<u>106</u>
<u>Setting Database Options</u>	<u>115</u>

<u>Database Snapshots</u>	<u>127</u>
<u>The <i>tempdb</i> Database</u>	<u>132</u>
<u>Database Security</u>	<u>137</u>
<u>Moving or Copying a Database</u>	<u>142</u>
<u>Compatibility Levels</u>	<u>147</u>
<u>Summary</u>	<u>148</u>

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 installation 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 and views 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 1 megabyte (MB) to a technical limit of 1,048,516 terabytes.
- It can grow and shrink, either automatically or by command.
- 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 options set 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.

While 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 2005 installation always includes four databases: *master*, *model*, *tempdb*, and *msdb*. It also contains a fifth, "hidden" database that you will 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 *mssql/systemresource*.

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 it contains, the *master* database has system catalogs that keep information about disk space, file allocations and usage, systemwide configuration settings, endpoints, login accounts, databases on the current instance, and the existence of other SQL servers (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 after performing such actions, you should back up *master*.

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 will inherit them. You can also change most properties of the *model* database by using the ALTER DATABASE command, and those property values will then be 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 will 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 will be lost the next time you restart your SQL Server instance. An alternative is to create the object in the *model* database, from which *tempdb* is copied.

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.

mssqlsystemresource

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 it 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 will 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 data directory by using Windows Explorer. My data directory is at C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\data; I can see a file called

`mssqlsystemresource.mdf`, which is 38 MB in size, and `mssqlsystemresource.ldf`, which is 0.5 MB. The created and modified date for both of these files is the day I installed this SQL Server instance, but their last accessed date is today.

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 SQL Server 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\Data\
mssqlsystemresource_COPY.mdf'), (NAME = log, FILENAME =
'C:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\Data\mssqlsystemresource_COPY.ldf')
FOR ATTACH;
```

SQL Server will treat this new `resource_COPY` database like any other user database, and it will 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` will obviously 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, and that connection will be able to 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](#), 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, which performs scheduled activities such as backups and replication tasks, and the Service Broker, which provides queuing and reliable messaging for SQL Server. When you are not performing backups and maintenance on this database, you should 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 SQL Server 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 tables: 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 Microsoft 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. In SQL Server 2005, 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 2005 does not automatically install any sample databases. However, three sample databases are widely available.

AdventureWorks

The *AdventureWorks* database was created by the Microsoft User Education group as an example of what a "real" database might look like. It is an optional component that you can choose to install during the installation process. The database was designed to showcase SQL Server 2005 features, in particular the organization of objects into different schemas. The design is also highly normalized. While normalized data and many separate schemas might closely map 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 will use simple tables that I create; if more than a few rows of data are needed, I'll copy data from one or more *AdventureWorks* tables into tables of my own. It's a good idea to become familiar with the design of the *AdventureWorks* database because many of the examples in Books Online and in white papers published on the Microsoft Web site use data from this database. Note that it is also possible to install an *AdventureWorksDW* database, which includes data and features relevant to a data

warehouse as well as SQL Server 2005 data warehousing features. I will not discuss that database in this book.

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 automatically installed 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.

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 completely 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 Access. Much of the preSQL Server 2005 documentation dealing with APIs 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*.

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 2005 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.

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. 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 4-1](#). The columns not mentioned here contain information dealing with transaction log backups relevant to the particular file, and I'll be discussing the transaction log in [Chapter 5](#).

Table 4-1. The `sys.database_files` View

Column	Description
<i>fileid</i>	The file identification number (unique for each database).
<i>file_guid</i>	GUID for the file. NULL = Database was upgraded from an earlier version of Microsoft SQL Server.
<i>type</i>	File type: 0 = Rows 1 = Log 2 = Reserved for future use. 3 = Reserved for future use. 4 = Full-text

Column	Description
<i>type_desc</i>	Description of the file type: ROWS LOG FULLTEXT
<i>data_space_id</i>	ID of the data space to which this file belongs. Data space is a filegroup. 0 = Log file.
<i>name</i>	The logical name of the file.
<i>physical_name</i>	Operating-system file name.

Column	Description
<i>state</i>	File state: 0 = ONLINE 1 = RESTORING 2 = RECOVERING 3 = RECOVERY_PENDING 4 = SUSPECT 5 = Reserved for future use. 6 = OFFLINE 7 = DEFUNCT
<i>state_desc</i>	Description of the file state: ONLINE RESTORING RECOVERING RECOVERY_PENDING SUSPECT OFFLINE DEFUNCT

Column	Description
<i>size</i>	<p>Current size of the file, in 8-kilobyte (KB) pages.</p> <p>0 = Not applicable</p>
	<p>For a database snapshot, size reflects the maximum space that the snapshot can ever use for the file.</p>
<i>max_size</i>	<p>Maximum file size, in 8-KB pages: 0 = No growth is allowed.</p> <p>1 = File will grow until the disk is full.</p> <p>268435456 = Log file will grow to a maximum size of 2 terabytes.</p>
<i>growth</i>	<p>0 = File is fixed size and will not grow.</p> <p>>0 = File will grow automatically.</p> <p>If <i>is_percent_growth</i> = 0, growth increment is in units of 8-KB pages, rounded to the nearest 64 KB.</p> <p>If <i>is_percent_growth</i> = 1, growth increment is expressed as a whole number percentage.</p>

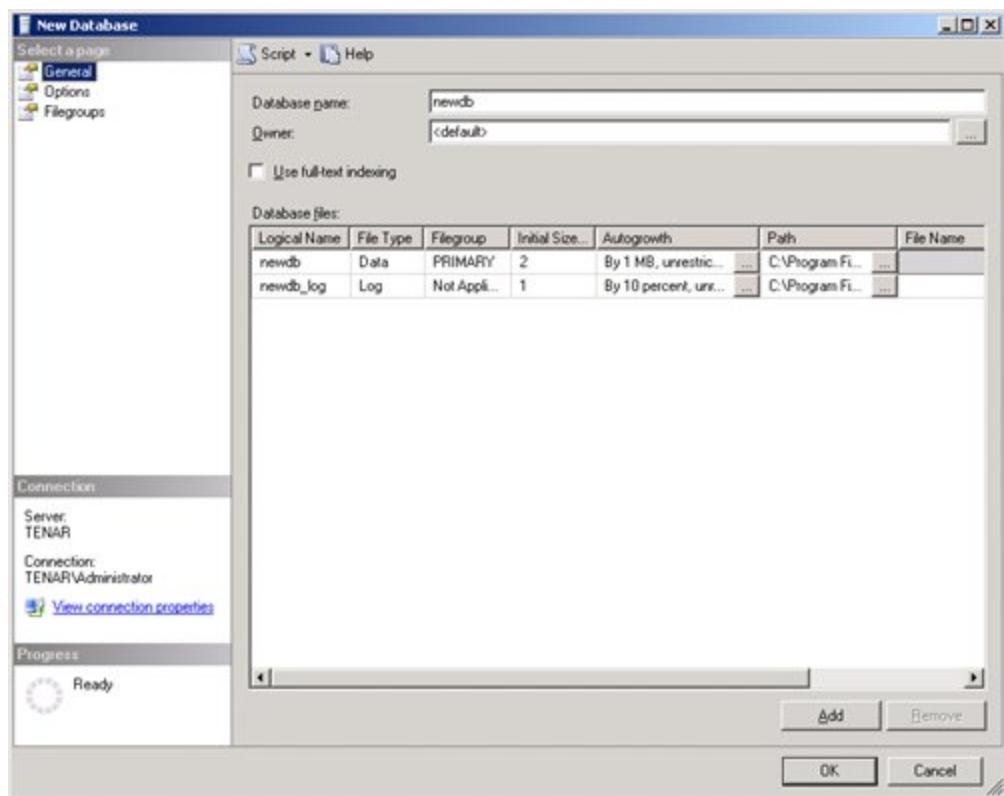
Column	Description
<i>is_media_read_only</i>	1 = File is on read-only media. 0 = File is on read/write media.
<i>is_read_only</i>	1 = File is marked read-only. 0 = File is marked read/write.
<i>is_sparse</i>	1 = File is a sparse file. 0 = File is not a sparse file. (Sparse files are used with database snapshots, discussed later in this chapter.)
<i>is_percent_growth</i>	See description for <i>growth</i> column, above.
<i>is_name_reserved</i>	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 SQL Server Management Studio, which provides a graphical front end to the Transact-SQL commands and stored procedures that actually create the database and set its properties. [Figure 4-1](#) shows the New Database dialog box, which represents the Transact-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.

Figure 4-1. The New Database dialog box, where you can create a new database

[[View full size image](#)]



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 47 objects, 41 system tables and 6 objects used for SQL Server Notification Services and Service Broker. 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 1788 objects. It turns out that most of these objects are not really

stored in the *model* database but are accessible through it. In [Chapter 6](#), 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.

A new user database must be 1 MB in size or larger, 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 (as determined at the time SQL Server was 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, and the default size of the log file is 1 MB. Whether the database name, *newdb*, is case sensitive depends on the sort order 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 will grow 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 terabyte, gigabyte (GB), 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. This is a change from SQL Server 2000, where the default growth increment for data files was 10 percent. In SQL Server 2005, the log file FILEGROWTH default is 10 percent, the same as it was in SQL Server 2000.

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 = 200,
FILEGROWTH = 20),
( NAME = Arch2,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\archdat2.ndf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20)
LOG ON
( NAME = Archlog1,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\archlog1.ldf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20);
```

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 than manual shrinkage. Log files have their own rules for growing and shrinking; I'll discuss changes in log file size in [Chapter 5](#).

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 recover disk space.

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 specified when the file is first defined can be qualified using the suffix *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. (Ten percent of 10,000 MB is 1000 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 will fail. The query would have failed anyway if autogrow wasn't set, but with autogrow enabled, SQL Server will spend a lot of time trying to grow the file, and you won't be informed of the problem immediately.

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. 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 manually expand a database file by using the ALTER DATABASE command 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

In SQL Server 2005, data 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 2005 documentation describes the instant file initialization feature as an "option," it is not 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 (MSSQLSERVER) service account is in the Windows Administrator role and your SQL Server is running on a Windows XP or Windows 2003 file system, instant file initialization will be used. If you want to make sure your database files are zeroed out as they are created and expanded, you can use traceflag 1806 to always zero the space, as previous SQL Server versions did.

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 autoshrinkwhich always has a session ID (SPID) of 6 in SQL Server 2005 (but there's no guarantee SQL Server will use the same SPID in future versions)shrinks databases at 30-minute intervals. I'll discuss the DBCC SHRINKDATABASE command in more detail momentarily.

Manual Shrinkage

You can manually shrink a database 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 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 statement 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. The database can't be made smaller than the *model* database, and DBCC SHRINKDATABASE 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 will then shrink the file to a size of 80 MB, and you'll have 20 MB of free space in addition to the original 60 MB of data. In other words, the 80-MB file will have 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 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 is the same as that used with DBCC SHRINKFILE. 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.

Two options for the DBCC SHRINKDATABASE and DBCC SHRINKFILE commands can force SQL Server to do either of the two steps just mentioned, while a third option is available only to DBCC SHRINKFILE:

- **NOTRUNCATE** This option causes all the freed file space to be retained in the database files. SQL Server compacts the data only by moving it to the front of the file. The default is to release the freed file space to the operating system.
- **TRUNCATEONLY** This option causes any unused space in the data files to be released to the operating system. No attempt is made to relocate rows to unallocated pages. When TRUNCATEONLY is used, *target_size* and *target_percent* are ignored.
- **EMPTYFILE** This option, available only with DBCC SHRINKFILE, empties the contents of a data file and moves them to other files in the filegroup.

Note



DBCC SHRINKFILE specifies a target size in megabytes. DBCC SHRINKDATABASE specifies a target percentage of free space to leave in the database.

Both the DBCC SHRINKFILE command and the DBCC SHRINKDATABASE command give a report for each file that can be shrunk. For example, if my *pubs* database currently has an 8-MB data file and a log file of about the same size, I get the following report when I issue this DBCC SHRINKDATABASE command:

```
DBCC SHRINKDATABASE(pub, 10);
```

RESULTS:

DbId	FileId	CurrentSize	MinimumSize	UsedPages
EstimatedPages				
5	1	256	80	152
152				
5	2	1152	63	1152
56				

The current size is the size in pages after any shrinking takes place. In this case, the database file (FileId = 1) was shrunk to 256 pages of 8 KB each, which is 2 MB. But only 152 pages were used. There might be several reasons for the difference between used pages and current pages:

- If I asked to leave a certain percentage free, the current size will be bigger than the used pages because of that free space.
- If the minimum size to which I can shrink a file is bigger than the used pages, the current size cannot become smaller than the minimum size.

- If the size of the data file for the *model* database is bigger than the used pages, the current size cannot become smaller than the size of *model*'s data file.

For the log file (FileId = 2), the only values that really matter are the current size and the minimum size. The other two values are basically meaningless for log files because the current size is always the same as the used pages and because there is really no simple way to estimate how small a log file can be shrunk. 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 5](#).

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 will have 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. I'll discuss fragmentation and defragmentation in [Chapter 7](#).

It is possible for shrink operations to be blocked by a transaction that has been enabled for the snapshot isolation level. 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 2005 also provides progress reporting for the SHRINK commands, available through the sys.*dm_exec_requests* view. I discuss progress

reporting in the section on DBCC commands, or you can get the full details from the Books Online page for *sys.dm_exec_requests*.

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 disk drives. The filegroup containing the primary data file is called the *primary filegroup*. There is only one primary filegroup, and if you don't specifically ask to place files in other filegroups when you create your database, *all* of your data files will be 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 statement.

Don't confuse the primary filegroup and the primary file:

- 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 called *sysfiles1* 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 statement. The default filegroup contains the pages for all tables and indexes that aren't placed in a specific filegroup.

Most SQL Server databases have a single data file in one (default) filegroup. In fact, most users will 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 filegroupsshe 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 2005. Only then will they need to use filegroups. They can then use Object Explorer in SQL Server Management Studio to create the database on multiple filegroups. Then they can right-click on the database name in Object Explore 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 12-GB file, you would need to restore it to a database with one file of that size. If you don't have another 12-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 4-GB drives available than one large 12-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.

Objects that have space allocated to them, namely tables and indexes, are created on a particular filegroup. If the filegroup is not specified, they 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 will have two extents (or units of space) allocated from it for each extent allocated from the second file. I'll discuss extents in more detail later in this chapter.

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 (VLDBs) 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 it in more detail in [Chapter 5](#).

A FILEGROUP CREATION Example

This example creates a database named *sales* with three filegroups:

- The primary filegroup with the files Spri1_dat and Spri2_dat. The FILEGROWTH increment for both of these files is specified as 15 percent.
- A filegroup named SalesGroup1 with the files SGrp1Fi1 and SGrp1Fi2.
- A filegroup named SalesGroup2 with the files SGrp2Fi1 and SGrp2Fi2.

```
CREATE DATABASE Sales
ON PRIMARY
```

```
( NAME = SPri1_dat,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\SPri1dat.mdf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
( NAME = SPri2_dat,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\SPri2dat.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\SG1Fi1dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
( NAME = SGrp1Fi2_dat,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\SG1Fi2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
FILENAME =
    'c:\program files\microsoft sql
server\mssql.1\mssql\data\SG2Fi1dt.ndf',
SIZE = 10,
MAXSIZE = 50,
```

```
FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
FILENAME =
      'c:\program files\microsoft sql
server\mssql.1\mssql\data\SG2Fi2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
FILENAME =
      'c:\program files\microsoft sql
server\mssql.1\mssql\data\saleslog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB );
```

Altering a Database

You can use the ALTER DATABASE statement 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. You can optionally put these files in a user-defined filegroup. All files added in a single ALTER DATABASE statement 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 statement.)
- 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. (In SQL Server 2000, only files in the *tempdb* database can be moved in this way.) 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 is a new feature in SQL Server 2005; it 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 statement 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 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 = 20MB);
```

The following example creates a new filegroup in a database, adds two 5-MB files to the filegroup, and makes the new filegroup the default filegroup. We need three ALTER DATABASE statements.

```
ALTER DATABASE Test1
ADD FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
ADD FILE
```

```
( NAME = 'test1dat3',
FILENAME =
    'c:\program files\microsoft sql server\
mssql.1\mssql\data\t1dat3.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB),
( NAME = 'test1dat4',
FILENAME =
    'c:\program files\microsoft sql server\
mssql.1\mssql\data\t1dat4.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB)
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, or DBID, 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, DBID, 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 2005 doesn't allocate entire extents to tables with small amounts of data. SQL Server 2005 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. Almost 8000 bytes, or 64,000 bits, are available on the page after the header and other overhead are accounted for, 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 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 4-2](#) shows the bit patterns that each extent has set in the GAM and SGAM, based on its current use.

Table 4-2. 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

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 GAM of 0 and a value in the SGAM of 1. 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 quickly locate the GAMs in a file because a GAM is always the third page in any database file (page 2). An SGAM is the fourth page (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 (which I'll discuss shortly). In [Chapter 6](#), I'll say more about how individual pages within a table look. For now, because I'm talking about space allocation, I'll examine how to keep track of which pages belong to which tables.

Index Allocation Map (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 types: pages holding regular in-row data, pages holding Large Object (LOB) data, or pages holding row-overflow data. I'll discuss these three types of pages, and when each kind is used, in [Chapter 6](#).

For example, a table on four partitions that has all three types of data (in-row, LOB, and row-overflow) will have at least 12 IAM pages. Again, a single IAM 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 page header; an IAM page header, which contains eight page-pointer slots; and 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 its extents are uniform extents which means that an object will never need 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. I'll discuss allocation units in more detail in [Chapter 6](#) when I discuss object data storage.

In addition to GAMs, SGAMs, and IAMs, a database file has three other types of special allocation pages. Page Free Space (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 8088th page thereafter. I'll talk about them more in [Chapter 6](#). 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 5](#). 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.

Checking Database Consistency

DBCC stood for Database Consistency Checker in versions of SQL Server prior to SQL Server 2000. However, since Microsoft acquired the code base for the product from Sybase, DBCC began to take on more and more functionality, and eventually went way beyond mere consistency checking. For example, DBCC is used to shrink a database or a data file and to clear out the data or plan cache. Starting in SQL Server 2000, Microsoft finally acknowledged this evolution, and the glossary in Books Online for both SQL Server 2000 and SQL Server 2005 actually defines DBCC as Database Console Command and divides the commands into four categories: validation, maintenance, informational, and miscellaneous.

In this section, I will discuss the DBCC commands that actually do consistency checking of the database, that is, the validation commands. These commands are the CHECK commands: DBCC CHECKTABLE, DBCC CHECKDB, DBCC CHECKALLOC, DBCC CHECKFILEGROUP, and DBCC CHECKCATALOG. Two others, DBCC CHECKCONSTRAINT and DBCC CHECKIDENT, will be

described in [Chapter 7](#), where I'll also discuss some of the table and index maintenance DBCC commands, such as DBCC CLEANTABLE and DBCC UPDATEUSAGE. I will cover DBCC INDEXDEFRAG and its SQL Server 2005 replacement when I cover indexes in [Chapter 7](#).

The most comprehensive of the DBCC validation commands is DBCC CHECKDB. Here is the complete syntax:

```
DBCC CHECKDB
[ 
    [ ( 'database_name' | database_id | 0
        [ , NOINDEX
        | , { REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST
    | REPAIR_REBUILD } ]
        ) ]
    [ WITH
        {
            [ ALL_ERRORMSGS ]
            [ , NO_INFOMSGS ]
            [ , TABLOCK ]
            [ , ESTIMATEONLY ]
            [ , { PHYSICAL_ONLY | DATA_PURITY } ]
        }
    ]
]
```

I'll discuss most of these options to the DBCC CHECKDB command shortly. As part of its operation, DBCC CHECKDB runs all of the other DBCC validation commands in this order:

- **DBCC CHECKALLOC is run on the database.** DBCC CHECKALLOC validates the allocation information maintained

in the GAM, SGAM, and IAM pages. You can think of DBCC CHECKALLOC as performing cross-reference checks to verify that every extent that the GAM or SGAM indicates has been allocated really has been allocated, and that any extents not allocated are indicated in the GAM and SGAM as not allocated. DBCC CHECKALLOC also verifies the IAM chain for each allocation unit, including the consistency of the links between the IAM pages in the chain. Finally, DBCC CHECKALLOC verifies that all extents marked as allocated to the allocation unit really are allocated.

- **DBCC CHECKTABLE is run on every table and indexed view in the database.** DBCC CHECKTABLE performs a comprehensive set of checks on the structure of a table, and by default these checks are both physical and logical. With the `physical_only` option to the DBCC command specified, you can exclude the logical checks and only validate the physical structure of the page and the record headers. The `physical_only` option is intended to provide a lightweight check of the physical consistency of the table and common hardware failures that can compromise data. In SQL Server 2005, a full run of DBCC CHECKTABLE can take considerably longer than in earlier versions.

Indexed views are verified by regenerating the view's rowset from the underlying SELECT statement definition and comparing the results with the data stored in the indexed view. SQL Server performs two left-anti-semi joins between the two rowsets to make sure that there are no rows in one that are not in the other.

- **DBCC CHECKCATALOG is run on the database.** DBCC CHECKCATALOG performs more than 50 crosschecks between various metadata tables. You cannot fix errors that it finds by running the DBCC operation with any of the REPAIR

options. Prior to SQL Server 2005, DBCC CHECKCATALOG was not included in a DBCC CHECKDB operation and had to be run separately.

- **The Service Broker data in the database is verified.**

Running this command is the only way to check the Service Broker data because there is no specific DBCC command to perform the checks. You can also consider DBCC CHECKFILEGROUP to be a subset of DBCC CHECKDB because DBCC CHECKFILEGROUP performs DBCC CHECKTABLE on all tables and views in a specified filegroup.

Because they are included as part of DBCC CHECKDB, the DBCC CHECKALLOC, DBCC CHECKTABLE, and DBCC CHECKCATALOG commands do not have to be run separately if DBCC CHECKDB is run regularly. If you choose to run any of these commands individually, you can refer to Books Online for the complete syntax.

On an upgraded database with no 2005 features or indexed views, DBCC CHECKDB will actually run slightly faster than its SQL Server 2000 counterpart. However, on a new SQL Server 2005 database, some of the logical checks added to complement new features in SQL Server 2005 are necessarily complex and do add to the runtime when invoked, so you may find that DBCC CHECKDB takes longer to run than you might have expected.

Performing Validation Checks

In SQL Server 2005, all of the DBCC validation commands use database snapshot technology to keep the validation operation from interfering with ongoing database operations and to allow the validation operation to see a quiescent, consistent view of the data, no matter how many changes were made to the underlying data.

while the operation was under way. I'll discuss database snapshots in more detail later in this chapter. A snapshot of the database is created at the beginning of the CHECK command, and no locks are acquired on any of the objects being checked. The actual check operation is executed against the snapshot.

As you'll see when we discuss database snapshots, the original version of a page is copied into the snapshot database when updates occur in the source, so the snapshot always reflects the original version of the data. Unlike regular database snapshots, the "snapshot file" that DBCC CHECKDB creates with the original page images is not visible to the end user and its location cannot be configured; it always uses space on the same volume as the database being checked. This capability is available only when your data directory is on an NTFS partition.

If you aren't using NTFS, or if you don't want to use the space necessary for the snapshot, you can avoid creating the snapshot by using the WITH TABLOCK option with the DBCC command. In addition, if you are using one of the REPAIR options to DBCC, a snapshot is not created because the database is in single-user mode, so no other transactions can be altering data. Without the TABLOCK option, the DBCC validation commands are considered online operations because they don't interfere with other work taking place in a database. With the TABLOCK option, however, a Shared Table lock is acquired for each table as it processed, so concurrent modification operations will be blocked. Similarly, if modification operations are in progress on one or more tables, a DBCC validation command being run with TABLOCK will block until the transaction performing the modifications is completed.

The DBCC validation checks can require a significant amount of space because SQL Server needs to temporarily store information about pages and structures that have been observed during the check operation, for cross-checking against pages and structures that are observed later during the DBCC scan. To determine the

tempdb needs in advance, you can run a DBCC validation check with the ESTIMATEONLY option. For example, if I want to see how much *tempdb* space I might need to run DBCC CHECKDB on the AdventureWorks database, I can run the following:

```
SET NOCOUNT ON;
DBCC CHECKDB ('Adventureworks') WITH ESTIMATEONLY;
```

Here is the output I receive:

Estimated TEMPDB space needed for CHECKALLOC (KB)

72

Estimated TEMPDB space needed for CHECKTABLES (KB)

198542

Note that even though AdventureWorks is considered just a sample database, it can require up to 193 MB of *tempdb* space to run to completion. There are several large indexes in *tempdb* that contribute to this large space requirement, and in addition, this value is computed as a worst-case estimate and assumes there will not be room in memory for any of the sort operations required.

SQL Server keeps track of the last error-free run of DBCC CHECKDB in the bootpage for every database, and it reports the date and time of the operation in the error log when SQL Server is started. Here is what the message might look like for the AdventureWorks database:

Date 1/24/2006 2:15:52 PM

Message

DBCC CHECKDB (AdventureWorks) executed by
TENAR\Administrator found 0 errors and repaired 0
errors. Elapsed time: 0 hours 5 minutes 8 seconds.

Validation Checks

SQL Server 2005 includes a set of logical validation checks to verify that data is appropriate for the column's datatype. These checks can be expensive and can affect the server's performance, so you can choose to disable this, along with all the other non-core logical validations by using the PHYSICAL_ONLY option. All new databases created in SQL Server 2005 have the DATA_PURITY logical validations enabled by default. For databases that have been upgraded from previous SQL Server versions, you must run DBCC CHECKDB with the DATA_PURITY option once, preferably immediately after the upgrade, as follows:

```
DBCC CHECKDB (<db_name>) WITH DATA_PURITY
```

After the purity check completes without any errors for a database, performing the logical validations is the default behavior in all future executions of DBCC CHECKDB, and there is no way to change this default. You can, of course, override the default with the PHYSICAL_ONLY option. This option not only skips the data purity checks, but it also skips any checks that actually have to analyze the contents of individual rows of data and basically limits the

checks that DBCC performs to the integrity of the physical structure of the page and the row headers.

If the CHECKSUM option is enabled for a database, which is the default in all new SQL Server 2005 databases, a checksum will be performed on each allocated page as it is read by the DBCC CHECK commands. As I will mention again in the upcoming section on database options, when the CHECKSUM option is on, a page checksum is calculated and written on each page as it is written to disk, so only pages that have been written since CHECKSUM was enabled will have this check done. The page checksum value is checked during the read and compared with the original checksum value stored on the page. If they do not match, an error is generated. In addition, pages with errors are recorded in the suspect_pages table in the *msdb* database.

DBCC Repair Options

The validation commands DBCC CHECKDB, DBCC CHECKTABLE, and DBCC CHECKALLOC allow you to indicate whether you want SQL Server to attempt to repair any errors that might be found. The syntax for the DBCC validation commands (except for DBCC CHECKCATALOG) allows you to specify either REPAIR_ALLOW_DATA_LOSS or REPAIR_REBUILD. Syntactically, you can also specify REPAIR_FAST, but that option is maintained only for backward compatibility, and no repair actions are performed.

Almost all the possible errors that the DBCC command can detect can be repaired. The exceptions are errors found through DBCC CHECKCATALOG and data purity errors found through DBCC CHECKTABLE. When you run DBCC CHECKDB with one of the REPAIR options, SQL Server first runs DBCC CHECKALLOC and repairs what it can, and then it runs DBCC CHECKTABLE on all tables and makes the appropriate repairs on all the tables. The

possible repairs for each table are ranked as SQL Server compiles the list of what needs repairing, to make the entire DBCC operation as efficient as possible. In this way, you won't end up, for example, in a situation where an index is being rebuilt, and then a page from table has to be removed, invalidating the work of rebuilding the index.

If you're running a DBCC command with REPAIR_ALLOW_DATA_LOSS, SQL Server tries to repair almost *all* detected errors, even at the risk of losing some data. Keep in mind that for almost any severe error, some data will be lost when the repair is run. During the repair, rows might be deleted if they are found to be inconsistent, such as when a computed column value is incorrect. Whole pages can be deleted if checksum errors are discovered. During the repair, no attempt is made to maintain any constraints on the tables, or between tables. Some errors SQL Server won't even try to repair particularly if the GAM or SGAM pages themselves are corrupted and unreadable.

If you use the REPAIR_REBUILD option, SQL Server performs both minor, relatively fast repair actions such as repairing extra keys in nonclustered indexes and time-consuming repairs such as rebuilding indexes. These types of repairs can be performed without risk of data loss. After the successful completion of the DBCC command, the database is physically consistent and online but might not be in a logically consistent state in terms of constraints and your business rules. For this reason, you should use the REPAIR options only as a last resort. A much better solution in the case of non-fixable errors is to restore a database from a backup or restore a smaller unit of the database, such as a single filegroup. If you are going to use the REPAIR_ALLOW_DATA_LOSS option, you should back up the database before you run the DBCC command.

You can run the REPAIR options for DBCC inside a user-defined transaction, which means you can perform a ROLLBACK to undo the repairs that have been made. The exception is when you are

running the REPAIR options on a database in EMERGENCY mode, which I discuss later in the section on database options. (If a repair in EMERGENCY mode fails, there are no further options except to restore the database from a backup.) Each individual repair in the DBCC operation runs in its own system transaction, which means that if a repair is not possible, it will not affect any of the other repairs, unless subsequent repairs depended on an earlier success repair. If you do run one of the REPAIR options, you can provide a partial safeguard by creating a database snapshot before the repair is initiated, starting a transaction, and then running DBCC with the REPAIR option. Before committing or rolling back, you can compare the repaired database with the original in the snapshot. If you are not satisfied with the changes made as part of the repair, you can roll back the repair operation.

Progress Reporting

Many of the DBCC commands in SQL Server 2005 provide progress reporting in the dynamic management view called `sys.dm_exec_requests`. Take a look at the following columns:

- `command` indicates current DBCC command phase
- `percent_complete` represents [%] completion of DBCC command phase
- `estimated_completion_time` (in milliseconds) represents an estimate of how long it will take to finish the task, based on past progress

Progress reporting is available for DBCC CHECKDB, DBCC CHECKTABLE, and DBCC CHECKFILEGROUP. DBCC CHECKALLOC is not included in this list because it is such a fast

command there would be no need (and usually no time) to check the progress. The command would be done before you had a chance to select from `sys.dm_exec_requests`. Progress reporting is also available for some of the maintenance commands, such as `DBCC SHRINKFILE` and `DBCC SHRINKDATABASE`. SQL Server will also populate the progress report columns when defragmenting an index using `ALTER INDEX` with the `REORG` option, because this command is equivalent to `DBCC INDEXDEFrag`, which also supports progress reporting.

DBCC Best Practices

Consider the following guidelines when planning how and when to use the DBCC validation commands:

- Use `CHECKDB` with the `CHECKSUM` database options and a sound backup strategy to protect the integrity of your data from hardware-caused corruption.
- There is no hard-and-fast rule for how often to run DBCC; it depends on how critical your data is, the quality of your hardware, and the frequency of your backups.
- Perform `DBCC CHECKDB` with the `DATA_PURITY` option after upgrading a database to SQL Server 2005 to check for invalid data values.
- Make sure you have enough disk space available to accommodate the database snapshot that will be created.
- Make sure you have space available in `tempdb` to allow the DBCC command to run. Note that you can use the `ESTIMATEONLY` option to find out how much `tempdb` space

will be required for DBCC CHECKDB, DBCC CHECKTABLE, DBCC CHECKFILEGROUP, and DBCC CHECKALLOC.

Warning



Use REPAIR_ALLOW_DATA_LOSS only as a last resort.

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 of 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* will have the same values as *model*. You can easily change the value of some of these options by using SQL Server 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 a future version of SQL Server.)

Examining the *sys.databases* catalog view can show you the values of all the options that have been set. The procedure also returns 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	2006-05-27 12:02:35.327	MULTI_USER	ONLINE
model	3	sa	2003-04-08 09:13:36.390	MULTI_USER	ONLINE
msdb	4	sa	2005-10-14 01:54:05.240	MULTI_USER	ONLINE

The *sys.databases* view actually contains both a number and a name for both 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. 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 SQL Server can participate in, including database

mirroring, Service Broker activities, 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 will 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

SINGLE_USER | RESTRICTED_USER | MULTI_USER

OFFLINE | ONLINE | EMERGENCY

READ_ONLY | READ_WRITE

Cursor options

CURSOR_CLOSE_ON_COMMIT { ON | OFF }

CURSOR_DEFAULT { LOCAL | GLOBAL }

Auto options

AUTO_CLOSE { ON | OFF }

AUTO_CREATE_STATISTICS { ON | OFF }

AUTO_SHRINK { ON | OFF }

AUTO_UPDATE_STATISTICS { ON | OFF }

AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }

Parameterization options

DATE_CORRELATION_OPTIMIZATION { ON | OFF }

PARAMETERIZATION { SIMPLE | FORCED }

SQL options

ANSI_NULL_DEFAULT { ON | OFF }

ANSI_NULLS { ON | OFF }

ANSI_PADDING { ON | OFF }

ANSI_WARNINGS { ON | OFF }

ARITHABORT { ON | OFF }

CONCAT_NULL_YIELDS_NULL { ON | OFF }

NUMERIC_ROUNDABORT { ON | OFF }

QUOTED_IDENTIFIER { ON | OFF }

RECURSIVE_TRIGGERS { ON | OFF }

Database recovery options

RECOVERY { FULL | BULK_LOGGED | SIMPLE }

TORN_PAGE_DETECTION { ON | OFF }

PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }

External access options

DB_CHAINING { ON | OFF }

TRUSTWORTHY { ON | OFF }

Database mirroring options

```
PARTNER { = 'partner_server'  
| FAILOVER  
| FORCE_SERVICE_ALLOW_DATA_LOSS  
| OFF  
| RESUME  
| SAFETY { FULL | OFF }  
| SUSPEND  
| TIMEOUT integer  
}  
  
WITNESS { = 'witness_server' |  
OFF  
}
```

Service Broker options

```
ENABLE_BROKER | DISABLE_BROKER  
NEW_BROKER  
ERROR_BROKER_CONVERSATIONS
```

Snapshot Isolation options

```
ALLOW_SNAPSHOT_ISOLATION {ON | OFF }
```

```
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

These three options 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 *AdventureWorks* database to single-user mode, use the following code:

```
ALTER DATABASE AdventureWorks 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 will be 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 these three 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. Any snapshots for the data are automatically dropped. 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 Adventureworks SET OFFLINE;
SELECT state_desc from sys.databases
WHERE name = 'AdventureWorks';
```

A database can be explicitly set to EMERGENCY mode, and I'll explain why you might want to do that after I discuss the database status values that cannot be set.

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 restore process is done on one database at a time, and until SQL Server has finished restoring 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 5](#).

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 look at the `DATABASEPROPERTYEX` values of a suspect database and see its status 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.

Emergency Mode Repair

You can run the `DBCC CHECKDB` command while in EMERGENCY mode, and when you specify the `REPAIR_ALLOW_DATA_LOSS` option, SQL Server can perform some special repairs on the database, which may allow for ordinarily unrecoverable databases to be made physically consistent and brought back online. These repairs should be used as a last resort and only when you cannot restore the database from a backup.

When the database is set to EMERGENCY mode, the database is internally set to `READ_ONLY`, logging is disabled, and access is limited to members of the `sysadmin` role.

However, the properties of the database that you see in `sys.databases` will not reflect these restrictions.

When the database is in emergency mode and DBCC CHECKDB with the REPAIR_ALLOW_DATA_LOSS clause is run, the following actions are taken:

- DBCC CHECKDB uses pages that have been marked inaccessible because of I/O or checksum errors, as if the errors have not occurred in order to increase the chances for data recovery.
- DBCC CHECKDB attempts to recover the database using regular log-based recovery techniques.
- If database recovery is unsuccessful, the transaction log is rebuilt. Rebuilding the transaction log may result in the loss of transactional consistency.

If the DBCC CHECKDB command succeeds, the database is in a physically consistent state and the database status is set to ONLINE. However, the database may contain one or more transactional or logical inconsistencies. You should consider running DBCC CHECKCONSTRAINTS to identify any business logic flaws and immediately back up the database.

If the DBCC CHECKDB command fails, the database cannot be repaired.

In some cases, EMERGENCY mode is not possible, in particular if some of the metadata related to space allocation, which is needed to start up the database, is missing or corrupt.

You can attempt to set a database that is in EMERGENCY mode into ONLINE mode (if the missing files have been made available, for example), and SQL Server will try to run recovery on the database. If the transition to ONLINE cannot be completed, the database will be left in either RECOVERY_PENDING or SUSPECT

status, just like when you first bring up your SQL Server instance and try to recover the database. Once again, you can change the state of the RECOVERY_PENDING database to EMERGENCY mode to allow the data to be read.

It's relatively easy to test emergency status value for a database on a test server. You can create a simple database with the three-word command CREATE DATABASE TESTDB, and then stop your SQL Server instance and rename (or remove) the log file. When you restart your instance, check the status of the new database:

```
SELECT name, database_id, user_access_desc,  
state_desc  
FROM sys.databases  
WHERE name = 'testdb';
```

The state_desc should show RECOVERY_PENDING, which you can now change to EMERGENCY:

```
ALTER DATABASE testdb SET EMERGENCY;
```

The database will now be available for reading data, even though there is no transaction log. If you try to update the database in any way, you'll get the following error:

```
Msg 3908, Level 16, State 1, Line 1  
Could not run BEGIN TRANSACTION in database  
'testdb' because the database is in bypass  
recovery mode.  
The statement has been terminated.
```

If you try to set the database state back to ONLINE, you will get an error indicating that recovery is not possible, and the database will be put back in RECOVERY_PENDING mode. As previously mentioned, running DBCC CHECKDB with the repair option while in EMERGENCY mode can put the database back in ONLINE mode if the database can be repaired.

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 AdventureWorks SET READ_ONLY;
SELECT name, is_read_only FROM sys.databases
WHERE name = 'AdventureWorks';
```

When READ_ONLY is enabled for database, the `is_read_only` column will return 1; otherwise, for a READ_WRITE database, it will return 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 statement. 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 statement to fail if certain connections exist. If the database is being set to SINGLE_USER mode, the ALTER DATABASE statement fails if any other connections exist. If the transition is to RESTRICTED_USER mode, the ALTER DATABASE statement fails if any unqualified connections exist.

The following command changes the user access option of the *AdventureWorks* database to SINGLE_USER and generates an

error if any other connections to the *AdventureWorks* database exist:

```
ALTER DATABASE Adventureworks 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 Transact-SQL commands for defining and manipulating cursors: DECLARE, OPEN, FETCH, CLOSE, and DEALLOCATE. Transact-SQL cursors are discussed in detail in *Inside SQL Server 2005: TSQL Programming*.

- **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 of 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 mode to SIMPLE. The log files shrink at the point that the log is backed up or truncated.

- **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 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.) I'll discuss statistics in more detail in [Chapter 7](#).

The preceding two statistics options, as well as AUTO_UPDATE_STATISTICS_ASYNC and the parameterization options DATE_CORRELATION_OPTIMIZATION and PARAMETERIZATION (all of which are new in SQL Server 2005), will be discussed in more detail in *Inside SQL Server 2005: Query Optimization and Tuning*.

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 SQL Server 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 Transact-SQL rules for identifiers when QUOTED_IDENTIFIER is ON. They can be keywords and can include characters not normally allowed in Transact-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 Transact-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 5](#).

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 2005, 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 will insert a row into the system table *suspect_pages* in the *msdb* database. (Books Online has more information on "Understanding and Managing the *suspect_pages* table.")

SQL Server will retry 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 will be written to the error log and the command that triggered the read will continue. If the attempts fail, the command will fail 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 2005, the default is CHECKSUM. In SQL Server 2000, TORN_PAGE_DETECTION is the default, and CHECKSUM is not available. If you upgrade a database from SQL Server 2000, the PAGE_VERIFY value will be NONE or TORN_PAGE_DETECTION. If it is TORN_PAGE_DETECTION, you should consider changing it to CHECKSUM. Although TORN_PAGE_DETECTION uses fewer resources, it provides less protection than CHECKSUM.

Other Database Options

Of the four other main categories of database options, I will cover only one category in detail—the external access options. The snapshot isolation options will be discussed in [Chapter 8](#). The Service Broker options are discussed in *Inside SQL Server 2005: TSQL Programming*. The database mirroring options are also beyond the scope of this book. Database mirroring is a new SQL Server 2005 technology that provides more options for high

availability and is fully supported as of Service Pack 1. (Microsoft did not fully support database mirroring in the initial RTM release of SQL Server 2005, but mirroring can be enabled in that release by using trace flag 1400 as a startup parameter.) All the details, both internal and external, that you might want to know about database mirroring in SQL Server 2005 can be found in the white paper "Database Mirroring in SQL Server 2005" by Ron Talmage and the Microsoft TechNet article "Database Mirroring Best Practices and Performance Considerations" by Sanjay Mishra, which are both included with the companion content for this book.

Database Snapshots

One of the most interesting new features in SQL Server 2005 is database snapshots, which allows 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 only stores pages that have changed, as we'll see 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. As of this writing, there is no graphical equivalent through Object Explorer, so you must use the Transact-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. No other properties of the files can be specified, and no log file is used.

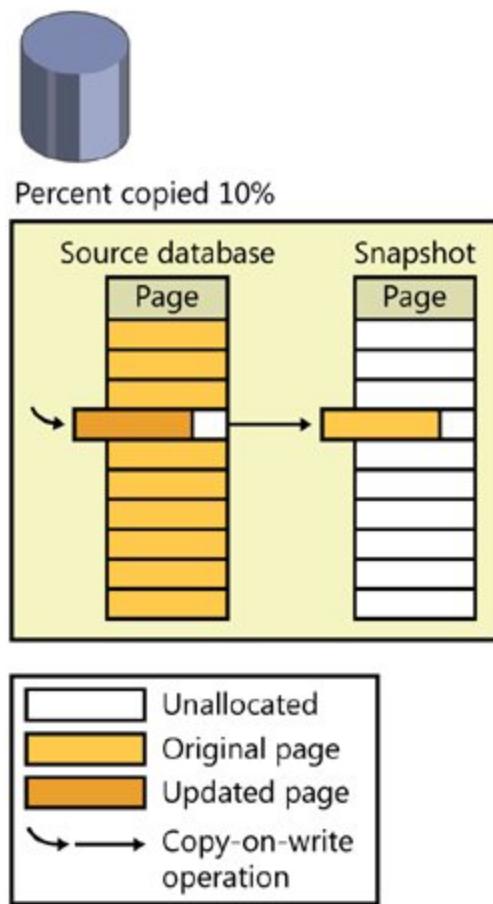
Here is the syntax to create a snapshot of the *AdventureWorks* database, putting the snapshot files in the SQL Server 2005 default data directory:

```
CREATE DATABASE AdventureWorks_snapshot ON
( NAME = N'AdventureWorks_Data',
  FILENAME =
    N'C:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\Data\AW_snapshot.mdf ')
AS SNAPSHOT OF Adventureworks;
```

Each file in the snapshot is created as a sparse file, which is a feature of the NTFS file system. 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 4-2](#) shows a database with a snapshot that contains 10 percent of the data (one page) from the source.

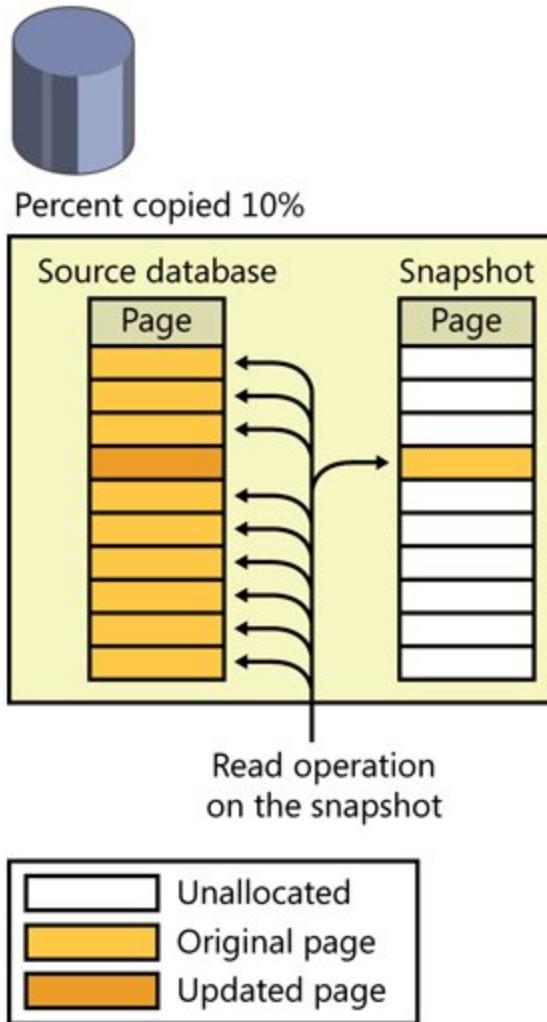
Figure 4-2. A database snapshot that contains one page of data from the source database



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 4-3](#) shows read operations from the same database as in [Figure 4-2](#). 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 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.

**Figure 4-3. Read operations from a database snapshot,
reading changed pages from the snapshot and
unchanged pages from the source database**



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 5](#), 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 will undo uncommitted transactions before the snapshot database is usable, so the snapshot will contain 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.

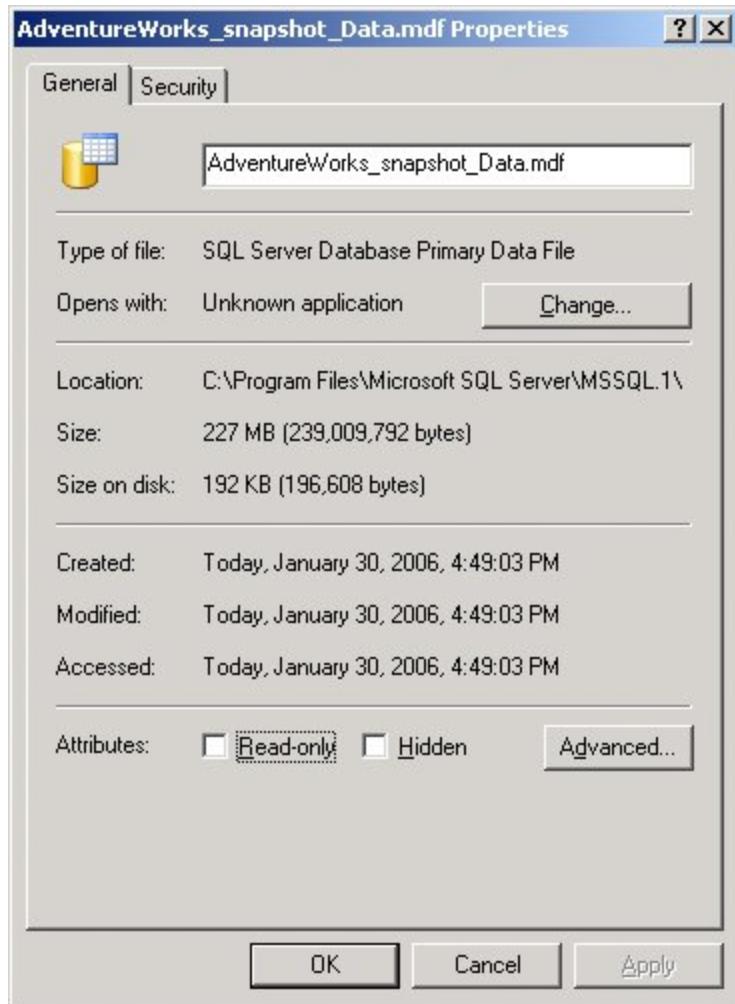
The first error is basically the generic failure message, and the second message provides more details about why the operation

failed.

Space Used by Database Snapshots

You can find out the number of bytes 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. Right-click on the file name and look at the properties, as shown in [Figure 4-4](#). 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 4-4. The snapshot file's Properties dialog box in Windows Explorer shows 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 will start out relatively small, but as the source database is updated, each snapshot will grow. 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 over-commit 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, because you have no control of the placement of the internal snapshot that the command uses it's placed on the same volume that the files of the parent database reside on. If this happens, the DBCC check will fail.) 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. Snapshots will be dropped automatically if you change a database to OFFLINE status. 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 AdventureWOrks  
FROM 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 will try to finish reverting when the database starts up 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 of 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* databases. (Internally, snapshots can be created to run the online DBCC checks on the *master* database, but they cannot be explicitly created.)
- 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, your *tempdb* database is just like any other database, but it has some unique behaviors. They are not all relevant to the topic of this chapter, so I will provide some references to other chapters where you can find additional information.

As mentioned earlier, 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 mode is not copied because *tempdb* always uses simple recovery, which will be discussed in detail in [Chapter 5](#). Certain database options cannot be set for *tempdb*, such as OFFLINE, READONLY, and CHECKSUM. You also cannot drop the *tempdb* database.

In SIMPLE mode, 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) will be gone.

Logging for *tempdb* is also different than for other databases. (Normal logging will be discussed in [Chapter 5](#).) 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 earlier, 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 (or, in fact, most of the DBCC validation commands) in online mode. Another difference with running DBCC in *tempdb* is that SQL Server will skip 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 8](#).)

Objects in *tempdb*

Three types of objects are stored in *tempdb*: user objects, internal objects, and the version store, which is new in SQL Server 2005.

User Objects

All users have the privileges to create and use private and global temporary tables that reside in *tempdb*. (Private and global table names have the # or ## prefix, respectively, which are discussed in *Inside SQL Server 2005: TSQL Programming*.) 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 add such privileges to *model*, from which *tempdb* is copied every time SQL Server is restarted, or you can grant the privileges in an autostart procedure that runs each time SQL Server is restarted. If you choose to add those privileges to the *model* database, you must remember to revoke them on any other new databases that you subsequently create if you don't want them to appear there as well.

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 [Chapter 6](#).

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 first 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 7](#), 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 a new technology in SQL Server 2005 for row-level versioning of data. Older versions of updated rows are kept in *tempdb* in the following situations:

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

Versioning, which is a new concurrency control feature in SQL Server 2005, and snapshot transactions will be discussed in detail in [Chapter 8](#).

Optimizations in *tempdb*

Because *tempdb* is so much more heavily used in SQL Server 2005 than in previous versions, you have to take much more care in managing it. The next section will present some best practices and monitoring suggestions. In this section, I'll 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 2005, allocation pages are accessed much more efficiently to determine where free extents are available; you should see far less contention on the allocation pages than in previous versions. SQL Server 2005 also has a more efficient search algorithm for finding an available single page from mixed extents. When a database has multiple files, SQL Server 2005 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 Index Allocation Map (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 done 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. We'll look at `sys.allocation_units` and the other system views that keep track of space usage in [Chapter 6](#).

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 and SGAM 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 7](#), we'll look at estimating the size of tables and indexes, and we'll talk more about online index building and the `tempdb` space required for that operation. Finally, in [Chapter 8](#), you'll see how to determine the size of the version store. All these factors affect the space needed for your `tempdb`.

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 less of *tempdb*.
- 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 update).

- 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 [Chapter 6](#) and [Chapter 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 file in *tempdb*. It returns the following columns:

- `database_id` (even though the database ID 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 one 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, and vast changes have been made in the security realm for SQL Server 2005, so I can't leave the topic completely untouched here.

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 new concept of schemas in SQL Server 2005 and describe how they are used to access objects.

SQL Server 2005 uses some new terms to describe features of the SQL Server security model, and some old terms are used in slightly new ways. In particular, the following two terms now have a broader meaning than in SQL Server 2000:

- **Securable** Known as an *object* in SQL Server 2000, a *securable* is an entity on which permissions can be granted. Securables include databases, schemas, and objects.
- **Principal** Known as a *user* in SQL Server 2000, a principal is an entity that can access securable objects. A *primary principal* represents a single user (such as a SQL Server 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 2005 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 makes use of *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 security identifier (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 2000 and Windows 2003, SQL Server can use Kerberos to support mutual authentication between the client and

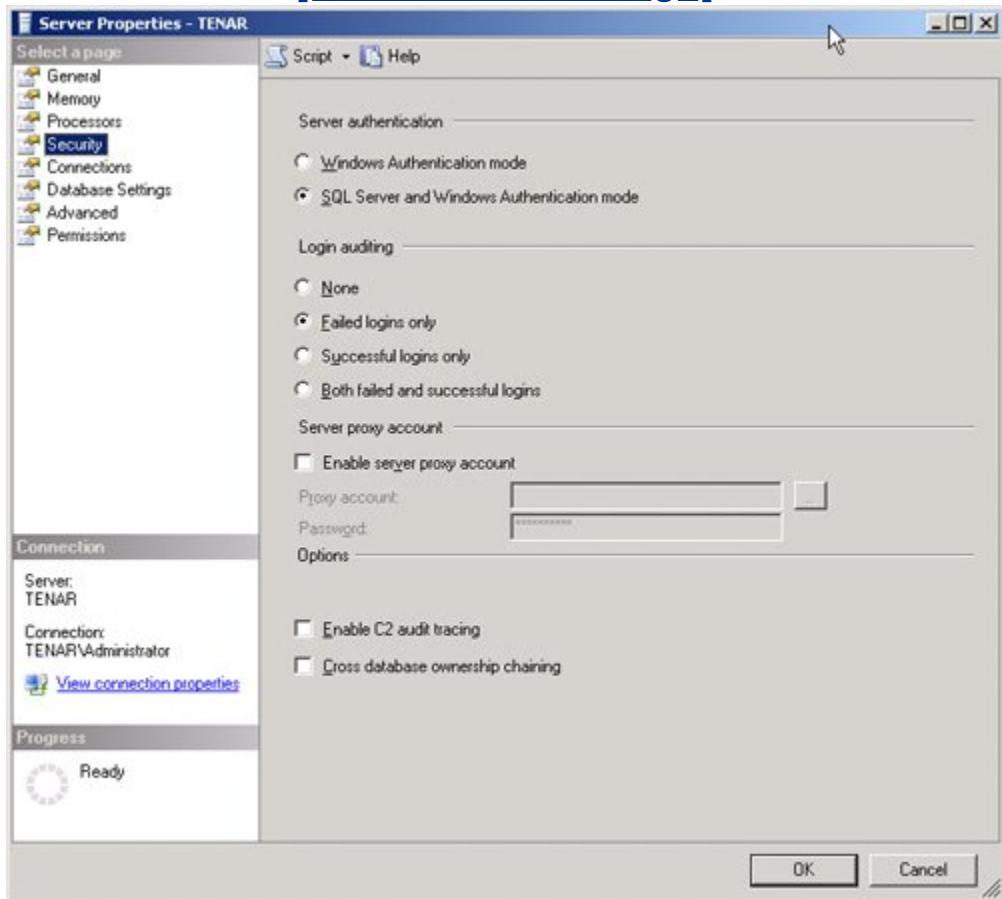
the server, as well as to pass a client's security credentials between computers so that work on a remote server can proceed using the credentials of the impersonated client. With Windows 2000 and Windows 2003, SQL Server uses Kerberos and delegation to support Windows Authentication as well as SQL Server 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. As of SQL Server 2005, when running on Windows 2003, SQL Server Authentication can also take advantage of the Windows password policies. Take a look at the ALTER LOGIN command in Books Online for the full details. Another change in SQL Server 2005 is that if you choose Windows Authentication during setup, the default SQL Server sa login will be 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 SQL Server 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 4-5](#).

Figure 4-5. Choosing an authentication mode for your SQL Server instance in the Server Properties dialog box

[View full size image]



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 explicitly supply a SQL Server login name. 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 SID used by Windows 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.

Note



Not everyone who can log in to SQL Server can see the data in the sys.server_principals view. In SQL Server 2005, metadata is fully secured, and unless you are a very privileged user or have been granted the VIEW DEFINITION permission at the server level, you cannot select from this view.

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 users.

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 `AdventureWorks` 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 `AdventureWorks` database, these are the results I get back:

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 login name `sa` has the user name `dbo`. This name is a special login that is used by the `sa` login, by all logins in the `sysadmin` role, and by whatever login is listed in `sys.databases` as the owner of the database. 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 `AdventureWorks` 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.)

Separation of Principals and Schemas

The previous version, SQL Server 2000, provides a `CREATE SCHEMA` statement, but it effectively does nothing because there is an implicit relationship between users and schemas that cannot be

changed or removed. In fact, the relationship is so close that many users of SQL Server 2000 are unaware that users and schemas are different things. Every user is the owner of a schema that has the same name as the user. If you create a user *sue*, for example, SQL Server 2000 creates a schema called *sue*, which is *sue*'s default schema. Permissions are granted to users, but objects are placed in schemas.

In SQL Server 2000, the statement *GRANT CREATE TABLE TO sue* refers to the user *sue*. Let's say *sue* then creates a table:

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

This table is put in the schema *sue* because that is *sue*'s default schema. 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.

In the new version, SQL Server 2005 breaks apart the linking of users to schemas. Schemas can be owned by either primary or secondary principals. Although every object in a SQL Server 2005 database is owned by a user, we never reference an object by its owner; we reference it by the schema in which it is contained. 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 2005 creates both a user and a schema, 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. When you create a user, you can optionally specify a default schema, but the default for the default schema is the *dbo* schema.

Default Schemas

When you create a new database in SQL Server 2005, several schemas are included in it. These include schemas that correspond to the default user names in SQL Server 2000: *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 predefined database role from SQL Server 2000 has a schema in SQL Server 2005.

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 will try 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 2005 you should always specify the schema name for all object access as well as object management.

Regardless of a user's default schema, SQL Server 2005 always checks the *sys* schema first for any object access. For example, if a user Sue runs the query *select col1 from mytable* and the default

schema for Sue is *SueSchema*, the name resolution process is as follows:

- 1.** Look for sys.table1.
- 2.** Look for SueSchema.table1.
- 3.** Look for dbo.table1.

Note that 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 (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.

We'll look again at schemas and the objects within them in [Chapter 6](#), when we discuss metadata and tables.

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 will not succeed. 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 trace of the

database remains within the SQL Server instance. 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 removes all traces of the database from your instance, but dropping a database is more severe. 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 *sp_attach_db* stored procedure, or you can use the CREATE DATABASE command with the FOR ATTACH option. In SQL Server 2005, the CREATE DATABASE option is the recommended one because it gives you more control over all the files and their placement and because *sp_attach_db* is being deprecated. With *sp_attach_db*, the limit is 16 files. CREATE DATABASE has no such limit in fact, you can specify up to 32,767 files and 32,767 file groups for each database.

```
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 will use 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 will break the log backup chain. You

should consider making a full backup after performing this 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 will therefore require less log space than the original database.

Although the documentation says that you should use sp_attach_db or 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 will be closed, just as if you had detached the database. However, you will not be 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 will have a record of all completed transactions, and a full recovery will be done 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 will know that the database was shut down cleanly, and the log file does not have to be available to attach the database. SQL Server will build 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 will be the minimum sizeless 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 in [Chapter 5](#). 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. The steps for moving *tempdb*, *model*, and *msdb* are slightly different than for moving the *master* database or the resource database.

Here are the steps for moving an undamaged system database (that is not the *master* database or the resource 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 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 SQL Server Management Studio or the SQLCMD utility.
4. Exit SQL Server 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. 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 and Resource Database

The location of the resource database (which is actually named *mssql/systemresource*) depends on the location of the master database. If you move the *master* database, you must move the resource database files to the same directory.

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

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

- 1.** Open the SQL Server Configuration Manager. Right-click on the desired instance of SQL Server, choose Properties, and then click on the Advanced tab.
- 2.** Edit the Startup Parameters values to point to the new directory location for the *master* database data and log files. You can optionally choose to 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;-
eC:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\LOG\ERRORLOG;
-lE:\SQLData\mastlog.ldf;
-eE:\ SQLData\LOG\ERRORLOG; -
```

3. Stop the instance of SQL Server and physically move the files for *master* and *mssqlsystemresource* to the new location.
4. Start the instance of SQL Server in *master-only* recovery mode by using the /f and / T3608 flags, as shown previously.
5. Using SQLCMD commands or SQL Server Management Studio, use ALTER DATABASE to change the FILENAME path for the *mssqlsystemresource* database to match the new location of the *master* data file. Do not change the name of the database or the file names.

```
ALTER DATABASE mssqlsystemresource MODIFY FILE
(NAME=data, FILENAME=
'new_path_of_master\mssqlsystemresource.mdf');
ALTER DATABASE mssqlsystemresource MODIFY FILE
(NAME=log, FILENAME=
'new_path_of_master\mssqlsystemresource.ldf');
```

6. Set the *mssqlsystemresource* database to read-only, and stop the instance of SQL Server.

7. Move the resource database's data and log files to the new location.
8. Restart the instance of SQL Server.
9. Verify the file change for the *master* database by running the following query. Note that you cannot view the *resource* database metadata by using the system catalog views or system tables.

```
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 tremendous amount of new functionality, much of which requires new keywords and also changes certain behaviors that existed in earlier versions. To provide maximum backward compatibility, Microsoft allows you to set the compatibility level of a database to one of the following modes: 90, 80, 70, 65, or 60. Compatibility levels 65 and 60 are being deprecated, and 60 is not supported by SQL Server Management Studio or SMO. All newly created databases in SQL Server 2005 have a compatibility level of 90 unless you change the level for the *model* database. A database that has been upgraded or attached from an older version will have 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 90 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 90 compatibility mode by executing this procedure:

```
EXEC sp_dbcmptlevel '<database name>';
```

To change to a different compatibility level, run the procedure using a second argument of one of the possible modes:

```
EXEC sp_dbcmptlevel '<database name>',  
<compatibility-level>;
```

Note



The compatibility-level options merely provide a transition period while you're upgrading a database or an application to SQL Server 2005. 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 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 PIVOT is a new reserved keyword in SQL Server 2005 (compatibility level 90), by setting your compatibility level to 80 you can create a table called PIVOT without using any special delimiter or a table you already have in a SQL Server 2000 database will continue to be accessible if the database stays in 80 compatibility level.

For a complete list of the behavioral differences between the compatibility levels and the new keywords, see the online documentation for the *sp_dbcmptlevel* procedure.

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.) 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 easily increase and decrease the size of databases and their files either manually or automatically.

Chapter 5. Logging and Recovery

In this chapter:

<u>Transaction Log Basics</u>	<u>149</u>
<u>Changes in Log Size</u>	<u>154</u>
<u>Reading the Log</u>	<u>162</u>
<u>Backing Up and Restoring a Database</u>	<u>162</u>
<u>Summary</u>	<u>174</u>

In the previous chapter, I told you about the data files that are created to hold information in a Microsoft SQL Server database. Every database also has at least one file that stores its transaction log. I made reference to SQL Server transaction logs and log files in [Chapter 4](#), but I did not really go into detail about how a log file is different from a data file and exactly how SQL Server uses its log files. In this chapter, I'll tell you about the structure of SQL Server log files and how they're managed when transaction information is logged. I'll explain how SQL Server log files grow and when and how a log file can be reduced in size. Finally, we'll look at how log

files are used during SQL Server backup and restore operations and how they are affected by your database's recovery model.

Transaction Log Basics

The transaction log records changes made to the database and stores enough information to allow SQL Server to recover the database. As we'll see later in this chapter, the recovery process takes place every time a SQL Server instance is started, and it can optionally take place every time SQL Server restores a database or a log from backup. *Recovery* is the processing of reconciling the data files and the log files. Any changes to the data that the log indicates have been committed must appear in the data files, and any changes that are not marked as committed must not appear in the data files. The log also stores information needed to roll back an operation if SQL Server receives a request to roll back a transaction from the client (using the ROLLBACK TRAN command) or if an error, such as a deadlock, generates an internal ROLLBACK.

Physically, the transaction log is one or more files associated with a database at the time the database is created or altered. Operations that perform database modifications write records in the transaction log that describe the changes made (including the page numbers of the data pages modified by the operation), the data values that were added or removed, information about the transaction that the modification was part of, and the date and time of the beginning and end of the transaction. SQL Server also writes log records when certain internal events happen, such as checkpoints. Each log record is labeled with a log sequence number (LSN) that is guaranteed to be unique. All log entries that are part of the same transaction are linked together so that all parts of a transaction can be easily located for both undo activities (as with a rollback) and redo activities (during system recovery).

The Buffer Manager guarantees that the transaction log will be written before the changes to the database are written. (This is called *write-ahead logging*.) This guarantee is possible because

SQL Server keeps track of its current position in the log by means of the LSN. Every time a page is changed, the LSN corresponding to the log entry for that change is written into the header of the data page. Dirty pages can be written to disk only when the LSN on the page is less than or equal to the LSN for the last page written to the log. The Buffer Manager also guarantees that log pages are written in a specific order, making it clear which log blocks must be processed after a system failure, regardless of when the failure occurred.

The log records for a transaction are written to disk before the commit acknowledgement is sent to the client process, but the actual changed data might not have been physically written out to the data pages. Although the writes to the log are asynchronous, at commit time the thread must wait for the writes to complete to the point of writing the commit record in the log for the transaction. (SQL Server must wait for the commit record to be written so that it knows the relevant log records are safely on disk.) Writes to data pages are completely asynchronous. That is, writes to data pages need only be posted to the operating system, and SQL Server can check later to see that they were completed. They don't have to be completed immediately because the log contains all the information needed to redo the work, even in the event of a power failure or system crash before the write completes. The system would be much slower if it had to wait for every I/O request to complete before proceeding.

Logging involves demarcating the beginning and end of each transaction (and savepoints, if a transaction uses them). Between the beginning and ending demarcations is information about the changes made to the data. This information can take the form of the actual "before and after" data, or it can refer to the operation that was performed so that those values can be derived. The end of a typical transaction is marked with a Commit record, which indicates that the transaction must be reflected in the database's data files or redone if necessary. A transaction aborted during normal runtime

(not system restart) due to an explicit rollback or something like a resource error (for example, an out-of-memory error) actually undoes the operation by applying changes that undo the original data modifications. The records of these changes are written to the log and marked as "compensation log records."

As mentioned earlier, there are two types of recovery, and both have the goal of making sure the log and the data are in agreement. A *restart recovery* runs every time SQL Server is started. The process runs on each database because each database has its own transaction log. Your SQL Server error log will report the progress of restart recovery, and for each database the log will tell you how many transactions were rolled forward and how many were rolled back. This type of recovery is sometimes referred to as "crash" recovery because a crash, or unexpected stopping of the SQL Server service, requires the recovery process to be run when the service is restarted. If the service was shut down cleanly with no open transactions in any database, only minimal recovery is necessary upon system restart. In SQL Server 2005, restart recovery can be run on multiple databases in parallel, each handled by a different thread.

The other type of recovery, *restore recovery* (or *media recovery*), is run by request when a restore operation is executed. This process makes sure that all the committed transactions in the backup of the transaction log are reflected in the data and that any transactions that did not complete do not show up in the data. I'll talk more about restore recovery later in the chapter.

Both types of recovery must deal with two situations: when there are transactions recorded as committed in the log but not yet written to the data files and when there are changes to the data files that don't correspond to committed transactions. These two situations can occur because committed log records are written to the log files on disk every time a transaction commits. Changed data pages are written to the data files on disk completely asynchronously, every

time a checkpoint occurs in a database. As I mentioned in [Chapter 2](#), data pages can also be written to disk at other times, but the regularly occurring checkpoint operations give SQL Server a point at which *all* changed (or dirty) pages are known to have been written to disk. Checkpoint operations also write log records from transactions in progress to disk because the cached log records are also considered to be dirty.

If the SQL Server service is stopped after a transaction commits but before the data is written out to the data pages, when SQL Server starts up and runs recovery, the transaction must be rolled forward. SQL Server essentially redoing the transaction by reapplying the changes indicated in the transaction log. All the transactions that need to be redone are processed first (even though some of them might need to be undone later during the next phase). This is called the REDO phase of recovery.

If a checkpoint occurs before a transaction is committed, it will write the uncommitted changes out to disk. If the SQL Server service is then stopped before the commit occurs, the recovery process will find the changes for the uncommitted transactions in the data files, and it will have to roll back the transaction by undoing the changes reflected in the transaction log. Rolling back all the incomplete transactions is called the UNDO phase of recovery.

I'll continue to refer to recovery as a system startup function, which is its most common role by far. However, remember that recovery is also run during the final step of restoring a database from backup and can also be forced manually.

Later in this chapter, I'll cover some special issues related to recovery during a database restore. These include the three recovery modes that you can set using the ALTER DATABASE statement and the ability to place a named marker in the log to indicate a specific point to recover to. The discussion that follows deals with recovery in general, whether it's performed when the

SQL Server service is restarted or when a database is being restored from a backup.

Phases of Recovery

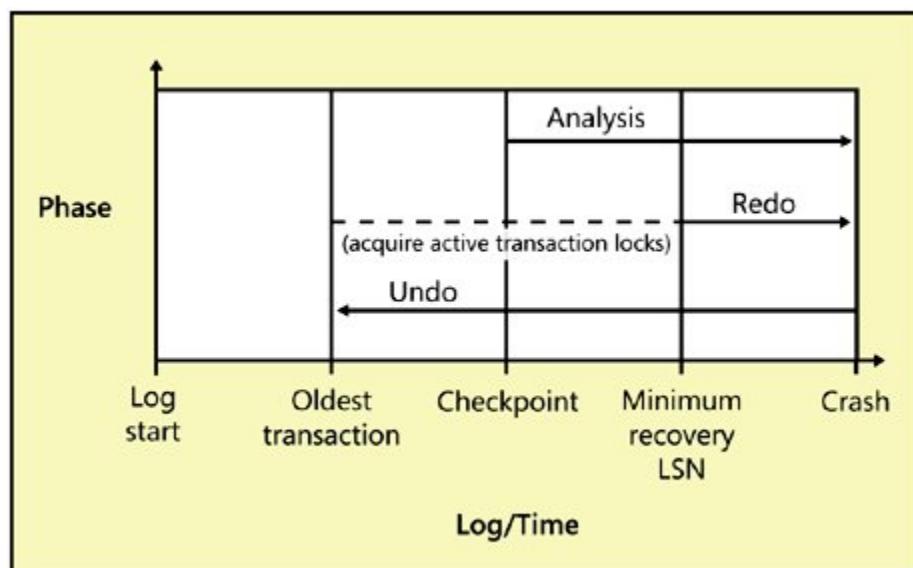
During recovery, only changes that occurred or were in progress since the last checkpoint are redone or undone. Any transactions that completed prior to the last checkpoint, either by being committed or rolled back, will be accurately reflected in the data pages, and no additional work will need to be done for them during recovery.

The recovery algorithm has three phases, which center around the last checkpoint record in the transaction log. The three phases are illustrated in [Figure 5-1](#).

- **Phase 1: Analysis** The first phase is a forward pass starting at the last checkpoint record in the transaction log. This pass determines and constructs a dirty page table (DPT) consisting of pages that might have been dirty at the time SQL Server stopped. An active transaction table is also built that consists of uncommitted transactions at the time SQL Server stops.
- **Phase 2: Redo** This phase returns the database to the state it was in at the time the SQL Server service stopped. The starting point for this forward pass is the start of the oldest uncommitted transaction. The minimum LSN in the DPT is the first time SQL Server expects to have to REDO an operation on a page, but it needs to REDO log starting all the way back at the start of the oldest open transaction so that the necessary locks can be acquired. (Prior to SQL Server 2005, it was just allocation locks that needed to be reacquired. In SQL 2005, all locks for those open transactions need to be reacquired.)

- **Phase 3: Undo** This phase uses the list of active transactions (uncommitted at the time SQL Server came down) which were found in the Phase 1 (Analysis). It rolls each of these active transactions back individually. SQL Server follows the links between entries in the transaction log for each transaction. Any transaction that was not committed at the time SQL Server stopped is undone so that none of the changes are actually reflected in the database.

Figure 5-1. The three phases of the SQL Server recovery process



SQL Server 2005 uses the log to keep track of the data modifications that were made, as well as any locks that were applied to the objects being modified. This allows SQL Server 2005 to support a feature called *fast recovery* when SQL Server is restarted (in the Enterprise and Developer editions only). With fast recovery, the database is available as soon as the redo phase is finished. The same locks that were acquired during the original modification can be reacquired to keep other processes from accessing the data that needs to have its changes undone; all other data in the database remains available. Fast recovery cannot be done during media recovery.

In addition, SQL Server 2005 uses multiple threads to process the recovery operations on the different databases, so databases with higher ID numbers don't have to wait for all databases with lower ID numbers to be completely recovered before their own recovery process starts.

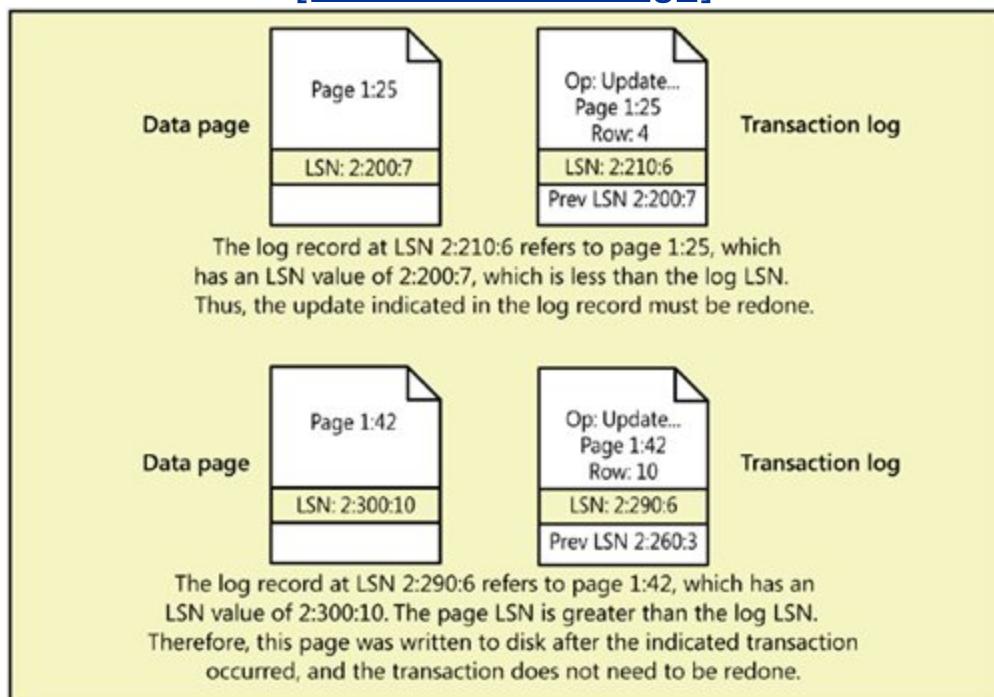
Page LSNs and Recovery

Every database page has an LSN in the page header that reflects the location in the transaction log of the last log entry that modified a row on this page. Each log record for changes to a data page has two LSNs associated with it. In addition to the LSN for the actual log record, it also keeps track of the LSN which was on the data page before the change recorded by this log record. During a redo operation of transactions, the LSNs on each log record are compared to the page LSN of the data page that the log entry modified. If the page LSN is equal to the previous page LSN in the log record, the operation indicated in the log entry is redone. If the LSN on the page is equal to or higher than the actual LSN for this log record, SQL Server will skip the REDO operation. These two possibilities are illustrated in [Figure 5-2](#). The LSN on the page

cannot be in between the previous and current value for the log record.

Figure 5-2. Comparing LSNs to decide whether to process the log entry during recovery

[[View full size image](#)]



Because recovery finds the last checkpoint record in the log (plus transactions that were still active at the time of the checkpoint) and proceeds from there, recovery time is short, and all changes committed before the checkpoint can be purged from the log or archived. Otherwise, recovery could take a long time and transaction logs could become unreasonably large. A transaction log cannot be truncated prior to the point of the earliest transaction that is still open, no matter how many checkpoints have occurred since the transaction started and no matter how many other transactions have started or completed. If a transaction remains open, the log must be preserved because it's still not clear whether the transaction is done or ever will be done. The transaction might ultimately need to be rolled back or rolled forward.

Note



Truncating of the transaction log is a logical operation and merely marks parts of the log as no longer needed, so the space can be reused. Truncation is not a physical operation and does not reduce the size of the transaction log files on disk. To reduce the physical size, a shrink operation must be performed.

Some SQL Server administrators have noted that the transaction log seems unable to be truncated, even after the log has been backed up. This problem often results from a process opening a transaction and then forgetting about it. For this reason, from an application development standpoint, you should ensure that

transactions are kept short. Another possible reason for an inability to truncate the log relates to a table being replicated using transactional replication when the replication log reader hasn't processed all the relevant log records yet. This situation is less common, however, because typically a latency of only a few seconds occurs while the log reader does its work. You can use DBCC OPENTRAN to look for the earliest open transaction or the oldest replicated transaction not yet processed and then take corrective measures (such as killing the offending process or running the sp_repldone stored procedure to allow the replicated transactions to be purged). I'll discuss problems with transaction management and some possible solutions in [Chapter 8](#). I'll discuss shrinking of the log in the next section.

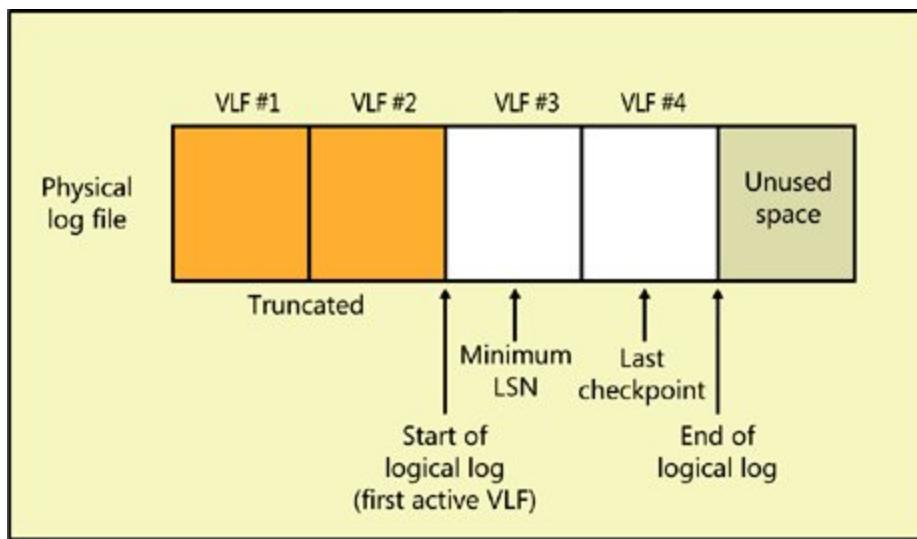
Changes in Log Size

No matter how many physical files have been defined for the transaction log, SQL Server always treats the log as one contiguous stream. For example, when the DBCC SHRINKDATABASE command (discussed in [Chapter 4](#)) determines how much the log can be shrunk, it does not consider the log files separately but instead determines the shrinkable size based on the entire log.

Virtual Log Files

The transaction log for any database is managed as a set of virtual log files (VLFs) whose size is determined internally by SQL Server based on the total size of all the log files and the growth increment used when enlarging the log. When a log file is first created, it always has between 4 and 16 VLFs. A log always grows in units of entire VLFs and can be shrunk only to a VLF boundary. ([Figure 5-3](#) illustrates a physical log file along with several VLFs.)

Figure 5-3. Multiple VLFs that make up a physical log file



A VLF can be in one of four states:

- **Active** The active portion of the log begins at the minimum LSN representing an active (uncommitted) transaction. The active portion of the log ends at the last LSN written. Any VLFs that contain any part of the active log are considered active VLFs. (Unused space in the physical log is not part of any VLF.)
[Figure 5-3](#) contains 04 active VLFs.
- **Recoverable** The portion of the log preceding the oldest active transaction is needed only to maintain a sequence of log backups for restoring the database to a former state.
- **Reusable** If transaction log backups are not being maintained or if you have already backed up the log, VLFs before the oldest active transaction are not needed and can be reused.

Truncating or backing up the transaction log will change recoverable VLFs into reusable VLFs.

- **Unused** One or more VLFs at the physical end of the log files might not have been used yet if not enough logged activity has taken place or if earlier VLFs have been marked as reusable and then reused.

Observing Virtual Log Files

You can observe the same key properties of virtual log files by executing the undocumented command DBCC LOGINFO. This command takes no parameters, so it must be run in the database for which you want information. It returns one row for each VLF. When I run this command in a newly created *pubs* database, I get the following three rows returned:

FileId	FileSize	StartOffset	FSeqNo	Status
2	253952	8192	18	0
2	253952	262144	19	0
2	270336	516096	20	2

The number of rows tells me how many VLFs are in my database. The results actually include two additional columns, but I have

omitted them here because I will not be discussing them. The FileID column indicates which of the log's physical files contains the VLF; for my *pubs* database, there is only one physical log file. *FileSize* and *StartOffset* are indicated in bytes, so you can see that the first VLF starts after 8192 bytes, which is the number of bytes in a page. The first physical page of a log file contains header information and not log records, so the VLF is considered to start on the second page. The *FileSize* column is actually redundant for most rows because the size value can be computed by subtracting the *StartOffset* values for two successive VLFs. The rows are listed in physical order, but that is not always the order in which the VLFs have been used. The use order (logical order) is reflected in the column called FSeqNo (which stands for File Sequence Number). For example, if I run the DBCC LOGINFO command in my *AdventureWorks* database, the following 10 rows are returned:

FileId	FileSize	StartOffset	FSeqNo	Status
2	458752	8192	4994	0
2	458752	466944	4995	0
2	458752	925696	4996	0
2	712704	1384448	4997	0
2	4194304	2097152	4998	2
2	4194304	6291456	4993	0

2	4194304	10485760	4989	0
2	4194304	14680064	4990	0
2	4194304	18874368	4991	0
2	4194304	23068672	4992	0

Again, you can see that the rows are listed in physical order according to the *StartOffset*, but the logical order does not match. The *FSeqNo* values indicate that the seventh VLF is actually the first one in use (logical) order; the last one in use order is the fifth VLF in physical order. The status column indicates whether the VLF is reusable. A status of 2 means that it is either active or recoverable; a status of 0 indicates that it is reusable or completely unused. As I mentioned earlier, truncating or backing up the transaction log changes recoverable VLFs into reusable VLFs, so a status of 2 will change to a status of 0 for all VLFs that don't include active log records. In fact, that's one way to tell which VLFs are active: the VLFs that still have a status of 2 after a log backup or truncation must contain records from active transactions. VLFs with a status of 0 can be reused for new log records, and the log will not need to grow to keep track of the activity in the database. On the other hand, if all the VLFs in the log have a status of 2, SQL Server will need to add new VLFs to the log to record new transaction activity.

Multiple Log Files

I mentioned earlier that SQL Server treats multiple physical log files as if they were one sequential stream. This means that all the VLFs in one physical file are used before any VLFs in the second file are used. If you have a well-managed log that is regularly backed up or truncated, you might never use any log files other than the first one. If one of the VLFs in multiple physical log files is available for reuse when a new VLF is needed, SQL Server will add new VLFs to each physical log file in a round-robin fashion. There is really no reason to need multiple physical log files if you have done thorough testing and have determined the optimal size of your database's transaction log. However, if you find that the log needs to grow more than expected (if the volume containing the log does not have sufficient free space to allow the log to grow enough), you might need to create a second log file on another volume.

Automatic Truncation of Virtual Log Files

SQL Server will assume you're not maintaining a sequence of log backups if any of the following is true:

- You have truncated the log using BACKUP LOG WITH NO_LOG or BACKUP LOG WITH TRUNCATE_ONLY.
- You have set the database to truncate the log on a regular basis by setting the recovery model to SIMPLE.
- You have never taken a full database backup.

Under any of these circumstances, SQL Server will truncate the database's transaction log every time it gets "full enough." (I'll explain this in a moment.) The database is considered to be in autotruncate mode.

Remember that truncation means that all log records prior to the oldest active transaction are invalidated, and all VLFs not containing any part of the active log are marked as reusable. It does not imply shrinking of the physical log file. In addition, if your database is a publisher in a replication scenario, the oldest open transaction could be a transaction marked for replication that has not yet been replicated.

"Full enough" means that there are more log records than can be redone during system startup in a reasonable amount of time—the *recovery interval*. You can manually change the recovery interval by using the `sp_configure` stored procedure or by using SQL Server Management Studio, as discussed in [Chapter 3](#). However, it is best to let SQL Server autotune this value. In most cases, this recovery interval value is set to 1 minute. By default, `sp_configure` shows zero minutes by default, meaning SQL Server will autotune the value. SQL Server bases its recovery interval on the estimate that 10 MB worth of transactions can be recovered in 1 minute.

The actual log truncation is invoked by the checkpoint process, which is usually sleeping and is woken up only on demand. Each time a user thread calls the log manager, the log manager checks the size of the log. If the size exceeds the amount of work that can be recovered during the recovery interval, the checkpoint thread is woken up. The checkpoint thread checkpoints the database and then truncates the inactive portion.

In addition, if the log ever gets to 70 percent full, the log manager wakes up the checkpoint thread to force a checkpoint. Growing the log is much more expensive than truncating it, so SQL Server truncates the log whenever it can.

Note



If the log manager is never needed, the

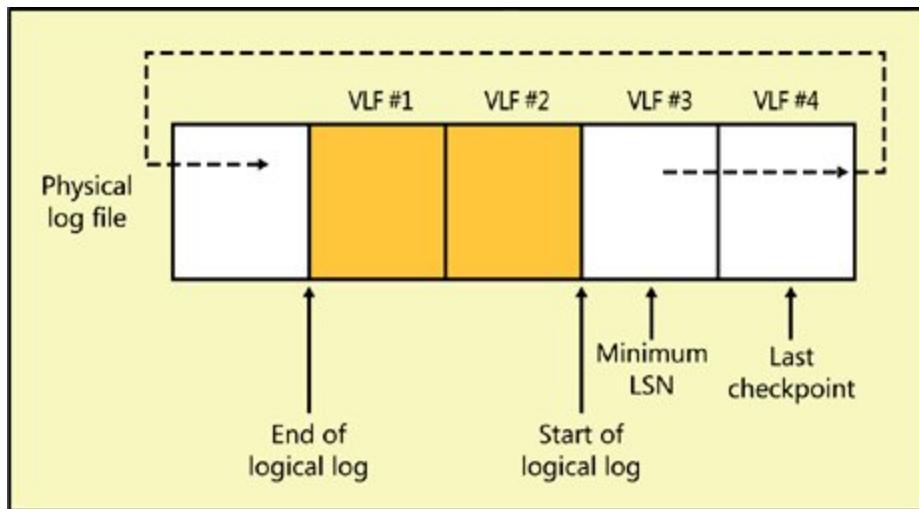
checkpoint process won't be invoked and the truncation will never happen. If you have a database in autotruncate mode, for which the transaction log has VLFs with a status of 2, you will not see the status change to 0 until some logging activity is required in the database.

If the log is regularly truncated, SQL Server can reuse space in the physical file by cycling back to an earlier VLF when it reaches the end of the physical log file. In effect, SQL Server recycles the space in the log file that is no longer needed for recovery or backup purposes. My *AdventureWorks* database is in this state because I have never taken a full database backup.

Maintaining a Recoverable Log

If a log backup sequence *is* being maintained, the part of the log before the minimum LSN cannot be overwritten until those log records have actually been backed up. The VLF status will stay at 2 until the log backup occurs. After the log backup, the status will change to 0 and SQL Server can cycle back to the beginning of the file. [Figure 5-4](#) depicts this cycle in a simplified fashion. As you can see from the *FSeqNo* values in the output from the earlier *AdventureWorks* database, SQL Server does not always reuse the log files in order of their physical sequence.

Figure 5-4. The active portion of the log circling back to the beginning of the physical log file



Note



If a database is not in autotruncate mode and you are not performing regular log backups, your transaction log will never be truncated. If you are doing only full database backups, you must manually truncate the log to keep it at a manageable size.

The easiest way to tell whether a database is in autotruncate mode is by using the catalog view called `sys.database_recovery_status` and looking in the column called `last_log_backup_lsn`. If that column has a value of 0, the database is in autotruncate mode.

You can actually observe the difference between a database in autotruncate mode and a database that isn't by running a simple script in the `pubs` database. This script will work as long as you have never made a full backup of the `pubs` database. If you have never made any modifications to `pubs`, the size of its transaction log file will be just about 0.75 MB, which is the size at creation. The following script creates a new table in the `pubs` database, inserts three records, and then updates those records 1000 times. Each update is an individual transaction, and each one is written to the transaction log. However, you should note that the log does not grow at all, and the number of VLFs does not increase even after 3000 update records are written. (If you've already taken a backup of `pubs`, you might want to re-create the database before trying this example. You can do that by running the script `instpubs.sql` from the companion CD, which I mentioned in [Chapter 4](#).) However, even though the number of VLFs does not change, you will see that the `FSeqNo` values changes. Log records are being generated, and as each VLF is reused, it gets a new `FSeqNo` value.

```
USE pubs
-- First look at the VLFs for the pubs database
DBCC LOGININFO
-- Now verify that pubs is in auto truncate mode
SELECT last_log_backup_lsn
FROM master.sys.database_recovery_status
WHERE database_id = db_id('pubs')
GO
CREATE TABLE newtable (a int)
GO
INSERT INTO newtable VALUES (10)
INSERT INTO newtable VALUES (20)
```

```
INSERT INTO newtable VALUES (30)
GO
DECLARE @counter int
SET @counter = 1
WHILE @counter < 1000 BEGIN
    UPDATE newtable SET a = a + 1
    SET @counter = @counter + 1
END
```

Now make a backup of the *pubs* database after making sure that the database is not in the SIMPLE recovery mode. I'll discuss recovery models later in this chapter, but for now you can just make sure that *pubs* is in the appropriate recovery mode by executing the following command:

```
ALTER DATABASE pubs SET RECOVERY FULL
```

You can use the following statement to make the backup, substituting the path shown with the path to your SQL Server installation:

```
BACKUP DATABASE pubs to disk =
    'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\backup\pubs.bak'
```

As soon as you take the full backup, you can verify that the database is not in autotruncate mode, again by looking at the *database_recovery_status* view:

```
SELECT last_log_backup_lsn  
FROM master.sys.database_recovery_status  
WHERE database_id = db_id('pubs')
```

This time, you should get a non-null value for *last_log_backup_lsn* to indicate that log backups are expected. Next, run the update script again, starting with the DECLARE statement. You should see that the physical log file has grown to accommodate the log records added and that there are more VLFs. The initial space in the log could not be reused because SQL Server assumed that you were saving that information for transaction log backups.

Now you can try to shrink the log back down again. If you issue the following command or if you issue the DBCC SHRINKFILE command for the log file, SQL Server will mark a shrinkpoint in the log, but no actual shrinking will take place until log records are freed by either backing up or truncating the log.

```
DBCC SHRINKDATABASE (pubs)
```

You can truncate the log in one of two ways. One way is to change the database's recovery mode to SIMPLE, as we'll see in the next section. Alternatively, you can run this command:

```
BACKUP LOG pubs WITH TRUNCATE_ONLY
```

At this point, you should notice that the physical size of the log file has been reduced. If a log is truncated without any prior shrink command issued, SQL Server marks the space used by the

truncated records as available for reuse but does not change the size of the physical file.

In SQL Server 7.0, where this log architecture was first introduced, running the preceding commands exactly as specified did not always shrink the physical log file. When the log file did not shrink, it was because the active part of the log was located at the end of the physical file. Physical shrinking can take place only from the end of the log, and the active portion is never shrinkable. To remedy this situation, you had to enter some dummy transactions after truncating the log to force the active part of the log to move around to the beginning of the file. In SQL Server 2000 and SQL Server 2005, this process is unnecessary. If a shrink command has already been issued, truncating the log internally generates a series of NO-OP (or dummy) log records that force the active log to move from the physical end of the file. Shrinking happens as soon as the log is no longer needed.

Automatic Shrinking of the Log

Remember, truncating is not shrinking. A database should be truncated so that it is most shrinkable, and if the log is in autotruncate mode and the autoshrink option is set, the log will be physically shrunk at regular intervals.

If a database has the autoshrink option on, an autoshrink process kicks in every 30 minutes (as discussed in [Chapter 4](#)) and determines the size to which the log should be shrunk. The log manager accumulates statistics on the maximum amount of log space used in the 30-minute interval between shrinks. The autoshrink process marks the shrinkpoint of the log as 125 percent of the maximum log space used or the minimum size of the log, whichever is larger. (Minimum size is the creation size of the log or the size to which it has been manually increased or decreased.) The

log then shrinks to that size whenever it gets the chance, which is when it gets truncated or backed up. It's possible to have autoshrink without having the database in autotruncate mode, although there's no way to guarantee that the log will actually shrink. For example, if the log is never backed up, none of the VLFs will be marked as reusable, so no shrinking can take place.

As a final note, you need to be aware that just because a database is in autotruncate mode, there is no guarantee that it won't grow. (It is the converse that you can be sure of that a database not in autotruncate mode *will* grow.) Autotruncate means only that VLFs that are considered recoverable will be marked as reusable at regular intervals. But VLFs in an active state will not be affected. If you have a long-running transaction (which might be a transaction that someone forgot to commit), all the VLFs that contain any log records since that long-running transaction started are considered active and can never be reused. One uncommitted transaction can mean the difference between a very manageable transaction log size and a log that uses more disk space than the database itself and continues to grow.

Log File Size

You can see the current size of the log file for all databases, as well as the percentage of the log file space that has been used, by running the DBCC command DBCC SQLPERF('logspace').

However, because it is a DBCC command, it's hard to filter the rows to get just the rows for a single database. In addition, if you want to create a stored procedure to return the log size, it can be tricky to manage the permissions mechanisms to allow someone who normally doesn't have permission to run the DBCC command. Instead, you can use the catalog view `sys.master_files`, which returns rows for all files in all databases or from one particular database, or you can use the view `sys.database_files`, which returns

the rows from `sys.master_files` for the current database. If you filter on `type = 1`, you'll get just the rows for the log files.

The following code, supplied by Cliff Dibble from Microsoft, returns the same information as `DBCC SQLPERF('logspace')` in a tabular format that can be further filtered or easily embedded into a stored procedure, table-valued function, or view:

```
SELECT rtrim(pc1.instance_name) AS [Database Name]
      , pc1.cntr_value/1024.0 AS [Log Size (MB)]
      , cast(pc2.cntr_value*100.0/pc1.cntr_value
as dec(5,2)) as [Log Space Used (%)]
  FROM sys.dm_os_performance_counters as pc1
  JOIN sys.dm_os_performance_counters as pc2
    ON pc1.instance_name = pc2.instance_name
   WHERE pc1.object_name LIKE '%Databases%'
     AND pc2.object_name LIKE '%Databases%'
     AND pc1.counter_name = 'Log File(s) Size
(KB)'
     AND pc2.counter_name = 'Log File(s) Used
Size (KB)'
     AND pc1.instance_name not in ('_Total',
'mssqlsystemresource')
     AND pc1.cntr_value > 0
go
```

The final condition is needed to filter out databases that have no log file size reported. This includes any database that is unavailable because it has not been recovered or is in a suspect state, as well as any database snapshots, which have no transaction log.

Reading the Log

Although the log contains a record of every change made to a database, it is not intended to be used as an auditing tool. The transaction log is used to enable SQL Server to guarantee recoverability in case of statement or system failure and to allow a system administrator to take backups of the changes to a SQL Server database. If you want to keep a readable record of changes to a database, you have to do your own auditing. You can do this by creating a trace of SQL Server activities, using SQL Server Profiler or the server-side trace capability.

You might be aware that some third-party tools can read the transaction log and show you all the operations that have taken place in a database and can allow you to roll back any of those operations. One such tool is Lumigent's Log Explorer. (You can download a trial version from the Lumigent Web site at <http://www.lumigent.com/downloads/>.) The developers of the original Log Explorer product, called Log Analyzer, did have access to an undocumented DBCC command for reading transaction log records, but even that was not enough information. They spent tens of thousands of hours looking at byte-level dumps of the transaction log files and correlating that information with the output of the undocumented DBCC command. Once they had a product on the market, Microsoft started working with them, which made their lives a bit easier in subsequent releases. The Lumigent engineers now spend several weeks in the labs at Microsoft before each new release and are given access to quite a bit of very low-level internal information about the structure of the log records. In return, Lumigent develops an internal document that other third-party vendors can potentially access.

Although you might assume that reading the transaction log directly would be interesting or even useful, it's just too much information. If

you know in advance that you want to keep track of what your SQL Server is doing, you're much better off defining a trace with the appropriate filter to capture just the information that is useful to you. Otherwise, using a product such as Lumigent's Log Explorer to save yourself all the research and testing hours could be a real bargain.

Backing Up and Restoring a Database

As you're probably aware by now, this book is not intended to be a how-to book for database administrators. The bibliography in the companion content lists several excellent books that can teach you the mechanics of making database backups and restoring and can offer best practices for setting up a backup-and-restore plan for your organization. Nevertheless, there are some important issues relating to backup and restore processes that can help you understand why one backup plan might be better suited to your needs than another. Most of these issues involve the role the transaction log plays in backup and restore operations, so I'll discuss the main ones in this section.

Types of Backups

No matter how much fault tolerance you have implemented on your database system, it is no replacement for regular backups. Backups can provide a solution to accidental or malicious data modifications, programming errors, and natural disasters (if you store backups in a remote location). If you opt for the fastest possible speed for data file access at the cost of fault tolerance, backups provide insurance in case your data files are damaged.

The degree to which you can restore the lost data depends on the type of backup. SQL Server 2005 has four main types of backups (and a couple of variations on those types):

- **Full backup** A full database backup basically copies all the pages from a database onto a backup device, which can be a local or network disk file, or a local tape drive.

- **Differential backup** A differential backup copies only the extents that were changed since the last full backup was made. The extents are copied onto a specified backup device. SQL Server can quickly tell which extents need to be backed up by examining the bits on the Differential Change Map DCM pages for each data file in the database. DCM pages are big bitmaps, with one bit representing an extent in a file, just like the GAM and SGAM pages I discussed in [Chapter 2](#). Each time a full backup is made, all the bits in the DCM are cleared to 0. When any page in an extent is changed, its corresponding bit in the DCM page is changed to 1.
- **Log backup** In most cases, a log backup copies all the log records that have been written to the transaction log since the last full or log backup was made. However, the exact behavior of the BACKUP LOG command depends on your database's recovery mode setting. I'll discuss recovery modes shortly.
- **File and Filegroup backup** File and filegroup backups are intended to increase flexibility in scheduling and media handling compared to full backups, in particular for very large databases. File and filegroup backups are also useful for large databases that contain data with varying update characteristics, meaning some filegroups allow both read and write operations and some are read-only.

Note



For full details on the mechanics of defining backup devices, making backups, and scheduling backups to occur at regular intervals, consult SQL Server Books Online or one of the SQL

Server administration books listed in the bibliography in the online companion content.

A full backup can be made while your SQL Server instance is in use. This is considered a "fuzzy" backup that is, it is not an exact image of the state of the database at any particular point in time. The backup threads just copy extents, and if other processes need to make changes to those extents while the backup is in progress, they can do so.

To maintain consistency for either full, differential, or file backups, SQL Server records the current log sequence number (LSN) at the time the backup starts and again at the time the backup ends. This allows the backup to also capture the relevant parts of the log. The relevant part starts with the oldest open transaction at the time of the first recorded LSN and ends with the second recorded LSN.

As mentioned previously, what gets recorded with a log backup depends on the recovery model you are using. So before I talk about log backup in detail, I'll tell you about recovery models.

Recovery Models

As I told you in [Chapter 4](#) when I discussed database options, the RECOVERY option has three possible values: FULL, BULK_LOGGED, or SIMPLE. The value you choose determines the speed and size of your transaction log backups as well as the degree to which you are at risk of losing committed transactions in case of media failure.

FULL Recovery Model

The FULL recovery model provides the least risk of losing work in the case of a damaged data file. If a database is in this mode, all operations are fully logged, which means that in addition to logging every row added with the INSERT operation, removed with the DELETE operation, or changed with the UPDATE operation, SQL Server also writes to the transaction log in its entirety every row inserted using a bcp or BULK INSERT operation. If you experience a media failure for a database file and need to recover a database that was in FULL recovery mode and you've been making regular transaction log backups preceded by a full database backup, you can restore to any specified point in time up to the time of the last log backup. In addition, if your log file is available after the failure of a data file, you can restore up to the last transaction committed before the failure. SQL Server 2005 also supports a feature called *log marks*, which allows you to place reference points in the transaction log. If your database is in FULL recovery mode, you can choose to recover to one of these log marks. I'll talk a bit more about log marks in [Chapter 8](#).

In FULL recovery mode, SQL Server also fully logs CREATE INDEX operations. When you restore from a transaction log backup that includes index creations, the recovery operation is much faster because the index does not have to be rebuilt all the index pages have been captured as part of the database backup. Prior to SQL Server 2000, SQL Server logged only the fact that an index had been built, so when you restored from a log backup, the entire index would have to be built all over again!

So, FULL recovery mode sounds great, right? As always, there are tradeoffs. The biggest tradeoff is that the size of your transaction log files can be enormous, so it can take much longer to make log backups than with any previous release of SQL Server.

BULK_LOGGED Recovery Model

The BULK_LOGGED recovery model allows you to completely restore a database in case of media failure and also gives you the best performance and least log space usage for certain bulk operations. These bulk operations include BULK INSERT, bcp, CREATE INDEX, SELECT INTO, WRITETEXT, and UPDATETEXT. In FULL recovery mode, these operations are fully logged, but in BULK_LOGGED recovery mode, they are only minimally logged.

When you execute one of these bulk operations, SQL Server logs only the fact that the operation occurred and information about space allocations. However, the operation is fully recoverable because SQL Server keeps track of what extents were actually modified by the bulk operation. Every data file in a SQL Server 2005 database has at least one special page called a BCM (Bulk Change Map) page, which is managed much like the GAM and SGAM pages that I discussed in [Chapter 2](#) and the DCM pages that I mentioned earlier. Each bit on a BCM page represents an extent, and if the bit is 1, it means that this extent has been changed by a minimally logged bulk operation since the last full database backup. A BCM page is located at the eighth page of every data file and every 511,230 pages thereafter. All the bits on a BCM page are reset to 0 every time a log backup occurs.

Because of the ability to minimally log bulk operations, the operations themselves can be carried out much faster than in FULL recovery mode. Setting the bits in the appropriate BCM page requires a little overhead, but compared with the cost of logging each individual change to a data or index row, the cost of flipping bits is an order of less magnitude.

If your database is in BULK_LOGGED mode and you have not actually performed any bulk operations, you can restore your database to any point in time or to a named log mark because the

log will contain a full sequential record of all changes to your database.

The tradeoff comes during the backing up of the log. In addition to copying the contents of the transaction log to the backup media, SQL Server scans the BCM pages and backs up all the modified extents along with the transaction log itself. The log file itself stays small, but the backup of the log can be many times larger. So the log backup takes more time and might take up a lot more space than in FULL recovery mode. The time it takes to restore a log backup made in BULK_LOGGED recovery mode is similar to the time it takes to restore a log backup made in FULL recovery mode. The operations don't have to be redone; all the information necessary to recover all data and index structures is available in the log backup.

SIMPLE Recovery Model

The SIMPLE recovery model offers the simplest backup-and-restore strategy. Your transaction log is truncated whenever a checkpoint occurs, which happens at regular, frequent intervals. Therefore, the only types of backups that can be made are those that don't require log backups. These types of backups are full database backups, differential backups, partial full and differential backups, and filegroup backups for read-only filegroups. You get an error if you try to back up the log while in SIMPLE recovery mode. Because the log is not needed for backup purposes, sections of it can be reused as soon as all the transactions it contains are committed or rolled back, and the transactions are no longer needed for recovery from server or transaction failure. In fact, as soon as you change your database to SIMPLE recovery model, the log will be truncated.

Keep in mind that SIMPLE logging does not mean no logging. What's "simple" is your backup strategy because you'll never need to worry about log backups. However, all operations are logged in

SIMPLE mode, even though the individual log records are not as big as they are in FULL mode. A log for a database in SIMPLE mode might not grow as much as a database in FULL mode because the bulk operations discussed under BULK_LOGGED recovery model will also be minimally logged in SIMPLE mode. This does not mean you don't have to worry about the size of the log in SIMPLE mode. As in any recovery mode, log records for active transactions cannot be truncated and neither can log records for any transaction that started after the oldest open transaction. So, if you have large or long-running transactions, you still might need lots of log space.

Migrating from SQL Server 7.0

Microsoft introduced these recovery models in SQL Server 2000 and intended them to replace the *select into/bulkcopy* and *trunc. log on chkpt.* database options. SQL Server 7.0 and earlier versions required that the *select into/bulkcopy* option be set in order for you to perform a SELECT INTO or bulk copy operation. The *trunc. log on chkpt.* option forced your transaction log to be truncated every time a checkpoint occurred in the database. This option was recommended only for test or development systems, not for production servers. You can still set these options by using the *sp_dboption* procedure but not by using the ALTER DATABASE command. However, in SQL Server 2000 and SQL Server 2005, changing either of these options using *sp_dboption* also changes your recovery model, and changing your recovery model changes the value of one or both of these options, as you'll see below. The recommended method for changing your database recovery mode is to use the ALTER DATABASE command:

```
ALTER DATABASE <database_name>
    SET RECOVERY [FULL | BULK_LOGGED | SIMPLE]
```

To see what mode your database is in, you can inspect the `sys.databases` view. For example, this query returns the recovery mode and the state of the `AdventureWorks` database:

```
SELECT name, database_id, suser_sname(owner_sid)
as owner ,
       state_desc, recovery_model_desc
FROM sys.databases
WHERE name = 'AdventureWorks'
```

As I just mentioned, you can change the recovery mode by changing the database options. For example, if your database is in FULL recovery mode and you change the `select into/bulkcopy` option to `true`, your database recovery mode changes to BULK_LOGGED. Conversely, if you force the database back into FULL mode by using `ALTER DATABASE`, the value of the `select into/bulkcopy` option changes. If you're using SQL Server 2005 Standard Edition or Enterprise Edition, the `model` database starts in FULL recovery mode, so all your new databases will also be in FULL mode. You can change the mode of the `model` database or any other user database by using the `ALTER DATABASE` command.

To make best use of your transaction log, you can switch between the FULL and BULK_LOGGED modes without worrying about your backup scripts failing. Prior to SQL Server 2000, once you performed a `SELECT INTO` or a bulk copy, you could no longer back up your transaction log. So if you had automatic log backup scripts scheduled to run at regular intervals, these would break and generate an error. This can no longer happen. You can run `SELECT INTO` or bulk copy in any recovery mode, and you can back up the log in either FULL or BULK_LOGGED mode. You might want to switch between FULL and BULK_LOGGED modes if you usually operate in FULL mode but occasionally need to perform a bulk

operation quickly. You can change to BULK_LOGGED and pay the price later when you back up the log; the backup will simply take longer and be larger.

You can't easily switch to and from SIMPLE mode. Switching into SIMPLE mode is no problem, but when you switch back to FULL or BULK_LOGGED, you need to plan your backup strategy and be aware that there are no log backups up to that point. So when you use the ALTER DATABASE command to change from SIMPLE to FULL or BULK_LOGGED, you should first make a complete database backup in order for the change in behavior to be complete. Remember that in SIMPLE recovery mode, your transaction log will be truncated at regular intervals. This recovery mode isn't recommended for production databases, where you need maximum transaction recoverability. The only time that SIMPLE mode is really useful is in test and development situations or for small databases that are primarily read-only. I suggest that you use FULL or BULK_LOGGED for your production databases and switch between those modes whenever you need to.

Choosing a Backup Type

If you're responsible for creating the backup plan for your data, you'll need to choose not only a recovery mode but also the kind of backup to make. I mentioned the three main types: full, differential, and log. In fact, you can use all three types together. To accomplish any type of full restore of a database, you must occasionally make a full database backup. In addition, you must choose between a differential backup and a log backup. Here are characteristics of these last two types that can help you decide between them:

A differential backup:

- Is faster if your environment includes a lot of changes to the same data. It backs up only the most recent change, whereas a log backup captures every individual update.
- Captures the entire B-tree structures for new indexes, whereas a log backup captures each individual step in building the index.
- Is cumulative. When you recover from a media failure, only the most recent differential backup needs to be restored because it will contain all the changes since the last full database backup.

A log backup:

- Allows you to restore to any point in time because it is a sequential record of all changes.
- Can be made after a failure of the database media, as long as the log is available. This allows you to recover right up to the point of the failure. The last log backup (called the tail of the log) must specify the WITH NO_TRUNCATE option in the BACKUP LOG command if the database itself is unavailable.
- Is sequential and discrete. Each log backup contains completely different log records. When you use a log backup to restore a database after a media failure, all log backups must be applied in the order that they were made.

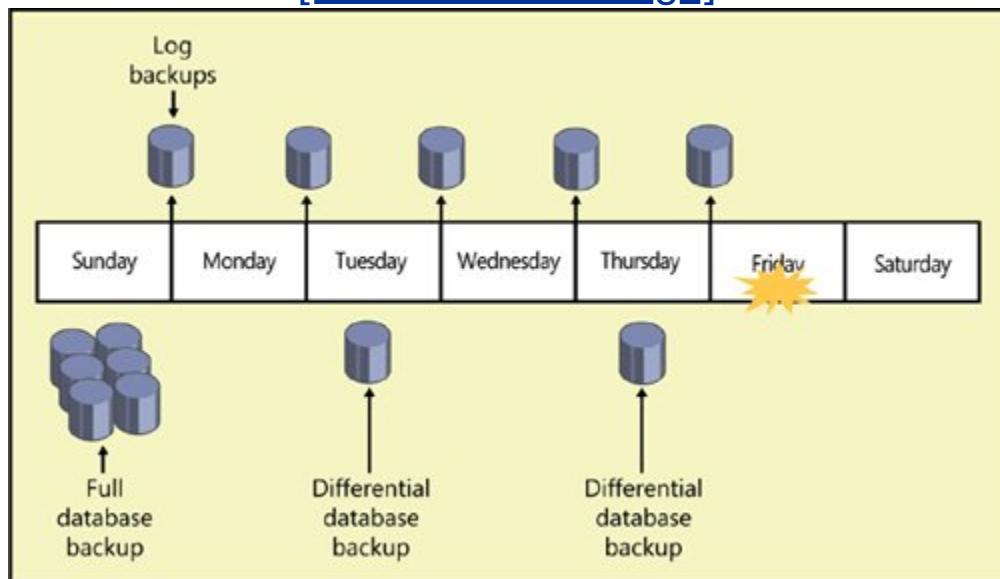
Restoring a Database

How often you make each type of backup determines two things: how fast you can restore a database and how much control you

have over which transactions are restored. Consider the schedule in [Figure 5-5](#), which shows a database fully backed up on Sunday. The log is backed up daily, and a differential backup is made on Tuesday and Thursday. A drive failure occurs on Friday. If the failure does not include the log files or if you have mirrored them using RAID 1, you should back up the tail of the log with the NO_TRUNCATE option.

Figure 5-5. Combined use of log and differential backups reduces total restore time.

[[View full size image](#)]



Warning



If you are operating in BULK_LOGGED recovery mode, backing up the log also backs up any data that was changed with a BULK_LOGGED operation, so you might need to have more than just the log file available to back up the tail of the log. You'll also need to have available any filegroups containing data inserted with a bulk copy or SELECT INTO command.

To restore this database after a failure, you must start by restoring the full backup made on Sunday. This does two things: it copies all the data, log, and index pages from the backup media to the database files, and it applies all the transactions in the log. You must determine whether incomplete transactions are rolled back. You can opt to recover the database by using the WITH RECOVERY option of the RESTORE command. This rolls back any incomplete transactions and opens the database for use. No further restoring can be done. If you choose not to roll back incomplete transactions by specifying the WITH NORECOVERY option, the database will be left in an inconsistent state and will not be usable.

If you choose WITH NORECOVERY, you can then apply the next backup. In the scenario depicted in [Figure 5-5](#), you would restore the differential backup made on Thursday, which would copy all the changed extents back into the data files. The differential backup

also contains the log records spanning the time the differential backup was being made, so you have to decide whether to recover the database. Complete transactions are always rolled forward, but you determine whether incomplete transactions are rolled back.

After the last differential backup is restored, you must restore, in sequence, all the log backups made after the last differential backup was made. This includes the tail of the log backed up after the failure if you were able to make this last backup.

Note



Restore recovery (media recovery) works almost exactly the same way as restart recovery, which I described earlier in this chapter. It includes an analysis pass to determine how much work might need to be done, a roll-forward pass to redo completed transactions and return the database to the state it was in when the backup was complete, and a roll-back pass to undo incomplete transactions. The big difference between restore recovery and restart recovery is that with restore recovery, you have control over when the roll-back pass is done. It should not be done until all the rolling forward from all the backups has been applied. Only then should you roll back any transactions that are still not complete. In addition, SQL Server might need to make some adjustments to metadata after the recovery is complete, so no access to the database is allowed until all phases of recovery are finished. In other words,

there is no option to use "fast" recovery as part of a RESTORE.

Backing Up and Restoring Files and Filegroups

SQL Server 2005 allows you to back up individual files or filegroups. This can be useful in environments with extremely large databases. You can choose to back up just one file or filegroup each day, so the entire database does not have to be backed up as often. This also can be useful when you have an isolated media failure on a single drive, and you think that restoring the entire database would take too long.

Here are some details to keep in mind about backing up and restoring files and filegroups:

- Individual files and filegroups with the READWRITE property can be backed up only when your database is in FULL or BULK_LOGGED recovery mode because you must apply log backups after you restore a file or filegroup, and you can't make log backups in SIMPLE mode. Read-only filegroups, and the files in them, can be backed up in SIMPLE mode.
- You can restore individual file or filegroup backups from a full database backup.
- Immediately before restoring an individual file or filegroup, you must back up the transaction log. You must have an unbroken

chain of log backups from the time the file or filegroup backup was made.

- After restoring a file or filegroup backup, you must restore all the transaction logs made between the time you backed up the file or filegroup and the time you restored it. This guarantees that the restored files are in sync with the rest of the database.

For example, suppose you back up Filegroup FG1 at 10 A.M. on Monday. The database is still in use, and changes happen to data in FG1 and transactions are processed that change data in both FG1 and other filegroups. You back up the log again at 4 P.M. More transactions are processed that change data in both FG1 and other filegroups. At 6 P.M., a media failure occurs and you lose one or more of the files that make up FG1.

To restore, you must first back up the tail of the log containing all changes that occurred between 4 P.M. and 6 P.M. The tail of the log is backed up using the special WITH NO_TRUNCATE option, but you can also use the NORECOVERY option. When backing up the tail of the log WITH NORECOVERY, the database is put into the RESTORING state and can prevent an accidental background change from interfering with the restore sequence.

You can then restore FG1 using the RESTORE DATABASE command, specifying just filegroup FG1. Your database will not be in a consistent state because the restored FG1 will have changes only through 10 A.M., and the rest of the database will have changes through 6 P.M. However, SQL Server knows when the last change was made to the database because each page in a database stores the LSN of the last log record that changed that page. When restoring a filegroup, SQL Server makes a note of the maximum LSN in the database. You must restore log backups until the log reaches at least the maximum LSN in the database, and you will not reach that point until you apply the 6 P.M. log backup.

Partial Backups

A partial backup can be based either on a full or a differential backup, but a partial backup does not contain all of the filegroups. Partial backups contain all the data in the primary filegroup and all of the read-write filegroups. In addition, you can specify that any read-only files also be backed up. If the entire database is marked as read-only, a partial backup will contain only the primary filegroup. Partial backups are particularly useful for VLDBs using the SIMPLE recovery model because they allow you to back up only specific filegroups, even without having log backups.

Page Restore

SQL Server 2005 also allows you to restore individual pages. When SQL Server detects a damaged page, it marks it as suspect and stores information about the page in the `suspect_pages` table in the `msdb` database.

Damaged pages can be detected when activities such as the following take place:

- A query needs to read a page.
- DBCC CHECKDB or DBCC CHECKTABLE is being run.
- BACKUP or RESTORE is being run.
- The database is dropped.
- You are trying to repair a database with DBCC DBREPAIR.

Several types of errors can require a page to be marked as suspect and entered into the suspect_pages table. These can include checksum and torn page errors, as well as internal consistency problems such as a bad page ID in the page header. The column event_type in the suspect_pages table indicates the reason for the status of the page, which usually reflects the reason the page has been entered into the suspect_pages table. SQL Server Books Online lists the following possible values for the event_type column:

Event_type	Description
1	824 errors other than a bad checksum or a torn page (for example, a bad page ID).
2	Bad checksum.
3	Torn page.
4	Restored. (The page was restored after it was marked as bad.)
5	Repaired. (DBCC repaired the page.)
6	Deallocated by DBCC.

Some of the errors recorded in the suspect_pages table might be transient errors such as an I/O error that occurs because a cable has been disconnected. Rows can be deleted from the suspect_pages table by someone with the appropriate permissions, such as someone in the sysadmin server role. In addition, not all errors that cause a page to be inserted in the suspect_pages table require that the page be restored. A problem that occurs in cached data, such as in a nonclustered index, might be resolved by rebuilding the index. If a sysadmin drops a nonclustered index and rebuilds it, the corrupt data, although fixed, will not be indicated as fixed in the suspect_pages table.

Page restore is specifically intended to replace pages that have been marked as suspect because of an invalid checksum or a torn write. Although multiple database pages can be restored at once, you aren't expected to be replacing a large number of individual pages. If you do have many damaged pages, you should probably consider a full file or database restore. In addition, you should probably try to determine the cause of the errors; if you discover pending device failure, you should do your full file or database restore to a new location. Log restores must be done after the page restores to bring the new pages up-to-date with the rest of the database. Just like with file restore, the log backups are applied to the database files containing a page that is being recovered.

In an online page restore, the database is online for the duration of the restore, and only the data being restored is offline. Note that not all damaged pages can be restored with the database online.

Note



Online restore is allowed only in Enterprise Edition of SQL Server 2005.

Books Online lists the following basic steps for a page restore:

1. Obtain the page IDs of the damaged pages to be restored. A checksum or torn write error returns the page ID, which is the information needed for specifying the pages. You can also get page IDs from the `suspect_pages` table.
2. Start a page restore with a full, file, or filegroup backup that contains the page or pages to be restored. In the `RESTORE DATABASE` statement, use the `PAGE` clause to list the page IDs of all the pages to be restored. The maximum number of pages that can be restored in a single file is 1000.
3. Apply any available differentials required for the pages being restored.
4. Apply the subsequent log backups.
5. Create a new log backup of the database that includes the final LSN of the restored pages that is, the point at which the last restored page was taken offline. The final LSN, which is set as part of the first restore in the sequence, is the redo target LSN. Online roll-forward of the file containing the page can stop at the redo target LSN. To learn the current redo target LSN of a file, see the `redo_target_lsn` column of `sys.master_files`.
6. Restore the new log backup. Once this new log backup is applied, the page restore is complete and the pages are usable. All the pages that were bad are affected by the log restore. All other pages will have a more recent LSN in their page header and there will be nothing to redo. In addition, no UNDO phase is

needed for page-level restore.

Partial Restore

SQL Server 2005 lets you do a partial restore of a database in emergency situations. Although the description and the syntax look similar to file and filegroup backup and restore, there is a big difference. With file and filegroup restore, you start with a complete database and replace one or more files or filegroups with previously backed up versions. With a partial database restore, you don't start with a full database. You restore individual filegroups, which must include the primary filegroup containing all the system tables, to a new location. Any filegroups you don't restore are treated as OFFLINE when you attempt to reference data stored on them. You can then restore log backups or differential backups to bring the data in those filegroups to a later point in time. This allows you the option of recovering the data from a subset of tables after an accidental deletion or modification of table data. You can use the partially restored database to extract the data from the lost tables and copy it back into your original database.

Restoring with Standby

In normal recovery operations, you have the choice of either running recovery to roll back incomplete transactions or not running recovery. If you run recovery, no further log backups can be restored and the database is fully usable. If you don't run recovery, the database is inconsistent and SQL Server won't let you use it at all. You have to choose one or the other because of the way log backups are made.

For example, in SQL Server 2005, log backups do not overlap; each log backup starts where the previous one ended. Consider a transaction that makes hundreds of updates to a single table. If you back up the log during the update and also after it, the first log backup will have the beginning of the transaction and some of the updates, and the second log backup will have the remainder of the updates and the commit. Suppose you then need to restore these log backups after restoring the full database. If, after restoring the first log backup, you run recovery, the first part of the transaction will be rolled back. If you then try to restore the second log backup, it will start in the middle of a transaction and SQL Server won't know what the beginning of the transaction did. You certainly can't recover transactions from this point because their operations might depend on this update that you've partially lost. SQL Server therefore will not allow any more restoring to be done. The alternative is to not run recovery to roll back the first part of the transaction and instead to leave the transaction incomplete. SQL Server will know that the database is inconsistent and will not allow any users into the database until you finally run recovery on it.

What if you want to combine the two approaches? It would be nice to be able to restore one log backup and look at the data before restoring more log backups, particularly if you're trying to do a point-in-time recovery, but you won't know what the right point is. SQL Server provides an option called STANDBY that allows you to recover the database and still restore more log backups. If you restore a log backup and specify WITH STANDBY = '<some filename>', SQL Server will roll back incomplete transactions but keep track of the rolled-back work in the specified file, which is known as a *standby file*. The next restore operation will read the contents of the standby file and redo the operations that were rolled back, and then it will restore the next log. If that restore also specifies WITH STANDBY, incomplete transactions will again be rolled back, but a record of those rolled back transactions will be saved. Keep in mind that you can't modify any data if you've restored WITH STANDBY (SQL Server will generate an error).

message if you try), but you can read the data and continue to restore more logs. The final log must be restored WITH RECOVERY (and no standby file will be kept) to make the database fully usable.

Summary

In addition to one or more data files, every database in a SQL Server instance has one or more log files that keep track of changes to that database. (In SQL Server 2005, database snapshots do not have log files because no changes are ever made directly to a snapshot.) SQL Server uses the transaction log to guarantee consistency of your data, at both a logical and a physical level. In addition, an administrator can make backups of the transaction log to make restoring a database more efficient. An administrator or database owner can also set a database's recovery mode to determine the level of detail stored in the transaction log.

Chapter 6. Tables

In this chapter:

<u>System Objects</u>	<u>176</u>
<u>Creating Tables</u>	<u>183</u>
<u>User-Defined Data Types</u>	<u>198</u>
<u>IDENTITY Property</u>	<u>200</u>
<u>Internal Storage</u>	<u>203</u>
<u>Constraints</u>	<u>237</u>
<u>Altering a Table</u>	<u>242</u>
<u>Summary</u>	<u>248</u>

In this chapter, we'll look at some in-depth implementation examples. But we'll start with a basic introduction to tables. Simply

put, a *table* is a collection of data about a specific *entity* (person, place, or thing) that has a discrete number of named *attributes* (for example, quantity or type). Tables are at the heart of Microsoft SQL Server and the relational model in general. In SQL Server, a table is often referred to as a *base table* to emphasize where data is stored. Calling it a base table also distinguishes the table from a *view*—a virtual table that's an internal query referencing one or more base tables or other views.

Attributes of a table's data (such as color, size, quantity, order date, and supplier's name) take the form of named *columns* in the table. Each instance of data in a table is represented as a single entry, or *row* (formally called a *tuple*). In a true relational database, each row in a table is unique and has a unique identifier called a *primary key*. (SQL Server, in accordance with the ANSI SQL standard, doesn't require you to make a row unique or declare a primary key. However, because both of these concepts are central to the relational model, you should always implement them.) Most tables will have some relationship to other tables. For example, in a typical order-entry system, the *orders* table has a *customer_number* column for keeping track of the customer number for an order, and *customer_number* also appears in the *customer* table. Assuming that *customer_number* is a unique identifier, or primary key, of the *customer* table, a foreign key relationship is established by which the *orders* and *customer* tables can subsequently be joined.

So much for the 30-second database design primer. You can find plenty of books that discuss logical database and table design, but this isn't one of them. I'll assume that you understand basic database theory and design and that you generally know what your tables will look like. The rest of this chapter discusses the internals of tables in SQL Server 2005.

System Objects

SQL Server maintains a set of tables that store information about all the objects, data types, constraints, configuration options, and resources available to SQL Server. This set of tables is sometimes called the *system catalog*. In SQL Server 2005, these tables are called the *system base tables*. Some of the system base tables exist only in the *master* database and contain systemwide information, and others exist in every database (including *master*) and contain information about the objects and resources belonging to that particular database. In SQL Server 2005, the system base tables are not always visible by default, in *master* or any other database. You won't see them when you expand the *tables* node in the Object Explorer in SQL Server Management Studio, and unless you are a system administrator, you won't see them when you execute the *sp_help* system procedure. In addition, if you log in as a system administrator and select from the catalog view (discussed shortly) called *sys.objects*, you can see the names of all the system tables. For example, the following query returns 51 rows of output on my SQL Server 2005 Service Pack 1 instance:

```
USE master
SELECT name FROM sys.objects
WHERE type_desc = 'SYSTEM_TABLE'
```

But even as a system administrator, if you try to select from one of the tables whose names are returned by the preceding query, you will get a 208 error, indicating that the object name is invalid. The only way to see the data in the system tables is to make a connection using the dedicated administrator connection (DAC), which I told you about in [Chapter 2](#). Keep in mind that the system base tables are used for internal purposes only within the SQL

Server 2005 Database Engine and are not intended for general customer use. They are subject to change, and compatibility is not guaranteed. In SQL Server 2005, the recommended way of accessing metadata is to use one of three new types of system objects. One type is dynamic management objects, which I introduced in [Chapter 2](#). We looked at some of the dynamic management views (DMVs) and dynamic management functions (DMFs) that provide information about SQL Server memory and schedulers in [Chapter 2](#), and we looked at other dynamic management objects in subsequent chapters. Remember that these dynamic management objects don't really correspond to physical tablesthey contain information gathered from internal structures to allow you to observe the current state of your SQL Server instance. The other two types of system objects are actual views built on top of the system base tables.

Compatibility Views

Although SQL Server 2000 allows you to see data in the system tables, it doesn't strongly encourage you to do this. Nevertheless, many people have made use of system tables for developing their own troubleshooting and reporting tools and techniques, providing result sets that aren't available using the supplied system procedures. You might assume that due to the inaccessibility of the SQL Server 2005 system tables, you would have to use the DAC to make use of your homegrown tools after you upgrade to SQL Server 2005. However, you still might be disappointed. Many of the names and much of the content of the SQL Server 2000 system tables have changed, so any code that uses them might be completely unusable even with the DAC. The DAC is intended only for emergency access, and no support is provided for any other use of the DAC interface. To save you from this grief, SQL Server 2005 offers a set of compatibility views that allow you to continue to access a subset of the SQL Server 2000 system tables. These

views are accessible from any database, although they were created in the hidden resource database.

Here are the names of the compatibility views. Please see SQL Server Books Online for complete details.

- *sysconfigures*
- *syscharsets*
- *syslanguages*
- *syscacheobjects*
- *sysaltfiles*
- *syssegments*
- *sysfiles*
- *sysfilegroups*
- *sysmembers*
- *sysusers*
- *systypes*
- *sysreferences*
- *sysprotects*

- *syspermissions*
- *sysindexkeys*
- *sysindexes*
- *sysfulltextcatalogs*
- *sysconstraints*
- *sysforeignkeys*
- *sysdepends*
- *syscomments*
- *syscolumns*
- *sysobjects*

If you've upgraded from SQL Server 2000, some of these object names might look familiar, such as *sysobjects* and *sysindexes*. Others, like *syssegments*, are still around only for backward compatibility with much older versions.

For compatibility reasons, the views in SQL Server 2005 have the same names as their SQL Server 2000 counterparts, as well as the same column names, which means that your existing code that uses the SQL Server 2000 system tables won't break. However, when you select from these views, you are not guaranteed to get exactly the same results that you get from the corresponding tables in SQL Server 2000. In addition, the compatibility views do not contain any metadata related to new SQL Server 2005 features,

such as partitioning. You should consider the compatibility views to be for backward compatibility only, and going forward, you should consider using other metadata mechanisms, such as the catalog view discussed in the next section. All these compatibility views will be removed in a future version of SQL Server.

SQL Server 2005 also provides compatibility views for the SQL Server 2000 pseudo-tables, such as *sysprocesses* and *syscacheobjects*. Pseudo-tables are tables that are not based on data stored on disk, but are built as needed from internal structures and can be queried exactly as if they are tables. SQL Server 2005 replaces the pseudo-tables with dynamic management objects, of which there are more than 80 DMVs and DMFs, as I mentioned earlier. Note that there is not always a one-to-one correspondence between the SQL Server 2000 pseudo-tables and the SQL Server 2005 dynamic management objects. For example, for SQL Server 2005 to retrieve all the information available in *sysprocesses*, you must access three DMVs: *sys.dm_exec_connections*, *sys.dm_exec_sessions*, and *sys.dm_exec_requests*.

Catalog Views

SQL Server 2005 introduces a set of catalog views as a general interface to the persisted system metadata. All the catalog views (as well as the dynamic management objects and compatibility views) are in the *sys* schema, and you must reference the schema name when you access the objects. Some of the names are easy to remember because they are similar to the SQL Server 2000 system table names. For example, there is a catalog view called *objects* in the *sys* schema, so to reference the view we can execute the following:

```
SELECT * FROM sys.objects
```

Similarly, there are catalog views called *sys.indexes* and *sys.databases*, but the columns displayed for these catalog views are very different from the columns in the compatibility views. Because the output from these types of queries is too wide to reproduce, let me just suggest that you run these two queries yourself and observe the difference.

```
SELECT * FROM sys.databases  
SELECT * FROM sysdatabases
```

Note that you must be in the *master* database to access the *sysdatabases* compatibility view directly, just as you have to be in the *master* database in SQL Server 2000 to access the *sysdatabases* table directly. The *sysdatabases* compatibility view is in the *sys* schema, so you can reference it as *sys.sysdatabases*. You can also reference it using *dbo.sysdatabases*. But again, for compatibility reasons, the schema name is not required, as it is for the catalog views. (That is, you cannot simply select from a view called *databases*; you must use the schema *sys* as a prefix.) When you compare the output from the two preceding queries, you might notice that there are a lot more columns in the *sys.databases* catalog view. Instead of a bitmap status field that needs to be decoded, each possible database property has its own column in *sys.databases*. With SQL Server 2000, running the system procedure *sp_helpdb* decodes all these database options, but because *sp_helpdb* is a procedure, it is difficult to filter the results. As a view, *sys.databases* can be queried and filtered. For example, if we want to know which databases are in SIMPLE recovery mode, we can run the following:

```
SELECT name FROM sys.databases  
WHERE recovery_model_desc = 'SIMPLE'
```

The catalog views are built on an inheritance model, so sets of attributes common to many objects don't have to be redefined internally. For example, `sys.objects` contains all the columns for attributes common to all types of objects, and the views `sys.tables` and `sys.views` contain all the same columns as `sys.objects`, as well as some additional columns that are relevant only to the particular type of objects. If you select from `sys.objects`, you get 12 columns, and if you then select from `sys.tables`, you get exactly the same 12 columns in the same order, plus 12 additional columns that aren't applicable to all types of objects but are meaningful for tables. In addition, although the base view `sys.objects` contains a subset of columns compared to the derived views such as `sys.tables`, it contains a superset of rows compared to a derived view. For example, the `sys.objects` view shows metadata for procedures and views in addition to that for tables, whereas the `sys.tables` view shows only rows for tables. So we can summarize the relationship between the base view and the derived views as follows: "The base views contain a subset of columns and a superset of rows, and the derived views contain a superset of columns and a subset of rows."

Just like in SQL Server 2000, some of the metadata appears only in the `master` database, and it keeps track of systemwide data, such as databases and logins. Other metadata is available in every database, such as objects and permissions. The SQL Server Books Online topic "Mapping SQL Server 2000 System Tables to SQL Server 2005 System Views" categorizes its objects into two lists those appearing only in `master` and those appearing in all databases. Note that metadata appearing only in the `msdb` database is not available through catalog views but is still available in system tables, in the schema `dbo`. This includes metadata for replication, backup, Database Maintenance Plans, and SQL Server Agent jobs and alerts.

As views, these metadata objects are based on an underlying Transact-SQL definition. The most straightforward way to see the definition of these views is by using the *object_definition* function. (You can also see the definition of these system views by using *sp_helptext* or by selecting from the catalog view *sys.system_sql_modules*.) So to see the definition of *sys.objects*, we can execute the following:

```
SELECT object_definition (object_id('sys.tables'))
```

If you execute the preceding SELECT statement, you'll see that the definition of *sys.tables* references several completely undocumented system objects. On the other hand, some system object definitions refer only to objects that are documented. For example, the definition of the compatibility view *syscacheobjects* refers only to three dynamic management objects (two views, *sys.dm_exec_cached_plans* and *sys.dm_exec_plan_attributes*, and one function, *sys.dm_exec_sql_text*) that are fully documented.

Other Metadata

Although the catalog views are the recommended interface for accessing SQL Server 2005 metadata, other tools are available as well.

Information Schema Views

Information schema views were the original system table-independent view of the SQL Server metadata, introduced in SQL Server 7.0. The information schema views included in SQL Server 2005 comply with the SQL-92 standard definition for the

INFORMATION_SCHEMA, and all these views are in a schema called INFORMATION_SCHEMA. Most of the information available through the catalog views is available through the information schema views, and if you need to write a portable application that accesses the metadata, you should consider using these objects. However, the information schema views only show objects compatible with the SQL-92 standard. This means there is no information schema view for certain features, such as indexes, that are not defined in the standard. (Indexes are an implementation detail.) If your code does not need to be strictly portable, or if you need metadata about non-standard features such as indexes, filegroups, the CLR, and SQL Server Service Broker, I suggest using the Microsoft-supplied catalog views. Most of the examples in the documentation, as well as in this and other reference books, are based on the catalog view interface.

System Functions

Most SQL Server system functions are property functions, which were introduced in SQL Server 7.0 and greatly enhanced in SQL Server 2000. SQL Server 2005 has enhanced these functions still further. Property functions give us individual values for many SQL Server objects and also for SQL Server databases and the SQL Server instance itself. The values returned by the property functions are scalar as opposed to tabular, so they can be used as values returned by SELECT statements and as values to populate columns in tables. Here is the list of property functions available in SQL Server 2005:

- SERVERPROPERTY
- COLUMNPROPERTY

- DATABASEPROPERTY (deprecated in favor of the following function)
- DATABASEPROPERTYEX
- INDEXPROPERTY
- INDEXKEY_PROPERTY
- OBJECTPROPERTY (deprecated)
- OBJECTPROPERTYEX
- SQL_VARIANT_PROPERTY
- FILEPROPERTY
- FILEGROUPPROPERTY
- TYPEPROPERTY

The only way to find out what the possible property values are for the various functions is to check Books Online.

Some of the information returned by the property functions can also be seen using the catalog views. For example, the DATABASEPROPERTYEX function has a property called *Recovery* that returns the recovery model of a database. To view the recovery model of a single database, we can use the property function:

```
SELECT DATABASEPROPERTYEX('AdventureWorks',  
'Recovery').
```

To view the recovery models of all our databases, we can use the `sys.databases` view:

```
SELECT name, recovery_model_desc  
FROM sys.databases
```

Note



Columns with names ending `in_desc` are the so-called "friendly name" columns, and they are always paired with another column that is much more compact, but cryptic. In this case, the `sys.databases.recovery_model` column is a `tinyint` with a value of 1, 2, or 3. Both columns are available in the view because different consumers have different needs. For example, internally at Microsoft, the teams building the internal interfaces wanted to bind to more compact columns, whereas DBAs running ad hoc queries might prefer the friendly names.

Tip



If your application doesn't need the friendly name columns, don't include them in your SELECT list. As a result, fewer underlying objects will need to be accessed and your queries will run faster.

In addition to the property functions, the system functions also include functions that are merely shortcuts for catalog view access. For example, to find out the database ID for the *AdventureWorks* database, we can either query the *sys.databases* catalog view or use the *DB_ID()* function. Both of the following SELECT statements should return the same result:

```
SELECT database_id  
FROM sys.databases  
WHERE name = 'AdventureWorks'
```

```
SELECT DB_ID('AdventureWorks')
```

System Stored Procedures

System stored procedures are the original metadata access tool, in addition to the system tables themselves. Most of the system stored procedures introduced in the very first version of SQL Server are still available. However, catalog views are a big improvement: you have control over how much of the metadata you see because you can query the views as if they were tables. With the system stored

procedures, you basically have to accept the data it returns. Some of the procedures allow parameters, but they are very limited. So for the *sp_helpdb* procedure, we can pass a parameter to see just one database's information or not pass a parameter and see information for all databases. However, if we want to see only databases that the login *sue* owns, or just see databases that are in a lower compatibility level, we cannot do it using the supplied stored procedure. Using the catalog views, these queries are straightforward:

```
SELECT name FROM sys.databases  
WHERE suser_sname(owner_sid) = 'sue'
```

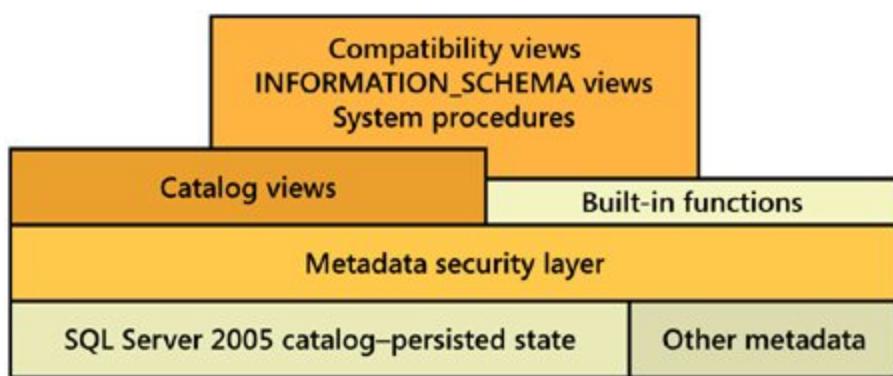
```
SELECT name FROM sys.databases  
WHERE compatibility_level < 90
```

Metadata Wrap-Up

[Figure 6-1](#) shows the multiple layers of metadata available in SQL Server 2005, with the lowest layer being the actual system tables. Any interface that accesses the information contained in the system tables is subject to the metadata security policies. For SQL Server 2005, that means nobody can see any metadata that they don't need to see or to which they haven't specifically been granted permissions. The "other metadata" refers to system information not contained in system tables, such as the internal information provided by the dynamic management objects. Remember that the preferred interface to the system metadata is the catalog views and system functions. Although not all the compatibility views, INFORMATION_SCHEMA views, and system procedures are actually defined in terms of the catalog views, conceptually it is

useful to think of them as another layer on top of the catalog view interface.

Figure 6-1. Layers of metadata in SQL Server 2005



Keep in mind that the reason for providing metadata is really to give you information about the objects you have created. So for the rest of this chapter, we'll look at the most important object you can create in a database: the table.

Creating Tables

To create a table, SQL Server uses the ANSI SQL standard CREATE TABLE syntax. SQL Server Management Studio provides a front-end, fill-in-the-blanks table designer that can sometimes make your job easier. Ultimately, the SQL syntax is always sent to SQL Server to create a table, no matter what interface you use. In this chapter, I'll emphasize direct use of the data definition language (DDL) rather than discuss the graphical interface tools. You should keep all DDL commands in a script so you can run them easily at a later time to re-create the table. (Even if you use one of the friendly front-end tools, it's critical that you be able to re-create the table later.) SQL Server Management Studio and other front-end tools can create and save operating system files using the SQL DDL commands necessary to create any object. This DDL is essentially source code, and you should treat it as such. Keep a backup copy. You should also consider keeping these files under version control using a source control product such as Microsoft Visual SourceSafe.

At the basic level, creating a table requires little more than knowing what you want to name it, what columns it will contain, and what range of values (domain) each column can store. Here's the basic syntax for creating the *customer* table in the *dbo* schema, with three fixed-length character (*char*) columns. (Note that this table definition isn't necessarily the most efficient way to store data because it always requires 46 bytes per entry for data plus a few bytes of overhead, regardless of the actual length of the data.)

```
CREATE TABLE dbo.customer
(
    name        char(30),
    phone       char(12),
    emp_id      char(4)
)
```

This example shows each column on a separate line, for readability. As far as the SQL Server parser is concerned, white spaces created by tabs, carriage returns, and the spacebar are identical. From the system's standpoint, the following CREATE TABLE example is identical to the preceding one, but it's harder to read from a user's standpoint:

```
CREATE TABLE customer (name char(30), phone  
char(12), emp_id char(4))
```

Naming Tables and Columns

A table is always created within one schema of one database. Tables also have owners, but unlike in earlier versions of SQL Server, the table owner is not used to access the table. The schema is used for all object access. Normally, a table is created in the default schema of the user who is creating it, but the CREATE TABLE statement can indicate the schema in which the object is to be created. A user can create a table only in a schema for which the user has ALTER permissions. Any user in the `sysadmin`, `db_ddladmin`, or `db_owner` roles can create a table in any schema. A database can contain multiple tables with the same name, as long as the tables are in different schemas. The full name of a table has three parts, in the following form:

`database.schema.tablename`

The first two parts of the three-part name specification have default values. The default for the name of the database is whatever database context you're currently working in. The table schema

actually has two possible defaults when querying. If no schema name is specified when you reference a table, SQL Server first checks for an object in your default schema. If there is no such table in your default schema, SQL Server then checks to see if there is an object of the specified name in the *dbo* schema.

Note



To access a table in a schema other than your default schema or the *dbo* schema, you must include the schema name along with the table name. In fact, you should get in the habit of always including the schema name when referring to any object in SQL Server 2005. Not only does this remove any possible confusion about which schema you are interested in, but it can lead to some performance benefits.

You should make column names descriptive, and because you'll use them repeatedly, you should avoid wordiness. The name of the column (or any object in SQL Server, such as a table or a view) can be whatever you choose, as long as it conforms to the SQL Server rule for regular identifiers: it must consist of a combination of 1 through 128 letters, digits, or the symbols #, \$, @, or _. (Alternatively, you can use a delimited identifier that includes any characters you like. For more about identifier rules, see "Using Identifiers" in Books Online. The discussion there applies to all SQL Server object names, not just column names.)

In some cases, you can access a table using a four-part name, in which the first part is the name of the SQL Server instance. However, you can refer to a table using a four-part name only if the SQL Server instance has been defined as a linked server. You can read more about linked servers in Books Online; I won't discuss them further here.

Reserved Keywords

Certain reserved keywords, such as TABLE, CREATE, SELECT, and UPDATE, have special meaning to the SQL Server parser, and collectively they make up the SQL language implementation. You should avoid using reserved keywords for your object names. In addition to the SQL Server reserved keywords, the SQL-92 standard has its own list of reserved keywords. In some cases, this list is more restrictive than the SQL Server list; in other cases, it's less restrictive. Books Online includes both lists.

Watch out for the SQL-92 reserved keywords. Some of the words aren't reserved keywords in SQL Server yet, but they might become reserved keywords in a future SQL Server version. If you use a SQL-92 reserved keyword, you might end up having to alter your application before upgrading it if the word becomes a SQL Server reserved keyword.

Delimited Identifiers

You can't use keywords in your object names unless you use a delimited identifier. In fact, if you use a delimited identifier, not only can you use keywords as identifiers but you can also use any other string as an object name whether or not it follows the rules for identifiers. This includes spaces and other non-alphanumeric

characters that are normally not allowed. Two types of delimited identifiers exist:

- Bracketed identifiers, which are delimited by square brackets (*[object name]*)
- Quoted identifiers, which are delimited by double quotation marks ("*object name*")

You can use bracketed identifiers in any environment, but to use quoted identifiers, you must enable a special option using SET QUOTED_IDENTIFIER ON. If you turn on QUOTED_IDENTIFIER, double quotes are interpreted as referencing an object. To delimit string or date constants, you must use single quotes.

Let's look at some examples. Because *column* is a reserved keyword, the first statement that follows is illegal in all circumstances. The second statement is illegal unless QUOTED_IDENTIFIER is on. The third statement is legal in any circumstance.

```
CREATE TABLE dbo.customer(name char(30), column  
char(12), emp_id char(4))
```

```
CREATE TABLE dbo.customer(name char(30), "column"  
char(12), emp_id char(4))
```

```
CREATE TABLE dbo.customer(name char(30), [column]  
char(12), emp_id char(4))
```

The SQL Native Client ODBC driver and SQL Native Client OLE DB Provider for SQL Server automatically set QUOTED_IDENTIFIER to ON when connecting. You can configure this in ODBC data sources,

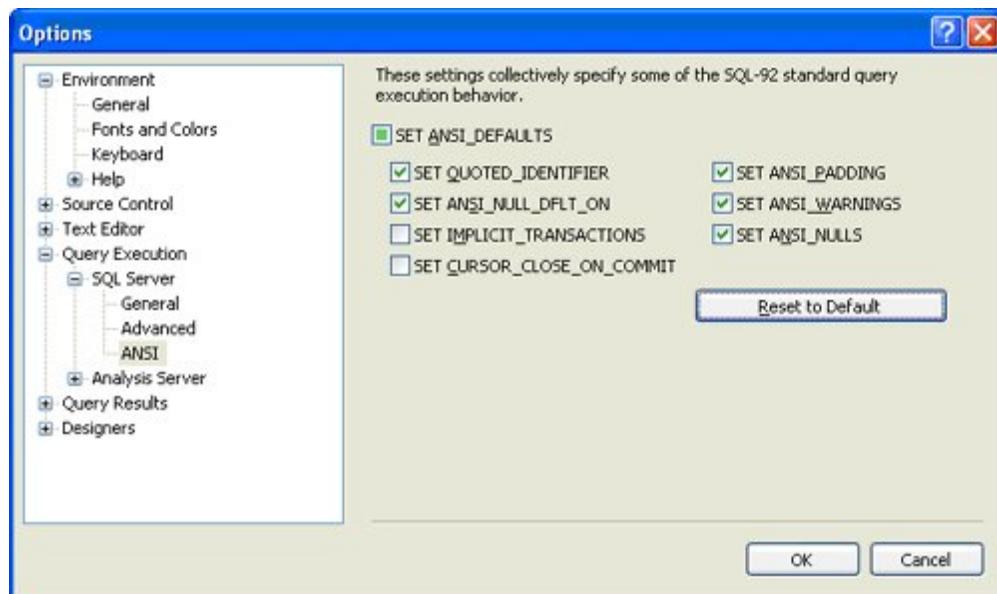
ODBC connection attributes, or OLE DB connection properties. You can determine whether this option is on or off for your session by executing the following query:

```
SELECT quoted_identifier  
FROM sys.dm_exec_sessions  
WHERE session_id = @@spid
```

A result value of 1 indicates that QUOTED_IDENTIFIER is ON. If you're using SQL Server Management Studio, you can check the setting by running the preceding command in a query window or by choosing Options from the Tools menu and then expanding the Query Execution/SQL Server node and examining the ANSI Properties information, as shown in [Figure 6-2](#).

Figure 6-2. Examining the ANSI properties for a connection in SQL Server Management Studio

[[View full size image](#)]



Theoretically, you can always use delimited identifiers with all object and column names, so you never have to worry about reserved keywords. However, I don't recommend this. Many third-party tools for SQL Server don't handle quoted identifiers well, and they can make your code difficult to read. Using quoted identifiers might also make upgrading to future versions of SQL Server more difficult.

Rather than using delimited identifiers to protect against reserved keyword problems, you should simply adopt some simple naming conventions. For example, you can precede column names with the first few letters of the table name and an underscore. This naming style makes the column or object name more readable and also greatly reduces your chances of encountering a keyword or reserved word conflict.

Naming Conventions

Many organizations and multi-user development projects adopt standard naming conventions. This is generally a good practice. For example, assigning a standard moniker of *cust_id* to represent a customer number in every table clearly shows that all the tables share common data. If an organization instead uses several monikers in the tables to represent a customer number, such as *cust_id*, *cust_num*, *customer_number*, and *customer_#*, it won't be as obvious that these monikers represent common data.

One naming convention is the Hungarian-style notation for column names. Hungarian-style notation is a widely used practice in C programming, whereby variable names include information about their data types. This notation uses names such as *sint_nn_custnum* to indicate that the *custnum* column is a small integer (*smallint* of 2 bytes) and is NOT NULL (doesn't allow nulls). Although this practice makes good sense in C programming, it defeats the data type independence that SQL Server provides; therefore, I recommend against using it. (SQL Server 2005 gives you the ability to create DDL triggers, so you can consider a trigger for CREATE TABLE that enforces your chosen naming conventions. DDL triggers are discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*.)

Data Types

SQL Server provides many data types, most of which are straightforward. Choosing the appropriate data type is simply a matter of mapping the domain of values you need to store to the corresponding data type. In choosing data types, you want to avoid wasting storage space while allowing enough space for a sufficient range of possible values over the life of your application. Some

problematic issues can arise when you work with a few of the available data types, and these issues are covered in detail in *Inside Microsoft SQL Server 2005: T-SQL Programming*. In this volume, I'll just cover some of the basic issues related to dealing with the various data types.

Choosing a Data Type

The decision about what data type to use for each column depends primarily on the nature of the data the column will hold and the operations you will want to perform on the data. The five basic data type categories in SQL Server 2005 are numeric, character, date and time, LOB (large object), and miscellaneous. SQL Server 2005 also supports a variant data type called *sql_variant*. Values stored in a *sql_variant* column can be of almost any data type. I'll discuss LOB and *sql_variant* columns later in this chapter when I discuss the internal storage of data on a page. *Inside Microsoft SQL Server 2005: T-SQL Programming* provides details and examples of working with many of the more interesting or possibly problematic data types. In this section, I'll examine some of the issues related to storing data of different data types.

Numeric Data Types

You should use numeric data types for data on which you want to perform numeric comparisons or arithmetic operations. Your main decisions are the maximum range of possible values you want to be able to store and the accuracy you need. The tradeoff is that data types that can store a greater range of values take up more space.

Numeric data types can also be classified as either exact or approximate. Exact numeric values are guaranteed to store exact representations of your numbers. Approximate numeric values have

a far greater range of values, but the values are not guaranteed to be stored precisely. The greatest range of values that exact numeric values can store data is $10^{38} + 1$ to $10^{38} 1$. Unless you need numbers with greater magnitude, I recommend that you not use the approximate numeric data types.

The exact numeric data types can be divided into two groups: integers and decimals. Integer types range in size from 1 to 8 bytes, with a corresponding increase in the range of possible values. The *money* and *smallmoney* data types are frequently included among the integer types because internally they are stored in the same way. For the *money* and *smallmoney* data types, it is just understood that the rightmost four digits are after the decimal point. For the other integer types, no digits come after the decimal point. [Table 6-1](#) lists the integer data types along with their storage size and range of values.

Table 6-1. SQL Server Integer Data Types

Data Type	Range	Storage (Bytes)
<i>bigint</i>	2^{63} to $2^{63} 1$	8
<i>int</i>	2^{31} to $2^{31} 1$	4
<i>Smallint</i>	2^{15} to $2^{15} 1$	2

Data Type	Range	Storage (Bytes)
<i>Tinyint</i>	0 to 255	1
<i>Money</i>	922,337,203,685,477.5808 8 to 922,337,203,685,477.5807, with accuracy of one ten- thousandth of a monetary unit	
<i>Smallmoney</i>	214,748.3648 to 214,748.3647, with accuracy of one ten- thousandth of a monetary unit	4

The decimal and numeric data types allow quite a high degree of accuracy as well as a large range of values. For those two synonymous data types, you can specify a *precision* (the total number of digits stored) and a *scale* (the maximum number of digits to the right of the decimal point). The maximum number of digits that can be stored to the left of the decimal point is (precision scale). Two different decimal values can have the same precision and very different ranges. For example, a column defined as decimal (8,4) can store values from 9999.9999 to 9999.9999, and a column

defined as decimal (8,0) can store values from 99,999,999 to 99,999,999.

[Table 6-2](#) shows the storage space required for decimal and numeric data based on the defined precision.

Table 6-2. SQL Server Decimal and Numeric Data Type Storage Requirements

Precision	Storage (Bytes)
1 to 9	5
10 to 19	9
20 to 28	13
29 to 38	17

Date and Time Data Types

SQL Server supports two data types for storing date and time information: *datetime* and *smalldatetime*. Again, the difference between these types is the range of possible dates and the number of bytes needed for storage. Both types have both a date and a time component. The range and storage requirements are shown in [Table 6-3](#). If no date is supplied, the default of January 1, 1900, is assumed; if no time is supplied, the default of 00:00:00.000 (midnight) is assumed. *Inside Microsoft SQL Server 2005: T-SQL Programming* addresses many issues related to how SQL Server interprets *datetime* values entered by an application and how you can control the formatting of the displayed *datetime* value.

Table 6-3. SQL Server Date and Time Data Types

Data Type	Range	Storage (Bytes)
<i>datetime</i>	January 1, 1753, through December 31, 9999, with an accuracy of three hundredths of a second	8
<i>smalldatetime</i>	January 1, 1900, through June 6, 2079, with an accuracy of 1 minute	4

Internally, *datetime* and *smalldatetime* values are stored completely differently from how you enter them or how they are displayed. They are stored as two separate components, a date component and a time component. The date is stored as the number of days before or after the base date of January 1, 1900. For *datetime* values, the time is stored as the number of clock ticks after midnight, with each tick representing 3.33 milliseconds, or 1/300 of a second. For *smalldatetime* values, the time is stored as the number of minutes after midnight. You can actually see these two parts if you convert a *datetime* value to a binary string of 8 hexadecimal bytes. The first 4 hexadecimal bytes are the number of days before or after the base date, and the second 4 bytes are the number of clock ticks after midnight. You can then convert these 4-byte hexadecimal strings to integers. I'll provide more details about hexadecimal strings later in the chapter.

The following example shows how to see the component parts of the current date and time by using the parameterless system function CURRENT_TIMESTAMP. The example stores the current date and time in a local variable so we can know we're using the same value for both computations.

```
DECLARE @today datetime  
SELECT @today = CURRENT_TIMESTAMP  
  
SELECT @today  
SELECT CONVERT (varbinary(8), @today)  
SELECT CONVERT (int, SUBSTRING (CONVERT  
(varbinary(8), @today), 1, 4))  
SELECT CONVERT (int, SUBSTRING (CONVERT  
(varbinary(8), @today), 5, 4))
```

Character Data Types

Character data types come in four varieties. They can be fixed-length or variable-length strings of single-byte characters (*char* and *varchar*) or fixed-length or variable-length strings of Unicode characters (*nchar* and *nvarchar*). Unicode character strings need two bytes for each stored character; use them when you need to represent characters that can't be stored in the single-byte characters that are sufficient for storing most of the characters in the English and European alphabets. Single-byte character strings can store up to 8,000 characters, and Unicode character strings can store up to 4,000 characters. You should know the type of data you'll be dealing with in order to decide between single-byte and double-byte character strings. Keep in mind that the catalog view *sys.types* reports length in number of bytes, not in number of characters. In SQL Server 2005, you can also define a variable-length character string with a MAX length. Columns defined as *varchar(max)* will be treated as normal variable-length columns when the actual length is less than or equal to 8,000 bytes, and they will be treated as a large object value (discussed later in this section) when the actual length is greater than 8,000 bytes.

Deciding whether to use a variable-length or a fixed-length data type is a more difficult decision, and it isn't always straightforward or obvious. As a general rule, variable-length data types are most appropriate when you expect significant variation in the size of the data for a column and when the data in the column won't be frequently changed.

Using variable-length data types can yield important storage savings. It can sometimes result in a minor performance loss, and at other times it can result in improved performance. A row with variable-length columns requires special offset entries in order to be internally maintained. These entries keep track of the actual length of the column. Calculating and maintaining the offsets requires slightly more overhead than does a pure fixed-length row, which needs no such offsets. This task requires a few addition and subtraction operations to maintain the offset value. However, the

extra overhead of maintaining these offsets is generally inconsequential, and this alone would not make a significant difference on most systems, if any.

Another potential performance issue with variable-length fields is the cost of increasing the size of a row on an almost full page. If a row with variable-length columns uses only part of its maximum length and is later updated to a longer length, the enlarged row might no longer fit on the same page. If the table has a clustered index, the row must stay in the same position relative to the other rows, so the solution is to split the page and move some of the rows from the page with the enlarged row onto a newly linked page. This can be an expensive operation. If the table has no clustered index, the row can move to a new location and leave a forwarding pointer in the original location. I'll talk about the details of page splitting and moving rows when I discuss data modification operations in [Chapter 7](#).

On the other hand, using variable-length columns can sometimes improve performance because it can allow more rows to fit on a page. But the efficiency results from more than simply requiring less disk space. A data page for SQL Server is 8 KB (8,192 bytes), of which 8,096 bytes are available to store data. (The rest is for internal use to keep track of structural information about the page and the object to which it belongs.) One I/O operation brings back the entire page. If you can fit 80 rows on a page, a single I/O operation brings back 80 rows. But if you can fit 160 rows on a page, one I/O operation is essentially twice as efficient. In operations that scan for data and return lots of adjacent rows, this can amount to a significant performance improvement. The more rows you can fit per page, the better your I/O and cache-hit efficiency will be.

For example, consider a simple customer table. Suppose that you could define it in two ways, fixed length and variable length, as shown in [Figures 6-3](#) and [6-4](#).

Figure 6-3. A customer table with all fixed-length columns

```
USE testdb
GO

CREATE TABLE customer_fixed
(
    cust_id           smallint
    NULL,
    cust_name         char(50)
    NULL,
    cust_addr1        char(50)
    NULL,
    cust_addr2        char(50)
    NULL,
    cust_city          char(50)
    NULL,
    cust_state         char(2)
    NULL,
    cust_postal_code   char(10)
    NULL,
    cust_phone         char(20)
    NULL,
    cust_fax           char(20)
    NULL,
    cust_email          char(30)
    NULL,
    cust_web_url        char(100)
    NULL,
)
```

Figure 6-4. A customer table with variable-length columns

[View full width]

```
USE testdb
GO

CREATE TABLE customer_var
(
    cust_id           smallint
    NULL,
    cust_name         varchar(50)
    NULL,
    cust_addr1        varchar(50)
    NULL,
    cust_addr2        varchar(50)
    NULL,
    cust_city          varchar(50)
    NULL,
    cust_state         char(2)
```

```

    NULL,
cust_postal_code      varchar(10)
    NULL,
cust_phone           varchar(20)
    NULL,
cust_fax             varchar(20)
    NULL,
cust_email            varchar(30)
    NULL,
cust_web_url          varchar(100)
    NULL
)

```

Columns that contain addresses, names, or URLs all have data that varies significantly in length. Let's look at the differences between choosing fixed-length columns vs. choosing variable-length columns. In [Figure 6-3](#), which uses all fixed-length columns, every row uses 384 bytes for data regardless of the number of characters actually inserted in the row. SQL Server also needs an additional 10 bytes of overhead for every row in this table, so each row needs a total of 394 bytes for storage. But let's say that even though the table must accommodate addresses and names up to the specified size, the average row is only half the maximum size.

In [Figure 6-4](#), assume that for all the variable-length (varchar) columns, the average entry is actually only about half the maximum. Instead of a row length of 394 bytes, the average length is 224 bytes. This length is computed as follows: The *smallint* and *char(2)*

columns total 4 bytes. The *varchar* columns' maximum total length is 380, half of which is 190 bytes. And a 2-byte overhead exists for each of nine *varchar* columns, for 18 bytes. Add 2 more bytes for any row that has one or more variable-length columns. In addition, these rows require the same 10 bytes of overhead that the fixed-length rows from [Figure 6-3](#) require, regardless of the presence of variable-length fields. So the total is $4 + 190 + 18 + 2 + 10$, or 224. (I'll discuss the actual meaning of each of these bytes of overhead later in this chapter.)

In the fixed-length example in [Figure 6-3](#), you always fit 20 rows on a data page (8,096/394, discarding the remainder). In the variable-length example in [Figure 6-4](#), you can fit an average of 36 rows per page (8,096/224). The table using variable-length columns will consume about half as many pages in storage, a single I/O operation will retrieve almost twice as many rows, and a page cached in memory is twice as likely to contain the row you're looking for.

Note



Additional overhead bytes are needed for each row if you are using snapshot isolation. I'll discuss this concurrency option as well as the extra row overhead needed to support it in [Chapter 8](#).

When you choose lengths for columns, don't be wasteful but don't be cheap, either. Allow for future needs, and realize that if the additional length doesn't change how many rows will fit on a page,

the additional size is free anyway. Consider again the examples in [Figures 6-3](#) and [6-4](#). The *cust_id* is declared as a *smallint*, meaning that its maximum positive value is 32,767 (unfortunately, SQL Server doesn't provide any unsigned *int* or unsigned *smallint* data types), and it consumes 2 bytes of storage. Although 32,767 customers might seem like a lot to a new company, the company might be surprised by its own success and, in a couple of years, find out that 32,767 is too limited.

The database designers might regret that they tried to save 2 bytes and didn't simply make the data type an *int*, using 4 bytes but with a maximum positive value of 2,147,483,647. They'll be especially disappointed if they realize they didn't really save any space. If you compute the rows-per-page calculations just discussed, increasing the row size by 2 bytes, you'll see that the same number of rows still fit on a page. The additional 2 bytes are free—they were simply wasted space before. They never cause fewer rows per page in the fixed-length example, and they'll rarely cause fewer rows per page even in the variable-length case.

So which strategy wins? Potentially better update performance? Or more rows per page? Like most questions of this nature, no one answer is right. It depends on your application. If you understand the tradeoffs, you'll be able to make the best choice. Now that you know the issues, this general rule merits repeating: Variable-length data types are most appropriate when you expect significant variation in the size of the data for that column and when the column won't be updated frequently.

Miscellaneous Data Types

I'll end this part of the discussion by showing you a few additional data types that you might have use for.

Binary Data Types

These data types are *binary* and *varbinary*. They are used to store strings of bits, and the values are entered and displayed using their hexadecimal (hex) representation, which is indicated by a prefix of 0x. So a hex value of 0x270F corresponds to a decimal value of 9999 and a bit string of 10011100001111. In hex, each two displayed characters represent a byte, so the value of 0x270F represents 2 bytes. You need to decide whether you want your data to be fixed or variable length, and you can use some of the same considerations we discussed for deciding between *char* and *varchar* to make your decision. The maximum length of *binary* or *varbinary* data is 8,000 bytes.

bit Data Type

The *bit* data type can store a 0 or a 1 and can consume only a single bit of storage space. However, if there is only one bit column in a table, it will take up a whole byte. Up to 8 bit columns are stored in a single byte.

Large Object Data Types

These data types are *text*, *ntext*, and *image*. The *text* data type can store up to 2^{31} 1 non-Unicode characters, *ntext* can store up to 2^{30} 1 (half as many) Unicode characters, and *image* can store up to 2^{31} 1 bytes of binary data. We'll look at the storage mechanisms used for these large object (LOB) data types later in the chapter.

cursor Data Type

The *cursor* data type can hold a reference to a cursor. Although you can't declare a column in a table to be of type *cursor*, this data type can be used for output parameters and local variables. I've included the *cursor* data type in this list for completeness, but I won't be talking more about it. Cursors are discussed in detail in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

***rowversion* Data Type**

The *rowversion* data type is a synonym for what was formerly called a *timestamp*. When using the *timestamp* data type name, many people might assume that the data has something to do with dates or times, but it doesn't. A column of type *rowversion* holds an internal sequence number that SQL Server automatically updates every time the row is modified. The value of any *rowversion* column is actually unique within an entire database, and a table can have only one column of type *rowversion*. Any operation that modifies any *rowversion* column in the database generates the next sequential value. The actual value stored in a *rowversion* column is seldom important by itself. The column is used to detect whether a row has been modified since the last time it was accessed, by finding out whether the *rowversion* value has changed.

***sql_variant* Data Type**

The *sql_variant* data type allows a column to hold values of any data type except *text*, *ntext*, *image*, *XML*, user-defined data types, variable-length data types with the MAX specifier, or *rowversion* (*timestamp*). I'll describe the internal storage of *sql_variant* data later in this chapter, and you can see some examples of using *sql_variant* data in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

***table* Data Type**

The *table* data type can be used to store the result of a function and can be used as the data type of local variables. Columns in tables cannot be of type *table*. Table variables are discussed in detail in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

***xml* Data Type**

The *xml* datatype lets you store XML documents and fragments in a SQL Server database. You can use the *xml* datatype as a column type when you create a table, or as the data type for variables, parameters, and the return value of a function. XML data has its own methods for retrieval and manipulation, and these are discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*. I will not be covering details of working with XML data in this volume.

***uniqueidentifier* Data Type**

The *uniqueidentifier* data type is sometimes referred to as a globally unique identifier (GUID) or universal unique identifier (UUID). A GUID or UUID is a 128-bit (16-byte) value generated in a way that, for all practical purposes, guarantees uniqueness worldwide, even among unconnected computers. It is becoming an important way to identify data, objects, software applications, and applets in distributed systems. Because I don't talk about *uniqueidentifier* data anywhere else in any of the *Inside Microsoft SQL Server 2005* volumes, I'll give you a bit more detail about it here.

The Transact-SQL language supports the system functions NEWID and NEWSEQUENTIALID (which is new in SQL Server 2005), which you can use to generate *uniqueidentifier* values. A column or

variable of data type *uniqueidentifier* can be initialized to a value in one of the following two ways:

- Using the system-supplied function NEWID or NEWSEQUENTIALID as a default value.
- Using a string constant in the following form (32 hexadecimal digits separated by hyphens): xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. (Each x is a hexadecimal digit in the range 0 through 9 or a through f.)

This data type can be quite cumbersome to work with, and the only operations that are allowed against a *uniqueidentifier* value are comparisons (=, <>, <, >, <=, >=) and checking for NULL. However, using this data type internally can offer several advantages.

One reason to use the *uniqueidentifier* data type is that the values generated by NEWID or NEWSEQUENTIALID are guaranteed to be globally unique for any machine on a network because the last 6 bytes of a *uniqueidentifier* value make up the node number for the machine. When the SQL Server machine does not have an Ethernet/TOKEN Ring (IEEE 802.x) address, there is no node number and the generated GUID is guaranteed to be unique among all GUIDs generated on that computer. However, the possibility exists that another computer without an Ethernet/TOKEN Ring address will generate the identical GUID. The GUIDs generated on computers with network addresses are guaranteed to be globally unique.

The primary reason that SQL Server needed a way to generate a GUID was for use in merge replication, in which identifier values for the same table could be generated on any one of many different SQL Server machines. There needed to be a way to determine whether two rows really were the same row and there had to be no way that two rows not referring to the same entity would have the same identifier. Using GUID values provides that functionality. Two

rows with the same GUID value must indicate that they really are the same row.

The difference between the NEWSEQUENTIALID and the NEWID functions is that NEWSEQUENTIALID creates a GUID that is greater than any GUID previously generated by this function on a specified computer and can be used to introduce a sequence to your GUID values. This turns out to greatly increase the scalability of systems using merge replication. If the *uniqueidentifier* values are being used as the clustered key for the replicated tables, the new rows are then inserted in random disk pages. (You'll see the details in [Chapter 7](#), when I discuss clustered indexes in detail.) If the machines involved are performing a large amount of I/O operations, the nonsequential GUID generated by the NEWID function results in lots of random B-tree lookups and inefficient insert operations. The new function, NEWSEQUENTIALID, which is a wrapper around the Windows function *UuidCreateSequential*, does some byte scrambling and creates an ordering to the generated UUID values.

The list of *uniqueidentifier* values can't be exhausted. This is not the case with other data types frequently used as unique identifiers. In fact, SQL Server uses this data type internally for row-level merge replication. A *uniqueidentifier* column can have a special property called the ROWGUIDCOL property; at most, one *uniqueidentifier* column can have this property per table. The ROWGUIDCOL property can be specified as part of the column definition in CREATE TABLE and ALTER TABLE ADD *column*, or it can be added or dropped for an existing column using ALTER TABLE ALTER COLUMN.

You can reference a *uniqueidentifier* column with the ROWGUIDCOL property using the keyword ROWGUIDCOL in a query. This is similar to referencing an identity column using the IDENTITYCOL keyword. The ROWGUIDCOL property does not imply any automatic value generation, and if automatic value generation is needed, the NEWID function should be defined as the

default value of the column. You can have multiple *uniqueidentifier* columns per table, but only one of them can have the ROWGUIDCOL property. You can use the *uniqueidentifier* data type for whatever reason you come up with, but if you're using one to identify the current row, an application must have a generic way to ask for it without needing to know the column name. That's what the ROWGUIDCOL property does.

Much Ado About NULL

The issue of whether to allow NULL has become an almost religious one for many in the industry, and no doubt the discussion here will outrage a few people. However, my intention isn't to engage in a philosophical debate. Pragmatically, dealing with NULL brings added complexity to the storage engine because SQL Server keeps a special bitmap in every row to indicate which nullable columns actually are NULL. If NULLs are allowed, SQL Server must decode this bitmap for every row accessed. Allowing NULL also adds complexity in application code, which can often lead to bugs. You must always add special logic to account for the case of NULL.

As the database designer, you might understand the nuances of NULL and three-valued logic in aggregate functions when you do joins and when you search by values. In addition, you must also consider whether your development staff really understands how to work with NULLs. I recommend, if possible, that you use all NOT NULL columns and define *default* values for missing or unknown entries (and possibly make such character columns *varchar* if the default value is significantly different in size from the typical entered value).

In any case, it's good practice to explicitly declare NOT NULL or NULL when you create a table. If no such declaration exists, SQL Server assumes NOT NULL. (In other words, no NULLs are

allowed.) This might surprise many people who assume that the default for SQL Server is to allow NULLs. The reason for this misconception is that most of the tools and interfaces for working with SQL Server enable a session setting that makes it the default to allow NULLs. However, you can set the default to allow NULLs by using a session setting or a database option, which, as I just mentioned, is what most tools and interfaces already do. If you script your DDL and then run it against another server that has a different default setting, you'll get different results if you don't explicitly declare NULL or NOT NULL in the column definition.

Several database options and session settings can control SQL Server's behavior regarding NULL values. You can set database options using the ALTER DATABASE command, as I showed you in [Chapter 4](#). And you can enable session settings for one connection at a time using the SET command.

Note



The database option *ANSI null default* corresponds to the two session settings `ANSI_NULL_DFLT_ON` and `ANSI_NULL_DFLT_OFF`. When the *ANSI null default* database option is false (the default setting for SQL Server), new columns created with the `ALTER TABLE` and `CREATE TABLE` statements are, by default, NOT NULL if the nullability status of the column isn't explicitly specified. `SET ANSI_NULL_DFLT_OFF` and `SET ANSI_NULL_DFLT_ON` are mutually exclusive options that indicate whether the database option should be overridden. When on, each option forces the opposite option off. Neither option,

when off, turns the opposite option on it only discontinues the current on setting.

You use the function GETANSINULL to determine the default nullability for your current session. This function returns 1 when new columns allow null values and the column or data type nullability wasn't explicitly defined when the table was created or altered. I strongly recommend declaring NULL or NOT NULL explicitly when you create a column. This removes all ambiguity and ensures that you're in control of how the table will be built, regardless of the default nullability setting.

The database option *concat null yields null* corresponds to the session setting SET CONCAT_NULL_YIELDS_NULL. When CONCAT_NULL_YIELDS_NULL is on, concatenating a NULL value with a string yields a NULL result. For example, SELECT 'abc' + NULL yields NULL. When SET CONCAT_NULL_YIELDS_NULL is off, concatenating a NULL value with a string yields the string itself. In other words, the NULL value is treated as an empty string. For example, SELECT 'abc' + NULL yields abc. If the session-level setting isn't specified, the value of the database option *concat null yields null* applies.

The database option *ANSI nulls* corresponds to the session setting SET ANSI_NULLS. When this option is set to true, all comparisons to a null value evaluate to FALSE. When it is set to false, comparisons of non-Unicode values to a null value evaluate to TRUE if both values are NULL. In addition, when this option is set to true, your code must use the condition IS NULL to determine whether a column has a NULL value. When this option is set to

false, SQL Server allows = NULL as a synonym for IS NULL and <> NULL as a synonym for IS NOT NULL.

You can see this behavior yourself by looking at the *titles* table in the *pubs* database. The *titles* table has two rows with a NULL price. The first batch of statements that follows, when executed from a query window, should return two rows, and the second batch should return no rows.

```
-- First batch will return 2 rows
USE pubs
SET ANSI_NULLS OFF
GO
SELECT * FROM titles WHERE price = NULL
GO
-- Second batch will return no rows
USE pubs
SET ANSI_NULLS ON
GO
SELECT * FROM titles WHERE price = NULL
GO
```

A fourth session setting is ANSI_DEFAULTS. Setting this to ON is a shortcut for enabling both ANSI_NULLS and ANSI_NULL_DFLT_ON as well as other session settings not related to NULL handling. The SQL Server ODBC driver and the SQL Server OLE DB provider automatically set ANSI_DEFAULTS to ON. You can change the ANSI_NULLS setting when you define your data source name (DSN). You should be aware that the tool you are using to connect to SQL Server might set certain options ON or OFF. To see which options are enabled in your current connection, you can run the following command:

```
DBCC USEROPTIONS
```

Here's a sample of the output it might return:

Set Option	Value
<hr/>	
<hr/>	
textsize	64512
language	us_english
dateformat	mdy
datefirst	7
ansi_null_dflt_on	SET
ansi_warnings	SET
ansi_padding	SET
ansi_nulls	SET
concat_null_yields_null	SET

Warning



The SET options always overrides the corresponding database options, whether the SET option has been turned ON or OFF. The only time the database option applies is when the SET option has not been touched. So you can think of the ANSI_NULLS and CONCAT_NULL_YIELDS_NULL options as having three possible values: ON, OFF, and "never been set." If the SET option is either ON or OFF, the database option is ignored. Because the SQL Native Client ODBC driver and SQL

Native Client OLE DB Provider for SQL Server automatically set both of these options to ON, the database options are never taken into account.

I recommend that you try to write your queries so it doesn't matter which option is in effect. When that is impossible, you should rely on the session-level option only and not the database-level option.

Note that the DBCC USEROPTIONS command only shows options that have been explicitly set in your connection. Any option that just relies on the default will not be shown. In addition, you cannot see the session setting for any other connections. In SQL Server 2005, an alternative is to use the *sys.dm_exec_sessions* dynamic management view, to see all the session option values.

The following query shows the values for your current session, but if you have VIEW SERVER STATE permission you can change or remove the WHERE clause to return information about other sessions:

```
SELECT * FROM sys.dm_exec_sessions  
WHERE session_id = @@spid
```

As you can see, you can configure and control the treatment and behavior of NULL values in several ways, and you might think it would be impossible to keep track of all the variations. If you try to

control every aspect of NULL handling separately within each individual session, you can cause immeasurable confusion and even grief. However, most of the issues become moot if you follow a few basic recommendations:

- Never allow NULL values in your tables.
- Include a specific NOT NULL qualification in your table definitions.
- Don't rely on database properties to control the behavior of NULL values.

If you must use NULLs in some cases, you can minimize problems by always following the same rules, and the easiest rules to follow are the ones that ANSI already specifies.

User-Defined Data Types

A user-defined data type (UDT) provides a convenient way for you to guarantee consistent use of underlying native data types for columns known to have the same domain of possible values. For example, perhaps your database will store various phone numbers in many tables. Although no single, definitive way exists to store phone numbers, in this database consistency is important. You can create a *phone_number* UDT and use it consistently for any column in any table that keeps track of phone numbers to ensure that they all use the same data type. Here's how to create this UDT:

```
CREATE TYPE phone_number FROM varchar(20) NOT  
NULL;
```

And here's how to use the new UDT when you create a table:

```
CREATE TABLE customer  
(  
    cust_id          smallint      NOT NULL,  
    cust_name        varchar(50)   NOT NULL,  
    cust_addr1       varchar(50)   NOT NULL,  
    cust_addr2       varchar(50)   NOT NULL,  
    cust_city        varchar(50)   NOT NULL,  
    cust_state       char(2)       NOT NULL,  
    cust_postal_code varchar(10)   NOT NULL,  
    cust_phone       phone_number NOT NULL,  
    cust_fax         varchar(20)   NOT NULL,  
    cust_email       varchar(30)   NOT NULL,  
    cust_web_url    varchar(100)  NOT NULL  
)
```

When the table is created, internally the *cust_phone* data type is known to be *varchar(20)*. Notice that both *cust_phone* and *cust_fax* are *varchar(20)*, although *cust_phone* has that declaration through its definition as a UDT.

Information about the columns in your tables is available through the catalog view *sys.columns*, which we'll look at in more detail in the section on internal storage later in this chapter. For now, we'll just look at a basic query to show us two columns in *sys.columns*, one containing a number representing the underlying system data type and one containing a number representing the data type used when creating the table. The following query selects all the rows from *sys.columns* and displays the *column_id*, the column name, the data type values, and the maximum length, and the results are shown immediately after:

```
SELECT column_id, name, system_type_id,
user_type_id,
      type_name(user_type_id) as user_type_name,
max_length
FROM sys.columns
WHERE object_id=object_id('customer');
```

column_id	type_name	system_type_id	
user_type_id	user_type_name	max_length	
1	<i>cust_id</i>	52	52
<i>smallint</i>	2		
2	<i>cust_name</i>	167	167
<i>varchar</i>	50		
3	<i>cust_addr1</i>	167	167
<i>varchar</i>	50		
4	<i>cust_addr2</i>	167	167

varchar	5	50	
		cust_city	167
varchar	6	50	
		cust_state	175
char	7	2	
		cust_postal_code	167
varchar	8	10	
		cust_phone	167
phone_number	9	20	
		cust_fax	167
varchar	10	20	
		cust_email	167
varchar	11	30	
		cust_web_url	167
varchar		100	

You can see that both the *cust_phone* and *cust_fax* columns have the same *system_type_id* value, although the *cust_phone* column shows that the *user_type_id* is a UDT (*user_type_id* = 257). The type is resolved when the table is created, and the UDT can't be dropped or changed as long as one or more tables are currently using it. Once declared, a UDT is static and immutable, so no inherent performance penalty occurs in using a UDT instead of the native data type.

The use of UDTs can make your database more consistent and clear. SQL Server implicitly converts between compatible columns of different types (either native types or UDTs of different types).

Currently, UDTs don't support the notion of subtyping or inheritance, nor do they allow a DEFAULT value or a CHECK constraint to be declared as part of the UDT itself. These powerful object-oriented concepts will likely make their way into future versions of SQL

Server. These limitations notwithstanding, UDT functionality is a dynamic and often underused feature of SQL Server.

CLR Data Types

SQL Server 2005 introduces the ability to integrate .NET programming into SQL Server objects, allowing you to create such CLR-based objects such as stored procedures, functions, and triggers using any .NET language. You can even define your data types with .NET, which allows you to have a UDT that is more than just a new name for a preexisting construct. *Inside Microsoft SQL Server 2005: T-SQL Programming* has an extensive discussion of creating CLR-based UDTs, including several code examples, so I will refer you to that volume for that discussion.

IDENTITY Property

It is common to provide simple counter-type values for tables that don't have a natural or efficient primary key. Columns such as *cust_id* are usually simple counter fields. The IDENTITY property makes generating unique numeric values easy. IDENTITY isn't a data type; it's a *column property* that you can declare on a whole-number data type such as *tinyint*, *smallint*, *int*, *bigint*, or *numeric/decimal* (with which only a scale of zero makes any sense). Each table can have only one column with the IDENTITY property. The table's creator can specify the starting number (seed) and the amount that this value increments or decrements. If not otherwise specified, the seed value starts at 1 and increments by 1, as shown in this example:

```
CREATE TABLE customer
(
    cust_id      smallint      IDENTITY NOT NULL,
    cust_name    varchar(50)   NOT NULL
)
```

To find out which seed and increment values were defined for a table, you can use the *IDENT_SEED(tablename)* and *IDENT_INCR(tablename)* functions. Take a look at this statement:

```
SELECT IDENT_SEED('customer'),
IDENT_INCR('customer')
```

It produces the following result for the *customer* table because values weren't explicitly declared and the default values were used.

This next example explicitly starts the numbering at 100 (seed) and increments the value by 20:

```
CREATE TABLE customer
(
    cust_id      smallint      IDENTITY(100, 20) NOT
    NULL,
    cust_name    varchar(50)   NOT NULL
)
```

The value automatically produced with the IDENTITY property is normally unique, but that isn't guaranteed by the IDENTITY property itself, nor are the IDENTITY values guaranteed to be consecutive. (I will expand on the issues of non-unique and non-consecutive IDENTITY values later in this section.) For efficiency, a value is considered used as soon as it is presented to a client doing an INSERT operation. If that client doesn't ultimately commit the INSERT, the value never appears, so a break occurs in the consecutive numbers. An unacceptable level of serialization would exist if the next number couldn't be parceled out until the previous one was actually committed or rolled back. (And even then, as soon as a row was deleted, the values would no longer be consecutive. Gaps are inevitable.)

Note



If you need exact sequential values without gaps, IDENTITY isn't the

appropriate feature to use. Instead, you should implement a *next_number*-type table in which you can make the operation of bumping the number contained within it part of the larger transaction (and incur the serialization of queuing for this value).

To temporarily disable the automatic generation of values in an identity column, you use the `SET IDENTITY_INSERT tablename ON` option. In addition to filling in gaps in the identity sequence, this option is useful for tasks such as bulk-loading data in which the previous values already exist. For example, perhaps you're loading a new database with customer data from your previous system. You might want to preserve the previous customer numbers but have new ones automatically assigned using `IDENTITY`. The `SET` option was created exactly for cases like this.

Because the `SET` option allows you to determine your own values for an `IDENTITY` column, the `IDENTITY` property alone doesn't enforce uniqueness of a value within the table. Although `IDENTITY` generates a unique number if `IDENTITY_INSERT` has never been enabled, the uniqueness is not guaranteed once you have used the `SET` option. To enforce uniqueness (which you'll almost always want to do when using `IDENTITY`), you should also declare a `UNIQUE` or `PRIMARY KEY` constraint on the column. If you insert your own values for an identity column (using `SET IDENTITY_INSERT`), when automatic generation resumes, the next value is the next incremented value (or decremented value) of the highest value that exists in the table, whether it was generated previously or explicitly inserted.

Tip



If you use the bcp utility for bulk loading data, be aware of the *-E* (uppercase) parameter if your data already has assigned values that you want to keep for a column that has the IDENTITY property. You can also use the Transact-SQL BULK INSERT command with the KEEPIDENTITY option. For more information, see the SQL Server documentation for bcp and BULK INSERT.

The keyword IDENTITYCOL automatically refers to the specific column in a table that has the IDENTITY property, whatever its name. If that column is *cust_id*, you can refer to the column as IDENTITYCOL without knowing or using the column name or you can refer to it explicitly as *cust_id*. For example, the following two statements work identically and return the same data:

```
SELECT IDENTITYCOL FROM customer  
SELECT cust_id FROM customer
```

The column name returned to the caller is *cust_id*, not IDENTITYCOL, in both cases.

When inserting rows, you must omit an identity column from the column list and VALUES section. (The only exception is when the IDENTITY_INSERT option is on.) If you do supply a column list, you must omit the column for which the value will be automatically supplied. Here are two valid INSERT statements for the *customer* table shown earlier:

```
INSERT customer VALUES ('ACME Widgets')
INSERT customer (cust_name) VALUES ('AAA Gadgets')
```

Selecting these two rows produces this output:

<code>cust_id</code>	<code>cust_name</code>
1	ACME Widgets
2	AAA Gadgets

(2 row(s) affected)

In applications, it's sometimes desirable to immediately know the value produced by IDENTITY for subsequent use. For example, a transaction might first add a new customer and then add an order for that customer. To add the order, you probably need to use the *cust_id*. Rather than selecting the value from the *customer* table, you can simply select the special system function @@IDENTITY, which contains the last identity value used by that connection. It doesn't necessarily provide the last value inserted in the table, however, because another user might have subsequently inserted data. If multiple INSERT statements are carried out in a batch on the same or different tables, the variable has the value for the last statement only. In addition, if there is an INSERT trigger that fires after you insert the new row and if that trigger inserts rows into a

table with an identity column, @@IDENTITY will not have the value inserted by the original INSERT statement. To you, it might look like you're inserting and then immediately checking the value:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets')
SELECT @@IDENTITY
```

However, if a trigger was fired for the INSERT, the value of @@IDENTITY might have changed.

There are two other functions that you might find useful when working with identity columns: SCOPE_IDENTITY and IDENT_CURRENT. SCOPE_IDENTITY returns the last identity value inserted into a table in the same scope, which could be a stored procedure, trigger, or batch. So if we replace @@IDENTITY with the SCOPE_IDENTITY function in the preceding code snippet, we can see the identity value inserted into the *customer* table. If an INSERT trigger also inserted a row that contained an identity column, it would be in a different scope:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets')
SELECT SCOPE_IDENTITY()
```

In other cases, you might want to know the last identity value inserted in a specific table from any application or user. You can get this value using the IDENT_CURRENT function, which takes a table name as an argument:

```
SELECT IDENT_CURRENT('customer')
```

This doesn't always guarantee that you can predict the next identity value to be inserted because another process could insert a row between the time you check the value of IDENT_CURRENT and the time you execute your INSERT statement.

You can't define the IDENTITY property as part of a UDT, but you can declare the IDENTITY property on a column that uses a UDT. A column that has the IDENTITY property must always be declared NOT NULL (either explicitly or implicitly); otherwise, error number 8147 will result from the CREATE TABLE statement and CREATE won't succeed. Likewise, you can't declare the IDENTITY property and a DEFAULT on the same column. To check that the current identity value is valid based on the current maximum values in the table, and to reset it if an invalid value is found (which should never be the case), use the DBCC CHECKIDENT(*tablename*) statement.

Identity values are fully recoverable. If a system outage occurs while insert activity is taking place with tables that have identity columns, the correct value is recovered when SQL Server is restarted. SQL Server does this during the checkpoint processing by flushing the current identity value for all tables. For activity beyond the last checkpoint, subsequent values are reconstructed from the transaction log during the standard database recovery process. Any inserts into a table that have the IDENTITY property are known to have changed the value, and the current value is retrieved from the last INSERT statement (post-checkpoint) for each table in the transaction log. The net result is that when the database is recovered, the correct current identity value is also recovered.

In rare cases, the identity value can get out of sync. If this happens, you can use the DBCC CHECKIDENT command to reset the identity value to the appropriate number. In addition, the RESEED option to this command allows you to set a new starting value for the identity sequence. See the online documentation for complete details.

Internal Storage

This section covers system catalogs and the internal data storage of tables. Although you can use SQL Server effectively without understanding the internals, understanding the details of how SQL Server stores data will help you develop efficient applications.

When you create a table, one or more rows are inserted into a number of system tables to manage that table. At a minimum, you can see metadata for your new table in the *sys.tables*, *sys.indexes*, and *sys.columns* catalog views. When you define the new table with one or more constraints, you'll also be able to see information in the *sys.check_constraints*, *sys.default_constraints*, *sys.key_constraints*, or *sys.foreign_keys* views. For every table created, a single row that contains among other things the name, object ID, and ID of the schema containing the new table is available through the *sys.tables* view. Remember that the *sys.tables* view inherits all the columns from *sys.objects* (which shows information relevant to all types of objects) and then includes additional columns pertaining only to tables. The *sys.columns* view shows you one row for each column in the new table, and each row will contain information such as the column name, data type, and length. Each column receives a column ID, which initially corresponds to the order in which you specified the columns when you created the table—that is, the first column listed in the CREATE TABLE statement will have a column ID of 1, the second column will have a column ID of 2, and so on. [Figure 6-5](#) shows the rows returned by the *sys.tables* and *sys.columns* views when you create a table. (Not all columns are shown for each view.)

Figure 6-5. Basic catalog information stored after a table is created

[\[View full width\]](#)

```
CREATE TABLE dbo.employee (
    emp_lname      varchar(15)
NOT NULL,
    emp_fname      varchar(10)
NOT NULL,
    address        varchar(30)
NOT NULL,
    phone          char(12)
NOT NULL,
    job_level      smallint
NOT NULL
)
```

sys.tables	object_id	name
schema_id	type_desc	
---	-----	-----
---	-----	-----
1	917578307	employee

1

USER_TABLE

sys.columns	object_id	column_id
name	system_type_id	max_length
-----	-----	-----
-----	-----	-----
917578307	1	

emp_lname	167	15
	917578307	2
emp_fname	167	10
	917578307	3
address		
	167	30
	917578307	4
phone		
	175	12
	917578307	5
job_level	52	2

Note



There can be gaps in the column ID sequence if the table is altered to drop columns. However, the information schema view gives you a value called ORDINAL_POSITION because that is what the SQL standard demands. The ordinal position is the order the column will be listed when you `SELECT *` on the table. So the `column_id` is not necessarily the ordinal position of that column.

The `sys.indexes` Catalog View

In addition to `sys.columns` and `sys.tables`, the `sys.indexes` view returns at least one row for each table. In versions of SQL Server prior to SQL Server 2005, the `sysindexes` table contains all the physical storage information for both tables and indexes, which are the only objects that actually use storage space. The `sysindexes` table has columns to keep track of the space used by all tables and indexes, the physical location of each index root page, and the first page of each table and index. (In [Chapter 7](#), I'll tell you more about root pages and what the "first" page actually means.) In SQL Server 2005, the compatibility view `sys.sysindexes` contains much of the same information, but it is incomplete because of changes in the storage organization in SQL Server 2005. The catalog view `sys.indexes` contains only basic property information about indexes, such as whether the index is clustered or nonclustered, unique or non-unique, and other properties, which I discuss in [Chapter 7](#). To get all the storage information in SQL Server 2005 that previous versions provided in the `sysindexes` table, we have to look at two other catalog views in addition to `sys.indexes`: `sys.partitions` and `sys.allocation_units` (or alternatively, the undocumented `sys.system_internals_allocation_units`). I'll discuss the basic contents of these views shortly, but first let's focus on `sys.indexes`.

You might be aware that if a table has a clustered index, the table's data is actually considered part of the index, so the data rows are actually index rows. For a table with a clustered index, SQL Server has a row in `sys.indexes` with an `index_id` value of 1 and the name column in `sys.indexes` contains the name of the index. The name of the table that is associated with the index can be determined from the `object_id` column in `sys.indexes`. If a table has no clustered index, there is no organization to the data itself, and we call such a

table a *heap*. A heap in *sys.indexes* table has an *index_id* value of 0, and the name column contains NULL. Every additional index has a row in *sys.indexes* with an *indid* (index ID) value between 2 and 250. Because there can be as many as 249 non-clustered indexes on a single table and there is one row for the heap or clustered index, every table has between 1 and 250 rows in the *sys.indexes* view for relational indexes. A table can have additional rows for XML indexes. Metadata for XML indexes is available in the *sys.xml_indexes* catalog view, which inherits columns from the *sys.indexes* view.

In SQL Server 2000, each row in the *sysindexes* table contains information about how much space that table or index takes up, and also information about where the pages for that structure can be found. LOB data stored in a table is stored in a separate location from the other table data, and it has its own row in *sysindexes* to reflect the size of the LOB data and its location. A row with an *indid* value of 255 is always used to keep track of storage information for LOB data.

Two main changes in SQL Server 2005 make it no longer efficient to store all storage information in a single table. First, SQL Server 2005 adds the ability to store a table or index on multiple partitions, so the space used by each partition, as well as its location, must be kept track of separately. Second, LOB data is now much more flexible. Not only is there an additional kind of LOB data called *row-overflow data*, but unlike in SQL Server 2000, LOB data can now be a part of an index. No longer can one extra row in *sysindexes* be sufficient for keeping track of LOB data if it occurs in multiple indexes and in two different formats. I'll discuss partitioning tables and indexes in [Chapter 7](#), and I'll discuss the storage of row-overflow data, along with the storage of LOB data, later in this chapter.

Data Storage Metadata

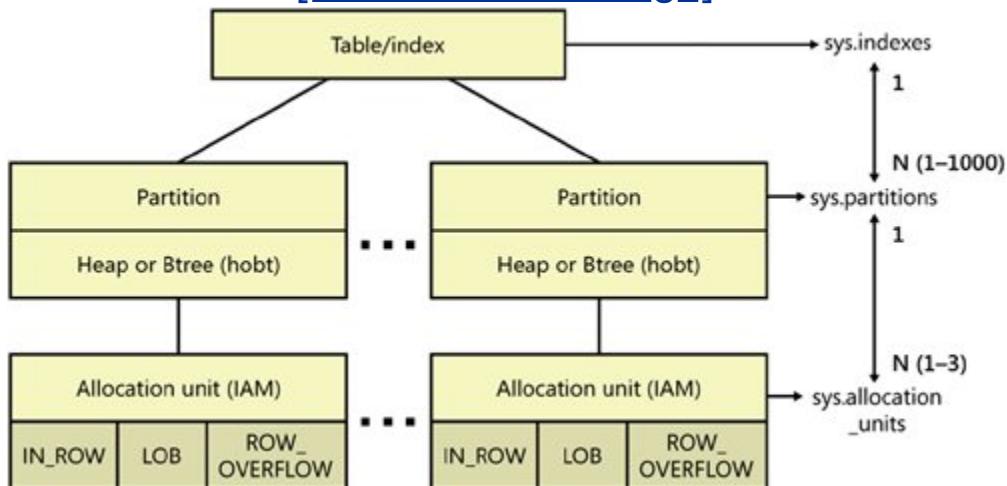
Each table and index has a row in `sys.indexes`, and each table and index in a SQL Server 2005 database can be stored on multiple partitions. The `sys.partitions` view contains one row for each partition of each table or index. Every table or index has at least one partition, even if you haven't specifically partitioned the structure, but there can be up to 1,000 partitions for one table or index. So there is a one-to-many relationship between `sys.indexes` and `sys.partitions`. The `sys.partitions` view contains a column called `partition_id` as well as the `object_id`, so we can join `sys.indexes` to `sys.partitions` on the `object_id` column to retrieve all the partition ID values for a particular table or index. The term used in SQL Server 2005 to describe a subset of a table or index on a single partition is *hobt*, which stands for Heap Or B-Tree and is pronounced (you guessed it) "hobbit." (A B-Tree is the storage structure used for indexes.) The `sys.partitions` view includes a column called `hobt_id`, and in SQL Server 2005 there is always a one-to-one relationship between `partition_id` and `hobt_id`. In fact, in the `sys.partitions` table you will always see that these two columns have the same value. (Future versions of SQL Server might change the relationship between partitions and hobts so that it is not always one-to-one, and the `partition_id` and `hobt_id` might have different values, but any comments about such a future enhancement would be pure speculation at this point.)

Each partition (whether for a table or an index) can have three types of rows, each stored on its own set of pages. These types are called *in-row data pages* (for our "regular" data or index information), *row-overflow data pages*, and *LOB data pages*. A set of pages of one particular type for one particular partition is called an *allocation unit*, so the final catalog view I need to tell you about is `sys.allocation_units`. The view `sys.allocation_units` contains one to three rows per partition because there can be as many as three allocation units for each table or index on each partition. There is always an allocation unit for regular in-row pages, but there might also be an allocation unit for LOB data and one for row-overflow

data. [Figure 6-6](#) shows the relationship between `sys.indexes`, `sys.partitions`, and `sys.allocation_units`.

Figure 6-6. The relationship between `sys.indexes`, `sys.partitions`, and `sys.allocation_units`

[[View full size image](#)]



Querying the Catalog Views

Let's look at a specific example now to see information in these three catalog views. Let's first create the table described earlier in [Figure 6-5](#). You can create it in any database, but I suggest either using *tempdb*, so the table will be automatically dropped next time you restart your SQL Server, or creating a new database just for testing. Many of my examples will assume a database called *test*.

```
CREATE TABLE dbo.employee(
    emp_lname  varchar(15)      NOT NULL,
    emp_fname   varchar(10)      NOT NULL,
    address     varchar(30)      NOT NULL,
    phone       char(12)        NOT NULL,
    job_level   smallint        NOT NULL
)
```

This table will have one row in *sys.indexes* and one in *sys.partitions*, as we can see when we run the following queries. I am including only a few of the columns from *sys.indexes*, but *sys.partitions* only has six columns, so I have retrieved them all.

```
SELECT object_id, name, index_id, type_desc
FROM sys.indexes
WHERE object_id=object_id('dbo.employee')
```

```
SELECT *
FROM sys.partitions
WHERE object_id=object_id('dbo.employee')
```

Here are my results. Yours might vary slightly because your ID values will most likely be different.

object_id	name	index_id	type_desc
5575058	NULL	0	HEAP

partition_id	object_id	index_id	partition_number	hobt_id	rows
72057594038779904	5575058	0			1
72057594038779904	0				

Each row in the `sys.allocation_units` view has a unique `allocation_unit_id` value. Each row also has a value in the column called `container_id` that can be joined with `partition_id` in `sys.partitions`, as shown in this query:

```

SELECT object_name(object_id) AS name,
       partition_id, partition_number AS pnum, rows,
       allocation_unit_id AS au_id, type_desc as
page_type_desc,
       total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
      ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.employee')
    
```

Again, for this simple table, I get only one row because there is only one partition, no nonclustered indexes, and only one type of data (IN_ROW_DATA). I will use this query later in this chapter, and I will refer to it as the "allocation query." (This query might not work in future versions of SQL Server if and when the relationship between hobts and partitions changes. In SQL Server 2005, because the

relationship is always one-to-one, *partition_id* and *hobt_id* can be used interchangeably.)

name	partition_id	pnum	rows	au_id
page_type_desc	pages			
employee	72057594038779904	1	0	
	72057594043301888	IN_ROW_DATA	0	

Now let's add some new columns to the table that will need to be stored on other types of pages. *Varchar* data can be stored on row-overflow pages if the total row size exceeds the maximum of 8,060 bytes. By default, text data is stored on text pages. For *varchar* data that stored on row-overflow pages, and for text data, there is additional overhead in the row itself to store a pointer to the off-row data. We'll look at the details of row-overflow and text data storage later in this section, and we'll look at ALTER TABLE at the end of this chapter, but now I just want to look at the additional rows in *sys.allocation_units*.

```
ALTER TABLE dbo.employee ADD resume_short  
varchar(8000)  
ALTER TABLE dbo.employee ADD resume_long text
```

If we run the preceding query that joins *sys.partitions* and *sys.allocation_units*, we get the following three rows:

[\[View full width\]](#)

name	partition_id	pnum	rows	au_id
	page_type_desc	pages		
employee	72057594038779904	1	0	
	72057594043301888	IN_ROW_DATA		0
employee	72057594038779904	1	0	
	72057594043367424	ROW_OVERFLOW_DATA		0
employee	72057594038779904	1	0	
	72057594043432960	LOB_DATA		0

You might also want to add an index or two and check the contents of these catalog views again. You should notice that just adding a clustered index does not change the number of rows in `sys.allocation_units`, but it will change the `partition_id` numbers because the entire table will be rebuilt internally when you create a clustered index. Adding a nonclustered index will add at least one more row to `sys.allocation_units` to keep track of the pages for that index. The following query joins all three viewssys.indexes, `sys.partitions`, and `sys.allocation_units` to show you the table name, index name and type, page type, and space usage information for the `dbo.employee` table:

```

SELECT convert(char(8),object_name(i.object_id))
AS table_name,
       i.name AS index_name, i.index_id, i.type_desc
as index_type,
       partition_id, partition_number AS pnum, rows,
       allocation_unit_id AS au_id, a.type_desc as
page_type_desc,
       total_pages AS pages
FROM sys.indexes i JOIN sys.partitions p
      ON i.object_id = p.object_id AND
i.index_id = p.index_id
JOIN sys.allocation_units a
      ON p.partition_id = a.container_id
WHERE i.object_id=object_id('dbo.employee')

```

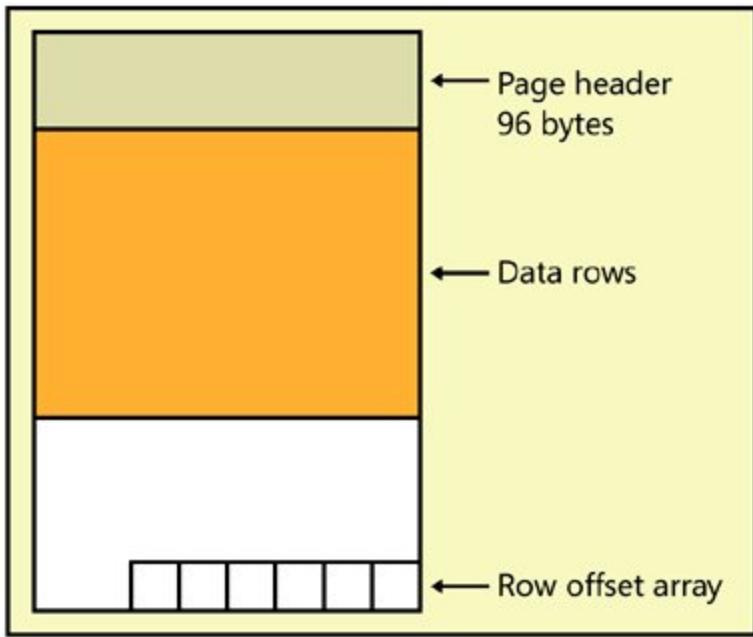
Because I have not inserted any data into this table, you should notice that the values for rows and pages are 0. When I discuss actual page structures, we'll insert data into our tables so we can look at the internal storage of the data at that time. The queries I've run so far do not provide us with any information about the location of pages in the various allocation units. In SQL Server 2000, the *sysindexes* table contains three columns that indicate where data is located; these columns are called *first*, *root*, and *firstIAM*. These columns are still available in SQL Server 2005 (with slightly different names: *first_page*, *root_page*, and *first_iam_page*), but they can be seen only in an undocumented view called *sys.system_internals_allocation_units*. This view is identical to *sys.allocation_units* except for the addition of these three additional columns, so you can replace *sys.allocation_units* with *sys.system_internals_allocation_units* in the preceding allocation query and add these three extra columns to the select list. Keep in mind that as an undocumented object, this view (and others starting with *system_internals*) are for internal use only and are subject to

change. Forward compatibility is not guaranteed. We'll briefly look at the *first_page* column in the next section, and the other two columns will be discussed in [Chapter 7](#) when we look in detail at index structures.

Data Pages

Data pages are the structures that contain user data that has been added to a database's tables. As we saw earlier, there are three varieties of data pages, each of which stores data in a different format. There are pages for in-row data, pages for row-overflow data, and pages for LOB data. As with all other types of pages in SQL Server, data pages have a fixed size of 8 KB, or 8,192 bytes. They consist of three major components: the page header, data rows, and the row offset array, as shown in [Figure 6-7](#).

Figure 6-7. The structure of a data page



Page Header

As you can see in [Figure 6-7](#), the page header occupies the first 96 bytes of each data page (leaving 8,096 bytes for data, row overhead, and row offsets). [Table 6-4](#) shows some of the information contained in the page header.

Table 6-4. Information Contained in the Page Header

Field	What It Contains
<i>pageID</i>	File number and page number of this page in the database
<i>nextPage</i>	File number and page number of the next page if this page is in a page chain
<i>prevPage</i>	File number and page number of the previous page if this page is in a page chain
<i>Metadata: ObjectId</i>	ID of the object to which this page belongs
<i>Metadata: PartitionId</i>	ID of the partition that this page is part of
<i>Metadata: AllocUnitId</i>	ID of the allocation unit that contains this page
<i>Lsn</i>	Log sequence number (LSN) value used for changes and updates to this page
<i>slotCnt</i>	Total number of slots (rows) used on this page

Field	What It Contains
<i>Level</i>	Level of this page in an index (always 0 for leaf pages)
<i>indexed</i>	Index ID of this page (always 0 for data pages)
<i>freeData</i>	Byte offset of the first free space on this page
<i>Pminlen</i>	Number of bytes in fixed-length portion of rows
<i>freeCnt</i>	Number of free bytes on page
<i>reservedCnt</i>	Number of bytes reserved by all transactions
<i>xactreserved</i>	Number of bytes reserved by the most recently started transaction
<i>tornBits</i>	1 bit per sector for detecting torn page writes (discussed in Chapters 4 and 5)
<i>flagBits</i>	2-byte bitmap that contains additional information about the page

Data Rows for In-Row Data

Following the page header is the area in which the table's actual data rows are stored. The maximum size of a single data row is 8,060 bytes of in-row data. Rows can also have row-overflow and LOB data stored on separate pages. The number of rows stored on a given page will vary depending on the structure of the table and on the data being stored. A table that has all fixed-length columns will always be able to store the same number of rows per page; variable-length rows can store as many rows as will fit based on the actual length of the data entered. Keeping row length shorter allows more rows to fit on a page, thus reducing I/O and improving the cache-hit ratio.

Row Offset Array

The row offset array is a block of 2-byte entries, each indicating the offset on the page at which the corresponding data row begins. Every row has a 2-byte entry in this array (as discussed earlier, when I mentioned the 10 overhead bytes needed by every row). Although these bytes aren't stored in the row with the data, they do affect the number of rows that will fit on a page.

The row offset array indicates the logical order of rows on a page. For example, if a table has a clustered index, SQL Server stores the rows in the order of the clustered index key. This doesn't mean the rows are physically stored on the page in the order of the clustered index key. Rather, slot 0 in the offset array refers to the first row in the clustered index key order, slot 1 refers to the second row, and so forth. As we'll see shortly when we examine an actual page, the physical location of these rows can be anywhere on the page.

There's no internal global row number for every row in a table. SQL Server uses the combination of file number, page number, and slot number on the page to uniquely identify each row in a table, and we can use those values with an undocumented command to see the actual bytes on a page.

Examining Data Pages

You can view the contents of a data page by using the DBCC PAGE command, which allows you to view the page header, data rows, and row offset table for any given page in a database. Only a system administrator can use DBCC PAGE. But because you typically won't need to view the contents of a data page, you won't find information about DBCC PAGE in the SQL Server documentation. Nevertheless, in case you want to use it, here's the syntax:

```
DBCC PAGE ({dbid | dbname}, filenum, pagenum[,  
printopt])
```

The DBCC PAGE command includes the parameters shown in [Table 6-5](#). [Figure 6-8](#) shows sample output from DBCC PAGE with a *printopt* value of 1. Note that DBCC TRACEON(3604) instructs SQL Server to return the results to the client. Without this traceflag, no output will be returned for the DBCC PAGE command.

Table 6-5. Parameters of the DBCC PAGE Command

Parameter	Description
<i>Dbid</i>	ID of the database containing the page
<i>Dbname</i>	Name of the database containing the page
<i>Filenum</i>	File number containing the page
<i>Pagenum</i>	Page number within the file
<i>Printopt</i>	Optional print option; takes one of these values: <ul style="list-style-type: none">• 0 Default; prints the buffer header and page header• 1 Prints the buffer header, page header, each row separately, and the row offset table• 2 Prints the buffer and page headers, the page as a whole, and the offset table• 3 Prints the buffer header, page header, each row separately, and the row offset table; each row is followed by each of its column values listed separately

Figure 6-8. Sample output from DBCC PAGE

[[View full width](#)]

```
DBCC TRACEON(3604)
GO
DBCC PAGE (pubs, 1, 153, 1)
GO
```

PAGE: (1:153)

BUFFER:

```
BUF @0x02C156E8
bpage = 0x057CC000
bhash = 0x00000000
bpageno = (1:153)
bdbid = 6
breferences = 0
bUse1 = 30779
bstat = 0xc00009
blog = 0x32159
bnext = 0x00000000
```

PAGE HEADER:

```
Page @0x057CC000
m_pageId = (1:153)
m_headerVersion =
```

```
m_type = 1
m_typeFlagBits = 0x4
m_level = 0
m_flagBits = 0x4200
m_objId (AllocUnitId.idObj) = 67
m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594042318848
Metadata: PartitionId = 72057594038321152
Metadata: IndexId = 1
Metadata: ObjectId = 2073058421
m_prevPage = (0:0)
m_nextPage = (0:0)
pminlen = 24
m_slotCnt = 23
m_freeCnt = 6010
m_freeData = 2136
m_reservedCnt = 0
m_lsn = (15:356:2)
m_xactReserved = 0
m_xdesId = (0:0)
m_ghostRecCnt = 0
m_tornBits = -1021426578
```

Allocation Status

```
GAM (1:2) = ALLOCATED
SGAM (1:3) = NOT ALLOCATED
PFS (1:1) = 0x60 MIXED_EXT ALLOCATED
0_PCT_FULL
DIFF (1:6) = CHANGED
ML (1:7) = NOT MIN_LOGGED
DATA:
```

```
Slot 0, Offset 0x631, Length 88, DumpStyle
BYTE
Record Type = PRIMARY_RECORD
```

```
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS
```

```
Memory Dump @0x5C0FC631
```

```
00000000: 30001800 34303820 3439362d
37323233 †0
... 408 496-7223
00000010: 43413934 30323501 090000fe
05003300
†CA94025.....3.
00000020: 38003f00 4e005800 3137322d
33322d31 †8
.?.N.X.172-32-1
00000030: 31373657 68697465 4a6f686e
736f6e31
†176WhiteJohnson1
00000040: 30393332 20426967 67652052
642e4d65
†0932 Bigge Rd.Me
00000050: 6e6c6f20 5061726b
††††††††††††††nlo Park
```

```
Slot 1, Offset 0xb8, Length 88, DumpStyle
BYTE
```

```
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS
```

```
Memory Dump @0x5C0FC0B8
```

```
00000000: 30001800 34313520 3938362d
37303230 †0
... 415 986-7020
```

Slot 2, Offset 0x110, Length 85, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C0FC110

00000000: 30001800 34313520 3534382d
37373233 †0
. . . 415 548-7723
00000010: 43413934 37303501 090000fe
05003300
†CA94705. 3.
00000020: 39003f00 4d005500 3233382d
39352d37 †9
. ?. M. U. 238-95-7
00000030: 37363643 6172736f 6e436865
72796c35

†766CarsonCheryl5
00000040: 38392044 61727769 6e204c6e
2e426572
†89 Darwin Ln.Ber
00000050: 6b656c65 79|||||||||||||||||||||
keley

/* Data for slots 3 through 20 not shown */

Slot 21, Offset 0x1c0, Length 89, DumpStyle BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C0FC1C0

00000000: 30001800 38303120 3832362d
30373532 †0
. . . 801 826-0752
00000010: 55543834 31353201 090000fe
05003300
†UT84152.....3.
00000020: 39003d00 4b005900 3839392d
34362d32 †9
. =.K.Y.899-46-2
00000030: 30333552 696e6765 72416e6e
65363720
†035RingerAnne67
00000040: 53657665 6e746820 41762e53
616c7420
†Seventh Av.Salt
00000050: 4c616b65 20436974
79|||||||||||||||||Lake City

Slot 22, Offset 0x165, Length 91, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C0FC165

00000000: 30001800 38303120 3832362d
30373532 †0
...801 826-0752
00000010: 55543834 31353201 090000fe
05003300
†UT84152.....3.
00000020: 39003f00 4d005b00 3939382d
37322d33 †9
.?.M.[.998-72-3
00000030: 35363752 696e6765 72416c62
65727436
†567RingerAlbert6
00000040: 37205365 76656e74 68204176
2e53616c †7
Seventh Av.Sal
00000050: 74204c61 6b652043 697479††††††††††††
†t
Lake City

OFFSET TABLE:

Row - Offset
22 (0x16) - 357 (0x165)
21 (0x15) - 448 (0x1c0)
20 (0x14) - 711 (0x2c7)

```
19 (0x13) - 1767 (0x6e7)
18 (0x12) - 619 (0x26b)
17 (0x11) - 970 (0x3ca)
16 (0x10) - 1055 (0x41f)
15 (0xf) - 796 (0x31c)
14 (0xe) - 537 (0x219)
13 (0xd) - 1673 (0x689)
12 (0xc) - 1226 (0x4ca)
11 (0xb) - 1949 (0x79d)
10 (0xa) - 1488 (0x5d0)
9 (0x9) - 1854 (0x73e)
8 (0x8) - 1407 (0x57f)
7 (0x7) - 1144 (0x478)
6 (0x6) - 96 (0x60)
5 (0x5) - 2047 (0x7ff)
4 (0x4) - 884 (0x374)
3 (0x3) - 1314 (0x522)
2 (0x2) - 272 (0x110)
1 (0x1) - 184 (0xb8)
0 (0x0) - 1585 (0x631)
```

DBCC execution completed. If DBCC printed error

messages, contact your system administrator.

As you can see, the output from DBCC PAGE is divided into four main sections: BUFFER, PAGE HEADER, DATA, and OFFSET

TABLE (really the offset array). The BUFFER section shows information about the buffer for the given page. A *buffer* in this context is the in-memory structure that manages a page, and the information in this section is relevant only when the page is in memory.

The PAGE HEADER section in [Figure 6-8](#) displays the data for all the header fields on the page. ([Table 6-4](#) shows the meaning of most of these fields.) The DATA section contains information for each row. When DBCC PAGE is used with a *printopt* value of 1 or 3, DBCC PAGE indicates the slot position of each row, the offset of the row on the page, and the length of the row. The row data is then divided into three parts. The left column indicates the byte position within the row where the displayed data occurs. The next section contains the actual data stored on the page, displayed in four columns of eight hexadecimal digits each. The right column contains an ASCII character representation of the data. Only character data is readable in this column, although some of the other data might be displayed.

The OFFSET TABLE section shows the contents of the row offset array at the end of the page. In [Figure 6-8](#), you can see that this page contains 23 rows, with the first row (indicated by slot 0) beginning at offset 1585 (0x631). The first row physically stored on the page is actually row 6, with an offset in the row offset array of 96. DBCC PAGE with a *printopt* value of 1 displays the rows in slot number order, even though, as you can see by the offset of each of the slots, that it isn't the order in which the rows physically exist on the page. If you use DBCC PAGE with a *printopt* value of 2, you see a dump of all 8,096 bytes of the page (after the header) in the order they are stored on the page.

The Structure of Data Rows

A table's data rows have the general structure shown in [Figure 6-9](#). The data for all fixed-length columns is stored first, followed by the data for all variable-length columns. [Table 6-6](#) shows the information stored in each row.

Figure 6-9. The structure of data rows

[[View full size image](#)]

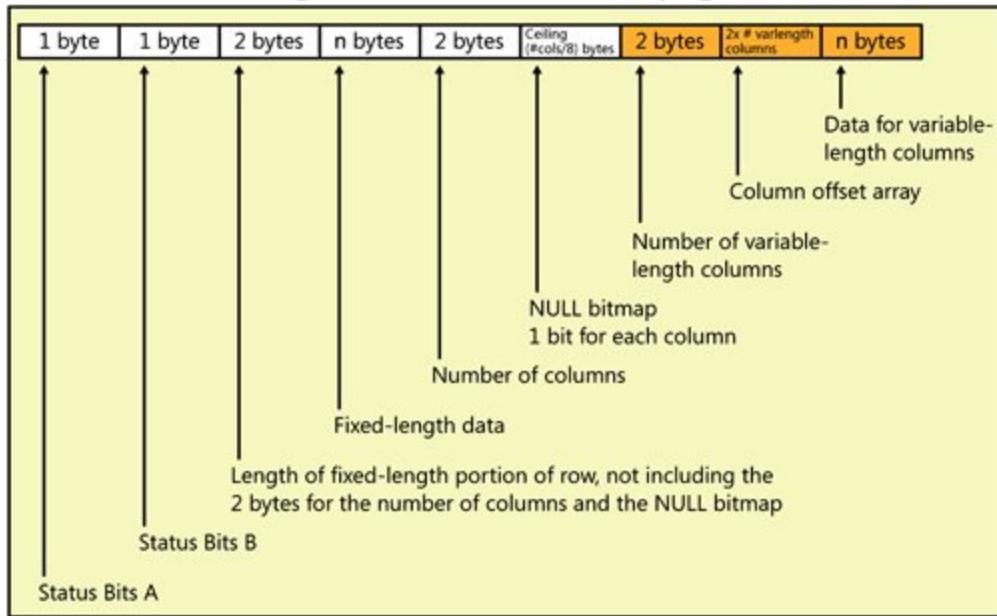


Table 6-6. Information Stored in a Table's Data Rows

Information	Mnemonic	Size
Status Bits A	TagA	1 byte
Status Bits B (not used in SQL Server 2000)	TagB	1 byte
Fixed-length size	Fsize	2 bytes
Fixed-length data	Fdata	Fsize 4
Number of columns	Ncol	2 bytes
NULL bitmap (1 bit for each column in table; a 1 indicates that the corresponding column is NULL or that the bit is unused.)	Nullbits	Ceiling (Ncol / 8)

Information	Mnemonic	Size
Number of variable-length columns	VarCount	2 bytes
Variable column offset array	VarOffset	$2 * \text{VarCount}$
Variable-length data	VarData	$\text{VarOff}[\text{VarCount}] - (\text{Fsize} + 4 + \text{Ceiling}(\text{Ncol} / 8) + 2 * \text{VarCount})$

Status Bits A contains a bitmap indicating properties of the row. The bits have the following meaning:

- **Bit 0** Versioning information; in SQL Server 2005, it's always 0.
- **Bits 1 through 3** Taken as a 3-bit value, 0 indicates a primary record, 1 indicates a forwarded record, 2 indicates a forwarding stub, 3 indicates an index record, 4 indicates a blob fragment or row-overflow data, 5 indicates a ghost index record, and 6 indicates a ghost data record. (I'll discuss forwarding and ghost records in [Chapter 7](#).)
- **Bit 4** Indicates that a NULL bitmap exists; in SQL Server 2005, a NULL bitmap is always present, even if no NULLs are allowed in any column.

- **Bit 5** Indicates that variable-length columns exist in the row.
- **Bits 6 and 7** Not used in SQL Server 2005.

Within each block of fixed-length or variable-length data, the data is stored in the column order in which the table was created. For example, suppose a table is created with the following statement:

```
CREATE TABLE Test1
(
Col1 int          NOT NULL,
Col2 char(25)     NOT NULL,
Col3 varchar(60)  NULL,
Col4 money         NOT NULL,
Col5 varchar(20)  NOT NULL
)
```

The fixed-length data portion of this row will contain the data for *Col1*, followed by the data for *Col2*, followed by the data for *Col4*. The variable-length data portion will contain the data for *Col3*, followed by the data for *Col5*. For rows that contain only fixed-length data, the following is true:

- The first hexadecimal digit of the first byte of the data row will be 1, indicating that no variable-length columns exist. (The first hexadecimal digit comprises bits 4 through 7; bits 6 and 7 are always 0, and if there are no variable-length columns, bit 5 is also 0. Bit 4 is always 1, so the value of the four bits is displayed as 1.)
- The data row ends after the NULL bitmap, which follows the fixed-length data (that is, the shaded portion shown in [Figure 6-](#)

[9](#) won't exist in rows with only fixed-length data).

- The total length of every data row will be the same.

Column Offset Arrays

A data row that has all fixed-length columns has no variable column count or column offset array. A data row that has variable-length columns has a column offset array in the data row with a 2-byte entry for each variable-length column, indicating the position within the row where each column ends. (The terms *offset* and *position* aren't exactly interchangeable. *Offset* is 0-based, and *position* is 1-based. A byte at an offset of 7 is in the eighth byte position in the row.)

The two examples that follow illustrate how fixed-length and variable-length data rows are stored.

Storage of Fixed-Length Rows

First let's look at the simpler case of an all fixed-length row:

```
CREATE TABLE Fixed
(
    Col1 char(5)      NOT NULL,
    Col2 int          NOT NULL,
    Col3 char(3)      NULL,
    Col4 char(6)      NOT NULL
)
```

When this table is created, you should be able to execute the following queries against the `sys.indexes` and `sys.columns` views to receive the information similar to the results shown:

```
SELECT object_id, type_desc,
       indexproperty(object_id, name, 'minlen') as minlen
      FROM sys.indexes where
object_id=object_id('fixed')
```

```
SELECT column_id, name, system_type_id,
max_length
FROM sys.columns
WHERE object_id=object_id('fixed')
```

RESULTS:

object_id	type_desc	minlen
53575229	HEAP	22

column_id	name	system_type_id
	max_length	
1	Col1	175
2	Col2	56
3	Col3	175
4	Col4	175

Note



The `sysindexes` table in SQL Server 2000 contains columns called *minlen* and

xmaxlen, which store the minimum and maximum length of a row. In SQL Server 2005, these values are available only by using undocumented parameters to the *indexproperty* function. As with all undocumented features, keep in mind that they are not supported by Microsoft and future compatibility is not guaranteed.

For tables containing only fixed-length columns, the value returned for *minlen* by the *indexproperty* function is equal to the sum of the column lengths (from *sys.columns.max_length*) plus 4 bytes. It doesn't include the 2 bytes for the number of columns or the bytes for the NULL bitmap.

To look at a specific data row in this table, you must first insert a new row:

```
INSERT Fixed VALUES ('ABCDE', 123, NULL, 'CCCC')
```

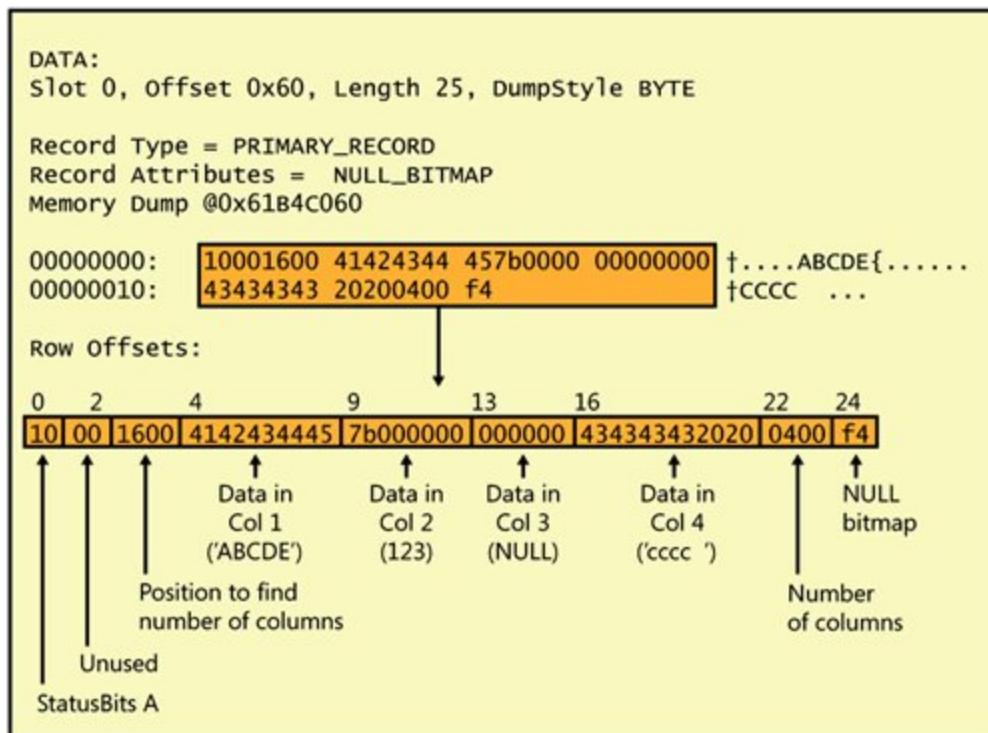
[Figure 6-10](#) shows this row's actual contents on the data page. To run the DBCC PAGE command, I needed to know what page number was used to store the row for this table. I mentioned earlier that a value for *first_page* was stored in an undocumented view called *sys.system_internals_allocation_units*, which is almost identical to the *sys.allocation_units* view. The following query gives me the value for *first_page* for the table called *fixed*:

```
SELECT object_name(object_id) AS name,
       rows, type_desc as page_type_desc,
       total_pages AS pages, first_page
  FROM sys.partitions p JOIN
       sys.system_internals_allocation_units a
      ON p.partition_id = a.container_id
 WHERE object_id=object_id('dbo.fixed')
```

RESULTS:

name	rows	page_type_desc	pages	first_page
-----	-----	-----	-----	-----
--				
Fixed	1	IN_ROW_DATA	2	
	0xCF0400000100			

Figure 6-10. A data row containing all fixed-length columns



I can then take the value of `first_page` from the preceding `sys.system_internals_allocation_units` output (0xCF0400000100) and convert it to a file and page address. In hexadecimal notation, each set of two hexadecimal digits represents a byte. I first had to swap the bytes to get 00 01 00 00 04 CF. The first two groups represent the 2-byte file number; the last four groups represent the page number. So the file is 0x0001, which is 1, and the page number is 0x000004CF, which is 1231 in decimal.

Unless you particularly enjoy playing with hexadecimal conversions, you might want to use one of two other options for determining the actual page numbers associated with your SQL Server tables and indexes. First you can create the function shown here to convert a

6-byte hexadecimal page number value (such as 0xCF0400000100) to a *file_number:page_number* format.

```
CREATE FUNCTION convert_page_nums (@page_num  
binary(6))  
    RETURNS varchar(11)  
AS  
    BEGIN  
        RETURN(convert(varchar(2), (convert(int,  
substring(@page_num, 6, 1))  
            * power(2, 8)) +  
            (convert(int, substring(@page_num, 5,  
1))) + ':' +  
            convert(varchar(11),  
                (convert(int, substring(@page_num, 4, 1)) *  
power(2, 24)) +  
                (convert(int, substring(@page_num, 3, 1)) *  
power(2, 16)) +  
                (convert(int, substring(@page_num, 2, 1)) *  
power(2, 8)) +  
                (convert(int, substring(@page_num, 1, 1)))) )  
    END
```

You can then execute this SELECT to call the function:

```
SELECT dbo.convert_page_nums(0xCF0400000100)
```

You should get back the result 1:1231.

Warning



SQL Server does not guarantee that the *first_page* column in `sys.system_internals_allocation_units` will always indicate the first page of a table. (The view is undocumented, after all.) I've found that *first_page* is reliable until you begin to perform deletes and updates on the data in the table.

The second option for determining the actual page numbers is to use another undocumented command called DBCC IND. Because most of the information returned is relevant only to indexes, I won't discuss this command in detail until [Chapter 7](#). However, for a sneak preview, you can run the following command and note the values in the first two columns of output (labeled *PageFID* and *PagePID*) in the row where *PageType* = 1, which indicates that the page is a data page.

```
DBCC IND(test, fixed, -1)
```

You would replace *test* with the name of whatever database you were in when you created this table. The values for *PageFID* and *PagePID* should be the same value you used when you converted the hexadecimal string for the *first_page* value. In my case, I see that *PageFID* value is 1 and the *PagePID* value is 1231. So those are the values I use when calling DBCC PAGE:

```
DBCC PAGE(test, 1, 1231, 1)
```

Reading the output of DBCC PAGE takes a bit of practice. First note that the output shows the data rows in groups of 4 bytes at a time. The shaded area in [Figure 6-10](#) has been expanded to show the bytes in an expanded form.

The first byte is Status Bits A, and its value (0x10) indicates that only bit 3 is on, so the row has no variable-length columns. The second byte in the row remains unused. The third and fourth bytes (1,600) indicate the length of the fixed-length fields, which is also the column offset in which the *Ncol* value can be found. (As a multi-byte numeric value, this information is stored in a byte-swapped form, so the value is really 0x0016, which translates to 22.) To know where in the row between 4 and 22 each column actually is located, we need to know the offset of each column. In SQL Server 2000, the *syscolumns* system table has a column indicating the offset within the row. Although in SQL Server 2005 you can still select from the compatibility view called *syscolumns*, the results you get back are not reliable. The offsets can be found in an undocumented view called *sys.system_internals_partition_columns* that we can then join to *sys.partitions* to get the information about the referenced objects and join to *sys.columns* to get other information about each column.

Here is a query to return basic column information, including the offset within the row for each column. I will use the same query for other tables later in this chapter, and I will refer to it as the "column detail query."

```
SELECT c.name AS column_name, column_id,
max_inrow_length,
    pc.system_type_id, leaf_offset
FROM sys.system_internals_partition_columns pc
JOIN sys.partitions p
    ON p.partition_id = pc.partition_id
```

```

JOIN sys.columns c
    ON column_id = partition_column_id
        AND c.object_id = p.object_id
WHERE p.object_id=object_id('fixed')

```

RESULTS:

column_name	column_id	max_inrow_length	system_type_id	leaf_offset
Col1	1	5	175	4
Col2	2	4	56	9
Col3	3	3	175	13
Col4	4	6	175	16

So now we can find the data in the row for each column simply by using the offset value in the preceding results: the data for column *Col1* begins at offset 4, the data for column *Col2* begins at offset 9, and so on. As an *int*, the data in *Col2* (7b000000) must be byte-swapped to give us the value 0x0000007b, which is equivalent to 123 in decimal.

Note that the 3 bytes of data for *Col3* are all zeros, representing an actual NULL in the column. Because the row has no variable-length columns, the row ends 3 bytes after the data for column *Col4*. The 2 bytes starting right after the fixed-length data at offset 22 (0400, which is byte-swapped to yield 0x0004) indicate that four columns are in the row. The last byte is the NULL bitmap. The value of 0xf4 is 11110100 in binary, and bits are shown from high order to low order. The low-order 4 bits represent the four columns in the table,

0100, which indicates that only the third column actually IS NULL. The high-order 4 bits are 1111 because those bits are unused. The null bitmap must have a multiple of 8 bits, and if the number of columns is not a multiple of 8, there will be unused bits.

Storage of Variable-Length Rows

Now let's look at the somewhat more complex case of a table with variable-length data. Each row has three *varchar* columns and two fixed length columns:

```
CREATE TABLE variable
(
    Col1 char(3)          NOT NULL,
    Col2 varchar(250)     NOT NULL,
    Col3 varchar(5)        NULL,
    Col4 varchar(20)       NOT NULL,
    Col5 smallint          NULL
)
```

When this table is created, you should be able to execute the following queries against the *sys.indexes*, *sys.partitions*, *sys.system_internals_partition_columns*, and *sys.columns* views to receive the information similar to the results shown here:

```
SELECT object_id, type_desc,
       indexproperty(object_id, name, 'minlen') as minlen
  FROM sys.indexes where
object_id=object_id('variable')
```

```
SELECT name, column_id, max_inrow_length,
```

```

pc.system_type_id, leaf_offset
FROM sys.system_internals_partition_columns pc
JOIN sys.partitions p
    ON p.partition_id = pc.partition_id
JOIN sys.columns c
    ON column_id = partition_column_id AND
c.object_id = p.object_id
WHERE p.object_id=object_id('variable')

```

RESULTS:

object_id	type_desc	minlen
69575286	HEAP	9

column_name	column_id	max_inrow_length	
system_type_id	leaf_offset		
Col1	1	3	175
4			
Col2	2	250	167
-1			
Col3	3	5	167
-2			
Col4	4	20	167
-3			
Col5	5	2	52
7			

Now you insert a row into the table:

```

INSERT variable VALUES
    ('AAA', REPLICATE('X', 250), NULL, 'ABC', 123)

```

The REPLICATE function is used here to simplify populating a column; this function builds a string of 250 Xs to be inserted into *Col2*.

In [Figure 6-11](#), the data for the fixed-length columns can be found by using the *leaf_offset* value in `sys.system_internals_partition_columns`, in the preceding query results. In this table, *Col1* begins at offset 4 and *Col5* begins at offset 7. Variable-length columns are not shown in the query output with fixed offset because the offset can be different in each row. Instead, the row itself holds the ending position of each variable-length column within that row in a part of the row called the Row Offset Array. The query output shows that *Col2* has an *leaf_offset* value of 1, which means that *Col2* is the first variable-length column; an offset for *Col3* of 2 means that *Col3* is the second variable-length column, and an offset of 3 for *Col4* means that *Col4* is the third variable-length column.

Figure 6-11. A data row with variable-length columns

DATA:

Slot 0, Offset 0x60, Length 273, DumpStyle BYTE

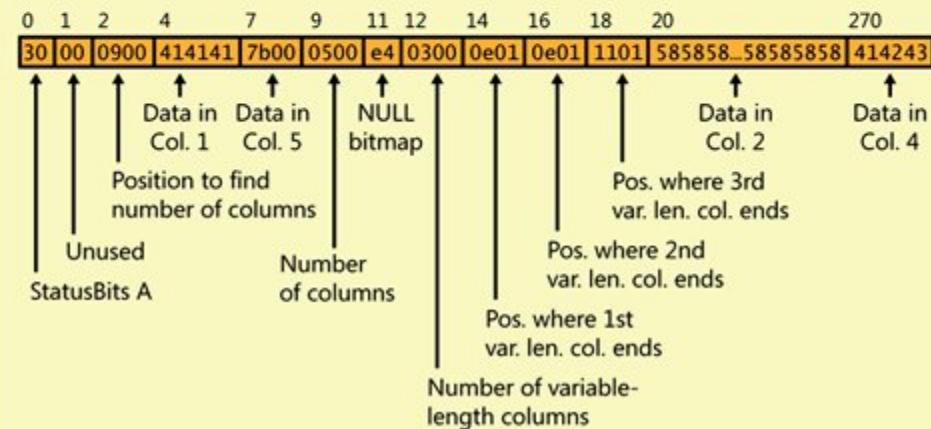
Record Type = PRIMARY_RECORD

Record Attributes = NULL_BITMAP VARIABLE_COLUMNS

Memory Dump @0x61C4C060

00000000:	30000900	4141417b	000500e4	03000e01	†0...AAA{.....}
00000010:	0e011101	58585858	58585858	58585858	†....xxxxxxxxxx
00000020:	58585858	58585858	58585858	58585858	†xxxxxxxxxxxxxx
00000030:	58585858	58585858	58585858	58585858	†xxxxxxxxxxxxxx
000000F0:	58585858	58585858	58585858	58585858	†xxxxxxxxxxxxxx
00000100:	58585858	58585858	58585858	58584142	†xxxxxxxxxxxxxxAB
00000110:	43				†C

Row Offsets:



To find the variable-length columns in the data row itself, you first locate the column offset array in the row. Right after the 2-byte field indicating the total number of columns (0x0500) and the NULL bitmap with the value 0xe4, a 2-byte field exists with the value 0x0300 (or 3, decimal) indicating that three variable-length fields exist. Next comes the column offset array. Three 2-byte values indicate the ending position of each of the three variable-length columns: 0x0e01 is byte-swapped to 0x010e, so the first variable

byte column ends at position 270. The next 2-byte offset is also 0x0e01, so that column has no length and has nothing stored in the variable data area. Unlike with fixed-length fields, if a variable-length field has a NULL value, it takes no room in the data row. SQL Server distinguishes between a *varchar* containing NULL and an empty string by determining whether the bit for the field is 0 or 1 in the NULL bitmap. The third 2-byte offset is 0x1101, which, when byte-swapped, gives us 0x0111. This means the row ends at position 273 (and is a total of 273 bytes in length).

The total storage space needed for a row depends on a number of factors. Variable-length fields add additional overhead to a row, and their actual size is probably unpredictable. Even for fixed-length fields, the number of bytes of overhead can change depending on the number of columns in the table. In the earlier example illustrated in [Figure 6-3](#), I mentioned that 10 bytes of overhead will exist if a row contains all fixed-length columns. For that row, 10 is the correct number. The size of the NULL bitmap needs to be long enough to store a bit for every column in the row. In the [Figure 6-3](#) example, the table has 11 columns, so the NULL bitmap needs to be 2 bytes. In the examples illustrated by [Figures 6-9](#) and [6-10](#), the tables have fewer than eight columns, so the NULL bitmaps need only a single byte. Don't forget that the total row overhead must also include the 2 bytes for each row in the row offset table at the bottom of the page.

Page Linkage

SQL Server 2005 doesn't connect the individual data pages of a table in a doubly linked list unless the table has a clustered index. Pages at each level of an index are linked together, and because the data is considered the leaf level of a clustered index, SQL Server does maintain the linkage. However, for a heap, there is no such linked list connecting the pages to each other. The only way

that SQL Server determines which pages belong to a table is by inspecting the Index Allocation Maps (IAMs) for the table.

If the table has a clustered index, you can use the *M_nextPage* and *M_prevPage* values in the page header information to determine the ordering of pages in the list. Alternatively, you can use the DBCC IND command to get a list of all pages that belong to an index, along with a column showing the next page and the previous page, for each page. We'll look at a lot more detailed examples of DBCC IND in [Chapter 7](#).

Row-Overflow Data

SQL Server 2005 lets you store some of a row's data off of the actual data page. This is the first change in the maximum row length in the SQL Server product since version 7.0. For SQL Server 7.0, the size of data pages changed from 2 KB to 8 KB, and along with that increase in page size came a corresponding increase in the maximum size of a data row. The maximum row size in SQL Server 6.5 was 1,962 bytes, and in SQL Server 7.0 it increased to 8,060 bytes; that maximum size stayed the same in SQL Server 2000. Along with the increase in page size, the maximum length of a variable-length column (*varchar* or *varbinary*) increased from 255 bytes to 8,000 bytes. In SQL Server 2005, 8,060 is still the maximum size of a row on a data page and 8,000 bytes is still the maximum defined length for a variable-length column. As discussed earlier, this maximum row size value includes several bytes of overhead stored with the row on the physical pages, so the total size of all the table's defined columns must be slightly less than this amount. In fact, SQL Server 2005 changed the error message you get if you try to create a table with more rows than the allowable maximum. If you execute the following CREATE TABLE statement with column definitions that add up to exactly 8,060 bytes, you'll get the error message shown.

```
CREATE TABLE dbo.bigrows  
(a char(3000),  
 b char(3000),  
 c char(2000),  
 d char(60) )
```

Msg 1701, Level 16, State 1, Line 1
Creating or altering table 'bigrows' failed
because the minimum row size would be 8067,
including 7 bytes of internal overhead. This
exceeds the maximum allowable table row size of
8060 bytes.

In this message, you can see the number of overhead bytes (7) that SQL Server 2005 wants to store with the row itself. There is also an additional 2 bytes for the row offset bytes at the end of the page, but those bytes are not included in this total here. The only way to exceed this size limit of 8,060 bytes is to use variable-length columns because for variable-length data, SQL Server 2005 can store the columns in special row-overflow pages if all the fixed-length columns will fit into the regular in-row size limit.

Now let's take a look at a table with all variable-length columns. Note that although my columns are all *varchar*, columns of other data types can potentially also be stored on row-overflow data pages. These other data types include *varbinary*, *nvarchar*, and *sqlvariant* columns, as well as columns that use CLR user-defined data types. The following code creates a table with rows that have a maximum defined length of much greater than 8,060 bytes.

```
CREATE TABLE dbo.bigrows  
(a varchar(3000),  
 b varchar(3000),
```

```
c varchar(3000),  
d varchar(3000) )
```

In fact, if you run this CREATE TABLE statement in SQL Server 7.0, you get an error and the table will not be created at all. In SQL Server 2000, the table will be created, but you'll get a warning that inserts or updates might fail if the row size exceeds the maximum.

Not only can the preceding *dbo.bigrows* table be created in SQL Server 2005, but you can insert a row with column sizes that add up to more than 8,060 bytes with a simple INSERT, as shown here:

```
INSERT INTO dbo.bigrows  
    SELECT REPLICATE('e', 2100), REPLICATE('f',  
2100),  
        REPLICATE('g', 2100), REPLICATE('h', 2100)
```

To determine whether SQL Server is storing any data in row-overflow data pages for a particular table, you can run the allocation query shown earlier:

```
SELECT object_name(object_id) AS name,  
    partition_id, partition_number AS pnum, rows,  
    allocation_unit_id AS au_id, type_desc as  
page_type_desc,  
    total_pages AS pages  
FROM sys.partitions p JOIN sys.allocation_units a  
    ON p.partition_id = a.container_id  
WHERE object_id=object_id('dbo.bigrows')
```

This query should return output similar to that shown here:

name	partition_id	pnum	rows	au_id
page_type_desc	pages			
bigrows	72057594039238656	1	1	
	72057594043957248	IN_ROW_DATA		2
bigrows	72057594039238656	1	1	
	72057594044022784	ROW_OVERFLOW_DATA	2	

You can see there are two pages for the one row of regular in-row data, and two pages for the one row of row-overflow data.

Alternatively, you can use the command *DBCC IND(test, bigrows, -1)* and see the four pages individually. Two pages are for the row-overflow data, and two are for the in-row data. The *PageType* values have the following meaning:

- Page type 1 = data page
- Page type 2 = index page
- Page type 3 = LOB or row-overflow page
- Page type 10 = IAM page

PageFID	PagePID	ObjectID	PartitionID
IAM_chain_type	PageType		
1	2252	85575343	72057594039238656
Row-overflow data	3		

1	2251	85575343	72057594039238656
Row-overflow data	10		
1	2254	85575343	72057594039238656
In-row data	1		
1	2253	85575343	72057594039238656
In-row data	10		

We can see that there is one data page and one IAM page for the in-row data, and one data page and one IAM page for the row-overflow data. With the results from DBCC IND, we could also look at the page contents with DBCC PAGE. On the data page for the in-row data, we would see three of the four *varchar* column values, and the fourth column would be stored on the data page for the row-overflow data. If you run DBCC PAGE for the data page storing the in-row data (page 1:2254 in my example), you'll notice that it isn't necessarily the fourth column in column order that is stored off the row. I won't show you the entire contents of the row because they are almost the entire page size. When I look at the in-row data page using DBCC PAGE, I see the column with e, the column with g, and the column with h, and it is the column with f that has moved to the new row. In the place of that column, we can see the bytes shown below. I have included the last byte with e (ASCII code 65) and the first byte with g (ASCII code 67), and in between there are 24 other bytes. The 17th through 22nd bytes of those 24 bytes are treated as a 6-byte numeric value: cc0800000100. We need to reverse the byte order and break it into a 1-byte hex value for the file number and a 4-byte hex value for the page number. So the file number is 0x01, or 1, for the file number, and 0x000008cc, or 2252, for the page number. This is the same file and page numbers that we saw using DBCC IND.

65020000 61010000 00c92900 00340800 00cc0800
00010000 0067

SQL Server stores variable-length columns on row-overflow pages only under certain conditions. The determining factor is the length of the row itself. It doesn't matter how full the regular page is into which SQL Server is trying to insert the new row. SQL Server constructs the row normally, and only if the row itself needs more than 8,060 bytes will some of its columns be stored on overflow pages. Each column in the table is either completely on the row or completely off the row. This means that a 4,000-byte variable-length column cannot have half its bytes on the regular data page and half on a row-overflow page. If a row is less than 8,060 bytes and there is no room on the page where SQL Server is trying to insert it, normal page-splitting algorithms (which I'll describe in [Chapter 7](#)) are applied.

One row can span many row-overflow pages if it contains many large variable-length columns. For example, you can create the table *dbo.hugerows* and insert a single row into it:

```
CREATE TABLE dbo.hugerows
    (a varchar(3000),
     b varchar(8000),
     c varchar(8000),
     d varchar(8000))
INSERT INTO dbo.hugerows
    SELECT REPLICATE('a', 3000), REPLICATE('b',
8000),
           REPLICATE('c', 8000), REPLICATE('d',
8000)
```

Now if I run the allocation query shown earlier, substituting *hugerows* for *bigrows*, I get the results shown here:

name	partition_id	pnum	rows	au_id
page_type_desc	pages			
hugerows	72057594039304192	1	1	
	72057594044088320	IN_ROW_DATA		2
hugerows	72057594039304192	1	1	
	72057594044153856	ROW_OVERFLOW_DATA	4	

There are four pages for the row overflow information, one for the IAM page and three for the columns that didn't fit in the regular row. The number of large variable-length columns that a table can have is not unlimited, although it is quite large. There is a limit of 1,024 columns in any table, so that is definitely a limit there. But another limit will be reached before that. When a column has to be moved off a regular page onto a row-overflow page, SQL Server keeps a pointer to the row-overflow information as part of the original row. The pointer is always 24 bytes, and the row still needs 2 bytes in the row for each variable-length column, whether or not the variable-length column is stored in the row. So it turns out that 308 is the maximum number of overflowing columns we can have, and such a row needs 8,008 bytes just for the 26 overhead bytes for each overflowing column in the row.

Note



Just because SQL Server can store lots of large columns on row-overflow pages doesn't mean it's always a good idea to do so. This capability does allow you more flexibility in the organization of your tables, but you might pay a heavy

performance price if many additional pages need to be accessed for every row of data. Row-overflow pages are intended to be a solution in the situation where most rows will fit completely on your data pages and you only occasionally need have row-overflow data. Using row-overflow pages, SQL Server can handle the extra data effectively, without requiring a redesign of your table.

In some cases, if a large variable-length column shrinks, it can be moved back to the regular row. However, for efficiency reasons, if the decrease is just a few bytes, SQL Server will not bother checking. Only when a column stored in a row-overflow page is reduced by more than 1,000 bytes will SQL Server even consider checking to see whether the column can now fit on the regular data page. You can observe this behavior if you previously created the *dbo.bigrows* table for the earlier example and inserted only the one row with 2,100 characters in each column.

The following update reduces the size of the first column by 500 bytes, reducing the row size to 7,900 bytes, which should all fit on the one data page.

```
UPDATE bigrows  
SET a = replicate('a', 1600)
```

However, if you run the allocation query again, you'll see that there are two row-overflow pages. Now reduce the size of the first column by more than 1,000 bytes and run the allocation query once more.

```
UPDATE bigrows  
SET a = 'aaaaaa'
```

You should see only three pages for the table now because there is no longer a row-overflow data page. The IAM for the row-overflow data pages has not been removed, but you will no longer have a data page for row-overflow data.

When a column is stored on a row-overflow page, SQL Server has to store a pointer to the row-overflow page from the regular data row, which includes the offset of the column data on the row-overflow page, which, as we've seen, takes 24 bytes in the original row.

Keep in mind that row-overflow data storage applies only to columns of variable-length data, which are defined to be no longer than the normal variable-length maximum of 8000 bytes per column. In addition, to store a variable-length column on a row-overflow page, you must meet the following conditions:

- All the fixed-length columns, including overhead bytes, must add up to no more than 8060 bytes. (The pointer to each row-overflow column adds 24 bytes of overhead to the row.)
- The actual length of the variable-length column must be more than 24 bytes.
- The column must not be part of the clustered index key.

If you have single columns that might need to store more than 8,000 bytes, you should use either large object (*text*, *image*, or *ntext*) columns or use the *varchar(MAX)* data type.

Large Object Data

If a table contains LOB data (*text*, *ntext*, or *image* types), by default the actual data is not stored on the regular data pages. Like row-overflow data, LOB data is stored in its own set of pages, and the allocation query shows you pages for LOB data as well as pages for regular in-row data and row-overflow data. For LOB columns, SQL Server stores a 16-byte pointer in the data row that indicates where the actual data can be found. Although the default behavior is to store all the LOB data off the data row, SQL Server 2005 allows you to change the storage mechanism by setting a table option to allow LOB data to be stored in the data row itself. We'll talk about the default storage of LOB data now.

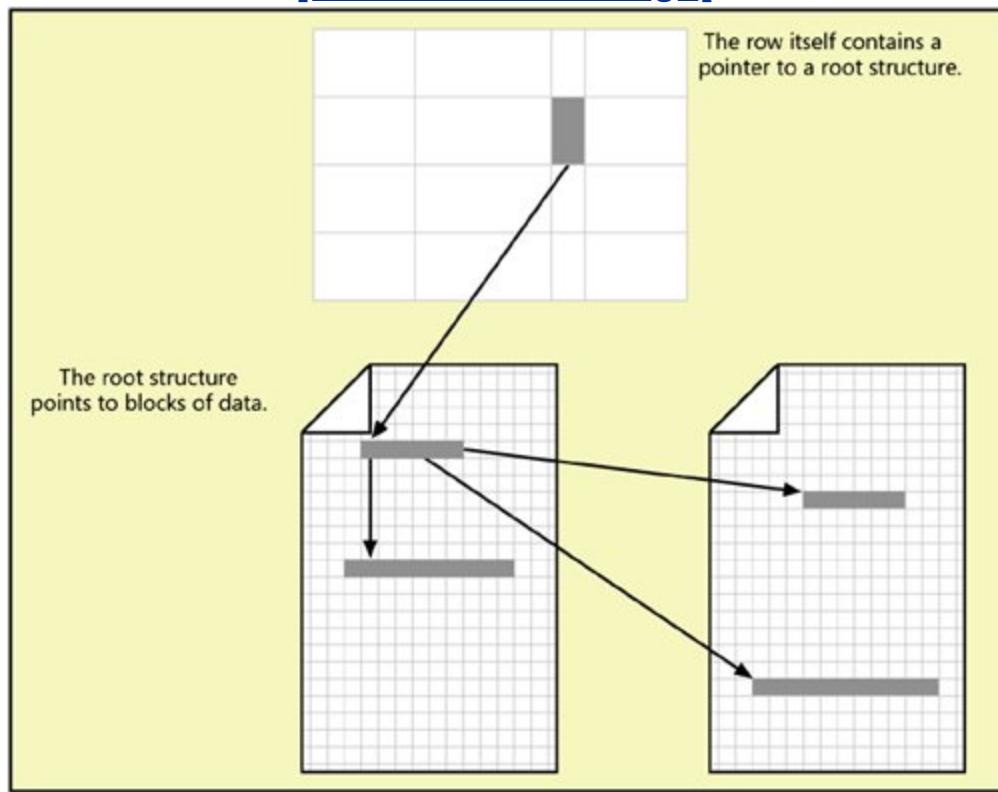
As stated earlier, by default no LOB data is stored in the data row. Instead, the data row contains only a 16-byte pointer to a page (or the first of a set of pages) where the data can be found. These pages are 8 KB in size, like any other page in SQL Server, and individual *text*, *ntext*, and *image* pages aren't limited to holding data for only one occurrence of a *text*, *ntext*, or *image* column. A *text*, *ntext*, or *image* page can hold data from multiple columns and from multiple rows; the page can even have a mix of *text*, *ntext*, and *image* data. However, one *text* or *image* page can hold only *text* or *image* data from a single table.

The collection of 8-KB pages that make up a LOB column aren't necessarily located next to each other. The pages are logically organized in a B-tree structure so operations starting in the middle of the LOB string are very efficient. The structure of the B-tree will vary slightly depending on whether the amount of data is less than

or more than 32 KB. (See [Figure 6-12](#) for the general structure.) I'll discuss B-trees in more detail when I talk about index internals in [Chapter 7](#).

Figure 6-12. A text column pointing to a B-tree that contains the blocks of data

[[View full size image](#)]



If the amount of data is less than 32 KB, the text pointer in the data row points to an 84-byte text root structure. This forms the root node of the B-tree structure. The root node points to the blocks of *text*, *ntext*, or *image* data. While the data for LOB columns is arranged logically in a B-tree, physically both the root node and the individual blocks of data are spread throughout LOB pages for the table.

They're placed wherever space is available. The size of each block of data is determined by the size written by an application. Small blocks of data are combined to fill a page. If the amount of data is less than 64 bytes, it's all stored in the root structure.

If the amount of data for one occurrence of a LOB column exceeds 32 KB, SQL Server starts building intermediate nodes between the data blocks and the root node. The root structure and the data blocks are interleaved throughout the *text* and *image* pages. The intermediate nodes, however, are stored in pages that aren't shared between occurrences of *text* or *image* columns. Each page storing intermediate nodes contains only intermediate nodes for one *text* or *image* column in one data row.

LOB Data Stored in the Data Row

If you store all your LOB data outside of your data pages, every time you access that data SQL Server will need to perform additional page reads, just like it does for row-overflow pages. In some cases, you might notice a performance improvement by allowing some of the LOB data to be stored in the data row. You can enable a table option called *text in row* for a particular table by setting the option to 'ON' or by specifying a maximum number of bytes to be stored in the data row. The following command enables up to 500 bytes of LOB data to be stored with the regular row data in a table called *employee*:

```
sp_tableoption employee, 'text in row', 500
```

Instead of a number, you can specify the option 'ON' (including the quote marks), which sets the maximum size to 256 bytes. Note that the value is in bytes, not characters. For *n*text data, each character needs 2 bytes so that any *n*text column will be stored in the data row if it is less than 250 characters. Once you enable the *text in row* option, you never get just the 16-byte pointer for the LOB data in the row, as is the case when the option is not on. If the data in the LOB field is more than the specified maximum, the row will hold the root structure containing pointers to the separate chunks of LOB data. The minimum size of a root structure is 24 bytes, and the possible range of values that *text in row* can be set to is 24 to 7,000 bytes.

To disable the *text in row* option, you can set the value to either 'OFF' or 0. To determine whether a table has the *text in row* property enabled, you can use the OBJECTPROPERTY function, as shown here:

```
SELECT OBJECTPROPERTY (object_id('employee'),  
'TableTextInRowLimit')
```

This function returns the maximum number of bytes allowed for storing LOBs in a data row. If a 0 is returned, the *text in row* option is disabled.

Let's create a table very similar to the one we created to look at row structures, but we'll change the *varchar(250)* column to the *text* data type. We'll use almost the same insert statement to insert one row into the table.

```

CREATE TABLE HasText
(
Col1 char(3)      NOT NULL,
Col2 varchar(5)   NOT NULL,
Col3 text         NOT NULL,
Col4 varchar(20)  NOT NULL
)

INSERT HasText VALUES
('AAA', 'BBB', REPLICATE('X', 250), 'CCC')

```

Now let's find the basic information for this table using the allocation query, and also look at the DBCC IND values for this table:

```

SELECT convert(char(7), object_name(object_id)) AS name,
       partition_id, partition_number AS pnum, rows,
       allocation_unit_id AS au_id,
convert(char(17), type_desc) as page_type_desc,
       total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
  ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.HasText')

```

```
DBCC IND (test, HasText, -1)
```

name	partition_id	pnum	rows	au_id
page_type_desc	pages			
HasText	72057594039435264	1	1	
	72057594044350464	IN_ROW_DATA		2
HasText	72057594039435264	1	1	
	72057594044416000	LOB_DATA		2

PageFID	PagePID	ObjectID	PartitionID
IAM_chain_type		PageType	
1	2251	133575514	72057594039435264
LOB data		3	
1	2264	133575514	72057594039435264
LOB data		10	
1	2265	133575514	72057594039435264
In-row data		1	
1	2266	133575514	72057594039435264
In-row data		10	

You can see that there are two LOB pages (the LOB data page and the LOB IAM page) and two pages for the in-row data (again, the data page and the IAM page). The data page for the in-row data is 2265, and the LOB data is on page 2251. [Figure 6-13](#) shows the output from running DBCC PAGE on page 2265. The row structure is very similar to the row structure in [Figure 6-10](#), except for the text field itself. Bytes 21 to 36 are the 16-byte text pointer, and you can see the value cb08 starting at offset 29. When we reverse the bytes, it becomes 0x08cb, or 2251 decimal, which is the page containing the text data, as we saw in the DBCC IND output.

Figure 6-13. A row containing a text pointer

[\[View full width\]](#)

DATA:

Slot 0, Offset 0x60, Length 40, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5CFEC060

Now let's enable text data in the row, for up to 500 bytes:

```
EXEC sp_tableoption HasText, 'text in row', 500
```

Enabling this option does not force the text data to be moved into the row. We have to update the text value to actually force the data movement:

```
UPDATE HasText  
SET col2 = REPLICATE('Z', 250)
```

If you run DBCC PAGE on the original data page, you'll see the text column of 250 z's is now in the data row and the row is practically identical to the row containing *varchar* data that we saw in [Figure 6-10](#).

Although enabling *text in row* does not move the data immediately, disabling the option does. If you turn off *text in row*, the LOB data moves immediately back onto its own pages, so you must make sure you don't turn this off for a large table during heavy operations.

A final issue when working with LOB data and the *text in row* option is when *text in row* is enabled but for some rows the LOB is longer than the maximum configured length. If you change the maximum length for *text in row* to 50 for the *HasText* table we've been working with, this also forces the LOB data for all rows with more than 50 bytes of LOB data to be moved off the page immediately, just as when you disable the option completely.

```
EXEC sp_tableoption HasText, 'text in row', 50
```

However, just setting the limit to a smaller value is different than disabling the option, in two ways. First, some of the rows might still have LOB data that is under the limit, and for those rows, the LOB data will be stored completely in the data row. Second, if the LOB data doesn't fit, the information stored in the data row itself will not simply be the 16-byte pointer, as would be the case if *text in row* were turned off. Instead, for LOB data that doesn't fit in the defined size, the row will contain a root structure for a B-tree that points to chunks of the LOB data. As long as the *text in row* option is not OFF (or 0), SQL Server will never store the simple 16-byte LOB pointer in the row. It will store either the LOB data itself, if it fits, or the root structure for the LOB data B-tree.

A root structure is at least 24 bytes long (which is why 24 is the minimum size for setting the *text in row* limit). Other information in the root includes:

- Bytes 0 through 1: the type of column; 1 indicates a LOB root
- Byte 2: level in the B-tree
- Byte 3: unused
- Bytes 4 through 7: a value used by optimistic concurrency control for cursors that increases every time a LOB is updated
- Bytes 8 through 11: a random value used by DBCC CHECKTABLE that remains unchanged during the lifetime of each LOB
- Bytes 12 through 23 and each succeeding group of 12 bytes in the column: links to LOB data on a separate page

As indicated earlier, when you first enable *text in row*, no data movement occurs until text data is actually updated. The same is true if the limit is increased—that is, even if the new limit is large enough to accommodate LOB data that was stored outside the row, the LOB data will not be moved onto the row automatically. You must update the actual LOB data first.

Another point to keep in mind is that even if the amount of LOB data is less than the limit, the data will not necessarily be stored in the row. You're still limited to a maximum row size of 8,060 bytes for a single row on a data page, so the amount of LOB data that can be stored in the actual data row might be reduced if the amount of non-LOB data is large. In addition, if a variable-length column needs to grow, it might push LOB data off the page so as not to exceed the 8,060-byte limit. Growth of variable-length columns always has priority over storing LOB data in the row. If no variable-length *char* fields need to grow during an update operation, SQL Server will check for growth of in-row LOB data, in column offset order. If one LOB needs to grow, others might be pushed off the row.

Finally, you should be aware that SQL Server logs all movement of LOB data, which means that reducing the limit of or turning OFF the *text in row* option can be a very time-consuming operation for a large table.

Although large data columns using the LOB data types can be stored and managed very efficiently, using them in your tables can be problematic. Data stored as *text*, *ntext*, or *image* cannot always be manipulated using the normal data manipulation commands, and in many cases you'll need to resort to using the operations *readtext*, *writetext*, and *updatetext*, which require dealing with byte offsets and data-length values. Prior to SQL Server 2005, you had to decide whether to limit your columns to a maximum of 8,000 bytes or deal with your large data columns using different operators than you used for your shorter columns. SQL Server 2005 provides a

solution that gives you the best of both worlds, as we'll see in the next section.

Storage of *varchar(MAX)* Data

SQL Server 2005 gives us the option of defining a variable-length field using the MAX specifier. Although this functionality is frequently described by referring only to *varchar(MAX)*, the MAX specifier can also be used with *nvarchar* and *varbinary*. You can indicate the MAX specifier instead of an actual size when you define a column, variable, or parameter using one of these types. By using the MAX specifier, you leave it up to SQL Server to determine whether to store the value as a regular *varchar*, *nvarchar*, or *varbinary* value or as a LOB. In general, if the actual length is 8,000 bytes or less, the value will be treated exactly as if it were one of the regular variable-length data types, including possibly overflowing onto the special row-overflow pages as discussed earlier. If the actual length is greater than 8,000 bytes, SQL Server will store and treat the value exactly as if it were *text*, *ntext*, or *image*. Because variable-length columns with the MAX specifier are treated either as regular variable-length columns or as LOB columns, no special discussion of their storage is needed. You can see examples of working with *varchar(MAX)* data in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

The size of values specified with MAX can reach the maximum size supported by LOB data, which is currently 2 gigabytes (GB). By using the MAX specifier, though, you are indicating that the maximum size should be the maximum the system supports. If, in the future, you upgrade a table with a *varchar(MAX)* column to some new SQL Server version, the MAX length will be whatever the new maximum is in the new version.

Storage of *sql_variant* Data

The *sql_variant* data type provides support for columns that contain any or all of the SQL Server base data types except LOBs and variable-length columns with the MAX qualifier, *rowversion* (*timestamp*), XML, and the types that can't be defined for a column in a table, namely *cursor* and *table*. For instance, a column can contain a *smallint* value in some rows, a *float* value in others, and a *char* value in the remainder.

This feature was designed to support what appears to be semi-structured data in products sitting above SQL Server. This semi-structured data exists in conceptual tables that have a fixed number of columns of known data types and one or more optional columns whose type might not be known in advance. An example is e-mail messages in Microsoft Outlook and Microsoft Exchange. With the *sql_variant* data type, you can pivot a conceptual table into a real, more compact table with sets of property-value pairs. Here is a graphical example: The conceptual table shown in [Table 6-7](#) has three rows of data. The fixed columns are the ones that exist in every row. Each row can also have values for one or more of the three different properties, which have different data types.

Table 6-7. Conceptual Table with an Arbitrary Number of Columns and Data Types

Row	Fixed Columns	Property 1	Property 2	Property 3
row -1	XXXXXX	value-11		value -13

Row	Fixed Columns	Property 1	Property 2	Property 3
row -2	YYYYYY	value-22		
row -3	ZZZZZZ	value-31	value-32	

This can be pivoted into [Table 6-8](#), where the fixed columns are repeated for each different property that appears with those columns. The column called *value* can be represented by *sql_variant* data and be of a different data type for each different property.

Table 6-8. Semi-Structured Data Stored Using the *sql_variant* Data Type

Fixed Columns	Property	Value
XXXXXX	property-1	value-11
XXXXXX	property-3	value-13
YYYYYY	property-2	value-22

Fixed Columns	Property	Value
ZZZZZZ	property-1	value-31
ZZZZZZ	property-2	value-32

Internally, columns of type *sql_variant* are always considered variable length. Their storage structure depends on the type of data, but the first byte of every *sql_variant* field always indicates the actual data type being used in that row.

I'll create a simple table with a *sql_variant* column and insert a few rows into it so we can observe the structure of the *sql_variant* storage.

```
CREATE TABLE variant (a int, b sql_variant)
GO
INSERT INTO variant VALUES (1, 3)
INSERT INTO variant VALUES (2, 3000000000)
INSERT INTO variant VALUES (3, 'abc')
INSERT INTO variant VALUES (4, current_timestamp)
```

SQL Server decides what data type to use in each row based on the data supplied. For example, the 3 in the first INSERT is assumed to be an integer. In the second INSERT, the 3000000000 is larger than the biggest possible integer, so SQL Server assumes a decimal with a precision of 10 and a scale of 0. (It could have used a *bigint*, but that would need more storage space.) We can now use DBCC IND

to find the first page of the table and use DBCC PAGE to see its contents:

```
DBCC IND (test, variant, -1)
-- (I got a value of file 1, page 2508 for the
data page in this table)
GO
DBCC TRACEON (3604)
DBCC PAGE (test, 1, 2508, 1)
```

[Figure 6-14](#) shows the contents of the four rows. I won't go into the details of every single byte because most are the same as what we've already examined.

Figure 6-14. Rows containing *sql_variant* data

[\[View full width\]](#)

DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C07C060

```
00000000: 30000800 01000000 0200fc01  
00150038 t0  
.....8  
00000010: 01030000 00|||||||||||||||||  
.....
```

Slot 1, Offset 0x75, Length 24, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C07C075

```
00000000: 30000800 02000000 0200fc01  
0018006c t0  
.....l  
00000010: 010a0001 005ed0b2 |||||||.....  
....^..
```

Slot 2, Offset 0x8d, Length 26, DumpStyle
BYTE

Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS

Memory Dump @0x5C07C08D

```
00000000: 30000800 03000000 0200fc01  
001a00a7 t0  
.....
```

```
00000010: 01401f08 d0003461 6263|||||||||||||||  
.  
@....4abc  
  
Slot 3, Offset 0xa7, Length 25, DumpStyle  
BYTE  
  
Record Type = PRIMARY_RECORD  
Record Attributes = NULL_BITMAP  
VARIABLE_COLUMNS  
  
Memory Dump @0x5C07C0A7  
  
00000000: 30000800 04000000 0200fc01  
0019003d †0  
.....=.....  
00000010: 011bd4e6 00bf9700 00|||||||||||||||  
. ....  
-----  
-----  
-----  
  
00000000: 30000800 01000000 0200fc01 00150038  
01030000 00  
  
00000000: 30000800 02000000 0200fc01 0018006c  
010a0001 005ed0b2  
  
00000000: 30000800 03000000 0200fc01 001a00a7  
01401f08 d0003461 6263  
  
00000000: 30000800 04000000 0200fc01 0019003d  
011bd4e6 00bf9700 00
```

The difference between the three rows starts at bytes 13 to 14, which indicate the position where the first variable-length column ends. Because there is only one variable-length column, this is also the length of the row. The *sql_variant* data begins at byte 15. Byte 15 is the code for the data type. You can find the codes in the *system_type_id* column of the *sys.types* catalog view. I've reproduced the relevant part of that view here:

<i>system_type_id</i>	<i>name</i>
34	image
35	text
36	uniqueidentifier
48	tinyint
52	smallint
56	int
58	smalldatetime
59	real
60	money
61	datetime
62	float
98	sql_variant
99	ntext
104	bit
106	decimal
108	numeric
122	smallmoney
127	bigint

165	<code>varbinary</code>
167	<code>varchar</code>
167	<code>tid</code>
167	<code>id</code>
173	<code>binary</code>
175	<code>char</code>
175	<code>empid</code>
189	<code>timestamp</code>
231	<code>sysname</code>
231	<code>nvarchar</code>
239	<code>nchar</code>

In our table, we have the data types 38 hex (which is 56 decimal, which is *int*), 6C hex (which is 108 decimal, which is *numeric*), A7 hex (which is 167 decimal, which is *varchar*), and 3D hex (which is 61 decimal, which is *datetime*). Following the byte for data type is a byte representing the version of the *sql_variant* format, and that is always 1 in SQL Server 2005. Following the version, there can be one of the following four sets of bytes:

- For numeric and decimal: 1 byte for the precision and 1 byte for the scale
- For strings: 2 bytes for the maximum length and 4 bytes for the collation ID
- For *binary* and *varbinary*: 2 bytes for the maximum length
- For all other types: no extra bytes

These bytes are then followed by the actual data in the *sql_variant* column.

There are many other issues related to working with *sql_variant* data and comparing data of different types. Some of the issues are discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

Constraints

Constraints provide a powerful yet easy way to enforce the data integrity in your database. Data integrity comes in three forms:

- **Entity integrity** Ensures that a table has a primary key. In SQL Server 2005, you can guarantee entity integrity by defining PRIMARY KEY or UNIQUE constraints or by building unique indexes. Alternatively, you can write a trigger to enforce entity integrity, but this is usually far less efficient.
- **Domain integrity** Ensures that data values meet certain criteria. In SQL Server 2005, domain integrity can be guaranteed in several ways. Choosing appropriate data types can ensure that a data value meets certain conditions for example, that the data represents a valid date. Other approaches include defining CHECK constraints or FOREIGN KEY constraints or writing a trigger. You might also consider DEFAULT constraints as an aspect of enforcing domain integrity.
- **Referential integrity** Enforces relationships between two tables, a referenced table, and a referencing table. SQL Server 2000 allows you to define FOREIGN KEY constraints to enforce referential integrity, and you can also write triggers for enforcement. It's crucial to note that there are always two sides to referential integrity enforcement. If data is updated or deleted from the referenced table, referential integrity ensures that any data in the referencing table that refers to the changed or deleted data is handled in some way. On the other side, if data is updated or inserted into the referencing table, referential integrity ensures that the new data matches a value in the referenced table.

In this section, I'll briefly describe some of the internal aspects of managing constraints. Constraints are also called *declarative data integrity* because they are part of the actual table definition. This is in contrast to *programmatic data integrity*, which uses stored procedures or triggers, which are discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

Here are the five types of constraints:

- PRIMARY KEY
- UNIQUE
- FOREIGN KEY
- CHECK
- DEFAULT

You might also sometimes see the IDENTITY property and the nullability of a column described as constraints. I typically don't consider these attributes to be constraints; instead, I think of them as properties of a column, for two reasons. First, each constraint has its own row in the `sys.objects` catalog view, but IDENTITY and nullability information is not available in `sys.objects`, only in `sys.columns` and `sys.identity_columns`. This makes me think that these properties are more like data types, which are also viewable through `sys.columns`. Second, when you use the command `SELECT INTO` to make a copy of a table, all column names and data types are copied, as well as IDENTITY information and column nullability, but constraints are *not* copied to the new table. This makes me think that IDENTITY and nullability are more a part of the actual table structure than constraints are.

Constraint Names and Catalog View Information

The following simple create table statement, which includes a primary key on the table, creates a primary key constraint along with the table, and the constraint has a very cryptic-looking name.

```
CREATE TABLE customer
(
    cust_id      int          IDENTITY NOT NULL PRIMARY
    KEY,
    cust_name    varchar(30) NOT NULL
)
```

If you don't supply a constraint name in the CREATE or ALTER TABLE statement that defines the constraint, SQL Server comes up with a name for you.

The constraint produced from the preceding simple statement has a name very similar to the nonintuitive name

PK_customer_0856260D. (The hexadecimal number at the end of the name will most likely be different for a customer table that you create.) All types of single-column constraints use this naming scheme, which I'll explain shortly. The advantage of explicitly naming your constraint rather than using the system-generated name is greater clarity. The constraint name is used in the error message for any constraint violation, so creating a name such as *CUSTOMER_PK* probably makes more sense to users than a name such as *PK_customer_cust_i_0856260D*. You should choose your own constraint names if such error messages are visible to your users. The first two characters (PK) show the constraint

typePK for PRIMARY KEY, UQ for UNIQUE, FK for FOREIGN KEY, and DF for DEFAULT.

Next are two underscore characters, which are used as a separator. (You might be tempted to use one underscore to conserve characters and to avoid having to truncate as much. However, it's common to use a single underscore in a table name or a column name, both of which appear in the constraint name. Using two underscore characters distinguishes the kind of a name it is and where the separation occurs.)

Note



Constraint names are schema scoped, which means they all share the same namespace and hence must be unique within a schema. Within a schema, you cannot have two tables with the same name for their PRIMARY KEY constraint.

Next comes the table name (*customer*), which is limited to 116 characters for a PRIMARY KEY constraint and slightly fewer characters for all other constraint names. For all constraints other than PRIMARY KEY, there are two more underscore characters for separation followed by the next sequence of characters, which is the column name. The column name is truncated to five characters if necessary. If the column name has fewer than five characters in it, the length of the table name portion can be slightly longer.

And finally, the hexadecimal representation of the object ID for the constraint (68E79C55) comes after another separator. This value is used in the *object_id* column of the *sys.objects* catalog view. Object names are limited to 128 characters in SQL Server 2005, so the total length of all portions of the constraint name must also be less than or equal to 128.

Several catalog views contain constraint information. They all inherit the columns from the *sys.objects* view and include additional columns specific to the type of constraint. These views are:

- *sys.key_constraints*
- *sys.check_constraints*
- *sys.default_constraints*
- *sys.foreign_keys*

The *parent_object_id* column, which indicates which object contains the constraint, is actually part of the base *sys.objects* view, but for objects that have no "parent," this column is NULL. Here is a query that allows you to take the name of a constraint, such as *PK_customer_0856260D* shown earlier, and use the hexadecimal number at the end of the name to get the *sys.objects* information for the object that includes this constraint:

```
SELECT * FROM sys.objects
WHERE object_id = (SELECT parent_object_id from
sys.objects
where object_id = 0x0856260D)
```

You might think a query like this is unnecessary because the constraint name includes the object name, but keep in mind that object names are not unique. You can have multiple objects with the name *customer* if they are in different schemas. The *object_id* however, is guaranteed to be unique.

Keep these additional facts about constraints in mind if you want to query the constraint metadata.

- **A constraint is an object.** A constraint has an entry in the *sys.objects* table with a value in the *type* column of C, D, F, PK, or UQ for CHECK, DEFAULT, FOREIGN KEY, PRIMARY KEY, or UNIQUE, respectively.
- **All four constraint catalog views relate to *sys.objects*.** The constraint views are just extensions of the *sys.objects* catalog view. The *object_id* column in the constraint views is the object ID of the constraint, and the *parent_column_id* column of the constraint views is the object ID of the base table on which the constraint is declared.
- ***parent_column_id* values in *sys.check_constraints* and *sys.default_constraints* indicate the column the constraint applies to.** If the constraint is a column-level CHECK or DEFAULT constraint, the constraint view has the column ID of the column in its *parent_column_id* column. This *parent_column_id* is related to the *column_id* of *sys.columns* for the base table represented by *parent_object_id*. A table-level constraint or any PRIMARY KEY/UNIQUE constraint (even if column level) always has 0 in the *parent_column_id* column.
- **To see the columns in a FOREIGN KEY constraint, you can look at the view *sys.foreign_key_columns*.** This view's

constraint_object_id column relates to the row for the constraint in *sys.objects*, and for a composite foreign key, there will be multiple rows with the same *constraint_object_id* value. The *constraint_column_id* indicates the column sequence, the *parent_object_id* and *parent_column_id* indicate the table containing the FOREIGN KEY constraint (the referencing table), and the *referenced_object_id* and *referenced_column_id* provide information about the referenced table.

- ***sys.key_constraints* contains rows for both PRIMARY KEY constraints and UNIQUE constraints** To see the names and order of the columns in a PRIMARY KEY or UNIQUE constraint, you can use the *parent_object_id* and *unique_index_id* columns to find the corresponding row in *sys.indexes* and then look at the *sys.index_columns* table. I'll discuss the relationship between PRIMARY KEY and UNIQUE constraints and indexes in [Chapter 7](#), and we'll look at the metadata in more detail at that time.

Note



Some of the documentation classifies constraints as either table-level or column-level and implies that any constraint defined on the line with a column is a column-level constraint and any constraint defined as a separate line in the table or added to the table with the ALTER TABLE command is a table-level constraint. However, this distinction does not hold true when you look at *constraint* catalog views. The only distinction in the catalog views is whether the constraint is on a single column.

Constraint Failures in Transactions and Multiple-Row Data Modifications

Many bugs occur in application code because developers don't understand how failure of a constraint affects a multiple-statement transaction declared by the user. The biggest misconception is that any error, such as a constraint failure, automatically aborts and rolls back the entire transaction. On the contrary, after an error is raised, it's up to the transaction to proceed and ultimately commit or to roll back. This feature provides the developer with the flexibility to decide how to handle errors. (The semantics are also in accordance with the ANSI SQL-92 standard for COMMIT behavior.) Because many developers have handled transaction errors incorrectly and because it can be tedious to add an error check after every command, SQL Server includes a SET option called XACT_ABORT that causes SQL Server to abort a transaction if it encounters any error during the transaction. The default setting is OFF, which is consistent with ANSI-standard behavior. For more details about handling errors in SQL Server 2005, see *Inside Microsoft SQL Server 2005: T-SQL Programming*, which has an entire chapter about error handling.

A final comment about constraint errors and transactions: a single data modification statement (such as an UPDATE statement) that affects multiple rows is automatically an atomic operation, even if it's not part of an explicit transaction. If such an UPDATE statement finds 100 rows that meet the criteria of the WHERE clause but one

row fails because of a constraint violation, no rows will be updated. I discuss implicit and explicit transactions a bit more in [Chapter 8](#).

The Order of Integrity Checks

The modification of a given row will fail if any constraint is violated or if a trigger rolls back the operation. As soon as a failure in a constraint occurs, the operation is aborted, subsequent checks for that row aren't performed, and no trigger fires for the row. Hence, the order of these checks can be important, as the following list shows.

1. Defaults are applied as appropriate.
2. NOT NULL violations are raised.
3. CHECK constraints are evaluated.
4. FOREIGN KEY checks of *referencing* tables are applied.
5. FOREIGN KEY checks of *referenced* tables are applied.
6. UNIQUE/PRIMARY KEY is checked for correctness.
7. Triggers fire.

Altering a Table

SQL Server 2005 allows existing tables to be modified in several ways. Using the ALTER TABLE command, you can make the following types of changes to an existing table:

- Change the data type or NULL property of a single column
- Add one or more new columns, with or without defining constraints for those columns
- Add one or more constraints
- Drop one or more constraints
- Drop one or more columns
- Enable or disable one or more constraints (applies only to CHECK and FOREIGN KEY constraints)
- Enable or disable one or more triggers

Changing a Data Type

By using the ALTER COLUMN clause of ALTER TABLE, you can modify the data type or NULL property of an existing column. But be aware of the following restrictions:

- The modified column can't be a *text*, *image*, *ntext*, or *rowversion (timestamp)* column.
- If the modified column is the ROWGUIDCOL for the table, only DROP ROWGUIDCOL is allowed; no data type changes are allowed.
- The modified column can't be a computed or replicated column.
- The modified column can't have a PRIMARY KEY or FOREIGN KEY constraint defined on it.
- The modified column can't be referenced in a computed column.
- The modified column can't have the type changed to *timestamp*.
- If the modified column participates in an index, the only type changes that are allowed are increasing the length of a variable-length type (for example, VARCHAR(10) to VARCHAR(20)), changing nullability of the column, or both.
- If the modified column has a UNIQUE OR CHECK constraint defined on it, the only change allowed is altering the length of a variable-length column. For a UNIQUE constraint, the new length must be greater than the old length.
- If the modified column has a default defined on it, the only changes that are allowed are increasing or decreasing the length of a variable-length type, changing nullability, or changing the precision or scale.

- The old type of the column should have an allowed implicit conversion to the new type.
- The new type always has ANSI_PADDING semantics if applicable, regardless of the current setting.
- If conversion of an old type to a new type causes an overflow (arithmetic or size), the ALTER TABLE statement is aborted.

Here's the syntax and an example of using the ALTER COLUMN clause of the ALTER TABLE statement:

SYNTAX

```
ALTER TABLE table-name ALTER COLUMN column-name
    { type_name [ ( prec [, scale] ) ]
  [COLLATE <collation name> ]
    [ NULL | NOT NULL ]
    | {ADD | DROP} {ROWGUIDCOL | PERSISTED}
}
```

EXAMPLE

```
/* Change the length of the emp_lname column in
the employee
    table from varchar(15) to varchar(30) */
ALTER TABLE employee
ALTER COLUMN emp_name varchar(30)
```

Adding a New Column

You can add a new column, with or without specifying column-level constraints. You can add only one column for each ALTER TABLE

statement. If the new column doesn't allow NULLs and isn't an identity column, the new column must have a default constraint defined. SQL Server populates the new column in every row with a NULL, the appropriate identity value, or the specified default. If the newly added column is nullable and has a default constraint, the existing rows of the table are not filled with the default value, but rather with NULL values. You can override this restriction by using the WITH VALUES clause so that the existing rows of the table are filled with the specified default value.

Adding, Dropping, Disabling, or Enabling a Constraint

You can use ALTER TABLE to add, drop, enable, or disable a constraint. The trickiest part of using ALTER TABLE to manipulate constraints is that the word CHECK can be used in three different ways:

- To specify a CHECK constraint.
- To defer checking of a newly added constraint. In the following example, we're adding a constraint to validate that *cust_id* in *orders* matches a *cust_id* in *customer*, but we don't want the constraint applied to existing data:

```
ALTER TABLE orders
WITH NOCHECK
ADD FOREIGN KEY (cust_id) REFERENCES customer
(cust_id)
```

Note



Instead of using WITH NOCHECK, I could use WITH CHECK to force the constraint to be applied to existing data, but that's unnecessary because it's the default behavior.

- To enable or disable a constraint. In this example, we enable all the constraints on the employee table:

```
ALTER TABLE employee  
    CHECK CONSTRAINT ALL
```

The only types of constraints that can be disabled are CHECK constraints and FOREIGN KEY constraints, and disabling tells SQL Server not to validate new data as it is added or updated. You should use caution when disabling and re-enabling constraints. If a constraint was part of the table when the table was created or was added to the table using the WITH CHECK option, SQL Server knows the data conforms to the data integrity requirements of the constraint. The SQL Server optimizer can then take advantage of this knowledge in some cases. For example, if you have a constraint that requires *col1* to be greater than 0, and then an application submits a query looking for all rows where *col1* < 0, if the constraint has always been in effect, the optimizer will know that no rows can satisfy this query and the plan is a very simple plan. However, if the constraint has been disabled and re-enabled without using the WITH CHECK option, there is no guarantee that some of the data in the table won't meet the integrity requirements. You

might not have any data less than or equal to 0, but the SQL Server optimizer will not know that when it is devising the plan.

The catalog views `sys.check_constraints` and `sys.foreign_keys` each have a column called `is_not_trusted`. If you re-enable a constraint and don't use the `WITH CHECK` option to tell SQL Server to revalidate all existing data, the `is_not_trusted` column will be set to 1.

Although you cannot use `ALTER TABLE` to disable or enable a PRIMARY KEY or UNIQUE constraint, you can use the `ALTER INDEX` command to disable the associated index. I'll discuss `ALTER INDEX` in [Chapter 7](#). You can use `ALTER TABLE` to drop PRIMARY KEY and UNIQUE constraints, but you need to be aware that dropping one of these constraints automatically drops the associated index. In fact, the only way to drop those indexes is by altering the table to remove the constraint.

Note



You can't use `ALTER TABLE` to modify a constraint definition. You must use `ALTER TABLE` to drop the constraint and then use `ALTER TABLE` to add a new constraint with the new definition.

Dropping a Column

You can use ALTER TABLE to remove one or more columns from a table. However, you can't drop the following columns:

- A replicated column
- A column used in an index
- A column used in a CHECK, FOREIGN KEY, UNIQUE, or PRIMARY KEY constraint
- A column associated with a default defined using the DEFAULT keyword or bound to a default object
- A column to which a rule is bound

You can drop a column using the following syntax:

```
ALTER TABLE table-name
    DROP COLUMN column-name [, next-column-name] ...
```

Note



Notice the syntax difference between dropping a column and adding a new column: the word COLUMN is required when dropping a column, but not when you add a new column to a table.

Enabling or Disabling a Trigger

You can enable or disable one or more (or all) triggers on a table using the ALTER TABLE command. Triggers are discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*.

Internals of Altering Tables

Note that not all the ALTER TABLE variations require SQL Server to change every row when the ALTER TABLE is issued. SQL Server can carry out an ALTER TABLE command in three basic ways:

- It might need to change only metadata.
- It might need to examine all the existing data to make sure it is compatible with the change but only need to make changes to metadata.
- It might need to physically change every row.

In many cases, SQL Server can just change the metadata (primarily the data seen through `sys.columns`) to reflect the new structure. In particular, the data isn't touched when a column is dropped, when a new column is added and NULL is assumed as the new value for all rows, when the length of a variable-length column is increased, or when a non-nullable column is changed to allow NULLs. The fact that data isn't touched when a column is dropped means that the disk space of the column is not reclaimed. You might have to reclaim the disk space of a dropped column when the row size of a

table approaches or has exceeded its limit. You can reclaim space by creating a clustered index on the table or rebuilding an existing clustered index by using ALTER INDEX, as we'll see in [Chapter 7](#).

Some changes to a table's structure require that the data be examined but not modified. For example, when you change the nullability property to disallow NULLs, SQL Server must first make sure there are no NULLs in the existing rows. A variable-length column can be shortened when all the existing data is within the new limit, so the existing data must be checked. If any rows have data longer than the new limit specified in the ALTER TABLE, the command will fail. So you do need to be aware that for a huge table, this can take some time. Changing a fixed-length column to a shorter type, such as changing an *int* column to *smallint* or changing a *char(10)* to *char(8)*, also requires examining all the data to verify that all the existing values can be stored in the new type. However, even though the new data type takes up fewer bytes, the rows on the physical pages are not modified. If you have created a table with an *int* column, which needs 4 bytes in each row, all rows will use the full 4 bytes. After altering the table to change the *int* to *smallint*, we are restricted in the range of data values we can insert, but the rows continue to use 4 bytes for each value, instead of the 2 bytes that *smallint* requires. You can verify this by using the DBCC PAGE command. Changing a *char(10)* to *char(8)* displays similar behavior, and the rows continue to use 10 bytes, but only 8 are allowed to be inserted until the table is rebuilt by creating or re-creating a clustered index.

Other changes to a table's structure require SQL Server to physically change every row, and as it makes the changes, it has to write the appropriate records to the transaction log, so these changes can be extremely resource intensive for a large table. Another negative side effect in most cases is that when a column is altered to increase its length, the old column is not actually replaced. A new column is added to the table, and DBCC PAGE shows you that the old data is still there. I'll let you explore the page

dumps for this situation on your own, but we can see some of this unexpected behavior by just looking at the column offsets using the column detail query that I showed you earlier in this chapter.

First create a table with all fixed-length columns, including a *smallint* in the first position.

```
CREATE TABLE change
(col1 smallint, col2 char(10), col3 char(5))
Now look at the column offsets:
SELECT c.name AS column_name, column_id,
max_inrow_length, pc.system_type_id, leaf_offset
  FROM sys.system_internals_partition_columns pc
    JOIN sys.partitions p
      ON p.partition_id = pc.partition_id
    JOIN sys.columns c
      ON column_id = partition_column_id
         AND c.object_id = p.object_id
WHERE p.object_id=object_id('fixed')
```

RESULTS:

column_name	column_id	max_inrow_length	system_type_id	leaf_offset
col1	1	2		52
4				
col2	2	10		175
6				
col3	3	5		175
16				

Now change the *smallint* to *int*:

```
ALTER TABLE change  
ALTER COLUMN col1 int
```

And finally, run the column detail query again to see that *col1* now starts much later in the row and that no column starts at offset 4 immediately after the row header information. This new column creation due to an ALTER TABLE takes place even before any data has been placed in the table.

column_name	column_id	max_inrow_length	system_type_id	leaf_offset
col1	1	4	21	56
col2	2	10	6	175
col3	3	5	16	175

Another drawback to SQL Server's behavior in not actually dropping the old column is that we are now more severely limited in the size of the row. The row size now includes the old column, which is no longer usable or visible (unless you use DBCC PAGE). For example, if I create a table with a couple of large fixed-length character columns, as shown here, I can then ALTER the *char(2000)* column to be *char(3000)*.

```
CREATE TABLE bigchange  
(col1 smallint, col2 char(2000), col3 char(1000))
```

```
ALTER TABLE bigchange  
    ALTER COLUMN col2 char(3000)
```

At this point, the length of the rows should be just over 4,000 bytes because there is a 3,000-byte column, a 1,000-byte column, and a *smallint*. However, if I try to add another 3000-byte column, it will fail.

```
ALTER TABLE bigchange  
    ADD col4 char(3000)
```

```
Msg 1701, Level 16, State 1, Line 1  
Creating or altering table 'bigchange' failed  
because the minimum row size would be 9009,  
including 7 bytes of internal overhead. This  
exceeds the maximum allowable table row size of  
8060 bytes.
```

However, if I just create a table with two 3,000-byte columns and a 1,000-byte column, there will be no problem.

```
CREATE TABLE nochange  
(col1 smallint, col2 char(3000), col3 char(1000),  
col4 char(3000))
```

Note that there is no way to ALTER a table to rearrange the logical column order or to add a new column in a particular position in the table. A newly added column always gets the next highest *column_id* value. When you execute SELECT * on a table or look at the metadata with *sp_help*, the columns are always returned in

column_id order. If you need a different order, you have several options:

- Don't use SELECT *; always SELECT a list of columns in the order you want to have them returned.
- Create a view on the table that SELECTs the columns in the order you want them, and then you can SELECT * from the view or run *sp_help* on the view.
- Create a new table, copy the data from the old table, drop the old table, and rename the new table to the old name. Don't forget to re-create all constraints, indexes, and triggers.

You might think that SQL Server Management Studio can add a new column in a particular position or rearrange the column order, but this is not true. Behind the scenes, the tool is actually using the preceding third option and creating a completely new table with all new indexes, constraints, and triggers. If you wonder why simply adding a new column to an existing (large) table is taking a long time, this is probably the reason.

Summary

Tables are at the heart of relational databases in general and SQL Server in particular. In this chapter, we looked at the internal storage issues of various data types, in particular comparing fixed- and variable-length data types. We saw that SQL Server 2005 provides multiple options for storing variable-length data, including data that is too long to fit on a single data page, and you saw that it's simplistic to think that using variable-length data types is either always good or always bad. SQL Server provides user-defined data types for support of domains, and it provides the IDENTITY property to make a column produce auto-sequenced numeric values. You also saw how data is physically stored in data pages, and we queried some of the metadata views that provide information from the underlying (and inaccessible) system tables. SQL Server also provides constraints, which offer a powerful way to ensure your data's logical integrity.

Chapter 7. Index Internals and Management

In this chapter:

<u>Index Organization</u>	<u>250</u>
<u>Creating an Index</u>	<u>254</u>
<u>The Structure of Index Pages</u>	<u>259</u>
<u>Index Space Requirements</u>	<u>275</u>
<u>Special Indexes</u>	<u>280</u>
<u>Table and Index Partitioning</u>	<u>288</u>
<u>Data Modification Internals</u>	<u>296</u>
<u>Managing Indexes</u>	<u>314</u>
<u>Using Indexes</u>	<u>329</u>

Indexes are the other significant user-defined, on-disk data structure besides tables. An index provides fast access to data when the data can be searched by the value that is the index key. I included some very basic information about indexes in [Chapter 2](#), but to really understand the benefit that indexes can provide and how to determine the best indexes for your environment, we need to take a deeper look into the organization of Microsoft SQL Server indexes.

In this chapter, I'll show you the physical organization of index pages for the two types of SQL Server relational indexes, clustered and nonclustered. I'll discuss the options available when you create and re-create relational indexes, and I'll tell you how, when, and why to rebuild your indexes. I'll tell you about the SQL Server 2005 online index rebuilding capability and about the metadata you can use to determine whether your indexes need defragmenting. I will not discuss XML indexes, which are non-relational; they are outside the scope of this book.

This chapter will also cover the internals of data modification operations. Although data modifications can obviously be performed on heaps as well as on indexes, a complete understanding of SQL Server processing of data modifications depends on an understanding of index structures. Finally, I will discuss the ability of SQL Server 2005 to separate data into partitions. Like data modification, this topic can also be relevant to heaps, but because I wanted to discuss table and index partitioning at the same time, I could not address that topic before this point.

Indexes allow data to be organized in a way that allows optimum performance when you access or modify it. SQL Server does not

need indexes to retrieve results for your SELECT statements successfully or to find and modify the specified rows in your data modification statements. However, as your tables get larger, the value of using proper indexes becomes obvious. You can use indexes to quickly find data rows that satisfy conditions in your WHERE clauses, to find matching rows in your JOIN clauses, to effectively enforce referential integrity, or to efficiently maintain uniqueness of your key columns during INSERT and UPDATE operations. Indexes are also useful for achieving vertical partitioning when subsets of columns are required in many queries. In some cases, you can use indexes to help SQL Server sort, aggregate, or group your data or to find the first few rows as indicated in a TOP clause (if the query also includes an ORDER BY clause).

It is the job of the query optimizer to determine which indexes, if any, are most useful in processing a specific query. The final choice of which indexes to use is one of the most important components of the query optimizer's execution plan. The basics of the optimizer's index selection criteria are described in [Chapter 3](#) of *Inside Microsoft SQL Server 2005: T-SQL Querying*. I'll discuss many more details about the optimizer in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*, including the factors that determine whether the query optimizer actually chooses to use indexes at all. In this chapter, I'll focus on what indexes look like and how they can speed up your queries.

Index Organization

Think of the indexes you might see in your everyday life those in books and other documents. In this era of massive amounts of online information and extremely fast online search engines, it's probably rare that you actually open a book to look for information, so you might have to stretch your imagination a bit to follow my analogy here. But assume that all you have is printed reference materials. Suppose that you're trying to create a view in SQL Server using the CREATE VIEW statement and you're using two SQL Server references to find out how to write the statement. One document is a *Microsoft SQL Server 2005 Transact-SQL Language Reference Manual*. The other is *Inside Microsoft SQL Server 2005: T-SQL Programming*. You can quickly find information in either book about views, even though the two books might be organized completely differently.

In the Transact-SQL language reference, all the commands and keywords are organized alphabetically. You know that CREATE VIEW will be near the front with all the other CREATE STATEMENTS, so you can just ignore the last 70 percent of the book. Keywords and phrases are shown at the top of each page to tell you what commands are on that page. So you can quickly flip through just a few pages and end up at a page that has CREATE TYPE as the first key phrase and CREATE XML INDEX as the last keyword, and you know that CREATE VIEW will be on this page. If CREATE VIEW is not on this page, it's not in the book (which is highly unlikely). And once you find the entry for CREATE VIEW, right after CREATE USER, you'll find complete syntax for this command.

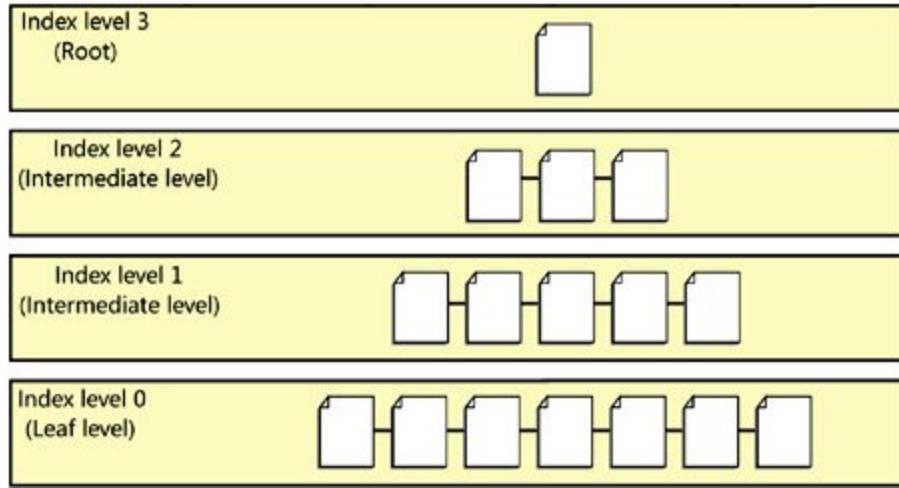
Next you try to find CREATE VIEW in the *Inside Microsoft SQL Server* book. There are no helpful keywords at the top of each page, but there's an index at the back of the book, and all the index entries

are organized alphabetically. So again, you can make use of the fact that CREATE VIEW is near the front of the alphabet and find it quickly. However, unlike in the reference manual, once you find the words CREATE VIEW, you won't see nice neat examples right in front of you. The index only gives you pointersit tells you what pages to look at. It might list two or three pages in the book. If you look up SELECT in the book's index, however, you might find dozens of pages listed. And if you look up *sp_addumpdevice* (a completely deprecated command), you won't find it at all.

These searches through the two books are analogous to using clustered and nonclustered indexes. In a clustered index, the data is actually stored in order, just as the reference manual has all the main topics in order. Once you find the data you're looking for, you're done with the search. In a nonclustered index, the index is a completely separate structure from the data itself. Once you find what you're looking for in the index, you have to follow pointers to the actual data. A nonclustered index in SQL Server is very much like the index in the back of a book. If only one page reference is listed in the index, you can quickly find the information. If a dozen pages are listed, it takes quite a bit longer to find what you need. If hundreds of pages are listed, you might think there's no point in using the index at all. Unless you can narrow your topic of interest to something more specific, the index might not be of much help.

In [Chapter 2](#), I explained that indexes in SQL Server store their information using standard B-trees, as shown in [Figure 7-1](#). A B-tree provides fast access to data by searching on a key value of the index. B-trees cluster records with similar keys. The *B* stands for *balanced*, and balancing the tree is a core feature of a B-tree's usefulness. The trees are managed, and branches are grafted as necessary, so navigating down the tree to find a value and locate a specific record takes only a few page accesses. Because the trees are balanced, finding any record requires about the same amount of resources, and retrieval speed is consistent because the index has the same depth throughout.

Figure 7-1. A B-tree for a SQL Server index



Note that although [Figure 7-1](#) shows two intermediate levels, the number will actually vary. An index consists of a tree with a single root page from which the navigation begins, possible intermediate index levels, and bottom-level leaf pages. You use the index to find the correct leaf page. The number of intermediate levels in an index will vary depending on the number of rows in the table and the size of the index rows. If you create an index using a large key, fewer entries will fit on a page, so more pages (and possibly more levels) will be needed for the index. On a qualified select, update, or delete, the correct leaf page will be the lowest page of the tree in which one or more rows with the specified key or keys reside. A qualified operation is one that affects only specific rows that satisfy the conditions of a WHERE clause, as opposed to one that accesses

the whole table. In any index, whether clustered or nonclustered, the leaf level contains every key value (or combination of values, for a composite index), in key sequence. The biggest difference between clustered and nonclustered indexes is what else is in the leaf level.

Clustered Indexes

The leaf level of a clustered index contains the data pages, not just the index keys. So the answer to the question "What else is in the leaf level of a clustered index besides the key value?" is "Everything else" that is, all the columns of every row are in the leaf level.

Another way to say this is that the data itself is part of the clustered index. A clustered index keeps the data in a table ordered around the key. The data pages in the table are kept in a doubly linked list called a *page chain*. (Note that pages in a heap are not linked together.) The order of pages in the page chain, and the order of rows on the data pages, is the order of the index key or keys. Deciding which key to cluster on is an important performance consideration. When the index is traversed to the leaf level, the data itself has been *retrieved*, not simply *pointed to*.

Because the actual page chain for the data pages can be ordered in only one way, a table can have only one clustered index. The query optimizer strongly favors a clustered index in many cases because such an index allows the data to be found directly at the leaf level. Because it defines the logical order of the data, a clustered index allows especially fast access for queries that look for a range of values. The query optimizer detects that only a certain range of data pages must be scanned.

Most tables should have a clustered index. If your table will have only one index, it generally should be clustered. Many documents describing SQL Server indexes will tell you that the clustered index

physically stores the data in sorted order. This can be misleading if you think of physical storage as the disk itself. If a clustered index had to keep the data on the actual disk in a particular order, it could be prohibitively expensive to make changes. If a page got too full and had to be split in two, all the data on all the succeeding pages would have to be moved down. Sorted order in a clustered index simply means that the data page chain is logically in order. If SQL Server follows the page chain, it can access each row in clustered index key order, but new pages can be added simply by adjusting the links in the page chain. I'll tell you more about page splitting and moving rows later in the chapter when I discuss data modification.

In SQL Server 2005, all clustered indexes are unique. If you build a clustered index without specifying the UNIQUE keyword, SQL Server guarantees uniqueness internally by adding a *uniqueifier* to the rows when necessary. This uniqueifier is a 4-byte value added as an additional clustered index key column. It is added only to the rows that have duplicates of their declared index key column(s). You'll be able to see this extra value when we look at the actual structure of index rows later in this chapter.

Nonclustered Indexes

In a nonclustered index, the leaf level does not contain all the data. In addition to the key values, each index row in the leaf level (the lowest level of the tree) contains a bookmark that tells SQL Server where to find the data row corresponding to the key in the index. A bookmark can take one of two forms. If the table has a clustered index, the bookmark is the clustered index key for the corresponding data row. If the table is a heap (in other words, it has no clustered index), the bookmark is a row identifier (RID), which is an actual row locator in the form *File#:Page#:Slot#*.

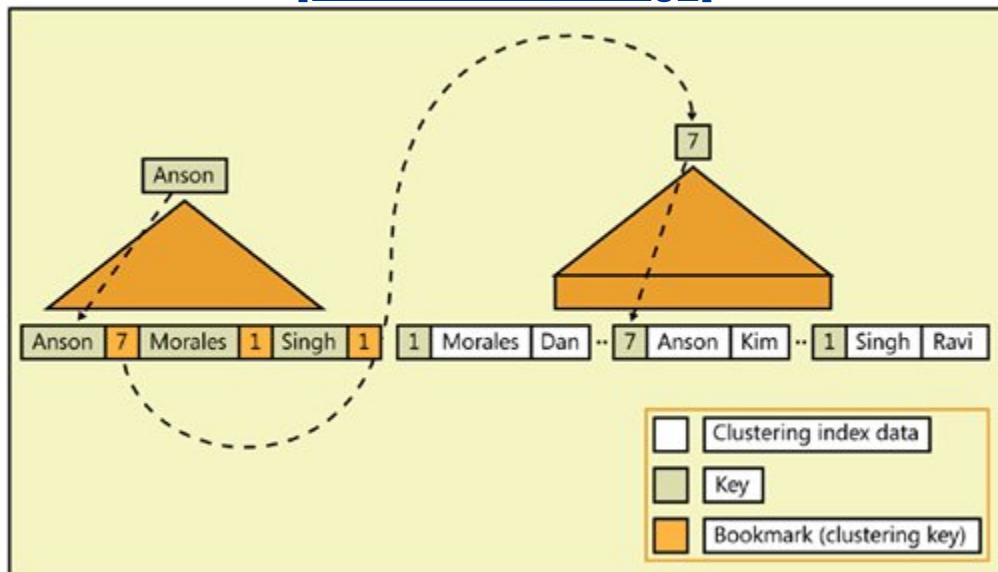
The presence or absence of a nonclustered index doesn't affect how the data pages are organized, so you're not restricted to having only one nonclustered index per table, as is the case with clustered indexes. Each table can include as many as 249 nonclustered indexes, but you'll usually want to have far fewer than that.

When you search for data using a nonclustered index, the index is traversed and then SQL Server retrieves the record or records pointed to by the leaf-level indexes. For example, if you're looking for a data page using a seek operation on an index with a depth of three—a root page, one intermediate page, and the leaf page—all three index pages must be traversed. If the leaf level contains a clustered index key, all the levels of the clustered index must then be traversed to locate the specific row. The clustered index will probably also have three levels, but in this case remember that the leaf level is the data itself. There are two additional index levels separate from the data, typically one less than the number of levels needed for a nonclustered index. The data page still must be retrieved, but because it has been exactly identified, there's no need to scan the entire table. Still, it takes six logical I/O operations to get one data page. You can see that a nonclustered index is a win only if it's highly selective.

[Figure 7-2](#) illustrates this process without showing you the individual levels of the B-trees. We'll see much more detailed illustrations of the content of clustered and nonclustered index pages in the section titled "[The Structure of Index Pages](#)." I want to find the first name for the employee named Anson, and I have a nonclustered index on the last name and a clustered index on the employee ID. The nonclustered index uses the clustered keys as its bookmarks. Searching the index for Anson, SQL Server finds that the associated clustered index key is 7. It then traverses the clustered index looking for the row with a key of 7, and it finds Kim as the first name in the row I'm looking for.

Figure 7-2. Traversing both the clustered and nonclustered indexes to find the first name for the employee named Anson

[[View full size image](#)]



Creating an Index

The basic syntax for creating an index is straightforward:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX  
index_name  
ON table_name (column_name [ASC | DESC][, . . . n])
```

When you create an index, you must specify a name for it. You must also specify the table on which the index will be built and one or more columns. For each column, you can specify that the leaf level store the key values sorted in either ascending (ASC) or descending (DESC) order. The default is ascending. You can specify that SQL Server must enforce uniqueness of the key values by using the keyword UNIQUE. If you don't specify UNIQUE, duplicate key values will be allowed. You can specify that the index be either clustered or nonclustered. Nonclustered is the default.

CREATE INDEX has some additional options available for specialized purposes. You can add a WITH clause to the CREATE INDEX command:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX  
index_name  
ON table_name (column_name [ASC | DESC][, . . . n])  
[WITH  
[FILLFACTOR = fillfactor]  
[,] [PAD_INDEX] = { ON | OFF }]  
[,] [DROP_EXISTING] = { ON | OFF }]  
[,] [IGNORE_DUP_KEY] = { ON | OFF }]  
[,] [SORT_IN_TEMPDB] = { ON | OFF }]  
[,] [STATISTICS_NORECOMPUTE] = { ON | OFF }]
```

```
[[,] ALLOW_ROW_LOCKS = { ON | OFF }]
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }]
[[,] MAXDOP = max_degree_of_parallelism]
[[,] ONLINE = { ON | OFF }]
]
```

Note



SQL Server 2000 does not use ON and OFF. If you want an option set to ON, specify the option; if you don't want the option, don't specify it. SQL Server 2005 offers a backward-compatible syntax without the ON and OFF keywords, but for new development the new syntax is recommended.

FILLFACTOR is probably the most commonly used of these options. FILLFACTOR lets you reserve some space on each leaf page of an index. In a clustered index, because the leaf level contains the data, you can use FILLFACTOR to control how much space to leave in the table itself. By reserving free space, you can later avoid the need to split pages to make room for a new entry. An important fact about FILLFACTOR is that the value is not maintained; it indicates only how much space is reserved with the existing data at the time the index is built. If you need to, you can use the ALTER INDEX command to rebuild the index and reestablish the original

FILLCOMPONENT specified. I'll talk about rebuilding indexes in the section titled "[Managing Indexes](#)."

FILLCOMPONENT isn't usually specified on an index-by-index basis, but you can specify it this way for fine-tuning. If FILLCOMPONENT isn't specified, the serverwide default is used. The value is set for the server via the `sp_configure` procedure, with the `fillfactor` option. This configuration value is 0 by default, which means that leaf pages of indexes are made as full as possible. FILLCOMPONENT generally applies only to the index's leaf page (the data page for a clustered index). In specialized and high-use situations, you might want to reserve space in the intermediate index pages to avoid page splits there, too. You can do this by specifying the `PAD_INDEX` option, which instructs SQL Server to use the same FILLCOMPONENT value at all levels of the index. Just like for FILLCOMPONENT, `PAD_INDEX` is only applicable when an index is created (or re-created).

The `DROP_EXISTING` option specifies that a given index should be dropped and rebuilt as a single transaction. This option is particularly useful when you rebuild clustered indexes. Normally, when a developer drops a clustered index, SQL Server must rebuild every nonclustered index to change its bookmarks to RIDs instead of the clustering keys. Then, if a developer builds (or rebuilds) a clustered index, SQL Server must again rebuild all nonclustered indexes to update the bookmarks. The `DROP_EXISTING` option of the `CREATE INDEX` command allows a clustered index to be rebuilt without having to rebuild the nonclustered indexes twice. If you are creating the index on exactly the same keys that it had previously, the non-clustered indexes do not need to be rebuilt at all. If you are changing the key definition, the non-clustered indexes are rebuilt only once, after the clustered index is rebuilt. Instead of using the `DROP_EXISTING` option to rebuild an existing index, you can use the `ALTER INDEX` command, which is new in SQL Server 2005. I'll discuss this command in the section titled "[ALTER INDEX](#)" later in this chapter.

You can ensure the uniqueness of an index key by defining it as UNIQUE or by defining a PRIMARY KEY or UNIQUE constraint. If an UPDATE or INSERT statement would affect multiple rows, or if even one row is found that would cause duplicates of keys defined as unique, the entire statement is aborted and no rows are affected. Alternatively, when you create the unique index, you can use the IGNORE_DUP_KEY option so that a duplicate key error on a multiple-row INSERT won't cause the entire statement to be rolled back. The nonunique row will be discarded, and all other rows will be inserted or updated. IGNORE_DUP_KEY doesn't allow the uniqueness of the index to be violated; instead, it makes a violation in a multiple-row data modification nonfatal to all the nonviolating rows.

The SORT_IN_TEMPDB option allows you to control where SQL Server performs the sort operation on the key values needed to build an index. The default is that SQL Server uses space from the filegroup on which the index is to be created. While the index is being built, SQL Server scans the data pages to find the key values and then builds leaf-level index rows in internal sort buffers. When these sort buffers are filled, they are written to disk. If the SORT_IN_TEMPDB option is specified, the sort buffers are allocated from *tempdb*, so much less space is needed in the source database. If you don't specify SORT_IN_TEMPDB, not only will your source database require enough free space for the sort buffers and a copy of the index (or the data, if a clustered index is being built), but the disk heads for the database will need to move back and forth between the base table pages and the work area where the sort buffers are stored. If, instead, your CREATE INDEX command includes the option SORT_IN_TEMPDB, performance can be greatly improved if your *tempdb* database is on a separate physical disk from the database you're working with. You can optimize head movement because two separate heads read the base table pages and manage the sort buffers. You can speed up index creation even more if your *tempdb* database is on a faster disk than your user database and you use the SORT_IN_TEMPDB option. As an

alternative to using the SORT_IN_TEMPDB option, you can create separate filegroups for a table and its indexes that is, the table is on one filegroup and its indexes are on another. If the two filegroups are on different disks, you can also minimize the disk head movement.

The STATISTICS_NORECOMPUTE option determines whether the statistics on the index should be updated automatically. Every index maintains a histogram representing the distribution of values for the leading column of the index. SQL Server's query optimizer uses these statistics to determine the usefulness of a particular index when determining a query plan. As data is modified, the statistics can get out of date, and this can lead to less than optimal query plans if the statistics are not updated. In [Chapter 4](#), I told you about a database option to enable all statistics in a database to be automatically updated when needed. The STATISTICS_NORECOMPUTE option allows a particular index to not have its statistics automatically updated when the data changes. Setting this option to OFF overrides an ON value for the AUTO_UPDATE_STATISTICS database option. If the database option is set to OFF, you cannot override that behavior for a particular index, and in that case, all statistics in the database must be manually updated using UPDATE STASTISTICS or `sp_updatestats`. I'll discuss statistics maintenance in a lot more detail in *Inside Microsoft SQL Server 2005: Tuning and Optimization*. I'll discuss the index options to ALLOW_PAGE_LOCKS or ALLOW_ROW_LOCKS in [Chapter 8](#).

The option MAXDOP controls the maximum number of processors that can be used for the index creation operation. It can override the server configuration option *max degree of parallelism* for index building. Allowing multiple processors to be used for the index creation operation can greatly enhance the performance of index build operations. As with other parallel operations, the SQL Server optimizer determines at run time the actual number of processors to use, based on the current load on the system. The MAXDOP value

just sets a maximum. Multiple processors can be used for index creation only when you run SQL Server 2005 Enterprise or Developer edition.

The final option available with the CREATE INDEX command is the ONLINE option, which is new in SQL Server 2005. This option actually deserves an entire section of its own, so I'll discuss it in detail later in the section titled "[Online Index Building](#)."

Included Columns

The key columns in SQL Server 2005 indexes are limited to 16 columns and a total of 900 bytes, just like in previous versions. However, SQL Server 2005 also allows you to define an index with *included columns*.

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX  
index_name  
ON table_name (column_name [ASC | DESC][,...n])  
[ INCLUDE ( column_name [ ,...n ] ) ]
```

These columns listed after the keyword INCLUDE allow you to exceed the 900-byte or 16-key column limits in the leaf level of a nonclustered index. The included columns appear only in the leaf level and do not control the sort order of the index rows in any way. Their purpose is to allow more information in the leaf level so you have more opportunity to use an index-tuning capability called *covering indexes*. A covering index is a nonclustered index in which all the information needed to satisfy a query can be found in the leaf level, so SQL Server doesn't have to access the data pages at all. In certain situations, SQL Server can silently add an included column to your indexes. This might happen when an index is

created on a partitioned table and no ON filegroup or ON *partition_scheme* is specified. I'll discuss this situation in the section titled "[Table and Index Partitioning](#)."

Covering indexes are discussed in *Inside Microsoft SQL Server 2005: T-SQL Querying* and will be discussed in more detail in *Inside Microsoft SQL Server: Tuning and Optimization*.

Index Placement

A final clause in the CREATE INDEX command allows you to specify the placement of the index. You can specify that an index should either be placed on a particular filegroup or should be partitioned according to a predefined partition scheme. By default, if no filegroup or partition scheme is specified, the index will be placed on the same filegroup as the base table. I discussed filegroups in [Chapter 4](#), and I'll tell you about partitioning later in this chapter.

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX  
index_name  
    ON table_name (column_name [ASC | DESC][,...n])  
    [ ON { partition_scheme_name ( column_name )  
          | filegroup_name  
        } ]
```

Constraints and Indexes

When you declare a PRIMARY KEY or UNIQUE constraint, a unique index is created on one or more columns, just as if you had used the CREATE INDEX command. The names of indexes that are built to support these constraints are the same as the constraint

names. In terms of internal storage and maintenance of indexes, there is no difference between unique indexes created using the CREATE INDEX command and indexes created to support constraints. The query optimizer makes decisions based on the presence of the unique index rather than on the fact that a column was declared as a primary key. How the index got there in the first place is irrelevant to the query optimizer.

When you create a table that includes PRIMARY KEY or UNIQUE constraints, you can specify whether the associated index will be clustered or nonclustered and you can also specify the *fillfactor*. Because the *fillfactor* applies only at the time the index is created, and because there is no data when you first create the table, it might seem that specifying the *fillfactor* at that time is completely useless. However, if after the table is populated you decide to rebuild your indexes, if no new *fillfactor* is specified, the original value will be used. You can also specify a *fillfactor* when you use ALTER TABLE to add a PRIMARY KEY or UNIQUE constraint to a table; if the table already has data in it, the *fillfactor* value is applied when you build the index to support the new constraint.

The biggest difference between indexes created using the CREATE INDEX command and indexes that support constraints is in how you can drop the index. The DROP INDEX command allows you to drop only indexes that were built with the CREATE INDEX command. To drop indexes that support constraints, you must use ALTER TABLE to drop the constraint. In addition, to drop a PRIMARY KEY or UNIQUE constraint that has any FOREIGN KEY constraints referencing it, you must first drop the FOREIGN KEY constraint. This can leave you with a window of vulnerability if your goal is to drop indexes and immediately rebuild them, perhaps with a new *fillfactor*. While the FOREIGN KEY constraint is gone, an INSERT statement can add a row to the table that violates your referential integrity.

One way to avoid this problem is to use ALTER INDEX, which allows you to drop and rebuild one or all of your indexes on a table in a single statement, without requiring the auxiliary step of removing FOREIGN KEY constraints. Alternatively, you can use the CREATE INDEX command with the DROP_EXISTING option if you want to rebuild existing indexes without having to drop and re-create them in two steps. Although you can normally use CREATE INDEX with DROP_EXISTING to redefine the properties of an index such as the key columns or included columns, or whether the index is unique if you use CREATE INDEX with DROP_EXISTING to rebuild an index that supports a constraint, you cannot make these kinds of changes. The index must be re-created with the same columns, in the same order, and the same values for uniqueness and clustering. We'll look at more details regarding rebuilding indexes in the section titled "Index Maintenance."

The issue of whether a unique index should be defined using a UNIQUE or PRIMARY KEY constraint is a common concern and a frequent cause of confusion. As I mentioned earlier, there is no internal difference in structure, or in the optimizer's choices, for a unique clustered index built using the CREATE INDEX command or one that was built to automatically support a PRIMARY KEY constraint. The difference is really a design issue, so it is beyond the scope of this book, which deals with internals. However, I will point out that a constraint is a logical construct and an index is a physical one. When you build an index, you are asking SQL Server to create a physical structure that takes up storage space and must be maintained during data modification operations. When you define a constraint, you are defining a property of your data and expecting SQL Server to enforce that property, but you are not telling SQL Server *how* to enforce it. In the current version, SQL Server supports PRIMARY KEY and UNIQUE constraints by creating unique indexes, but there is no requirement that it do so. In a future version, SQL Server might have some other way of enforcing uniqueness besides building an index, but SQL Server 2005 does not.

The Structure of Index Pages

Note



Basic index organization is discussed in *Inside Microsoft SQL Server 2005: T-SQL Querying*, but because it is crucial that you understand the basic structure before reading the rest of this chapter, some overlap between the two volumes is necessary. However, the T-SQL querying volume looks at indexes from the perspective of using them to retrieve results from a query, and this chapter looks in depth at their internal structure.

Index pages are structured much like data pages. As with all other types of pages in SQL Server, index pages have a fixed size of 8 kilobytes (KB), or 8,192 bytes. Index pages also have a 96-byte header, but just like in data pages, there is an offset array at the end of the page with 2 bytes for each row to indicate the offset of that row on the page. Each index has a row in the *sys.indexes* catalog view, with an *index_id* value of either 1, for a clustered index, or a number between 2 and 250, indicating a nonclustered index. In [Chapter 6](#), where I started telling you about *sys.indexes*, I said that even heaps have a row in *sys.indexes*, with an *index_id* value of 0. Most of the information in *sys.indexes* is basic property information about indexes, including the values for each of the index options

that we can specify in the CREATE INDEX command. To get space usage information about indexes, we can join `sys.indexes` with `sys.partitions` and `sys.allocation_units`. In [Chapter 6](#), I showed a query that I referred to as the "allocation query," which reports the number of pages used for each partition of each table or index, with a separate row for each type of storage (in-row, row-overflow, or LOB). We'll look at more details about getting the space used for indexes later in this chapter.

Although that allocation query returns space used for each table and index, it does not report any information about the location of the data. For that, we'll need to use the undocumented catalog view called `sys.system_internals_allocation_units` that I told you about in [Chapter 6](#). In that chapter, we looked at the `first_page` column and I showed you how to take the hexadecimal value and convert it to a file number and page number. Every IAM chain (a chain of the same type of page for one partition of one table or index) has a `first_page`, which indicates the first page in the leaf level of the index. The view `sys.system_internals_allocation_units` also has a column called `root_page` and one called `first_iam_page`. The value for `root_page` can be converted to the file number and page number of the index's root, and for a heap, `root_page` stores the last page of the table. For heaps, first and last really have no meaning, but you can just think of them as the first page and last page returned by running DBCC IND to get the complete list of pages.

As I mentioned in [Chapter 6](#), if you prefer not to have to do hex-to-decimal conversions to find out actual page numbers, or if you want to get ALL the pages from a table or index, you can use the DBCC IND command. We looked at some of the basic information returned by DBCC IND in [Chapter 6](#), but now we'll look at the rest of the details. Keep in mind that Microsoft doesn't support this command and that it isn't guaranteed to maintain the same behavior from release to release. In fact, the behavior of this command changed between the initial release of SQL Server 2005 and the release of Service Pack 1 (SP1). However, I find it enormously useful and will

continue to use it as long as it is available. The description below refers to the use of DBCC IND in SQL Server 2005 SP1.

The DBCC IND command has four parameters, but only the first three are required.

```
DBCC IND ( { 'dbname' | dbid }, { 'objname' |  
objid },  
           { nonclustered indid | 1 | 0 | -1 | -2 } [,  
partition_number] )
```

The first parameter is the database name or the database ID. The second parameter is an object name or object ID within the database; the object can be either a table or an indexed view. The third parameter is a specific nonclustered index ID or the value 1, 0, 1, or 2. The values for this parameter have the following meanings.

- 0 Displays information for in-row data pages and in-row IAM pages of the specified object.
- 1 Displays information for all pages, including IAM pages, data pages, and any existing LOB pages or row-overflow pages of the requested object. If the requested object has a clustered index, the index pages are included.
- 1 Displays information for all IAMs, data pages, and index pages for all indexes on the specified object. This includes LOB and row-overflow data.

2	Displays information for all IAMs for the specified object.
Nonclustered	Displays information for all IAMs, data pages, and index pages for one index. This includes LOB and row-overflow data that might be part of the indexes INCLUDED columns.

The final parameter is optional, to maintain backward compatibility with DBCC IND SQL Server 2000. It specifies a particular partition number, and if no value is specified or a 0 is given, information for all partitions is displayed.

Unlike DBCC PAGE, SQL Server does not require that you enable trace flag 3604 before running DBCC IND. However, because I usually run DBCC IND to find out page numbers in preparation for using DBCC PAGE, it's a good idea to turn the trace flag on before running either command.

The columns in the result set are described in [Table 7-1](#). Note that all page references have the file and page component conveniently split between two columns, so you don't have to do any conversion.

Table 7-1. Column Descriptions for DBCC IND Output

Column	Meaning
<i>PageFID</i>	Index file ID
<i>PagePID</i>	Index page ID
<i>IAMFID</i>	File ID of the IAM managing this page
<i>IAMPID</i>	Page ID of the IAM managing this page
<i>ObjectID</i>	Object ID
<i>IndexID</i>	Index ID
<i>PartitionNumber</i>	Partition number within the table or index for this page
<i>PartitionID</i>	ID for the partition containing this page (unique in the database)
<i>iam_chain_type</i>	Type of allocation unit this page belongs to: in-row data, row-overflow data, or LOB data
<i>PageType</i>	Page type: 1 = data page, 2 = index page, 3 = LOB_MIXED_PAGE, 4 = LOB_TREE_PAGE, 10 = IAM page

Column	Meaning
<i>IndexLevel</i>	Level of index; 0 is leaf
<i>NextPageFID</i>	File ID for next page at this level
<i>NextPagePID</i>	Page ID for next page at this level
<i>PrevPageFID</i>	File ID for previous page at this level
<i>PrevPagePID</i>	Page ID for previous page at this level

Most of the return values were described in [Chapter 6](#) because they are equally relevant to heaps. When dealing with indexes, we also can look at the *IndexID* column, which will be 0 for a heap, 1 for pages from a clustered index, and a number between 2 and 250 for any nonclustered index pages. The *IndexLevel* value allows us to see at what level of the index tree a page is at, with a value of 1 meaning the leaf level. The highest value for any particular index is therefore the root page of that index, and you should be able to verify that the root page is the same value you get from the `sys.system_internals_allocation_units` view in the *root_page* column. The remaining four columns indicate the page linkage, at each level of each index. For each page, there is a file ID and page ID for the next page and a file ID and page ID for the previous page. Of course, for the root pages, all these values will be 0. You can also determine the first page by finding one with zeros for the previous page, and you can find the last page because it will have

zeros for the next page. Because the output of this DBCC command is so wide and hard to display in a page of a book, I have created a script that copies the output of this command to a table. Once we have this information in a table, we can query it and retrieve just the columns we are interested in. Here is a script that creates a table called *sp_table_pages* with columns to hold all the returned information from DBCC IND. Note that any object created in the *master* database with a name that starts with *sp_* can be accessed from any database, without having to qualify it with the database name.

```
USE master;
GO
CREATE TABLE sp_table_pages
(PageFID tinyint,
PagePID int,
IAMFID tinyint,
IAMPID int,
ObjectID int,
IndexID tinyint,
PartitionNumber tinyint,
PartitionID bigint,
iam_chain_type varchar(30),
PageType tinyint,
IndexLevel tinyint,
NextPageFID tinyint,
NextPagePID int,
PrevPageFID tinyint,
PrevPagePID int,
Primary Key (PageFID, PagePID));
```

The following code truncates the *sp_table_pages* table and then fills it with DBCC IND results from the *Sales.SalesOrderDetail* table in the *AdventureWorks* database.

```
TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind ( AdventureWorks,
[Sales.SalesOrderDetail], -1)' );
```

Once you have the results of DBCC IND in a table, you can select any subset of rows or columns that you are interested in. I'll use *sp_table_pages* to report on DBCC IND information for many examples in this chapter. You can then use DBCC PAGE to examine index pages, just as you do for data pages. However, if you use DBCC PAGE with style 3 to print out the details of each column on each row on an index page, the output looks quite different. We'll see some examples after I talk about index the structure of index pages.

Index pages fall into three basic types: leaf level for nonclustered indexes, node (nonleaf) level for clustered indexes, and node level for nonclustered indexes. There isn't really a separate structure for leaf level pages of a clustered index because those are the data pages, which we've already seen in detail. There is, however, one special case for leaf-level clustered index pages which I'll tell you about now.

Clustered Index Rows with a Uniqueifier

If your clustered index was not created with the UNIQUE property, SQL Server adds a 4-byte field when necessary to make each key unique. Because clustered index keys are used as bookmarks to identify the base rows being referenced by nonclustered indexes, there needs to be a unique way to refer to each row in a clustered index.

SQL Server adds the uniqueifier only when necessary—that is, when duplicate keys are added to the table. As an example, I'll create a small table with all fixed-length columns and then add a clustered, nonunique index to the table. (I created an identical table with a different name in [Chapter 6](#).)

```
USE AdventureWorks;
GO
CREATE TABLE Clustered_Dupes
    (Col1 char(5)    NOT NULL,
     Col2 int        NOT NULL,
     Col3 char(3)    NULL,
     Col4 char(6)    NOT NULL);
GO
CREATE CLUSTERED INDEX Cl_dupes_col1 ON
Clustered_Dupes(col1);
```

If you look at the row in the `sysindexes` compatibility view for this table, you'll notice something unexpected.

```
SELECT first, indid, keycnt, name FROM sysindexes
WHERE id = object_id ('Clustered_Dupes');
```

RESULT:

indid	keycnt	name
1	2	Cl_dupes_col1

The column called `keycnt`, which indicates the number of keys an index has, has a value of 2. (Note that this column is available only in the compatibility view `sysindexes`, and not in the catalog view `sys.indexes`.) If I had created this index using the `UNIQUE` qualifier,

the *keycnt* value would be 1. If I had looked at the *sysindexes* row before adding a clustered index, when the table was still a heap, the row for the table would have had a *keycnt* value of 0 because a heap has no keys. I'll now add the same initial row that I added to the identical table I created in [Chapter 6](#). To find the first (or only) data page of the table, three methods are available. I can examine the first column from the *sysindexes* compatibility view, I can run a query that joins the catalog view *sys.indexes* to *sys.partitions* and *sys.system_internals_allocation_units*, or I can run DBCC IND and find the page with a *PageType* of 1 with no previous pagethat will be the first page in logical order in the data pages, which are the leaf level of the clustered index. I'll use the third option and make use of the *sp_table_pages* table that I created earlier.

```
INSERT Clustered_Dupes VALUES ( 'ABCDE', 123, null,
'CCCC');
GO
TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ( 'dbcc ind ( AdventureWorks,
Clustered_Dupes, -1)' );
SELECT PageFID, PagePID FROM sp_table_pages
    WHERE PageType = 1 and PrevPageFID = 0 and
PrevPagePID = 0;
```

Although this single-row table has only one data page, I can use the preceding WHERE clause to find the first data page in any table with a clustered index, after inserting the results of DBCC IND into the table. Here are my results:

PageFID	PagePID
1	21088

The output tells me that the first page of the table is on file 1, page 21088, so I can use DBCC PAGE to examine that page. Remember to turn on trace flag 3604 before executing the undocumented DBCC PAGE command.

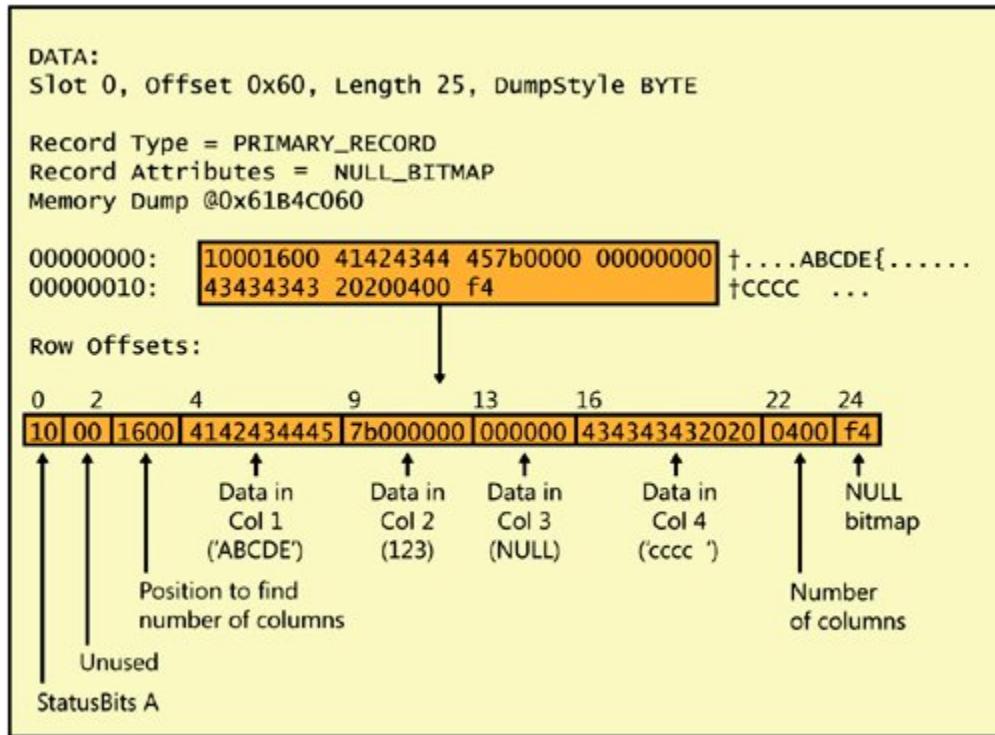
```
DBCC TRACEON (3604);
GO
DBCC PAGE (AdventureWorks,1,21088, 1);
```

The only row on the page looks exactly like the row shown in [Figure 6-10](#) in [Chapter 6](#), but I'll reproduce it again here, in [Figure 7-3](#).

When you read the row output from DBCC PAGE, remember that each two displayed characters represents a byte. For character fields, you can just treat each byte as an ASCII code and convert that to the associated character. Numeric fields are stored with the low-order byte first, so within each numeric field, we must swap bytes to determine what value is being stored. Right now, there are no duplicates of the clustered key, so no extra information has to be provided. However, if I add two additional rows, with duplicate values in *col1*, the row structure changes:

```
INSERT Clustered_Dupes VALUES ('ABCDE', 456, null,
'AAAA');
INSERT Clustered_Dupes VALUES ('ABCDE', 64, null,
'BBBB');
```

Figure 7-3. A data row containing all fixed-length columns and a unique value in the clustered key column

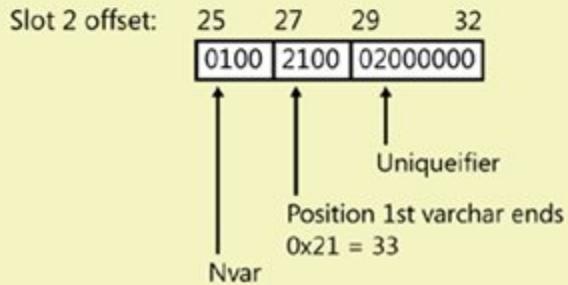


[Figure 7-4](#) shows the three rows that are now on the page.

Figure 7-4. Three data rows containing duplicate values in the clustered key column

[View full size image](#)

```
Slot 0, offset 0x60, Length 25, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP
Memory Dump @0x6211C060
00000000: 10001600 41424344 457b0000 00e76a3f +....ABCDE{....j?
00000010: 43434343 20200500 e8|||||||||||||||||CCCC ...
Slot 1, offset 0x79, Length 33, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Memory Dump @0x6211C079
00000000: 30001600 41424344 45c80100 00000000 +...ABCDE.....
00000010: 44444444 20200500 e8010021 00010000 +DDDD ....!....
00000020: 00|||||||||||||||||||||||||||||||||||||.
Slot 2, offset 0x9a, Length 33, DumpStyle BYTE
Record Type = PRIMARY_RECORD
Record Attributes = NULL_BITMAP VARIABLE_COLUMNS
Memory Dump @0x6211C09A
00000000: 30001600 41424344 45400000 00000000 +0...ABCDE@.....
00000010: 45454545 20200500 e8010021 00020000 +EEEE ....!....
00000020: 00|||||||||||||||||||||||||||||||||.
```



The first difference in the second two rows is that the bits in the first byte (*TagA*) are different. Bit 5 is on, giving *TagA* a value of 0x30,

which means the variable block is present in the row. Without this bit on, *TagA* would have a value of 0x10. The extra variable-length portions of the second two rows are shaded in the figure. You can see that 8 extra bytes are added when we have a duplicate row. In this case, the first 4 extra bytes are added because the uniqueifier is considered a variable-length column. Because there were no variable-length columns before, SQL Server adds 2 bytes to indicate the number of variable-length columns present. These bytes are at offsets 33-34 in these rows with the duplicate keys and have the value of 1. The next 2 bytes (offsets 35-36) indicate the position where the first variable-length column ends. In both these rows, the value is 0x29, which converts to 41 decimal. The last 4 bytes (offsets 37-40) are the actual uniqueifier. In the second row, which has the first duplicate, the uniqueifier is 1. The third row has a uniqueifier of 2.

Index Row Formats

The row structure of an index row is very similar to the structure of a data row. An index row does not use the *TagB* or *Fsize* row header values. In place of the *Fsize* field, which indicates where the fixed-length portion of a row ends, the page header *pminlen* value is used to decode an index row. The *pminlen* value indicates the offset at which the fixed-length data portion of the row ends. If the index row has no variable-length or nullable columns, that is the end of the row. Only if the index row has nullable columns are the field called *Ncol* and the null bitmap both present. The *Ncol* field contains a value indicating how many columns are in the index row; this value is needed to determine how many bits are in the null bitmap. Data rows have an *Ncol* field and null bitmap whether or not any columns allow NULL, but index rows have only a null bitmap and an *Ncol* field if any NULLs are allowed in an index column. [Table 7-2](#) shows the general format of an index row.

Table 7-2. Information Stored in an Index Row

Information	Mnemonic	Size
Status Bits A	<i>TagA</i>	1 byte
	Some of the relevant bits are: <ul style="list-style-type: none">• Bits 1 through 3 Taken as a 3-bit value, 0 indicates a primary record, 3 indicates an index record, and 5 indicates a ghost index record. (I'll discuss ghost records later in this chapter.)• Bit 4 Indicates that a NULL bitmap exists.• Bit 5 Indicates that variable-length columns exist in the row.	
Fixed-length data	<i>Fdata</i>	<i>pminlen 1</i>

Information	Mnemonic	Size
Number of columns	$Ncol$	2 bytes
NULL bitmap (1 bit for <i>Nullbits</i> each column in the table; a 1 indicates that the corresponding column is NULL)		Ceiling ($Ncol / 8$)
Number of variable-length columns; only present if > 0	$VarCount$	2 bytes
Variable column offset <i>VarOffset</i> array; only present if $VarCount > 0$		$2 * VarCount$
Variable-length data, <i>VarData</i> if any		

The data contents of an index row depend on the level of the index. All rows except those in the leaf level contain a 6-byte down-page pointer in addition to the key values and bookmarks. The down-page pointer is the last column in the fixed-data portion of the row. In nonclustered indexes, the nonclustered keys and bookmarks are

treated as normal columns. They can reside in either the fixed or variable portion of the row, depending on how each of the key columns was defined. A bookmark that is a RID, however, is always part of the fixed-length data.

If the bookmark is a clustered key value and the clustered and nonclustered indexes share columns, the actual data is stored only once. For example, if your clustered index key is *lastname* and you have a nonclustered index on (*firstname*, *lastname*), the index rows will not store the value of *lastname* twice. I'll show you an example of this shortly.

Clustered Index Node Rows

The node levels of a clustered index contain pointers to pages at the next level down in the index, along with the first key value on each page pointed to. Page pointers are 6 bytes: 2 bytes for the file number and 4 bytes for the page number in the file. The following code creates and populates a table that I'll use to show you the structure of clustered index node rows.

```
USE AdventureWorks;
GO
CREATE TABLE clustered_nodupes (
    id int NOT NULL ,
    str1 char (5) NOT NULL ,
    str2 char (600) NULL
);
GO
CREATE CLUSTERED INDEX idxCL ON
Clustered_Nodupes(str1);
GO
SET NOCOUNT ON;
GO
DECLARE @i int;
```

```

SET @i = 1240;
WHILE @i < 13000 BEGIN
    INSERT INTO Clustered_Nodupes
        SELECT @i, cast(@i AS char), cast(@i AS char);
    SET @i = @i + 1;
END;
GO

TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind ( AdventureWorks,
Clustered_Nodupes, -1)' );
SELECT PageFID, PagePID, IndexLevel, PageType FROM
sp_table_pages
WHERE IndexId = 1 and IndexLevel >= 0
    and PrevPageFID = 0 and PrevPagePID = 0;
RESULT:
PageFID PagePID      IndexLevel PageType
----- ----- -----
1       21090        0          1
1       21092        1          2
1       21738        2          2

```

```

DBCC TRACEON (3604);
GO
DBCC PAGE (AdventureWorks,1,21090, 1);

```

The results from the *sp_table_pages* table show us the first page at each level of the clustered index. We know it's the clustered index because the *IndexID* value is 1; we know it's the first page because the *PrevPage* values are 0. The filter on *IndexLevel* eliminates the

IAM page for this index because its *IndexLevel* is null. The root of the index is the page at the highest *IndexLevel*, so in this case, the root page is 21738. Using DBCC PAGE to look at the root, we can see two index entries, as shown in [Figure 7-5](#). When you examine index pages, you need to be aware that the first index key entry on each page is frequently either meaningless or empty. The down-page pointer is valid, but the data for the index key might not be a valid value. When SQL Server traverses the index, it starts looking for a value by comparing the search key with the second key value on the page. If the value being sought is less than the second entry on the page, SQL Server follows the page pointer indicated in the first index entry. In this example, the down-page pointer is at byte offsets 6 through 9, with a hex value of 0x5264. (The next two bytes are 0x0001 for the file ID.) In decimal, the page number for the first page at the next level down is 21092, which is the same value we saw earlier when looking at the output of DBCC IND.

Figure 7-5. The root page of a clustered index

```

DATA:
Slot 0, offset 0x60, Length 12, DumpStyle BYTE

Record Type = INDEX_RECORD
Record Attributes =
Memory Dump @0x61DBC060

00000000: 06206806 f03f6452 00000100 tttttttttt.
h..?dR....
```

Slot 1, offset 0x6c, Length 12, DumpStyle BYTE

```

Record Type = INDEX_RECORD
Record Attributes =
Memory Dump @0x61DBC06C

00000000: 06343532 3920eb54 00000100 tttttttttt.4529 .T..
```

Slot 1 offset:	0	1	5	6	9	10	11
	06	3435323920	eb540000	0100			

TagA

Clustered key

Down-page pointer
0x54eb = 21739

Down-page file ID

We can then use DBCC PAGE again to look at page 21092, but I'll leave that to you to examine on your own. The row structure is identical to the rows on page 21738, the root page.

When I look at the first row on page 21092 (not shown), I see that it has a meaningless key value, but the page-down pointer should be the first page in clustered index order. The hex value for the page-down pointer is 0x5262, which is 21090 decimal. That page is the leaf-level page, and it has a structure of a normal data page. If we use DBCC PAGE to look at that page, we'll see that the first row has

the value of '10000', which is the minimum value of the str1 column in the table. We can verify this with the following query:

```
SELECT min(str1), min(id)
FROM Clustered_Nodupes;
```

Remember that the clustered index column is a char(5) column. Although the lowest (and first) number I inserted was 1240, which is the minimum value for the *id* column, when converted to a char(5), that's '1240'. When sorted alphabetically, '10000' comes well before '1240'.

Nonclustered Index Leaf Rows

Index node rows in a clustered index contain only the first key value of the page they are pointing to in the next level and page-down pointers to guide SQL Server in traversing the index. Nonclustered index rows can contain much more information. The rows in the leaf level of a nonclustered index contain every key value and a bookmark. I'll show you three examples of nonclustered index leaf rows. First we'll look at the leaf level of a nonclustered index built on a heap. I'll use the same code I used to build the clustered index with no duplicates in the previous example, but the index I build on the *str1* column will be nonclustered. I'll also put only a few rows in the table so the root page will be the entire index.

```
USE AdventureWorks;
SET NOCOUNT ON;
CREATE TABLE NC_Heap_Nodupes (
    id int NOT NULL ,
    str1 char (5) NOT NULL ,
    str2 char (600) NULL
);
```

```

GO
CREATE UNIQUE INDEX idxNC_heap ON NC_Heap_Nodupes
(str1);
GO
SET NOCOUNT ON;
GO
DECLARE @i int;
SET @i = 1240;
WHILE @i < 1300 BEGIN
    INSERT INTO NC_Heap_Nodupes
        SELECT @i, cast(@i AS char), cast(@i AS char);
    SET @i = @i + 1;
END;
GO

TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind ( AdventureWorks,
NC_Heap_Nodupes, -1)' );
SELECT PageFID, PagePID, IndexID, IndexLevel,
PageType
FROM sp_table_pages
WHERE IndexLevel >= 0;
RESULTS:
PageFID PagePID      IndexID IndexLevel PageType
----- ----- -----
1       21107        0       0          1
1       21109        2       0          2
1       21111        0       0          1
1       22088        0       0          1
1       22089        0       0          1
1       22090        0       0          1
1       22091        0       0          1

```

You can see that there are six data pages in the heap and one index page in the nonclustered index. That one page is both the leaf and the root. As the leaf, its rows contain bookmarks or pointers to actual data rows. Because this index is built on a heap, the bookmark is an 8-byte RID. These RIDs are 8 bytes long, and we can see them if we look at the page for index 2 returned by the preceding query. For me, it is page 21109.

```
DBCC TRACEON (3604);  
GO  
DBCC PAGE (Adventureworks,1,21109, 1);
```

As you can see in [Figure 7-6](#), this RID is fixed-length and is located in the index row immediately after the index key value of '1240'. The first 4 bytes of the RID (0x5273) are the page number (21107), the next 2 bytes (0x0001) are the file ID, and the last 2 bytes are the slot number (0x0000).

Figure 7-6. An index row on a leaf-level page from a nonclustered index on a heap

```

DATA:
Slot 0, Offset 0x60, Length 14, DumpStyle BYTE

Record Type = INDEX_RECORD
Record Attributes =
Memory Dump @0x5C34C060

00000000: 06313234 30207352 00000100 0000ffff.1240 sR.....

```

Slot 0 offset:	0	1	5	6	9	10	11	12	13
	06	3132343020	73520000	0100	0000				

↑ ↑ ↑ ↑
 TagA nc key Bookmark page pointer
 ↓ ↓ ↓ ↓
 Bookmark slot # Bookmark file ID

0x5273 = 21107

I'll build a nonclustered index on a similar table, but first I'll build a clustered index on a *varchar* column so we can see what an index looks like when a bookmark is a clustered index key. Also, in order not to have the same values in the same sequence in both the *str1* and *str2* columns, I'll introduce a bit of randomness into the generation of the values for *str2*. If you run this script, the randomness might generate some duplicate values for the unique clustered index column *str2*. Because each row is inserted in its own INSERT statement, a violation of uniqueness will cause only that one row to be rejected. If you get error messages about PRIMARY KEY violation, just ignore them. You'll still have enough rows.

```

USE AdventureWorks;
GO
SET NOCOUNT ON;

```

```

GO
CREATE TABLE NC_Nodupes (
    id int NOT NULL ,
    str1 char (5) NOT NULL ,
    str2 varchar (10) NULL
);
GO
CREATE UNIQUE CLUSTERED INDEX idxcl_str2 on
NC_Nodupes (str2);
CREATE UNIQUE INDEX idxNC ON NC_Nodupes (str1);
GO
SET NOCOUNT ON;
GO
DECLARE @i int;
SET @i = 1240;
WHILE @i < 1300 BEGIN
    INSERT INTO NC_Nodupes
        SELECT @i, cast(@i AS char),
               cast(cast(@i * rand() AS int) as char);
    SET @i = @i + 1;
END;
GO

TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind ( AdventureWorks, NC_Nodupes,
-1)' );
SELECT PageFID, PagePID, IndexID, IndexLevel,
PageType
FROM sp_table_pages
WHERE IndexLevel >= 0;

RESULT:
PageFID PagePID      IndexID IndexLevel PageType
----- ----- ----- ----- -----

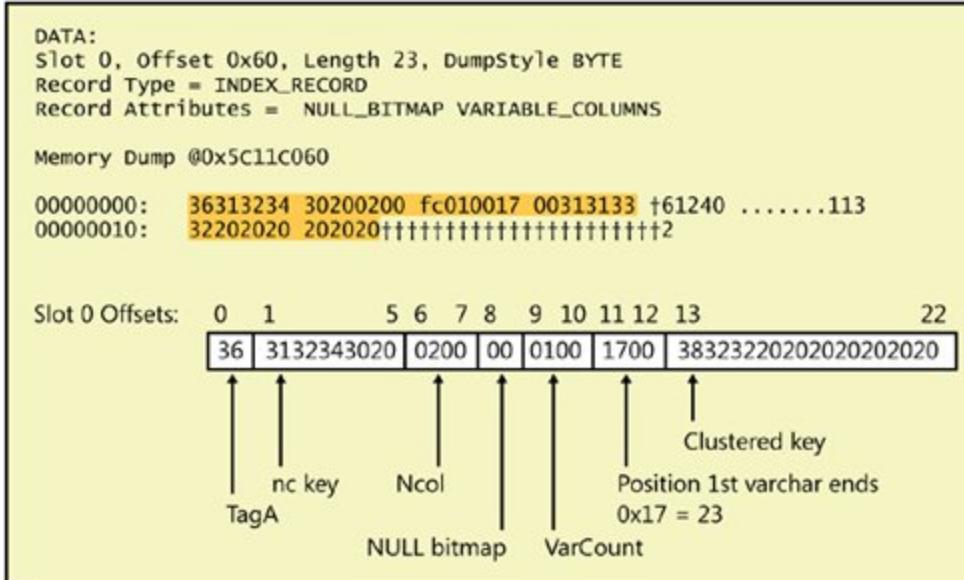
```

1	22104	1	0	1
1	22106	2	0	2

[Figure 7-7](#) shows the first index row from the nonclustered index on *NC_Nodupes*.

Figure 7-7. An index row on a leaf-level page from a nonclustered index built on a clustered table

[\[View full size image\]](#)



The index row contains the nonclustered key value of '1240', and the bookmark is the clustered key '1132'. Because the clustered key is variable-length, it comes at the end of the row after the *Ncol* value and null bitmap. A variable-length key column in an index uses only the number of bytes needed to store the value inserted. However, the preceding script creates the *varchar* value by first converting an *int* to a *char*, which will assume a length of 30 and pad the string with blanks to that length. When the session option *ANSI_PADDING* is enabled (the default), the trailing blanks up to the 10th character are preserved when you insert into the *varchar(10)* column in the table.

If you've had enough of translating the cryptic hex strings into meaningful values, another alternative is available. The third parameter to DBCC PAGE provides a different format for the output when you print an index page, as I mentioned earlier. Using format 3 on an index page basically provides tabular output showing the contents of each index row, including the bookmarks.

The following output is from DBCC PAGE on the one index page from my *NC_Nodupes* table. There are 60 rows of data, so the index page has 60 index rows. (Because of the random generation of values, you might have fewer than 60 rows in your table.) Each index row has the nonclustered key as well as the clustered key stored in the row, along with the file and page IDs and the row number on the page where the rows can be found. Because there is just one data page, all the file and page IDs are the same. The final value displayed is called the *KeyHashValue*, which is not actually stored in the index row. It is a fixed-length string derived using a hash formula on all the key columns. This value is used to represent the row in certain other tools. One such tool that we will see in [Chapter 8](#) is the dynamic management view that shows the locks

that are being held. When a lock is held on an index row, the list of locks displays the *KeyHashValue* to indicate which key is locked.

```
DBCC PAGE (AdventureWorks,1,21106, 3);
```

FileId	PageId	Row	Level	str1 (key)	str2
KeyHashValue					
1	22106 (510089f2f438)	0	0	1240	1132
1	22106 (5100c8c3ef21)	1	0	1241	706
1	22106 (51000b90c20a)	2	0	1242	344
1	22106 (51004aa1d913)	3	0	1243	682
1	22106 (51008d37985c)	4	0	1244	926
1	22106 (5100cc068345)	5	0	1245	359
1	22106 (51000f55ae6e)	6	0	1246	383
1	22106 (51004e64b577)	7	0	1247	1052
1	22106 (510081782df0)	8	0	1248	986
1	22106 (5100c04936e9)	9	0	1249	946
1	22106 (5100be983639)	10	0	1250	312
1	22106 (5100ffa92d20)	11	0	1251	735
1	22106 (51003cfa000b)	12	0	1252	1017
1	22106	13	0	1253	1097

(51007dcb1b12)					
1	22106	14	0	1254	615
(5100ba5d5a5d)					
1	22106	15	0	1255	185
(5100fb6c4144)					
1	22106	16	0	1256	394
(5100383f6c6f)					
1	22106	17	0	1257	1027
(5100790e7776)					
1	22106	18	0	1258	1111
(5100b612eff1)					
1	22106	19	0	1259	129
(5100f723f4e8)					
1	22106	20	0	1260	425
(5100e726703b)					
1	22106	21	0	1261	111
(5100a6176b22)					
1	22106	22	0	1262	220
(510065444609)					
1	22106	23	0	1263	503
(510024755d10)					
1	22106	24	0	1264	332
(5100e3e31c5f)					
1	22106	25	0	1265	12
(5100a2d20746)					
1	22106	26	0	1266	182
(510061812a6d)					
1	22106	27	0	1267	982
(510020b03174)					
1	22106	28	0	1268	535
(5100efaca9f3)					
1	22106	29	0	1269	48
(5100ae9db2ea)					
1	22106	30	0	1270	693
(5100d04cb23a)					
1	22106	31	0	1271	200

(5100917da923)					
1	22106	32	0	1272	49
(5100522e8408)					
1	22106	33	0	1273	1170
(5100131f9f11)					
1	22106	34	0	1274	815
(5100d489de5e)					
1	22106	35	0	1275	1114
(510095b8c547)					
1	22106	36	0	1276	1038
(510056ebe86c)					
1	22106	37	0	1277	321
(510017daf375)					
1	22106	38	0	1278	1023
(5100d8c66bf2)					
1	22106	39	0	1279	70
(510099f770eb)					
1	22106	40	0	1280	1107
(5100ed0bee31)					
1	22106	41	0	1281	1108
(5100ac3af528)					
1	22106	42	0	1282	464
(51006f69d803)					
1	22106	43	0	1283	1035
(51002e58c31a)					
1	22106	44	0	1284	50
(5100e9ce8255)					
1	22106	45	0	1285	1143
(5100a8ff994c)					
1	22106	46	0	1286	1036
(51006bacb467)					
1	22106	47	0	1287	943
(51002a9daf7e)					
1	22106	48	0	1288	695
(5100e58137f9)					
1	22106	49	0	1289	797

(5100a4b02ce0)					
1	22106	50	0	1290	269
(5100da612c30)					
1	22106	51	0	1291	202
(51009b503729)					
1	22106	52	0	1292	1063
(510058031a02)					
1	22106	53	0	1293	823
(51001932011b)					
1	22106	54	0	1294	960
(5100dea44054)					
1	22106	55	0	1295	316
(51009f955b4d)					
1	22106	56	0	1296	1251
(51005cc67666)					
1	22106	57	0	1297	1284
(51001df76d7f)					
1	22106	58	0	1298	266
(5100d2ebf5f8)					
1	22106	59	0	1299	1123
(510093daeee1)					

The last example will show a composite nonclustered index on *str1* and *str2* and an overlapping clustered index on *str2*. I'll just show you the code to create the table and indexes; the code to populate the table and to find the root of the index is identical to the code in the previous example.

```
USE AdventureWorks;
CREATE TABLE NC_Overlap (
    id int NOT NULL ,
    str1 char (5) NOT NULL ,
    str2 char (10) NULL
);
```

```

GO
CREATE UNIQUE CLUSTERED INDEX idxCL_overlap ON
NC_Overlap (str2);
CREATE UNIQUE INDEX idxNC_overlap ON NC_Overlap
(str1, str2);
GO

```

Here are several of the leaf-level index rows from the nonclustered composite index on the table *NC_Overlap*.

FileId	PageId	Row	Level	str1 (key)	str2
(key)		KeyHashValue			
1 (e700d6c3e711)	22110	0	0	1240	676
1 (e500845fb78e)	22110	1	0	1241	476
1 (e200b66ef631)	22110	2	0	1242	156

Note that the value in column *str2* (676 in the first row) is part of the index key for the nonclustered index as well as the bookmark because it is the clustered index key. The only real difference between this output and the previous output (besides the values in *str2*, which are generated randomly) is that in this output the *str2* column is labeled a key. It is included in the previous output only because it is the clustered index key, used as the bookmark. Although it serves two purposes in the rows in this output, its value is not duplicated. The value 676 occurs only once, and if you look at the format 1 output of the page, you'll see that it takes the maximum allowed space of 10 bytes. Because this is a leaf-level row, there is

no page-down pointer, and because the index is built on a clustered table, there is no RID, only the clustered key to be used as a bookmark.

Nonclustered Index Node Rows

You now know that the leaf level of a nonclustered index must have a bookmark because from the leaf level you want to be able to find the actual row of data. The nonleaf levels of a nonclustered index only need to help us traverse down to pages at the lower levels. If the nonclustered index is unique, the node rows need to have only the nonclustered key and the page-down pointer. If the index is *not* unique, the row contains key values, a down-page pointer, and a bookmark. I'll leave it to you to create a table that will have enough rows for more than a single nonclustered index level. You can use a script similar to the ones shown previously and change the upper limit of the WHILE loop to about 13000. You can create two tables, one with a non-unique, nonclustered index on *str1* and a clustered index on *str2* and another table with a unique nonclustered index on *str1* and a clustered index on *str2*.

Keep in the mind that for the purposes of creating the index rows, SQL Server doesn't care whether the keys in the non-unique index actually contain duplicates. If the index is not defined to be unique, even if all the values are unique, the nonleaf index rows will contain bookmarks. I'll let you use DBCC IND and DBCC PAGE to view the index rows for yourself.

Index Space Requirements

I've shown you the structure and number of bytes in individual index rows, but you really need to be able to translate that into overall index size. In general, the size of an index is based on the size of the index keys, which determines how many index rows can fit on an index page and the number of rows in the table.

B-Tree Size

When we talk about index size, we usually mean the size of the index tree. The clustered index does include the data, but because you still have the data even if you drop the clustered index, we're usually just interested in how much additional space the nonleaf levels require. A clustered index's node levels typically take up very little space. You have one index row for each page of table data, so the number of index pages at the level above the leaf (level 1) is the bytes available per page divided by the index key row size, and this quotient divided *into* the number of data pages. You can do a similar computation to determine the pages at level 2. Once you reach a level with only one page, you can stop your computations because that level is the root.

For example, consider a table of 10,000 pages and 500,000 rows with a clustered key of a 5-byte fixed-length character. As you saw in [Figure 7-5](#), the index key row size is 12 bytes, so we can fit 674 rows (8,096 bytes available on a page/12 bytes per row) on an index page. Because we need index rows to point to all 10,000 pages, we'll need 15 index pages ($10,000/674$) at level 1. Now, all these index pages need pointers at level 2, and because one page can easily store the 15 index rows needed to point to these 15 pages at level 1, level 2 is the root. If our 10,000-page table also

has a 5-byte fixed-length character nonclustered index, the leaf-level rows (level 0) will be 11 bytes long because they will each contain a clustered index key (5 bytes), a nonclustered index key (5 bytes), and 1 byte for the status bits. The leaf level will contain every single nonclustered key value along with the corresponding clustered key values. An index page can hold 736 index rows at this leaf level. We need 500,000 index rows, one for each row of data, so we need 680 leaf pages. If this nonclustered index is unique, the index rows at the higher levels will be 12 bytes each, so 674 index rows will fit per page, and we'll need two pages at level 1, with level 2 as the one root page.

So how big are these indexes compared to the table? For the clustered index, we had 16 index pages for a 10,000-page table, which is less than 1 percent of the size of the table. I frequently use 1 percent as a ballpark estimate for the space requirement of a clustered index, even though you can see that in this case it's an overly large estimate. On the other hand, our nonclustered index needed 683 pages for the 10,000-page table, which is about 6 percent additional space. For nonclustered indexes, it is much harder to give a good ballpark figure. My example used a very short key. Nonclustered index keys are frequently built on larger keys, or are even composite, so it's not unusual to have key sizes of over 100 bytes. In that case, we'd need a lot more leaf-level pages, and the total nonclustered index size could be 30 or 40 percent of the size of the table. I've even seen nonclustered indexes that are as big as or bigger than the table itself. Once you have two or three nonclustered indexes, you need to double the space to support these indexes. Remember that SQL Server allows you to have up to 249 nonclustered indexes! Disk space is cheap, but is it that cheap? You still need to plan your indexes carefully. *Inside Microsoft SQL Server 2005: T-SQL Querying* gives you an introduction to how to choose the most appropriate indexes for your tables based on the queries that you will be executing, and *Inside Microsoft SQL Server 2005: Tuning and Optimization* will provide even more information.

Actual vs. Estimated Size

The actual space used for tables or indexes, as opposed to the ballpark estimates I just described, can be seen in the `sys.allocation_units` catalog view. As mentioned earlier, in [Chapter 6](#) I showed you a query I referred to as the "allocation query," which joins `sys.partitions` with `sys.allocation_units` to give you the number of pages and the number of rows for each type of page in each partition of each table and index. Another way to get this information, in a slightly different format, is to use the dynamic management view called `sys.dm_db_partition_stats`. This view returns all the information for each partition in a single row, which includes the following result columns:

- `partition_id`
- `object_id`
- `index_id`
- `partition_number`
- `in_row_data_page_count`
- `in_row_used_page_count`
- `in_row_reserved_page_count`
- `lob_used_page_count`
- `lob_reserved_page_count`

- *row_overflow_used_page_count*
- *row_overflow_reserved_page_count*
- *used_page_count*
- *reserved_page_count*
- *row_count*
- Based on all the discussion of structures in [Chapter 6](#) and in this chapter, most of the columns should be self-explanatory. Note that there is a separate value for used pages and reserved pages for each type of page and then a column for total used space and one for total reserved space. (These columns are just the sum of the used and reserved space for the three types of storage.) Remember that after the first eight pages are allocated to a table or index, SQL Server grants all future allocations in units of eight pages, called *extents*. The total of all the pages in all the allocated extents for a particular type of page is the reserved page count. The total of all the pages that actually contain data is the *in_row_used_page_count*. It might be that only a few of the eight pages in some of the extents actually have data in them and are counted as used, so the reserved value is frequently higher than the used value.

I'll create a table called *hugerows_with_text* that contains regular in-row data, row-overflow data, and LOB data.

```
USE AdventureWork;
CREATE TABLE dbo.hugerows_with_text
(a varchar(3000),
b varchar(8000),
```

```

c varchar(8000),
d text);

INSERT INTO dbo.hugerows_with_text
    SELECT REPLICATE('a', 3000), REPLICATE('b',
8000),
        REPLICATE('c', 8000), REPLICATE('d',
8000);

```

We can compare the results of my 'allocation query' with the results of selecting from `sys.db_dm_partition_stats`:

```

-- Here is my 'allocation query'
SELECT object_name(object_id) AS name,
    partition_id, partition_number AS pnum, rows,
    allocation_unit_id AS au_id, type_desc as
page_type_desc,
    total_pages AS pages
FROM sys.partitions p JOIN sys.allocation_units a
    ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.hugerows_with_text
');

```

RESULTS:

name	partition_id	pnum	rows
au_id	page_type_desc	pages	
hugerows_with_text	72057594042449920	1	1
72057594048413696	IN_ROW_DATA	2	
hugerows_with_text	72057594042449920	1	1
72057594048479232	ROW_OVERFLOW_DATA	3	
hugerows_with_text	72057594042449920	1	1
72057594048544768	LOB_DATA	3	

```
SELECT * FROM sys.dm_db_partition_stats
```

```

WHERE object_id=object_id('dbo.hugerows_with_text
');

RESULTS: (wrapped)
partition_id          object_id   index_id
partition_number  in_row_data_page_count
-----
-----
72057594042449920      923866358    0           1
1
in_row_used_page_count in_row_reserved_page_count
lob_used_page_count   lob_reserved_page_count
-----
-----
2                      2
3                      3

row_overflow_used_page_count
row_overflow_reserved_page_count
-----
-----
3                      3

used_page_count reserved_page_count row_count
-----
-----
8                      8           1

```

The first query will return three rows, one for each type of page. The second query will return only one row. Note that for this first row, the used and reserved page count values are the same for all three types of pages. Also note that for the in-row data pages, three values are returned: a page count, a used page count, and a reserved page count. The simple page count is the number of

pages at the leaf level of an index, or if the structure is a heap, it's the total pages. The used page count includes all the other pages in the index, including the node pages and IAM pages. If you run the preceding INSERT statement several more times (I suggest at least eight more times), you start to get some bigger values returned from `sys.db_dm_partition_stats`, and you see that the values for used and reserved will no longer be the same.

Although `sys.db_dm_partition_stats` doesn't return the index name, it does return the object ID and index ID. For my `hugerow_with_text` table, that is irrelevant because the table is a heap. However, if you want to look at the space allocated to the `Sales.SalesOrderDetail` table, you might want to see the index name. Unfortunately, although there is a supplied system function that provides the name of an object given the object ID, there is no function to return an index name. So I have written one. If you create the `index_name` function in your `AdventureWorks` database, you can use it when selecting from `sys.db_dm_partition_stats` for a table in that database.

```
USE AdventureWorks;
CREATE FUNCTION dbo.index_name (@object_id int,
@index_id tinyint)
RETURNS sysname
AS
BEGIN
    DECLARE @index_name sysname
    SELECT @index_name = name FROM sys.indexes
        WHERE object_id = @object_id and index_id =
@index_id
    RETURN(@index_name)
END;
```

Now I will use this function to return just a few selected columns from `sys.db_dm_partition_stats`.

```
SELECT convert(char(25),dbo.index_name(object_id,
index_id)) AS index_name,
       used_page_count, reserved_page_count,
row_count
FROM sys.dm_db_partition_stats
WHERE
object_id=object_id('Sales.SalesOrderDetail');
RESULTS:
index_name                      used_page_count
reserved_page_count   row_count
-----  -----
PK_SalesOrderDetail_SalesOrder 1245
1278                  121317
AK_SalesOrderDetail_rowguid    417
454                   121317
IX_SalesOrderDetail_ProductID 237
253                  121317
```

SQL Server also supplies a stored procedure called `sp_spaceused`, which, when passed a table name as a parameter, reports the total amount of space used by the table. It reports total reserved space, total used data space, and total index space for the table and all its relational indexes, but it does not give an index by index breakdown. However, `sp_spaceused` does include the space used by XML indexes and full-text indexes that are dependent on the specified table, so the totals it gives might not agree with what you can compute by totaling all the values returned by `sys.db_dm_partition_stats`. In earlier versions of SQL Server, the values representing total space used are not always updated every time a table or index is modified. In particular, immediately after you

create a table or after certain bulk operations, `sp_spaceused` does not accurately reflect the total space reserved or used. You can force these earlier versions of SQL Server to update the internal metadata by passing a parameter called `@updateusage` to `sp_spaceused`, with the value `true`, or by running the command `DBCC UPDATEUSAGE`. SQL Server 2005 Books Online reports that this command is no longer needed because the space usage information is always maintained accurately. The only time you might want to use one of those options is after upgrading a SQL Server 2000 or SQL Server 7.0 database to SQL Server 2005.

It's great to have a way to see how much space is being used by a particular table or index, but what if we're getting ready to build a database and we want to know how much space our data and indexes will need? For planning purposes, it would be nice to know this information ahead of time.

If you've already created your tables and indexes, even if they're empty, SQL Server can get column and data type information from the `sys.columns` catalog view, and it should be able to see any indexes you've defined in the `sys.indexes` view. You shouldn't need to type all this information into a spreadsheet. The companion content includes a set of stored procedures you can use to calculate estimated space requirements when the table and indexes have already been built. The main procedure is called `sp_EstTableSize`, and it requires only two parameters: the table name and the anticipated number of rows. The procedure calculates the storage requirements for the table and all indexes by extracting information from the `sys.indexes`, `sys.columns`, and `sys.types` views. The result is only an estimate when you have variable-length fields. The procedure has no way of knowing whether a variable-length field will be completely filled, half filled, or mostly empty in every row, so it assumes that variable-length columns will be filled to the maximum size. If you know that this won't be the case with your data, you can create a second table that more closely matches the expected data. For example, if you have a `varchar(1000)` column that will only

rarely use the full 1000 bytes, and 99 percent of your rows will use only about 100 bytes, you can create a second table with a `varchar(100)` column and run `sp_EstTableSize` on that table.

Special Indexes

SQL Server 2005 allows you to create two special kinds of indexes: indexes on computed columns and indexes on views. Without indexes, both of these constructs are purely logical. There is no physical storage for the data involved. A computed column is not stored with the table data; it is recomputed every time a row is accessed. A view does not save any data; it basically saves a SELECT statement that is re-executed every time the data in the view is accessed. With the new special indexes, SQL Server actually materializes what was only logical data into the physical leaf level of an index.

Prerequisites

Before you can create indexes on either computed columns or views, certain prerequisites must be met. The biggest issue is that SQL Server must be able to guarantee that given the identical base table data, the same values will always be returned for any computed columns or for the rows in a view. To guarantee that the same values will always be generated, these special indexes have certain requirements, which fall into three categories. First, a number of session-level options must be set to a specific value. Second, there are some restrictions on the functions that can be used within the column or view definition. The third requirement, which applies only to indexed views, is that the tables that the view is based on must meet certain criteria.

SET Options

The following seven SET options can affect the result value of an expression or predicate, so you must set them as shown to create indexed views or indexes on computed columns:

```
SET CONCAT_NULL_YIELDS_NULL ON  
SET QUOTED_IDENTIFIER ON  
SET ANSI_NULLS ON  
SET ANSI_PADDING ON  
SET ANSI_WARNINGS ON  
SET NUMERIC_ROUNDABORT OFF
```

Note that all the options have to be ON except the NUMERIC_ROUNDABORT option, which has to be OFF. Technically, the option ARITHABORT must also be set to ON, but in SQL Server 2005, setting ANSI_WARNINGS to ON automatically sets ARITHABORT to ON, so you do not need to set it separately. If any of these options is not set as specified, you'll get an error message when you create a special index. In addition, if you've already created one of these indexes and then you attempt to modify the column or view on which the index is based, you'll get an error. If you issue a SELECT that normally should use the index, and if the SET options do not have the values indicated, the index will be ignored but no error will be generated.

There are a couple of ways to determine whether the SET options are set appropriately before you create one of these special indexes. You can use the property function SESSIONPROPERTY to test the settings for your current connection. A returned value of 1 means that the setting is ON, and a 0 means that it is OFF. The following example checks the current session setting for the option NUMERIC_ROUNDABORT:

```
SELECT SessionProperty('NUMERIC_ROUNDABORT')
```

Alternatively you can use the DMV `sys.dm_exec_session` to check the SET options for any connection. The following query returns the values for five of the previously discussed six set options for the current session:

```
SELECT quoted_identifier, arithabort,  
ansi_warnings,  
      ansi_padding, ansi_nulls,  
concat_null_yields_null  
FROM sys.dm_exec_sessions  
WHERE session_id = @@spid
```

Unfortunately, `NUMERIC_ROUNDABORT` is not included in the `sys.dm_exec_session` view. There is no way to see the setting for that value for other connections besides the current one.

Permissible Functions

A function is either *deterministic* or *nondeterministic*. If the function returns the same result every time it is called with the same set of input values, it is deterministic. If it can return different results when called with the same set of input values, it is nondeterministic. For the purposes of indexes, a function is considered deterministic if it always returns the same values for the same input values when all the SET options have the required settings. Any function used in a computed column's definition or used in the SELECT list or WHERE clause of an indexable view must be deterministic.

SQL Server 2005 Books Online contains a complete list of which supplied functions are deterministic and which are nondeterministic.

Some functions can be either deterministic or nondeterministic, depending on how they are used, and Books Online also describes these functions.

It might seem that the list of nondeterministic functions is quite restrictive, but SQL Server must be able to guarantee that the values stored in the index will be consistent. In some cases, the restrictions might be overly cautious, but the downside of being not cautious enough might be that your indexed views or indexes on computed columns are meaningless. The same restrictions apply to functions you use in your own user-defined functions (UDFs) that is, your own functions cannot be based on any nondeterministic built-in function. UDFs are explained in detail in *Inside Microsoft SQL Server 2005: T-SQL Programming*. You can verify the determinism property of any function by using the OBJECTPROPERTY function:

```
SELECT  
OBJECTPROPERTY(object_id('<function_name>'),  
'IsDeterministic')
```

Even if a function is deterministic, if it contains *float* or *real* expressions, the result of the function might vary with different processors, depending on the processor architecture or microcode version. Expressions or functions containing values of the data type *float* or *real* are therefore considered to be *imprecise*. To guarantee consistent values even when moving a database from one machine to another (by detaching and attaching, or by backup and restore), imprecise values can only be used in key columns of indexes if they are physically stored in the database and not recomputed. An imprecise value can be used if it is the value of a stored column in a table or if it is a computed column that is marked as persisted. I'll discuss persisted columns in more detail in the upcoming section titled "[Indexes on Computed Columns](#)."

Schema Binding

To create an indexed view, a requirement on the table itself is that the definition of any underlying object's schema cannot change. To prevent a change in schema definition, the CREATE VIEW statement allows the WITH SCHEMABINDING option. When you specify WITH SCHEMABINDING, the SELECT statement that defines the view must include the two-part names (*schema.object*) of all referenced tables. You can't drop or alter tables that participate in a view created with the SCHEMABINDING clause unless you've dropped that view or changed the view so that it no longer has schema binding. Otherwise, SQL Server raises an error. If any of the tables on which the view is based is owned by someone other than the user creating the view, the view creator doesn't automatically have the right to create the view with schema binding because that would restrict the table's owner from making changes to her own table. A user must be granted REFERENCES permission in order to create a view with schema binding on that table. We'll see an example of schema binding in a moment.

Indexes on Computed Columns

SQL Server 2005 allows you to build indexes on deterministic, precise computed columns where the resulting data type is otherwise indexable. This means that the column's data type cannot be *text*, *ntext*, or *image*. Such a computed column can participate at any position of an index or in a PRIMARY KEY or UNIQUE constraint. You cannot define a FOREIGN KEY, CHECK, or DEFAULT constraint on a computed column, and computed columns are always considered nullable unless you enclose the expression in the ISNULL function. You cannot create indexes on any computed columns in system tables. When you create an index

on computed columns, the six SET options must first have the correct values set.

Here's an example:

```
CREATE TABLE t1 (a int, b as 2*a);
GO
CREATE INDEX i1 on t1 (b);
GO
```

If any of your SET options does not have the correct value when you create the table, you get this message when you try to create the index:

```
Server: Msg 1934, Level 16, State 1, Line 2
CREATE INDEX failed because the following SET
options have
incorrect settings: '<OPTION NAME>' .
```

If more than one option has an incorrect value, the error message will report them all.

Here's an example that creates a table with a nondeterministic computed column:

```
CREATE TABLE t2 (a int, b datetime, c AS
datename(mm, b));
GO
CREATE INDEX i2 on t2 (c);
GO
```

When you try to create the index on the computed column c, you get this error:

```
Msg 2729, Level 16, State 1, Line 1  
Column 'c' in table 't2' cannot be used in an  
index or statistics or as a partition key  
because it is nondeterministic.
```

Column c is nondeterministic because the month value of *datename* can have different values depending on the language you're using.

Using the COLUMNPROPERTY Function

You can use the *IsDeterministic* column property to determine before you create an index on a computed column whether that column is deterministic. If you specify this property, the COLUMNPROPERTY function returns 1 if the column is deterministic and 0 otherwise. The result is undefined for noncomputed columns, so you should consider checking the *IsComputed* property before you check the *IsDeterministic* property. The following example detects that column c in table t2 in the previous example is nondeterministic:

```
SELECT COLUMNPROPERTY(object_id('t2'), 'c',  
'IsDeterministic');
```

The value 0 is returned, which means that column c is nondeterministic. Note that the COLUMNPROPERTY function requires an object ID for the first argument and a column name for the second argument.

Implementation of a Computed Column

If you create a clustered index on a computed column, the computed column is no longer a virtual column in the table. Its computed value will physically exist in the rows of the table, which is the leaf level of the clustered index. Updates to the columns that the computed column is based on will also update the computed column in the table itself. For example, in the t1 table created previously, if we insert a row with the value 10 in column a, the row will be created with both the values 10 and 20 in the actual data row. If we then update the 10 to 15, the second column will be automatically updated to 30.

Persisted Columns

The ability to mark a column as persisted, a new feature in SQL Server 2005, allows storage of computed values in a table, even before you build an index. In fact, this feature was added to the product to allow columns of computed values from underlying table columns of type *float* or *real* to have indexes built on them. The alternative, when you want an index on such a column, would be to drop and re-create the underlying column, which can involve an enormous amount of overhead on a large table.

Here's an example. In the *Northwind* database, the *Order Details* table has a column called *Discount* that is of type *real*. The following code adds a computed column called *Final* that shows the total price for an item after the discount is applied. The statement to build an index on *Final* will fail because the resultant column involving the *real* value is imprecise.

```
USE Northwind;
GO
ALTER TABLE [Order Details]
```

```
    ADD Final AS
        (Quantity * UnitPrice) - Discount * (Quantity
        * UnitPrice);
GO
CREATE INDEX OD_Final_Index on [Order Details]
(Final);
```

Error Message:

```
Msg 2799, Level 16, State 1, Line 1
Cannot create index or statistics 'OD_Final_Index'
on table 'Order Details' because the comp
uted column 'Final' is imprecise and not
persisted. Consider removing column from index or
statistics key or marking computed column
persisted.
```

Without persisted columns, the only way to create an index on a computed column containing the final price would be to drop the *Discount* column from the table and redefine it. Any existing indexes on *Discount* would have to be dropped also, and then rebuilt. With persisted columns, all you need to do is drop the computed column, which is a metadata-only operation, and then redefine it as a persisted column. You can then build the index on the computed, persisted column.

```
ALTER TABLE [Order Details]
    DROP COLUMN Final;
GO
ALTER TABLE [Order Details]
    ADD Final AS
        (Quantity * UnitPrice) - Discount * (Quantity *
        UnitPrice) PERSISTED;
GO
```

```
CREATE INDEX OD_Final_Index on [Order Details]  
    (Final);
```

When determining whether you have to use the PERSISTED option, use the COLUMNPROPERTY function and the *IsPrecise* property to determine whether a deterministic column is precise.

```
SELECT COLUMNPROPERTY (object_id('Order Details'),  
    'Final', 'IsPrecise');
```

You can also use persisted columns when you define partitions. A computed column that is used as the partitioning column must be explicitly marked as PERSISTED, whether it is imprecise. We'll look at partitioning later in this chapter.

Indexed Views

Indexed views in SQL Server are similar to what other products call *materialized views*. One of the most important benefits of indexed views is the ability materialize summary aggregates of large tables. For example, consider a customer table containing rows for several million U.S.-based customers, from which you want information regarding customers in each state. You can create a view based on a GROUP BY query, grouping by state and containing the count of orders per state. Normal views are only saved queries (as discussed in *Inside Microsoft SQL Server 2005: T-SQL Programming*) and do not store the results. Every time the view is referenced, the aggregation to produce the grouped results must be recomputed. When you create an index on the view, the aggregated data is stored in the leaf level of the index. So instead of millions of

customer rows, your indexed view has only 50 rows, one for each state. Your aggregate reporting queries can then be processed using the indexed views without having to scan the underlying large tables. The first index you must build on a view is a clustered index, and because the clustered index contains all the data at its leaf level, this index actually does materialize the view. The view's data is physically stored at the leaf level of the clustered index.

Additional Requirements

In addition to the requirement that all functions used in the view be deterministic and that the required SET options be set to the appropriate values, the view definition can't contain any of the following:

- TOP
- *text*, *ntext*, or *image* columns
- DISTINCT
- MIN, MAX, COUNT(*), COUNT(<expression>), STDEV, VARIANCE, AVG
- SUM on a nullable expression
- A derived table
- The ROWSET function
- Another view (you can reference only base tables)

- UNION
- Subqueries, OUTER joins, or self-joins
- Full-text predicates (CONTAINS, FREETEXT)
- COMPUTE, COMPUTE BY
- ORDER BY

Also, if the view definition contains GROUP BY, you must include the aggregate COUNT_BIG(*) in the SELECT list. COUNT_BIG returns a value of the data type BIGINT, which is an 8-byte integer. A view that contains GROUP BY can't contain HAVING, CUBE, ROLLUP, or GROUP BY ALL. Also, all GROUP BY columns must appear in the SELECT list. Note that if your view contains both SUM and COUNT_BIG(*), you can compute the equivalent of the AVG function even though AVG is not allowed in indexed views. Although these restrictions might seem severe, remember that they apply to the view definitions, not to the queries that might use the indexed views.

To verify that you've met all the requirements, you can use the OBJECTPROPERTY function's *IsIndexable* property. The following query tells you whether you can build an index on a view called *Product Totals*:

```
SELECT ObjectProperty(object_id('Product_Totals'),  
'IsIndexable');
```

A return value of 1 means you've met all requirements and can build an index on the view.

Creating an Indexed View

The first step in building an index on a view is to create the view itself. Here's an example from the *AdventureWorks* database:

```
USE AdventureWorks;
GO
CREATE VIEW Vdiscount1 WITH SCHEMABINDING
AS SELECT SUM(UnitPrice*OrderQty) AS SumPrice,
SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount))
AS SumDiscountPrice,
COUNT_BIG(*) AS Count,
ProductID
FROM Sales.SalesOrderDetail
GROUP BY ProductID;
```

Note the WITH SCHEMABINDING clause and the specification of the schema name (*dbo*) for the table. At this point, we have a normal view a stored SELECT statement that uses no storage space. In fact, if we look at the data in *sys.dm_db_partition_stats* for this view, we'll see that no rows are returned.

```
SELECT convert(char(25),dbo.index_name(object_id,
index_id)) AS index_name,
       used_page_count, reserved_page_count,
       row_count
  FROM sys.dm_db_partition_stats
 WHERE object_id=object_id('dbo.Vdiscount1');
```

To create an indexed view, you must create an index. The first index you create on a view must be a *unique clustered index*. Clustered

indexes are the only type of SQL Server index that contains data; the clustered index on a view contains all the data that makes up the view definition. This statement defines a unique clustered index for the view:

```
CREATE UNIQUE CLUSTERED INDEX VDiscount_Idx ON  
Vdiscount1 (ProductID);
```

After you create the index, you can reexamine *sys.dm_db_partition_stats*. My output looks like this:

index_name	used_page_count	reserved_page_count	row_count
-----	-----	-----	-----
-----	-----	-----	-----
VDiscountIdx	6		6
266			

Data that composes the indexed view is persistent, with the indexed view storing the data in the clustered index's leaf level. You could construct something similar by using temporary tables to store the data you're interested in. But a temporary table is static and doesn't reflect changes to underlying data. In contrast, SQL Server automatically maintains indexed views, updating information stored in the clustered index whenever anyone changes data that affects the view.

After you create the unique clustered index, you can create multiple nonclustered indexes on the view. You can determine whether a view is indexed by using the *OBJECTPROPERTY* function's *IsIndexed* property. For the *Vdiscount1* indexed view, the following statement returns a 1, which means the view is indexed:

```
SELECT ObjectProperty(object_id('Vdiscount1'),  
'IsIndexed');
```

Once a view is indexed, metadata about space usage and location is available through the catalog view, just as for any other index.

Using an Indexed View

One of the most valuable benefits of indexed views is that your queries don't have to directly reference a view to use the index on the view. Consider the *Vdiscount1* indexed view. Suppose that you issue the following SELECT statement:

```
SELECT ProductID, total_sales =  
SUM(UnitPrice*OrderQty)  
FROM Sales.SalesOrderDetail  
GROUP BY ProductID;
```

SQL Server's query optimizer will realize that the precomputed sums of all the *UnitPrice * OrderQty* values for each *ProductID* are already available in the index for the *Vdiscount1* view. The query optimizer will evaluate the cost of using that indexed view in processing the query, and the indexed view will very likely be used to access the information required to satisfy this query. The *Sales.SalesOrderDetail* table might never need to be touched at all.

Note



Although you can create indexed views in

any edition of SQL Server 2005, in order for the optimizer to consider using them even when they aren't referenced in the query, the engine edition of your SQL Server 2005 must be Enterprise. This is the case with the Enterprise Edition as well as the Developer and Evaluation editions.

Just because you have an indexed view doesn't mean the query optimizer will always choose it for the query's execution plan. In fact, even if you reference the indexed view directly in the FROM clause, the query optimizer might decide to directly access the base table instead. To make sure that an indexed view in your FROM clause is not expanded into its underlying SELECT statement, you can use the NOEXPAND hint in the FROM clause. In *Inside Microsoft SQL Server 2005: Tuning and Optimization*, I'll tell you more about how the query optimizer decides whether to use indexed views and how you can tell if indexed views are being used for a query. I'll also revisit indexes on computed columns and show you some situations where you can make good use of them.

Table and Index Partitioning

As we've already seen when looking at the metadata for table and index storage, partitioning is an integral feature of SQL Server space organization. [Figure 6-6](#) illustrated the relationship between tables and indexes (hobts), partitions, and allocation units. To determine where a table or index is stored, you must reference the partition of the table or index. Tables and indexes that are built without any reference to partitions are considered to be stored on a single partition. One of the more useful metadata objects for retrieving information about data storage is the dynamic management view called `sys.dm_db_partition_stats`, which combines information found in `sys.partitions`, `sys.allocation_units`, and `sys.indexes`.

A partitioned object is one that is internally split into separate physical units that can be stored in different locations. Partitioning is invisible to the users and programmers, who can use Transact-SQL code to select from a partitioned table exactly the same way they select from a non-partitioned table. Creating large objects on multiple partitions improves the manageability and maintainability of your database system and can greatly enhance the performance of activities such as purging historic data and loading large amounts of data. In SQL Server 2000, partitioning is available only by manually creating a view that combines multiple tables. That functionality is referred to as *partitioned views*. The SQL Server 2005 built-in partitioning of tables and indexes has many advantages over partitioned views, including improved execution plans and fewer prerequisites for implementation.

A full discussion of all the benefits and uses of partitioning is outside the scope of this book. In this section, we'll focus primarily on the partitioning metadata. In *Inside Microsoft SQL Server 2005: Tuning and Optimization*, we'll look at the use of partitions for performance improvement and examine query plans involving partitioned tables and partitioned indexes.

Partition Functions and Partition Schemes

To understand the partitioning metadata, we need a little background into how partitions are defined. I will use an example based on the SQL Server samples, combining the scripts in `PartitionAW.sql` and `sliding.sql`. You can find my script, called `Partition.sql`, with the companion content. This script defines two tables: `TransactionHistory` and `TransactionHistoryArchive`, along with a clustered index and two nonclustered indexes on each. Both tables are partitioned on the `TransactionDate` column, with each month of data in a separate partition. Initially there are 12 partitions in `TransactionHistory` and 2 in `TransactionHistoryArchive`.

Before you create a partitioned table or index, you must define a partition function, which is basically just a set of endpoints. The number of endpoints will be one less than the number of partitions. Here is the partition function that my example will use:

```
CREATE PARTITION FUNCTION [TransactionRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('20031001', '20031101', '20031201',
                           '20040101', '20040201', '20040301', '20040401',
                           '20040501', '20040601', '20040701', '20040801');
```

Note that the table name is not mentioned in the function definition because the partition function is not tied to any one particular table. The function *TransactionRangePF1* divides the data into 12 partitions because there are 11 datetime endpoints. The keyword RIGHT specifies that any value that equals one of the endpoints will go into the partition to the right of the endpoint. So for this function, all values less than October 1, 2003, will go in the first partition, and values greater than or equal to October 1, 2003, and less than November 1, 2003, will go in the second partition. I could have also specified LEFT (which is the default), in which case the value equal to the endpoint would go in the partition to the left. After you define the partition function, you define a partition scheme, which lists a set of filegroups onto which each range of data will be placed. Here is the partition schema for my example:

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
TO ([PRIMARY], [PRIMARY], [PRIMARY],
    [PRIMARY], [PRIMARY], [PRIMARY]
    , [PRIMARY], [PRIMARY], [PRIMARY]
    , [PRIMARY], [PRIMARY], [PRIMARY]);
GO
```

To avoid having to create 12 files and filegroups, I have put all the partitions on the PRIMARY filegroup, but for the full benefit of partitioning, you would most likely have each partition on its own filegroup. The CREATE PARTITION SCHEME command must list at least as many filegroups as there are partitions, but there can be more. And as you can see in the example, the listed filegroups do not have to be unique. Additional filegroups will be used in order, as more partitions are added, which can happen when a partition function is altered to split an existing range into two. If you do not specify extra filegroups at the time you create the partition scheme, you can alter the partition scheme to add another filegroup.

The partition function and partition scheme for a second table are shown here:

```
CREATE PARTITION FUNCTION [TransactionArchivePF2] (datetime)
AS RANGE RIGHT FOR VALUES ('20030901');
GO

CREATE PARTITION SCHEME [TransactionArchivePS2]
AS PARTITION [TransactionArchivePF2]
TO ([PRIMARY], [PRIMARY]);
GO
```

My script then creates two tables and loads data into them. I will not include all the details here. To partition a table, you must specify a partition scheme in the table creation statement. I create a table called *TransactionArchive* that includes this line as the last part of the CREATE TABLE:

```
ON [TransactionsPS1] (TransactionDate)
```

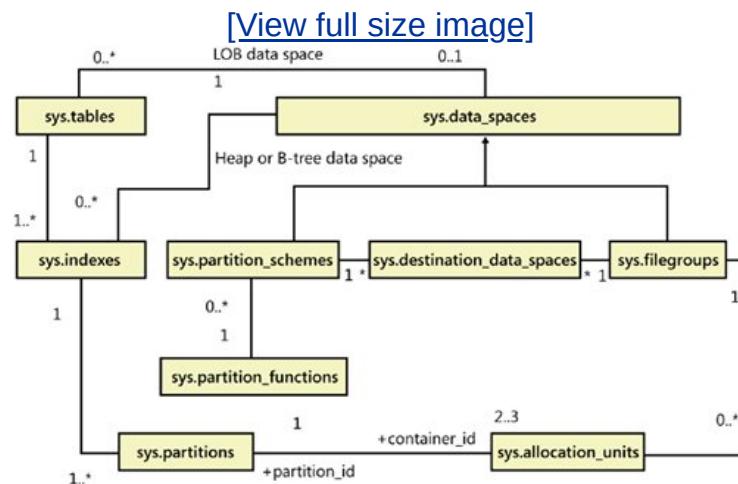
My second table, *TransactionArchiveHistory*, is created using the *TransactionsPS1* partitioning scheme.

My script then loads data into the two tables, and because the partition scheme has already been defined, as the data is loaded each row is placed in the appropriate partition. After the tables are loaded, we can examine the metadata.

Metadata for Partitioning

[Figure 7-8](#) shows most of the catalog views for retrieving information about partitions. Along the left and bottom edges you can see the *sys.tables*, *sys.indexes*, *sys.partitions*, and *sys.allocation_units* that I've discussed already.

Figure 7-8. Catalog views containing metadata relevant to partitioning and data storage



In some of my queries, I will use the undocumented `sys.system_internals_allocation_units` view instead of `sys.allocation_units` in order to retrieve page address information. Below, I'll describe the most relevant columns of each of the these views:

- `Sys.data_spaces` has a primary key called `data_space_id`, which is either a partition ID or a filegroup ID, and there is one row for each filegroup and one row for each partition scheme. One of the columns in `sys.data_spaces` specifies to which type of data space the row refers. If the row refers to a partition scheme, `data_space_id` can be joined with `sys.partition_schemes.data_space_id`. If the row refers to a filegroup, `data_space_id` can be joined with `sys.filegroups.data_space_id`. The `sys.indexes` view also has a `data_space_id` column to indicate how each heap or B-tree stored in `sys.indexes` is stored. So, if we know that a table is partitioned, we can directly join it with `sys.partition_schemes` without going through `sys.data_spaces`. Alternatively, you can use the following query to determine whether a table is partitioned, by replacing `Production.TransactionHistoryArchive` with the name of the table in which you're interested:

```
SELECT DISTINCT object_name(object_id) as TableName,
               ISNULL(ps.name, 'Not partitioned') as PartitionScheme
        FROM (sys.indexes i LEFT JOIN sys.partition_schemes ps
              ON (i.data_space_id = ps.data_space_id))
         WHERE (i.object_id =
object_id('Production.TransactionHistoryArchive'))
           AND (i.index_id IN (0,1));
```

- `Sys.partition_schemes` has one row for each partition scheme. In addition to the `data_space_id` and the name of the partition scheme, it has a `function_id` column to join with `sys.partition_functions`.
- `Sys.destination_data_spaces` is a linking table because `sys.partition_schemes` and `sys.filegroups` are in a many-to-many relationship with each other. For each partition scheme, there is one row for each partition. The partition number is in the `destination_id` column, and the filegroup ID is stored in the `data_space_id` column.
- `Sys.partition_functions` contains one row for each partition function, and its primary key `function_id` is a foreign key in `sys.partition_schemes`.
- `Sys.partition_range_values` (not shown) has one row for each endpoint of each partition function. Its `function_id` column can be joined with `sys.partition_functions`, and its `boundary_id` column can join with either `partition_id` in `sys.partitions` or with `destination_id` in `sys.destination_data_spaces`.

These views have other columns that I haven't mentioned, and there are additional views that provide information, such as the columns and their data types that the partitioning is based on. However, the preceding information should be sufficient to understand [Figure 7-8](#) and the view

below. This view returns information about each partition of each partitioned table. The WHERE clause filters out partitioned indexes (other than the clustered index), but you can change that condition if you desire. When selecting from the view, you can add your own WHERE clause to find information about just the table you're interested in.

```
CREATE VIEW Partition_Info AS
SELECT OBJECT_NAME(i.object_id) as Object_Name,
dbo.INDEX_NAME(i.object_id,i.index_id) AS Index_Name,
    p.partition_number, fg.name AS Filegroup_Name, rows,
au.total_pages,
CASE boundary_value_on_right
    WHEN 1 THEN 'less than'
    ELSE 'less than or equal to' END as 'comparison',
value
FROM sys.partitions p JOIN sys.indexes i
    ON p.object_id = i.object_id and p.index_id = i.index_id
    JOIN sys.partition_schemes ps
        ON ps.data_space_id = i.data_space_id
    JOIN sys.partition_functions f
        ON f.function_id = ps.function_id
    LEFT JOIN sys.partition_range_values rv
ON f.function_id = rv.function_id
    AND p.partition_number = rv.boundary_id
    JOIN sys.destination_data_spaces dds
        ON dds.partition_scheme_id = ps.data_space_id
            AND dds.destination_id = p.partition_number
    JOIN sys.filegroups fg
        ON dds.data_space_id = fg.data_space_id
    JOIN (SELECT container_id, sum(total_pages) as total_pages
        FROM sys.allocation_units
        GROUP BY container_id) AS au
        ON au.container_id = p.partition_id
WHERE i.index_id <2;
```

The LEFT JOIN is needed to get all the partitions because the `sys.partition_range_values` view has a row only for each boundary value, not for each partition. The LEFT JOIN gives the last partition with a boundary value of NULL, which means there is no upper limit to the last partition's value. There is a derived table that groups together all the rows in `sys.allocation_units` for a partition, so the space used for all the types of storage (in-row, row-overflow, and LOB) will be aggregated into a single value. This query will use the preceding view to get information about my `TransactionHistory` table's partitions:

```
SELECT * FROM Partition_Info
WHERE Object_Name = 'TransactionHistory';
```

Here are my results:

Object_Name	partition_number	Filegroup_Name	rows	total_pages	comparison	value
TransactionHistory	1	PRIMARY	11155	209	less than	2003-10-0
TransactionHistory	2	PRIMARY	9339	177	less than	2003-11-01
TransactionHistory	3	PRIMARY	10169	185	less than	2003-12-01
TransactionHistory	4	PRIMARY	12181	225	less than	2004-01-01
TransactionHistory	5	PRIMARY	9558	177	less than	2004-02-01
TransactionHistory	6	PRIMARY	10217	193	less than	2004-03-01
TransactionHistory	7	PRIMARY	10703	201	less than	2004-04-01
TransactionHistory	8	PRIMARY	10640	193	less than	2004-05-01
TransactionHistory	9	PRIMARY	12508	225	less than	2004-06-01
TransactionHistory	10	PRIMARY	12585	233	less than	2004-07-01
TransactionHistory	11	PRIMARY	3380	73	less than	2004-08-01

Object_Name	partition_number	Filegroup_Name	rows	total_pages	comparison	value
TransactionHistory	12	PRIMARY	1008	33	less than	NULL

This view contains details about the boundary point of each partition, as well as the filegroup that each partition is stored on, the number of rows in each partition, and the amount of space used. Note that although the comparison indicates that the values in the partitioning column for the rows in a particular partition are less than the specified value, you should assume that it also means that the values are greater than or equal to the specified value in the preceding partition. However, this view doesn't provide information about where in the particular filegroup the data is located. We'll look at a metadata query that gives us location information in the next section.

Partition Power

One of the main benefits of partitioning your data is that you can move data from one partition to another as a metadata-only operation. The data itself doesn't have to move. As I mentioned, this is not intended to be a complete how-to guide to SQL Server 2005 partitioning; rather, it is a look into the internal storage of partitioning information. However, to show the internals of rearranging partitions, we need to look at some additional partitioning operations.

The main operation you use when working with partitions is the SWITCH option to the ALTER TABLE command. This option allows you to:

- Assign a table as a partition to an already existing partitioned table
- Switch a partition from one partitioned table to another
- Reassign a partition to form a single table

In all these operations, no data is moved. Rather, the metadata is updated in the *sys.partitions* and *sys.system_internals_allocation_units* views to indicate that a given allocation unit now is part of a different partition. Let's look at an example. The following query returns information about each allocation unit in the first two partitions of my *TransactionHistory* and *TransactionHistoryArchive* tables, including the number of rows, the number of pages, the type of data in the allocation unit, and the page where the allocation unit starts.

```
SELECT convert(char(25),object_name(object_id)) AS name,
       rows, convert(char(15),type_desc) as page_type_desc,
       total_pages AS pages, first_page, index_id, partition_number
  FROM sys.partitions p JOIN sys.system_internals_allocation_units a
```

```

    ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
      OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
      AND index_id = 1 AND partition_number <= 2;

```

Here is the data I get back. (I left out the *page_type_desc* because all the rows are of type IN_ROW_DATA.)

<i>name</i>	<i>index_id</i>	<i>partition_number</i>	<i>rows</i>	<i>pages</i>	<i>first_page</i>
TransactionHistory	1	1	11155	209	0xD81B00000100
TransactionHistory	2	2	9339	177	0xA82200000100
TransactionHistoryArchive	1	1	89253	1553	0x981B00000100
TransactionHistoryArchive	2	2	0	0	0x000000000000

Now let's move one of my partitions. My ultimate goal is to add a new partition to *TransactionHistory* to store a new month's worth of data and to move the oldest month's data into *TransactionHistoryArchive*. The partition function used by my *TransactionHistory* table divides the data into 12 partitions, and the last one contains all dates greater than or equal to August 1, 2004. I'm going to alter the partition function to put a new boundary point in for September 1, 2004, so the last partition will be split. Before doing that, I must ensure that the partition scheme using this function knows what filegroup to use for the newly created partition. With this command, some data movement will occur, and all data from the last partition of any tables using this partition scheme will be moved to a new allocation unit. Please refer to Books Online for the complete details about each of the following commands:

```

ALTER PARTITION SCHEME TransactionsPS1
NEXT USED [PRIMARY];
GO

```

```

ALTER PARTITION FUNCTION TransactionRangePF1()
SPLIT RANGE ('20040901');
GO

```

Next, I'll do something similar for the function and partition scheme used by *TransactionHistoryArchive*. In this case, I'll add a new boundary point for October 1, 2003.

```

ALTER PARTITION SCHEME TransactionArchivePS2
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionArchivePF2()
SPLIT RANGE ('20031001');
GO

```

I want to move all data from *TransactionHistory* with dates earlier than October 1, 2003 to the second partition of *TransactionHistoryArchive*. However, the first partition of *TransactionHistory* technically has no lower limit; it is everything earlier than October 1, 2003. The second partition of *TransactionHistoryArchive* does have a lower limit, which is the first boundary point, or September 1, 2003. To SWITCH a partition from one table to another, I have to guarantee that all the data to be moved meets the requirements for the new location. So I add a CHECK constraint that guarantees that no data in *TransactionHistory* is earlier than September 1, 2003. After adding the CHECK constraint, I run the ALTER TABLE command with the SWITCH option to move the data in partition 1 of *TransactionHistory* to partition 2 of *TransactionHistoryArchive*. (For your testing purposes, you could try leaving out the next step that adds the constraint and try just executing the ALTER TABLE / SWITCH command. You'll get an error message, and then you add the constraint and run the SWITCH command again.)

```

ALTER TABLE [Production].[TransactionHistory]
ADD CONSTRAINT [CK_TransactionHistory_DateRange]
CHECK ([TransactionDate] >= '20030901');
GO
ALTER TABLE [Production].[TransactionHistory]
SWITCH PARTITION 1
TO [Production].[TransactionHistoryArchive] PARTITION 2;
GO

```

Now we run the metadata query that examines the size and location of the first two partitions of each table:

```

SELECT convert(char(25),object_name(object_id)) AS name,
       rows, convert(char(15),type_desc) as page_type_desc,
       total_pages AS pages, first_page, index_id, partition_number
FROM sys.partitions p JOIN sys.system_internals_allocation_units a
    ON p.partition_id = a.container_id
WHERE (object_id=object_id('[Production].[TransactionHistory]')
      OR object_id=object_id('[Production].[TransactionHistoryArchive]'))
      AND index_id = 1 AND partition_number <= 2;

```

RESULTS:

name	rows	pages	first_page	index_id
------	------	-------	------------	----------

<code>partition_number</code>					
<code>TransactionHistory</code>	0	0	<code>0x00000000000000</code>	1	
<code>1</code>					
<code>TransactionHistory</code>	9339	177	<code>0xA82200000100</code>	1	
<code>2</code>					
<code>TransactionHistoryAr</code>	89253	1553	<code>0x981B00000100</code>	1	
<code>1</code>					
<code>TransactionHistoryAr</code>	11155	209	<code>0xD81B00000100</code>	1	
<code>2</code>					

You'll notice that the second partition of `TransactionHistoryArchive` now has exactly the same information that the first partition of `TransactionHistory` had in the first result set. It has the same number of rows (11,155), the same number of pages (209), and the same starting page (0xD81B00000100, or file 1, page 7128). No data was moved; the only change was that the allocation unit starting at file 1, page 7128 is not recorded as belonging to the second partition of the `TransactionHistoryArchive` table.

Although my partitioning script created the indexes for my partitioned tables using the same partition scheme used for the tables themselves, this is not always necessary. An index for a partitioned table can be partitioned using the same partition scheme or a different one. If you do not specify a partition scheme or filegroup when you build an index on a partitioned table, the index is placed in the same partition scheme as the underlying table, using the same partitioning column. Indexes built on the same partition scheme as the base table are called *aligned indexes*. Whether to make your indexes aligned with an underlying partitioned table is primarily a performance-related decision; I will discuss it in *Inside Microsoft SQL Server 2005: Tuning and Optimization*.

However, there is an internal storage component associated with automatically aligned indexes. As previously mentioned, if you build an index on a partitioned table and do not specify a filegroup or partitioning scheme on which to place the index, SQL Server creates the index using the same partitioning scheme that the table uses. However, if the partitioning column is not part of the index definition, SQL Server adds the partitioning column as an extra included column in the index. If the index is clustered, adding an included column is not necessary because the clustered index always contains all the columns already. Another case in which SQL Server does not add an included column automatically is when you create a unique index, either clustered or nonclustered. Because it is a requirement of unique partitioned indexes that the partitioning column be contained in the unique key, a unique index for which you have not explicitly included the partitioning key will not be automatically partitioned.

Note



In addition to looking in Books Online, you can get more partitioning background, usage suggestions, and best practices

by referring to the white paper titled "SQL Server 2005 Partitioned Tables and Indexes" by Kimberly Tripp, which is included with the companion content.

Data Modification Internals

We've now seen how SQL Server stores data and index information. Now we'll look at what SQL Server actually does internally when your data is modified. We've seen how clustered indexes control space usage in SQL Server. As a rule of thumb, you should always have a clustered index on a table. There are some cases in which you might be better off with a heap, such as when the most important factor is the speed of INSERT operations, but until you do thorough testing to establish that you have one of these cases, it's better to have a clustered index than to have no organization to your data at all. In *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*, I'll examine the benefits and tradeoffs of clustered and nonclustered indexes and present some guidelines for their use. For now, we'll look only at how SQL Server deals with the existence of indexes when processing data modification statements.

Inserting Rows

When inserting a new row into a table, SQL Server must determine where to put it. When a table has no clustered index—that is, when the table is a heap—a new row is always inserted wherever room is available in the table. In [Chapter 6](#), I showed you how IAMs and the PFS pages keep track of which extents in a file already belong to a table and which of the pages in those extents have space available. Even without a clustered index, space management is quite efficient. If no pages with space are available, SQL Server tries to find unallocated pages from existing uniform extents that already belong to the object. If none exists, SQL Server must allocate a whole new extent to the table. [Chapter 4](#) discussed how the GAMS and SGAMs were used to find extents available to be allocated to an object.

A clustered index directs an insert to a specific page based on the value the new row has for the clustered index key columns. The insert occurs when the new row is the direct result of an INSERT statement or when it's the result of an UPDATE statement executed via a delete-followed-by-insert (delete/insert) strategy. New rows are inserted into their clustered position, splicing in a page via a page split if the current page has no room. You saw earlier in this chapter that if a clustered index isn't declared as unique and duplicate values are inserted for a key, an automatic uniqueifier is generated for all subsequent rows with the same key. Internally, SQL Server treats all clustered index keys as unique.

Because the clustered index dictates a particular ordering for the rows in a table, every new row has a specific location where it belongs. If there's no room for the new row on the page where it belongs, you must allocate a new page and link it into the list of pages. If possible, this new page is allocated from the same extent as the other pages to which it will be linked. If the extent is full, a new extent (eight pages, or 64 KB) is allocated to the object. As described in [Chapter 4](#), SQL Server uses the GAM pages to find an available extent.

Splitting Pages

After SQL Server finds the new page, the original page must be split; half the rows are left on the original page, and half are moved to the new page. In some cases, SQL Server finds that even after the split, there's no room for the new row, which, because of variable-length fields, could potentially be much larger than any of the existing rows on the pages. After the split, one or more rows are promoted to the parent page. One row is promoted if only a single split is needed. However, if the new row still won't fit after a single split, there can potentially be multiple new pages and multiple promotions to the parent page. For example, consider a page with

32 rows on it. Suppose that SQL Server tries to insert a new row with 8,000 bytes. It will split the page once, and the new 800-byte row won't fit. Even after a second split, the new row won't fit. Eventually, SQL Server will realize that the new row cannot fit on a page with any other rows, and it will allocate a new page to hold only the new row. Quite a few splits will occur, resulting in many new pages, and many new rows on the parent page.

An index tree is always searched from the root down, so during an insert operation it is split on the way down. This means that while the index is being searched on an insert, the index is protected in anticipation of possibly being updated. The protection mechanism is a latch, which you can think of as something like a lock. (Locks will be discussed in detail in [Chapter 8](#).) A latch is acquired while a page is being read from or written to disk and protects the physical integrity of the contents of the page. A parent node (not a leaf node) is latched until the child node is known to be available for its own latch. Then the parent latch can be released safely.

Before the latch on a parent node is released, SQL Server determines whether the page will accommodate another two rows; if not, it splits the page. This occurs only if the page is being searched with the objective of adding a row to the index. The goal is to ensure that the parent page always has room for the row or rows that result from a child page splitting. (Occasionally, this results in pages being split that don't need to beat least not yet. In the long run, it's a performance optimization.) The type of split depends on the type of page being split: a root page of an index, an intermediate index page, or a data page.

Splitting the Root Page of an Index

If the root page of an index needs to be split in order for a new index row to be inserted, two new pages are allocated to the index. All the rows from the root are split between these two new pages, and the

new index row is inserted into the appropriate place on one of these pages. The original root page is still the root, but now it has only two rows on it, pointing to each of the newly allocated pages. A root page split creates a new level in the index. Because indexes are usually only a few levels deep, this type of split doesn't occur often.

Splitting the Intermediate Index Page

An intermediate index page split is accomplished simply by locating the midpoint of the index keys on the page, allocating a new page, and then copying the lower half of the old index page into the new page. Again, this doesn't occur often, although it's more common than splitting the root page.

Splitting the Data Page

A data page split is the most interesting and potentially common case, and it's probably the only split that you, as a developer, should be concerned with. Data pages split only under insert activity and only when a clustered index exists on the table. If no clustered index exists, the insert goes on any page that has room for the new row, according to the PFS pages. Although splits are caused only by insert activity, that activity can be a result of an UPDATE statement, not just an INSERT statement. As you're about to learn, if the row can't be updated in place or at least on the same page, the update is performed as a delete of the original row followed by an insert of the new version of the row. The insertion of the new row can, of course, cause a page split.

Splitting a data page is a complicated operation. Much like an intermediate index page split, it's accomplished by locating the midpoint of the index keys on the data page, allocating a new page, and then copying half of the old page into the new page. It requires

that the index manager determine the page on which to locate the new row and then handle large rows that don't fit on either the old page or the new page. When a data page is split, the clustered index key values don't change, so the nonclustered indexes aren't affected.

Let's look at what happens to a page when it splits. The following script creates a table with large rows so large, in fact, that only five rows will fit on a page. Once the table is created and populated with five rows, we'll find its first (and only, in this case) page by inserting the output of DBCC IND in the *sp_table_pages* table, finding the information for the data page with no previous page, and then using DBCC PAGE to look at the contents of the page. Because we don't need to see all 8,020 bytes of data on the page, we'll look at only the row offset array at the end of the page and then see what happens to those rows when we insert a sixth row.

```
/* First create the table */
USE AdventureWorks;
GO

DROP TABLE bigrows;
GO

CREATE TABLE bigrows
(
    a int primary key,
    b varchar(1600)
);
GO

/* Insert five rows into the table */
INSERT INTO bigrows
VALUES (5, REPLICATE('a', 1600));
```

```

INSERT INTO bigrows
    VALUES (10, replicate('b', 1600));

INSERT INTO bigrows
    VALUES (15, replicate('c', 1600));

INSERT INTO bigrows
    VALUES (20, replicate('d', 1600));
INSERT INTO bigrows
    VALUES (25, replicate('e', 1600));
GO

TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind ( AdventureWorks, bigrows,
-1)' );
SELECT PageFID, PagePID
FROM sp_table_pages
WHERE PageType = 1;

RESULTS: (Yours will vary.)
PageFID PagePID
-----
2252

DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks, 1, 2252, 1);

```

Here is the row offset array from the DBCC PAGE output:

Row - Offset
4 (0x4) - 6556 (0x199c)
3 (0x3) - 4941 (0x134d)

```
2 (0x2) - 3326 (0xcf6)
1 (0x1) - 1711 (0x6af)
0 (0x0) - 96 (0x60)
```

Now we'll insert one more row and look at the row offset array again:

```
INSERT INTO bigrows
    VALUES (22, REPLICATE('x', 1600)) ;
GO
DBCC PAGE(AdventureWorks, 1, 2252, 1);
GO
```

When you inspect the original page after the split, you might find that it contains either the first half of the rows from the original page or the second half. SQL Server normally moves the rows so the new row to be inserted goes on the new page. Because the rows are moving anyway, it makes more sense to adjust their positions to accommodate the new inserted row. In this example, the new row, with a clustered key value of 22, would have been inserted in the second half of the page. So when the page split occurs, the first three rows stay on the original page, 2252. You can inspect the page header to find the location of the next page, which contains the new row.

The page number is indicated by the *m_nextPage* field. This value is expressed as a *file number:page number* pair, in decimal, so you can easily use it with the DBCC PAGE command. When I ran this query, I got an *m_nextPage* of 1:2303, so I ran the following command:

```
DBCC PAGE(AdventureWorks, 1, 2303, 1);
```

Here's the row offset array after the insert for the second page:

Row	-	Offset
2	(0x2)	- 1711 (0x6af)
1	(0x1)	- 3326 (0xcf6)
0	(0x0)	- 96 (0x60)

Note that after the page split, three rows are on the page: the last two original rows with keys of 20 and 25, and the new row with a key of 22. If you examine the actual data on the page, you'll notice that the new row is at slot position 1, even though the row itself is physically the last one on the page. Slot 1 (with value 22) starts at offset 3326, and slot 2 (with value 25) starts at offset 1711. The clustered key ordering of the rows is indicated by the slot number of the row, not by the physical position on the page. If a table has a clustered index, the row at slot 1 will always have a key value less than the row at slot 2 and greater than the row at slot 0.

Although the typical page split isn't too expensive, you'll want to minimize the frequency of page splits in your production system, at least during peak usage times. One page split is cheap, hundreds or thousands are not. You can avoid system disruption during busy times by reserving some space on pages using the FILLFACTOR clause when you're creating the clustered index on existing data. You can use this clause to your advantage during your least busy operational hours by periodically re-creating the clustered index with the desired FILLFACTOR. That way, the extra space is available during peak usage times, and you'll save the overhead of splitting then. If there are no "slow" times, you can reorganize your index with ALTER INDEX and readjust the FILLFACTOR without having to make the entire table unavailable. Note that ALTER INDEX with REORGANIZE can only adjust the fillfactor by compacting data and

removing pages; it will never add new pages to re-establish a fillfactor. Using SQL Server Agent, you can easily schedule the rebuilding or reorganizing of indexes to occur at times when activity is lowest. I'll talk about rebuilding and reorganizing your indexes in the section titled "[ALTER INDEX](#)."

Note



The FILLFACTOR setting is helpful only when you're creating the index on existing data; the FILLFACTOR value isn't maintained when you insert new data. Trying to maintain extra space on the page would actually defeat the purpose because you'd need to split pages anyway to keep that amount of space available.

Deleting Rows

When you delete rows from a table, you have to consider what happens both to the data pages and the index pages. Remember that the data is actually the leaf level of a clustered index, and deleting rows from a table with a clustered index happens the same way as deleting rows from the leaf level of a nonclustered index. Deleting rows from a heap is managed in a different way, as is deleting from node pages of an index.

Deleting Rows from a Heap

SQL Server 2005 doesn't automatically compress space on a page when a row is deleted. As a performance optimization, the compaction doesn't occur until a page needs additional contiguous space for inserting a new row. You can see this in the following example, which deletes a row from the middle of a page and then inspects that page using DBCC PAGE.

```
USE AdventureWorks;
GO

CREATE TABLE smallrows
(
    a int identity,
    b char(10)
);
GO

INSERT INTO smallrows
    VALUES ('row 1');
INSERT INTO smallrows
    VALUES ('row 2');
INSERT INTO smallrows
    VALUES ('row 3');
INSERT INTO smallrows
    VALUES ('row 4');
INSERT INTO smallrows
    VALUES ('row 5');
GO

TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ( 'dbcc ind (AdventureWorks, smallrows,
-1) ' );
```

```
SELECT PageFID, PagePID  
FROM sp_table_pages  
WHERE PageType = 1;
```

Results:

PageFID PagePID

1 4536

DBCC TRACEON(3604);

GO

```
DBCC PAGE(AdventureWorks, 1, 4536,1);
```

Here is the output from DBCC PAGE:

[\[View full width\]](#)

DATA:


```
00000010: 20200200 fc|||||||||||||||||||||||  
†† ...  
  
Slot 4, Offset 0xb4, Length 21, DumpStyle  
BYTE  
Record Type = PRIMARY_RECORD  
Record  
    Attributes = NULL_BITMAP  
Memory Dump @0x61D9C0B4  
00000000: 10001200 05000000 726f7720  
35202020 †.  
.....row 5  
00000010: 20200200 fc|||||||||||||||||||  
†† ...  
  
OFFSET TABLE:  
Row - Offset  
4 (0x4) - 180 (0xb4)  
3 (0x3) - 159 (0x9f)  
2 (0x2) - 138 (0x8a)  
1 (0x1) - 117 (0x75)  
0 (0x0) - 96 (0x60)
```

Now we'll delete the middle row (WHERE a = 3) and look at the page again.

```
DELETE FROM smallrows  
WHERE a = 3;
```

GO

```
DBCC PAGE(AdventureWorks, 1, 4536,1);  
GO
```

Here is the output from the second execution of DBCC PAGE.

[\[View full width\]](#)

DATA:

Slot 0, Offset 0x60, Length 21, DumpStyle
BYTE

Record Type = PRIMARY RECORD

Record

Attributes = NULL_BITMAP

Memory Dump @0x61B6C060

00000000: 10001200 01000000 726f7720

31202020 †.

www. row 1

三

Slot 1, Offset 0x75, Length 21, DumpStyle
BYTE

Record Type = PRTMARY RECORD

Record

Attributes = NUL | BTTMAP

Memory Dump @0x61B6C075

00000000: 10001200 02000000 726f7720



Note that in the heap, the row doesn't show up in the page itself. The row offset array at the bottom of the page shows that the third row (at slot 2) is now at offset 0 (which means there really is no row using slot 2), and the row using slot 3 is at its same offset as before the delete. The data on the page is *not* compressed.

In addition to space on pages not being reclaimed, empty pages in heaps frequently cannot be reclaimed. Even if you delete all the rows from a heap, SQL Server will not mark the empty pages as unallocated, so the space will not be available for other objects to use. The catalog view `sys.dm_db_partition_stats` will still show the space as belonging to the heap table.

Deleting Rows from a B-Tree

In the leaf level of an index, either clustered or nonclustered, when rows are deleted, they're marked as *ghost records*. This means that the row stays on the page but a bit is changed in the row header to indicate that the row is really a ghost. The page header also reflects the number of ghost records on a page. Ghost records are used for several purposes. They can be used to make rollback much more efficient; if the row hasn't physically been removed, all SQL Server has to do to roll back a delete is to change the bit indicating that the row is a ghost. It is also a concurrency optimization for key-range locking (which I'll discuss in [Chapter 8](#)), along with other locking modes. Ghost records are also used to support row-level versioning; that will also be discussed in [Chapter 8](#).

Ghost records are cleaned up sooner or later, depending on the load on your system, and sometimes they can be cleaned up before you have a chance to inspect them. In the code shown below, if you perform the DELETE and then wait a minute or two to run DBCC PAGE, the ghost record might really disappear. That is why I look at the page number for the table before I run the DELETE, so I can execute the DELETE and the DBCC page with a single click from my query window. To guarantee that the ghost will not be cleaned up, I can put the DELETE into a user transaction and not commit or roll back the transaction before examining the page. The housekeeping thread will not clean up ghost records that are part of an active transaction. Alternatively, I can use an undocumented trace flag 661 to disable ghost cleanup to ensure consistent results when running tests such as in this script. As usual, keep in mind that undocumented trace flags are not guaranteed to continue to work in any future release or service pack, and no support is available for them. Also be sure to turn off the trace flag when you're done with your testing.

The following example builds the same table used in the previous DELETE example, but this time the table has a primary key declared, which means a clustered index will be built. The data is the leaf level of the clustered index, so when the row is removed, it will be marked as a ghost.

```
USE AdventureWorks;
GO
DROP TABLE smallrows;
GO
CREATE TABLE smallrows
(
    a int IDENTITY PRIMARY KEY,
    b char(10)
);
GO
INSERT INTO smallrows
```

```
        VALUES ('row 1');
INSERT INTO smallrows
        VALUES ('row 2');
INSERT INTO smallrows
        VALUES ('row 3');
INSERT INTO smallrows
        VALUES ('row 4');
INSERT INTO smallrows
        VALUES ('row 5');
GO
TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind (Adventureworks, smallrows,
-1) ' );
SELECT PageFID, PagePID
FROM sp_table_pages
WHERE PageType = 1;
```

Results:

```
PageFID PagePID
```

```
-----  
1      4568
```

```
DELETE FROM smallrows
WHERE a = 3;
GO
DBCC TRACEON(3604);
DBCC PAGE(AdventureWorks, 1, 4544, 1);
GO
```

Here is the output from DBCC PAGE:



[\[View full width\]](#)

```
PAGE HEADER:  
Page @0x064AE000  
m_pageId = (1:4568)  
    m_headerVersion = 1  
m_type = 1  
m_typeFlagBits = 0x4  
m_level = 0  
                                m_flagBits =  
0x8000  
m_objId (AllocUnitId.idObj) = 172  
m_indexId  
    (AllocUnitId.idInd) = 256  
Metadata: AllocUnitId = 72057594049200128  
Metadata: PartitionId = 72057594043105280  
                                Metadata: In  
dexId = 1  
Metadata: ObjectId = 1179867270  
m_prevPage =  
    (0:0)                                m_nextPage =  
    (0:0)  
pminlen = 18  
m_slotCnt = 5  
                                m_freeCnt =  
7981  
m_freeData = 201  
m_reservedCnt  
    = 0                                m_lsn = (233  
:499:2)  
m_xactReserved = 0  
m_xdesId = (0  
:18856)                                m_ghostRecCn
```



```
0 (0x0) - 96 (0x60)
1 (0x1) - 117 (0x75)
```

Note that the row still shows up in the page itself because the table has a clustered index. The header information for the row shows that this is really a ghosted record. The row offset array at the bottom of the page shows that the row at slot 2 is still at the same offset and that all rows are in the same location as before the deletion. In addition, the page header gives us a value (*m_ghostRecCnt*) for the number of ghosted records in the page. To see the total count of ghost records in a table, you can look at the *sys.dm_db_index_physical_stats* function, which we'll see in detail later in this chapter.

Deleting Rows in the Node Levels of an Index

When you delete a row from a table, all nonclustered indexes must be maintained because every nonclustered index has a pointer to the row that's now gone. Rows in index node pages aren't ghosted when deleted, but just as with heap pages, the space isn't compressed until new index rows need space in that page.

Reclaiming Pages

When the last row is deleted from a data page, the entire page is deallocated. The exception is if the table is a heap, as I discussed earlier. (If the page is the only one remaining in the table, it isn't deallocated. A table always contains at least one page, even if it's empty.) Deallocation of a data page results in the deletion of the row in the index page that pointed to the deallocated data page. Index pages are deallocated if an index row is deleted (which, again, might occur as part of a delete/insert update strategy), leaving only one entry in the index page. That entry is moved to its neighboring page, and then the empty page is deallocated.

The discussion so far has focused on the page manipulation necessary for deleting a single row. If multiple rows are deleted in a single delete operation, you must be aware of some other issues. Because the issues of modifying multiple rows in a single query are the same for inserts, updates, and deletes, we'll discuss this issue in its own section, later in this chapter.

Updating Rows

SQL Server updates rows in multiple ways, automatically and invisibly choosing the fastest update strategy for the specific operation. In determining the strategy, SQL Server evaluates the number of rows affected, how the rows will be accessed (via a scan or an index retrieval, and via which index), and whether changes to the index keys will occur. Updates can happen either in place, by just changing one column's value to a new value in the original row, or as a delete followed by an insert. In addition, updates can be managed by the query processor or by the storage engine. In this section, we'll examine only whether the update happens in place or whether SQL Server treats it as two separate operations: delete the old row and insert a new row. The question of whether the update is controlled by the query processor or the storage engine is actually

relevant to all data modification operations (not just updates), so we'll look at that in a separate section.

Moving Rows

What happens if a table row has to move to a new location? In SQL Server 2005, this can happen because a row with variable-length columns is updated to a new, larger size so that it no longer fits on the original page. It can also happen when the clustered index column is changing, because rows are logically ordered by the clustering key. For example, if we have a clustered index on *lastname*, a row with a *lastname* value of *Abbot* will be stored near the beginning of the table. If the *lastname* value is then updated to *Zappa*, this row will have to move to near the end of the table.

Earlier in this chapter, we looked at the structure of indexes and saw that the leaf level of non-clustered indexes contains a row locator, or bookmark, for every single row in the table. If the table has a clustered index, that row locator is the clustering key for that row. So if and only if the clustered index key is being updated, modifications are required in every nonclustered index. Keep this in mind when you decide which columns to build your clustered index on. It's a great idea to cluster on a nonvolatile column.

If a row moves because it no longer fits on the original page, it will still have the same row locator (in other words, the clustering key for the row stays the same), and no nonclustered indexes will have to be modified.

In our discussion of index internals, you also saw that if a table has no clustered index (in other words, it's a heap), the row locator stored in the nonclustered index is actually the physical location of the row. In SQL Server 2005, if a row in a heap moves to a new page, the row will leave a forwarding pointer in the original location. The nonclustered indexes won't need to be changed; they'll still

refer to the original location, and from there they'll be directed to the new location.

Let's look at an example. I'll create a table a lot like the one we created for doing inserts, but this table will have a third column of variable length. After I populate the table with five rows, which will fill the page, I'll update one of the rows to make its third column much longer. The row will no longer fit on the original page and will have to move. I can then load the output from DBCC IND into the *sp_table_pages* table to get the page numbers used by the table.

```
USE AdventureWorks;
GO
DROP TABLE bigrows;
GO
CREATE TABLE bigrows
(
    a int IDENTITY ,
    b varchar(1600),
    c varchar(1600));
GO
INSERT INTO bigrows
    VALUES (REPLICATE('a', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('b', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('c', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('d', 1600), '');
INSERT INTO bigrows
    VALUES (REPLICATE('e', 1600), '');

GO
UPDATE bigrows
SET c = REPLICATE('x', 1600)
WHERE a = 3;
GO
```

```
TRUNCATE TABLE sp_table_pages;
INSERT INTO sp_table_pages
    EXEC ('dbcc ind (AdventureWorks, bigrows, -1)'
);
SELECT PageFID, PagePID
FROM sp_table_pages
WHERE PageType = 1;
```

RESULTS:

PageFID	PagePID
-----	-----
1	2252
1	4586

```
DBCC TRACEON(3604);
GO
DBCC PAGE(AdventureWorks, 1, 2252, 1);
GO
```

I won't show you the entire output from the DBCC PAGE command, but I'll show you what appears in the slot where the row with $a = 3$ formerly appeared:

```
Slot 2, Offset 0x1feb, Length 9, DumpStyle BYTE
Record Type = FORWARDING_STUB           Record
Attributes =
Memory Dump @0x61ADDFFEB
00000000: 04ea1100 00010000 00|||||||||||||||  
.....
```

The value of 4 in the first byte means that this is just a forwarding stub. The 0011ea in the next 3 bytes is the page number to which the row has been moved. Because this is a hexadecimal value, we need to convert it to 4586 decimal. The next group of 4 bytes tells us that the page is at slot 0, file 1. If you then use DBCC PAGE to look at that page 4586, you can see what the forwarded record looks like.

Managing Forward Pointers

Forward pointers allow you to modify data in a heap without worrying about having to make drastic changes to the nonclustered indexes. If a row that has been forwarded must move again, the original forwarding pointer is updated to point to the new location. You'll never end up with a forwarding pointer pointing to another forwarding pointer. In addition, if the forwarded row shrinks enough to fit in its original place, the record might move back to its original place, if there is still room on that page, and the forward pointer would be eliminated.

A future version of SQL Server might include some mechanism for performing a physical reorganization of the data in a heap, which will get rid of forward pointers. Note that forward pointers exist only in heaps, and that the ALTER TABLE option to reorganize a table won't do anything to heaps. You can defragment a nonclustered index on a heap, but not the table itself. Currently, when a forward pointer is created, it stays there forever with only a few exceptions. The first exception is the case I already mentioned, in which a row shrinks and returns to its original location. The second exception is when the entire database shrinks. The bookmarks are actually reassigned when a file is shrunk. The shrink process never generates forwarding pointers. For pages that were removed because of the shrink process, any forwarded rows or stubs they contain are effectively "unforwarded." Other cases in which the

forwarding pointers will be removed are the obvious ones: if the forwarded row is deleted or if a clustered index is built on the table so that it is no longer a heap.

To get a count of forward records in a table, you can look at the output from the `sys.dm_db_index_physical_stats` function, which I'll discuss shortly.

Updating in Place

In SQL Server 2005, updating a row in place is the rule rather than the exception. This means that the row stays in exactly the same location on the same page and only the bytes affected are changed. In addition, the log will contain a single record for each such updated row unless the table has an update trigger on it or is marked for replication. In these cases, the update still happens in place, but the log will contain a delete record followed by an insert record.

In cases where a row can't be updated in place, the cost of a not-in-place update is minimal because of the way the nonclustered indexes are stored and because of the use of forwarding pointers. In fact, you can have an update not-in-place for which the row stays on the original page. Updates will happen in place if a heap is being updated or if a table with a clustered index is updated without any change to the clustering keys. You can also get an update in place if the clustering key changes but the row does not need to move at all. For example, if you have a clustered index on a last name column containing consecutive key values of *Able*, *Becker*, and *Charlie*, you might want to update *Becker* to *Baker*. Because the row will stay in the same location even after the clustered index key changes, SQL Server will perform this as an update in place. On the other hand, if you update *Able* to *Buchner*, the update cannot occur in place, but the new row might stay on the same page.

Updating Not in Place

If your update can't happen in place because you're updating clustering keys, the update will occur as a delete followed by an insert. In some cases, you'll get a hybrid update: some of the rows will be updated in place and some won't. If you're updating index keys, SQL Server builds a list of all the rows that need to change as both a delete and an insert operation. This list is stored in memory, if it's small enough, and is written to *tempdb* if necessary. This list is then sorted by key value and operator (delete or insert). If the index whose keys are changing isn't unique, the delete and insert steps are then applied to the table. If the index is unique, an additional step is carried out to collapse delete and insert operations on the same key into a single update operation.

Let's look at a simple example. The following code builds a table with a unique clustered index on column X and then updates that column in both rows:

```
USE pubs
```

```
GO
```

```
DROP TABLE T1
```

```
GO
```

```
CREATE TABLE T1(X int PRIMARY KEY, Y int)
```

```
INSERT T1 VALUES(1, 10)
```

```
INSERT T1 VALUES(2, 20)
```

```
GO
```

```
UPDATE T1 SET X = 3 - X
```

```
GO
```

Here are the operations that SQL Server generates internally for the preceding UPDATE statement. This list of operations is called the *input stream* and consists of both the old and new values of every column and an identifier for each row that is to be changed. I have simplified the representation of the Row ID; SQL Server would use its own internal row identifier.

Operation	Original X	Original Y	New X	New Y	Row ID
Update	1	10	2	10	ROW1
Update	2	20	1	20	ROW2

In this case, the UPDATE statements must be split into DELETE and INSERT operations. If they aren't, the UPDATE to the first row, changing X from 1 to 2, will fail with a duplicate-key violation. A converted input stream is generated, as shown here.

Operation	Original X	Original Y	Row ID
Delete	1	10	ROW1

Operation	Original X	Original Y	Row ID
Insert	2	10	<null>
Delete	2	20	ROW2
insert	1	20	<null>

Note that for insert operations, the Row ID isn't relevant in the input stream of operations to be performed. Before a row is inserted, it has no Row ID. This input stream is then sorted by index key and operation, as shown here.

Operation	X	Y	Row ID
Delete	1	10	ROW1
Insert	1	20	<null>

Operation	X	Y	Row ID
Delete	2	20	ROW2
Insert	2	10	<null>

Finally, if the same key value has both a delete and an insert, the two rows in the input stream are collapsed into an update operation. The final input stream looks like this:

Operation	Original X	Original Y	New X	New Y	Row ID
Update	1	10	1	20	ROW1
Update	2	20	2	10	ROW2

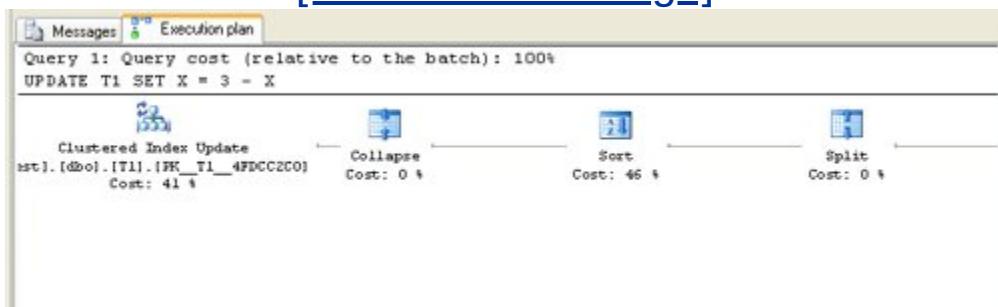
Note that although the original query was an update to column X, after the split, sort, and collapse of the input stream, the final set of

actions looks like we're updating column Y! This method of actually carrying out the update ensures that no intermediate violations of the index's unique key occur.

Part of the graphical query plan for this update ([Figure 7-9](#)) shows you the split, sort, and collapse phases. Basic query plans are described in *Inside Microsoft SQL Server 2005: T-SQL Querying*, and I'll go into much more detail about analyzing plans in *Inside Microsoft SQL Server 2005: Tuning and Optimization*.

Figure 7-9. A graphical query plan for an update operation using split and collapse

[\[View full size image\]](#)



Updates to any nonclustered index keys also affect the leaf level of the index by splitting the operation into deletes and inserts. Think of the leaf level of a nonclustered index as a miniclustered index, so any modification of the key can potentially affect the sequence of values in the leaf level. Just like for the data in a clustered index, the actual type of update is determined by whether the index is unique. If the nonclustered index is not unique, the update is split into delete and insert operations. If the nonclustered index is unique, the split is followed by a sort and an attempt to collapse any deletes and inserts on the same key back into an update operation.

Table-Level vs. Index-Level Data Modification

I've been discussing only the placement and index manipulation necessary for modifying either a single row or a few rows with no more than a single index. If you are modifying multiple rows in a single operation (insert, update, or delete) or by using BCP or the BULK INSERT command and the table has multiple indexes, you must be aware of some other issues. SQL Server 2005 offers two strategies for maintaining all the indexes that belong to a table: table-level modification and index-level modification. The query optimizer chooses between them based on its estimate of the anticipated execution costs for each strategy.

Table-level modification is sometimes called *row-at-a-time*, and index-level modification is sometimes called *index-at-a-time*. In table-level modification, all indexes are maintained for each row as that row is modified. If the update stream isn't sorted in any way, SQL Server has to do a lot of random index accesses, one access per index per update row. If the update stream is sorted, it can't be sorted in more than one order, so nonrandom index accesses can occur for at most one index.

In index-level modifications, SQL Server gathers all the rows to be modified and sorts them for each index. In other words, there are as many sort operations as there are indexes. Then, for each index, the updates are merged into the index, and each index page is never accessed more than once, even if multiple updates pertain to a single index leaf page.

Clearly, if the update is smallsay, less than a handful of rowsand the table and its indexes are sizable, the query optimizer usually considers table-level modification the best choice. Most OLTP operations use table-level modification. On the other hand, if the update is relatively large, table-level modifications require a lot of random I/O operations and might even read and write each leaf page in each index multiple times. In that case, index-level modification offers much better performance. The amount of logging required is the same for both strategies.

Let's look at a specific example with some actual numbers. Suppose that you use BULK INSERT to increase the size of a table by 1 percent (which could correspond to one week in a two-year sales history table). The table is stored in a heap (no clustered index), and the rows are simply appended because there is not much available space on other pages in the table. There's nothing to sort on for insertion into the heap, and the rows (and the required new heap pages and extents) are simply appended to the heap. Assume also that there are two non-clustered indexes on columns a and b. The insert stream is sorted on column a and is merged into the index on column a.

If an index entry is 20 bytes long, an 8-KB page filled at 70 percent (which is the natural, self-stabilizing value in a B-tree after lots of random insertions and deletions) would contain $8\text{ KB} * 70\% / 20\text{ bytes} = 280$ entries. Eight pages (one extent) would then contain 2,240 entries, assuming that the leaf pages are nicely laid out in extents. Table growth of 1 percent would mean an average of 2.8 new entries per page, or 22.4 new entries per extent.

Table-level (row-at-a-time) insertion would touch each page an average of 2.8 times. Unless the buffer pool is large, row-at-a-time insertion reads, updates, and writes each page 2.8 times. That's 5.6 I/O operations per page, or 44.8 I/O operations per extent. Index-level (index-at-a-time) insertion would read, update, and write each page (and extent, again presuming a nice contiguous layout) exactly once. In the best case, about 45 page I/O operations are replaced by 2 extent I/O operations for each extent. Of course, the cost of doing the insert in this way also includes the cost of sorting the inputs for each index. But the cost savings of the index-level insertion strategy can easily offset the cost of sorting.

Note



Earlier versions of SQL Server recommend that you drop all your indexes if you are going to import a lot of rows into a table using a bulkcopy interface because they offer no index-level strategy for maintaining all the indexes.

You can determine whether your updates were done at the table level or the index level by inspecting the query execution plan. If SQL Server performs the update at the index level, you'll see a plan produced that contains an update operator for each of the affected indexes. If SQL Server performs the update at the table level, you'll see only a single update operator in the plan.

Logging

Standard INSERT, UPDATE, and DELETE statements are always logged to ensure atomicity, and you can't disable logging of these operations. The modification must be known to be safely on disk in the transaction log (write-ahead logging) before the commit of the statement or transaction can be acknowledged to the calling application. Page allocations and deallocations, including those done by TRUNCATE TABLE, are also logged. As we saw in [Chapter 5](#), certain operations can be minimally logged when your database is in the BULK_LOGGED recovery mode, but even then, information about allocations and deallocations is written to the log, along with the fact that a minimally logged operation has been executed.

Locking

Any data modification must always be protected with some form of exclusive lock. For the most part, SQL Server makes all the locking decisions internally; a user or programmer doesn't need to request a particular kind of lock. [Chapter 8](#) explains the different types of locks and their compatibility. *Inside Microsoft SQL Server 2005: Tuning and Optimization* will discuss techniques for overriding SQL Server's default locking behavior. However, because locking is closely tied to data modification, you should always be aware of the following:

- Every type of data modification performed in SQL Server requires some form of exclusive lock. For most data modification operations, SQL Server considers row locking as the default, but if many locks are required, SQL Server can lock pages or even the whole table.

- Update locks can be used to signal the intention to do an update, and they are important for avoiding deadlock conditions. But ultimately, the update operation requires that an exclusive lock be performed. The update lock serializes access to ensure that an exclusive lock can be acquired, but the update lock isn't sufficient by itself.
- Exclusive locks must always be held until the end of a transaction in case the transaction needs to be undone (unlike shared locks, which can be released as soon as the scan moves off the page, assuming that READ COMMITTED isolation is in effect).
- If a full table scan must be employed to find qualifying rows for an update or a delete, SQL Server has to inspect every row to determine the row to modify. Other processes that need to find individual rows will be blocked even if they will ultimately modify different rows. Without inspecting the row, SQL Server has no way of knowing whether the row qualifies for the modification. If you're modifying only a subset of rows in the table, as determined by a WHERE clause, be sure that you have indexes available to allow SQL Server to access the needed rows directly so it doesn't have to scan every row in the table.

Managing Indexes

SQL Server maintains your indexes automatically. As you add new rows, it automatically inserts them into the correct position in a table with a clustered index, and it adds new leaf-level rows to your nonclustered indexes that will point to the new rows. When you remove rows, SQL Server automatically deletes the corresponding leaf-level rows from your nonclustered indexes. So, although your indexes will continue to contain all the correct index rows in the B-tree to help SQL Server find the rows you are looking for, you might still occasionally need to perform maintenance operations on your indexes. In addition, several properties of indexes can be changed.

ALTER INDEX

SQL Server 2005 introduces the ALTER INDEX command to allow you to use a single command to invoke various kinds of index changes that in previous versions required an eclectic collection of different commands, including *sp_indexoption*, UPDATE STATISTICS, DBCC DBREINDEX and DBCC INDEXDEFrag. Instead of having individual commands or procedures for each different index maintenance activity, they all can be done by using ALTER INDEX.

Basically, you can make four types of changes using ALTER INDEX, three of which have corresponding options that you can specify when you create an index using CREATE INDEX.

- **Rebuilding an index** This option replaces both the DBCC DBREINDEX command and the DROP_EXISTING option to the CREATE INDEX command. A new option allows indexes to be rebuilt online, in the same way you can create indexes

online (as I mentioned in the earlier section titled "[Creating an Index](#)"). I'll discuss online index building and rebuilding shortly.

- **Disabling an index** This option makes an index completely unavailable, so it can't be used for finding rows for any operations. Disabling an index also means that the index won't be maintained as changes to the data are made. You can disable one index or ALL indexes with a single command. There is no ENABLE option. Because no maintenance is done while an index is disabled, indexes must be completely rebuilt to make them useful again. Re-enabling, which can take place either online or offline, is done with the REBUILD option to ALTER INDEX. This feature was introduced mainly for SQL Server's own internal purposes when applying upgrades and service packs, but you can use it when you want to temporarily ignore the index for troubleshooting purposes.

Warning



If you disable the clustered index on a table, none of the data will be available because the data is stored in the clustered index leaf level.

- **Changing index options** Four of the options that you can specify during a CREATE INDEX operation can also be specified with the ALTER INDEX command. These options are ALLOW_ROW_LOCKS, ALLOW_PAGE_LOCKS, IGNORE_DUP_KEY, and STATISTICS_NORECOMPUTE. As

mentioned earlier in this chapter, the locking options will be discussed in [Chapter 8](#), and the statistics option will be discussed in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*. The IGNORE_DUP_KEY option was described in the section titled "[Creating an Index](#)."

- **Reorganizing an index** This is the only change that doesn't have a corresponding option in the CREATE INDEX command because when you create an index, there is nothing to reorganize. The REORGANIZE option replaces the DBCC INDEXDEFRAG command and removes some of the fragmentation from an index, but it is not guaranteed to remove all the fragmentation, just as DBCC INDEXDEFRAG doesn't remove all the fragmentation, in spite of its name. Before we discuss removing fragmentation, we must first discuss fragmentation, which we will do in the next section.

Types of Fragmentation

As I mentioned, you need to do very little in terms of index maintenance. However, indexes can become fragmented. Whether fragmentation is a bad thing depends on what you'll do with your data and indexes, as well as on the type of fragmentation. Although technically fragmentation can occur in heaps as well as in index B-trees, most of this discussion will center on index fragmentation. Fragmentation can be of two types: internal and external. Internal fragmentation occurs when space is available within your index pages that is, when the index pages have not been filled as full as possible. External fragmentation occurs when the logical order of pages does not match the physical order, or when the extents belonging to an index are not contiguous. (You might also think about fragmentation on the physical disk, which might occur when your database files grow in increments and the operating system allocates space for these new increments on different parts of the

disk. I can recommend that you plan carefully and preallocate space for your files so they will not need to grow in an unplanned manner, but a full discussion of fragmentation at the disk level is beyond the scope of this book.)

Internal fragmentation means that the index is taking up more space than it needs to. Scanning the entire table or index involves more read operations than if no free space were available on your pages. However, internal fragmentation is sometimes desirable. In fact, you can request internal fragmentation by specifying a low *fillfactor* value when you create an index. Having room on a page means that there is space to insert more rows without having to split a page. Splitting a page can be an expensive operation if it occurs frequently, and it can lead to external fragmentation because when a page is split, a new page must be linked into the indexes page chain, and usually the new page is not contiguous to the page being split.

External fragmentation is truly bad only when SQL Server is doing an ordered scan of all or part of a table or an index. If you're seeking individual rows through an index, it doesn't matter where those rows are physically locatedSQL Server can find them easily. If SQL Server is doing an unordered scan, it can use the IAM pages to determine which extents need to be fetched, and the IAM pages list the extents in disk order, so the fetching can be very efficient. Only if the pages need to be fetched in logical order, according to their index key values, do you need to follow the page chain. If the pages are heavily fragmented, this operation is more expensive than if there were no fragmentation.

Detecting Fragmentation

The `sys.dm_db_index_physical_stats` function replaces DBCC SHOWCONTIG as the recommended way to get a fragmentation report for a table or index. From a usage standpoint, the nicest

advantage of using the new function is that you can filter the information returned. If you request all the columns of information that this function could potentially return, you'll actually get a lot more information than you get with DBCC SHOWCONTIG, but because the data is returned with a table-valued function, you can restrict the columns as well as the rows that you want returned.

The function `sys.dm_db_index_physical_stats` takes five parameters, and they all have defaults. If you set all the parameters to their defaults and do not filter the rows or the columns, the function will return 20 columns of data for (almost) every level of every index on every table on every partition in every database of the current SQL Server instance. You would request that information as follows:

```
SELECT * FROM sys.dm_db_index_physical_stats  
(NULL, NULL, NULL, NULL, NULL);
```

When I run this command on a very small SQL Server instance, with only the AdventureWorks, pubs, and Northwind databases in addition to the system databases, I get more than 390 rows returned. Obviously, 20 columns and 390 rows is too much output to illustrate here, so I'll just let you run the command for yourself and see what you get.

Let's look at the parameters now.

- **Database_ID** The first parameter must be specified as a number, but you can embed the `db_id` function as a parameter if you want to specify the database by name. If you specify `NULL`, which is the default, the function returns information about ALL databases. If the database ID is null, the next three parameters must also be `NULL` (which is their default value).

- **Object_ID** The second parameter is the object ID, which again must be a number, not a name. Again, the NULL default means you want information about all objects, and in that case, the next two parameters, *index_id* and *partition_id* must be NULL. Just as for the database ID, you can use an embedded function to get the object ID if you know the object name. (If the object is in a different database than your current database, you should use a three-part object name, including the database name and the schema name.)
- **Index_id** The third parameter allows you to specify the index ID from a particular table, and again the default of NULL indicates that you want all the indexes.
- **Partition_number** The fourth parameter indicates the partition number, and NULL means you want information for all the partitions. (Remember that if you haven't explicitly created a table or index on a partition scheme, SQL Server considers it to be built on a single partition.)
- **Mode** The fifth and last parameter is the only one for which the default NULL does not imply returning the most information. The last parameter indicates the MODE of sampling that you want SQL Server to perform when retrieving the information. The mode in which the function is executed determines the level of scanning performed to obtain the information that the function uses. When the function is called, SQL Server traverses the page chains for the allocated pages for the specified partitions of the table or index. Unlike DBCC SHOWCONTIG in SQL Server 2000, which usually requires a shared (S) table lock, *sys.dm_db_index_physical_stats* (and DBCC SHOWCONTIG in SQL Server 2005) requests only an Intent-Shared (IS) table lock, which is compatible with most other kinds of lock, as we'll see in [Chapter 8](#). Valid inputs are

DEFAULT, NULL, LIMITED, SAMPLED, and DETAILED. The default is NULL, which corresponds to LIMITED. Here is what the latter three values mean:

- **LIMITED** The LIMITED mode is the fastest and scans the smallest number of pages. It scans all pages for a heap, but only the node pages for an index.
- **SAMPLED** The SAMPLED mode returns statistics based on a 1-percent sample of all the pages in the index or heap. However, if the table is relatively small, SQL Server converts SAMPLED to DETAILED. (For the purposes of this function, small means fewer than 10,000 pages.)
- **DETAILED** The DETAILED mode scans all pages and returns all statistics.

You must be careful when using the built-in *db_id* or *object_id* functions. If you specify an invalid name, or simply misspell the name, you do not receive an error message and the value returned is NULL. However, because NULL is a valid parameter, SQL Server will just assume that this is what you meant to use. For example, say I want to see all the previously described information, but only for the *AdventureWorks* database, and I mistype the command as follows:

```
SELECT * FROM sys.dm_db_index_physical_stats  
          (db_id('AdventureWords'), NULL, NULL,  
NULL, NULL);
```

There is no such database as *AdventureWorks*, so the *db_id* function will return NULL, and it will be as if I had called the function with all NULL parameters. No error or warning will be given.

You might be able to guess from the number of rows returned that you made an error, but of course, if you have no idea how much output you are expecting, it might not be immediately obvious. Books Online suggests that you can avoid this issue by capturing the IDs into variables and error-checking the values in the variables before calling the `sys.dm_db_index_physical_stats` function, as shown in this code:

```
DECLARE @db_id SMALLINT;
DECLARE @object_id INT;

SET @db_id = DB_ID(N'AdventureWorks');
SET @object_id =
OBJECT_ID(N'AdventureWorks.Person.Address');

IF @db_id IS NULL
BEGIN;
    PRINT N'Invalid database';
END;
ELSE IF @object_id IS NULL
BEGIN
    PRINT N'Invalid object';
END;

SELECT * FROM sys.dm_db_index_physical_stats
(@db_id, @object_id, NULL, NULL, NULL);
```

Another, more insidious problem is that the `object_id` function is called based on your current database, before any call to the `sys.dm_db_index_physical_stats` function is made. So if you are in the `AdventureWorks` database but want information from a table in the `pubs` database, you could try running the following code:

```
SELECT *
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'pubs'), OBJECT_ID(N'dbo.authors'),
null, null, null);
```

Because there is no *dbo.authors* table in my current database (*AdventureWorks*), the *object_id* is treated as NULL, and I get all the information from all the objects in *pubs*.

The only time SQL Server will report an error is when there are objects with the same name in two different databases. So if there were a *dbo.authors* table in *AdventureWorks*, the ID for that table would be used to try to retrieve data from the *pubs* database. SQL Server will return an error if the ID value returned by *OBJECT_ID* does not match the ID value of the object in the specified database. The following script shows the error:

```
USE AdventureWorks
GO
CREATE TABLE dbo.authors
    (ID char(11), name varchar(60))
GO
SELECT *
FROM sys.dm_db_index_physical_stats
    (DB_ID(N'pubs'), OBJECT_ID(N'dbo.authors'),
null, null, null);
```

When you run the preceding SELECT, the *dbo.authors* ID will be determined based on the current environment, which is still *AdventureWorks*. But when SQL Server tries to use that ID in *pubs*, an error will be generated:

Msg 2573, Level 16, State 40, Line 1
Could not find table or object ID 512720879. Check system catalog.

The best solution is to fully qualify the table name, either in the call to the `sys.dm_db_index_physical_stats` function itself or, as in the code sample from Books Online shown earlier, to use variables to get the ID of the fully qualified table name. If you write wrapper procedures to call the `sys.dm_db_index_physical_stats` function, you can concatenate the database name onto the object name before retrieving the object ID and thereby avoid the problem. Because the output of this function is a bit cryptic, you might find it beneficial to write your own procedure to access this function and return the information in a slightly friendlier fashion.

As I mentioned, you can potentially get a lot of rows back from this function if you use all the default parameters. But even for a subset of tables or indexes, and with careful use of the available parameters, you still might get more data back than you want. Because `sys.dm_db_index_physical_stats` is a table-valued function, you can add your own filters to the results being returned. For example, you can choose to look at the results for just the nonclustered indexes. Using the available parameters, your only choices are to see all the indexes or only one particular index. If we make the third parameter `NULL` to specify all indexes, we can then add a filter in a `WHERE` clause to indicate that we want only rows `WHERE index_id is BETWEEN 2 and 250`. Alternatively, we can choose to look only at rows that indicate a high degree of fragmentation.

Next we'll look at the table returned by this function and see what information about fragmentation is returned.

Fragmentation Report

I mentioned earlier that the output of `sys.dm_db_index_physical_stats` can return a row for each partition of each table or index, but it actually can return more than that. By default, the function returns a row for each level of each partition of each index. For a small index with only in-row data (no row-overflow or LOB pages) and only the one default partition, we might get only two or three rows back (one for each index level). But if there are multiple partitions and additional allocation units for the row-overflow and LOB data, we might see many more rows, even though only the leaf level of a clustered index (the data) can have row-overflow and LOB data. For example, a clustered index on a table containing row-overflow data, built on 11 partitions and being two levels deep, will have 33 rows (2 levels x 11 partitions + 11 partitions for the `row_overflow` allocation units) in the fragmentation report returned by `sys.dm_db_index_physical_stats`.

The following columns in the output will help you identify what each row of output refers to:

- ***database_id*** The ID of the database containing the table or view (in the case of indexed views).
- ***object_id*** The ID of the table or view.
- ***index_id*** An Index ID value of 1 indicates a clustered index; 0 indicates a heap. Other indexes have values greater than 1.
- ***partition_number*** A 1-based partition number within the table or index. A non-partitioned index or heap will have a partition number of 1.

- ***index_type_desc*** This value is one of the following: HEAP, CLUSTERED INDEX, NONCLUSTERED INDEX, PRIMARY XML INDEX, or XML INDEX. (XML indexes are beyond the scope of this book.)
- ***alloc_unit_type_desc*** A description of the allocation unit type. The value is IN_ROW_DATA, LOB_DATA, or ROW_OVERFLOW_DATA.
- ***index_level*** The current level of the index is 0 for index leaf levels, heaps, and LOB_DATA or ROW_OVERFLOW_DATA allocation units. Note that the leaf level of a clustered index is the data itself. The *index_level* is greater than 0 for nonleaf index levels. The *index_level* will be the highest at the root level of an index.

In SQL 2000, DBCC SHOWCONTIG provides information about four types of fragmentation, which include internal fragmentation and three types of external fragmentation. The values reported for two of the types of fragmentation are actually meaningless if a table or index spans multiple files, but the output is returned anyway. For SQL Server 2005, all values reported are valid across multiple files. However, not every type of fragmentation is relevant to every structure. Here are the columns in the output from *sys.dm_db_index_physical_stats* that report on fragmentation:

- ***avg_page_space_used_in_percent*** This *float* value represents the internal fragmentation, measured as the average across all pages for one particular index level for one type of allocation unit in one partition. This is the only type of fragmentation that is reported for the LOB pages and the row-overflow pages.

- ***avg_fragmentation_in_percent*** This float value represents the external fragmentation, which is logical fragmentation for indexes or extent fragmentation for heaps in the in-row allocation unit. A value of 0 is returned for LOB and row-overflow allocation units.

Logical fragmentation is the percentage of out-of-order pages in one level of an index. Each page has a pointer in the page header that indicates what the next page in that level in logical order should be, and an out-of-order page is one that has a lower page number than the previous page. For example, if the leaf level of an index contains page 86 and page 86 indicates that the next page in index key order is page 77, page 77 is an out-of-order page. If the next page after 86 is 102, it is not out of order. Logical fragmentation is not concerned with contiguosity, only that the logical order of pages matches the physical order on the disk. Logical order is meaningful only for an index, where the order is controlled by the data type of the index key column.

Extent fragmentation measures the contiguosity of the data belonging to a heap. SQL Server allocates space to a table or to index extents, which are eight contiguous pages. The first page number of every extent is a multiple of 8, so the extent starting on page 16 is contiguous with the extent starting on page 24. Extent fragmentation counts the gaps between extents belonging to the same object. If the extent starting on page 8 belongs to Table1 and the extent starting on page 24 also belongs to Table1 but the extent starting on page 16 belongs to a different table, there is one gap and two extents for Table1.

The value for *avg_fragmentation_in_percent*, whether it indicates logical or extent fragmentation, should be as close to zero as possible for maximum performance when SQL Server is scanning a table or index.

- ***Fragment_count* and *avg_fragment_size_in_pages*** A third type of fragmentation can be observed by looking at these two values reported by `sys.dm_db_index_physical_stats`. A fragment consists of physically consecutive leaf pages in the same file for an allocation unit. Every index has at least one fragment. The maximum number of fragments that an index can have equals the number of pages in the leaf level if none of the pages are contiguous or in order. If an index has larger fragments, it means that there is less I/O to read the same amount of pages because SQL Server can take advantage of Read Ahead. A bigger value for *avg_fragment_size_in_pages* results in better performance when scanning the data. As long as *avg_fragment_size_in_pages* is greater than 8 pages (64 KB), which is the size of an extent, scan performance will be reasonably good. However, there is an upper limit, and your performance improvement is unlikely to be measurable after an *avg_fragment_size_in_page* of more than 32 pages (256 KB). The *fragment_count* and *avg_fragment_size_in_pages* values are reported only for in-row allocation units for leaf levels of indexes, and for heaps. No fragment size information is stored or reported for nonleaf levels of an index or for LOB or row-overflow allocation units; the function returns NULL in these cases.

Not all the values can be computed in every mode. The LIMITED mode scans all pages for a heap, but only the parent-level pages for an index, which are the pages above the leaf level. This means that certain values cannot be computed and returned in LIMITED mode. In particular, any of the values that require SQL Server to examine the contents of the leaf-level pages, such as *avg_page_space_used_in_percent*, are not available in LIMITED mode. The function returns NULL for that value in LIMITED mode. Other values returned by the `sys.dm_db_index_physical_stats` function also always report NULL in LIMITED mode, and their meanings are pretty self-explanatory. These include:

- *record_count*
- *min_record_size_in_bytes*
- *max_record_size_in_bytes*
- *avg_record_size_in_bytes*

You can consider using SAMPLED mode when the table is very large because a SAMPLED scan looks at only 1 percent of the pages. SQL Server basically just looks at the first of every 100 pages to get this sample. However, if the table is less than 10,000 pages in size, SQL Server considers it too small to worry about and automatically converts a request for SAMPLED into DETAILED and examines all the pages. If the table is a heap, a sampled scan will not be able to report any information about fragments or fragment size. Fragments can be analyzed only if SQL Server knows ALL the pages that belong to a table. For a table with a clustered index, the upper levels of the index give SQL Server the information it needs to determine the number of fragments and their average size, but for a heap, there are no additional structures to provide this information.

The `sys.dm_db_index_physical_stats` function returns two additional pieces of information that can indicate fragmentation in your structures.

- ***Forwarded_record_count*** Forwarded records (discussed earlier in the section titled "[Data Modification Internals](#)") are possible only in a heap, and occur when a row with variable-length columns increases in size due to updates so that it no longer fits in its original location. If a table has lots of forwarded records, scanning the table can be very inefficient.

- ***Ghost_Record_Count*** and ***version_ghost_record_count***

Ghost records are rows that physically still exist on a page but logically have been removed, as I discussed earlier in the section titled "[Data Modification Internals](#)." Background processes in SQL Server clean up ghost records, but until that happens, no new records can be inserted in their place. So if there are lots of ghost records, your table has the drawback of lots of internal fragmentation (that is, the table is spread out over more pages and takes longer to scan) with none of the advantages (there is no room on the pages to insert new rows to avoid external fragmentation). A subset of ghost records is measured by *version_ghost_record_count*. This value reports the number of rows that have been retained by an outstanding snapshot isolation transaction. These will not be cleaned up until all relevant transactions have been committed or rolled back. Snapshot isolation will be discussed in [Chapter 8](#).

Removing Fragmentation

If fragmentation becomes too severe, you have several options for removing it. You might also wonder how severe is too severe. First of all, fragmentation is not always a bad thing. The biggest performance penalty from having fragmented data arises when your application needs to perform an ordered scan on the data. The more the logical order differs from the physical order, the greater the cost of scanning the data. If, on the other hand, your application needs only one or a few rows of data, it doesn't matter whether the table or index data is in logical order or is physically contiguous, or whether it is spread all over the disk in totally random locations. If you have a good index to find the rows you are interested in, SQL Server can find one or a few rows very efficiently, wherever they happen to be physically located.

If you are doing ordered scans of an index (such as table scans on a table with a clustered index, or a leaf-level scan of a nonclustered index), it is frequently recommended that if your *avg_fragmentation_in_percent* value is between 5 and 20, you should reorganize your index to remove the fragmentation. As we'll see shortly, reorganizing an index (also called *defragging*) rearranges the pages in the leaf level of an index to correct the logical fragmentation, using the same pages that the index originally occupied. No new pages are allocated, so extent fragmentation and fragment size are unaffected.

If the *avg_fragmentation_in_percent* value is greater than 30, or if the *avg_fragment_size_in_pages* value is less than 8, you should consider completely rebuilding your index. Rebuilding an index means that a whole new set of pages will be allocated for the index. This will remove almost all fragmentation, but it is not guaranteed to completely eliminate it. If the free space in the database is itself fragmented, you might not be able to allocate enough contiguous space to remove all gaps between extents. In addition, if other work is going on that needs to allocate new extents while your index is being rebuilt, the extents allocated to the two processes can end up being interleaved.

Keep in mind that it is possible to have a good value for *avg_fragmentation_in_percent* and a bad value for *avg_fragment_size_in_pages* (for example, they could both be 2), or vice-versa (they could both be 40). A suboptimal value for *avg_fragmentation_in_percent* means that ordered scan performance will suffer, and a suboptimal value for *avg_fragment_size_in_pages* means that even unordered scans will not have ideal performance.

Defragmentation is designed to remove logical fragmentation from the leaf level of an index while keeping the index online and as available as possible. When defragmenting an index, SQL Server acquires an Intent-Share lock on the index B-tree. Exclusive page

locks are taken on individual pages only while those pages are being manipulated, as we'll see below when I describe the defragmentation algorithm. Defragmentation in SQL Server 2005 is initiated using the ALTER INDEX command. The general form of the command to remove fragmentation is as follows:

```
ALTER INDEX { index_name | ALL }
    ON <object>
    REORGANIZE
        [ PARTITION = partition_number ]
        [ WITH ( LOB_COMPACTION = { ON | OFF } )
    ) ]
```

ALTER INDEX with the REORGANIZE option offers enhanced functionality compared to DBCC INDEXDEFrag in SQL Server 2000. It supports partitioned indexes, so you can choose to defragment just one particular partition. The default is to defragment all the partitions. It is an error to specify a partition number if the index is not partitioned. Another new capability allows you to control whether the LOB data is affected by the defragmenting.

Defragmenting an index takes place in two phases. The first phase is a compaction phase, and the second is the actual rearranging of data pages to allow the logical and physical order of the pages to match.

As mentioned earlier, every index is created with a specific fillfactor. The initial fillfactor value is stored with the index metadata, so when defragmenting is requested, SQL Server can inspect this value. During defragmentation, SQL Server attempts to reestablish the initial fillfactor if it is greater than the current fillfactor. Defragmentation is designed to compact data, and this can be done by putting more rows per page and increasing the fullness percentage of each page. SQL Server might end up then removing

pages from the index after the defragmentation. If the current fillfactor is greater than the initial fillfactor, SQL Server must add more pages to reestablish the initial value, and this will not happen during defragmentation. The compaction algorithm inspects adjacent pages (in logical order) to see if there is room to move rows from the second page to the first. SQL Server 2005 makes this process even more efficient by looking at a sliding window of eight logically consecutive pages. It determines whether rows from any of the eight pages can be moved to other pages to completely empty a page.

As mentioned earlier, SQL Server 2005 also provides the option to compact your LOB pages. The default is ON. Reorganizing a specified clustered index compacts all LOB columns that are contained in the clustered index before it compacts the leaf pages. Reorganizing a nonclustered index compacts all LOB columns that are nonkey (included) columns in the index.

In SQL Server 2000, the only way a user can compact LOBs in a table is to unload and reload the LOB data. LOB compaction in SQL Server 2005 finds low-density extentsthose less than 75-percent utilized. It moves pages out of these low-density uniform extents and places the data from them in available space in other uniform extents already allocated to the LOB allocation unit. This functionality allows much better use of disk space, which can be wasted with low-density LOB extents. No new extents are allocated either during this compaction phase or during the next phase.

The second phase of the reorganization operation actually moves data to new pages in the in-row allocation unit with the goal of having the logical order of data match the physical order. The index is kept online because only two pages at a time are processed in an operation similar to a bubble sort. The following example is a simplification of the actual process of reorganization. Consider an index on a column of datetime data. Monday's data logically precedes Tuesday's data, which precedes Wednesday's data, which

precedes data from Thursday. If, however, Monday's data is on page 88, Tuesday's is on page 50, Wednesday's is on page 100, and Thursday's is on page 77, the physical and logical ordering doesn't match in the slightest, and we have logical fragmentation. When defragmenting an index, SQL Server determines the first physical page belonging to the leaf level (page 50 in our case) and the first logical page in the leaf level (page 88, which holds Monday's data) and swaps the data on those two pages using one additional new page as a temporary storage area. After this swap, the first logical page with Monday's data is on the lowest numbered physical page, page 50. After each page swap, all locks and latches are released and the key of the last page moved is saved. The next iteration of the algorithm uses the saved key to find the next logical page Tuesday's data, which is now on page 88. The next physical page is 77, which holds Thursday's data. So another swap is made to place Tuesday's data on page 77 and Thursday's on page 88. This process continues until no more swaps need to be made. Note that no defragmenting is done for pages on mixed extents.

You might wonder whether there is any harm in initiating a defragmentation operation if there is no or little existing fragmentation, because the amount of page swapping seems to be proportional to the number of out-of-order pages. In most cases, if there is no fragmentation, the defragmentation will do no swapping of pages. However, a pathological situation can occur in which a small amount of fragmentation requires an enormous amount of work to be done. Imagine an index of thousands of pages, all in order except for the last one in logical sequence. The first page of data in order (say, for January 1, 2000) is on page 101, the next (for January 2, 2000) is on page 102, the next is on 103, and so on. But the last logical page of data for January 31, 2006, is stored on page 100. With a 1000-page table, the fragmentation value reported would be only 0.1 percent because 1 out of 1000 pages is out of order. If we decided to reorganize this index even with such a low fragmentation percentage, the defragmentation algorithm would switch page 100 with 101, then switch 101 with 102, then switch

102 with 103, in an effort to have the data for the last date in the table be on the last page. We might have just one page out of order, but we have to do a thousand page swaps. I strongly suggest that you don't defragment your data unless you know the data is fragmented and you have determined that you need unfragmented data for maximum performance.

You need to be aware of some restrictions on using the REORGANIZE option. Certainly, if the index is disabled it cannot be defragmented. Also, because the process of removing fragmentation needs to work on individual pages, you will get an error if you try to reorganize an index that has the option ALLOW_PAGE_LOCKS set to OFF. Reorganization cannot happen if a concurrent online index is built on the same index or if another process is concurrently reorganizing the same index.

You can observe the progress of each index's reorganization in the *sys.dm_exec_requests* dynamic management view in the *complete_percentage* and *estimated_completion_time* columns. The value in the column *complete_percentage* reports the percentage completed in one index's reorganization; the value in the column *estimated_completion_time* shows the estimated time (in milliseconds) required to complete the remainder of the index reorganization. If you are reorganizing multiple indexes in the same command, you might see these values go up and down as each index is defragmented in turn.

Rebuilding an Index

You can completely rebuild an index in several ways. You can use a simple DROP INDEX and CREATE INDEX, but this method is probably the least preferable. In particular, if you are rebuilding a clustered index in this way, all the nonclustered indexes must be rebuilt when you drop the clustered index. This nonclustered index

rebuilding is necessary to change the row locators in the leaf level from the clustered key values to row IDs. Then, when you rebuild the clustered index, all the nonclustered indexes must be rebuilt again. In addition, if the index supports a PRIMARY KEY or UNIQUE constraint, you can't use the DROP INDEX command at all. Better solutions are to use the ALTER INDEX command or to use the DROP_EXISTING clause along with CREATE INDEX. As an example, here are both methods for rebuilding the *PK_TransactionHistory_TransactionID* index on the *Production.TransactionHistory* table:

```
ALTER INDEX PK_TransactionHistory_TransactionID  
    ON Production.TransactionHistory  
REBUILD;
```

```
CREATE UNIQUE CLUSTERED INDEX  
PK_TransactionHistory_TransactionID  
    ON Production.TransactionHistory  
        (TransactionDate,  
        TransactionID)  
    WITH DROP_EXISTING;
```

Although the CREATE method requires more typing, it is actually more powerful and offers more options that you can specify. You can change the columns that make up the index, change the uniqueness property, or change a nonclustered index to clustered, as long as there isn't already a clustered index on the table. You can also specify a new filegroup or a partition scheme to use when rebuilding. Note that if you do change the properties, all nonclustered indexes must be rebuilt, but only once (not twice, as would happen if we were to execute a DROP INDEX followed by a CREATE INDEX).

With the ALTER INDEX option, the nonclustered indexes never need to be rebuilt just as a side effect, because you can't change the index definition at all. However, you can specify ALL instead of an index name and request that all indexes be rebuilt. Another advantage of the ALTER INDEX method is that you can specify just a single partition to be rebuilt if, for example, the fragmentation report from `sys.dm_db_index_physical_stats` shows fragmentation in just one partition or a subset of the partitions.

Online Index Building

The default behavior of either method of rebuilding an index is that SQL Server takes an exclusive lock on the index, so it is completely unavailable while the index is being rebuilt. If the index is clustered, the entire table is unavailable; if the index is nonclustered, there is a shared lock on the table, which means no modifications can be made but other processes can SELECT from the table. Of course, they cannot take advantage of the index you're rebuilding, so the query might not perform as well as it should.

SQL Server 2005 provides the option to rebuild one or all indexes online. The ONLINE option is available with both ALTER INDEX and CREATE INDEX, with or without also using the DROP_EXISTING option. Here's the syntax for building the preceding index, but doing it online:

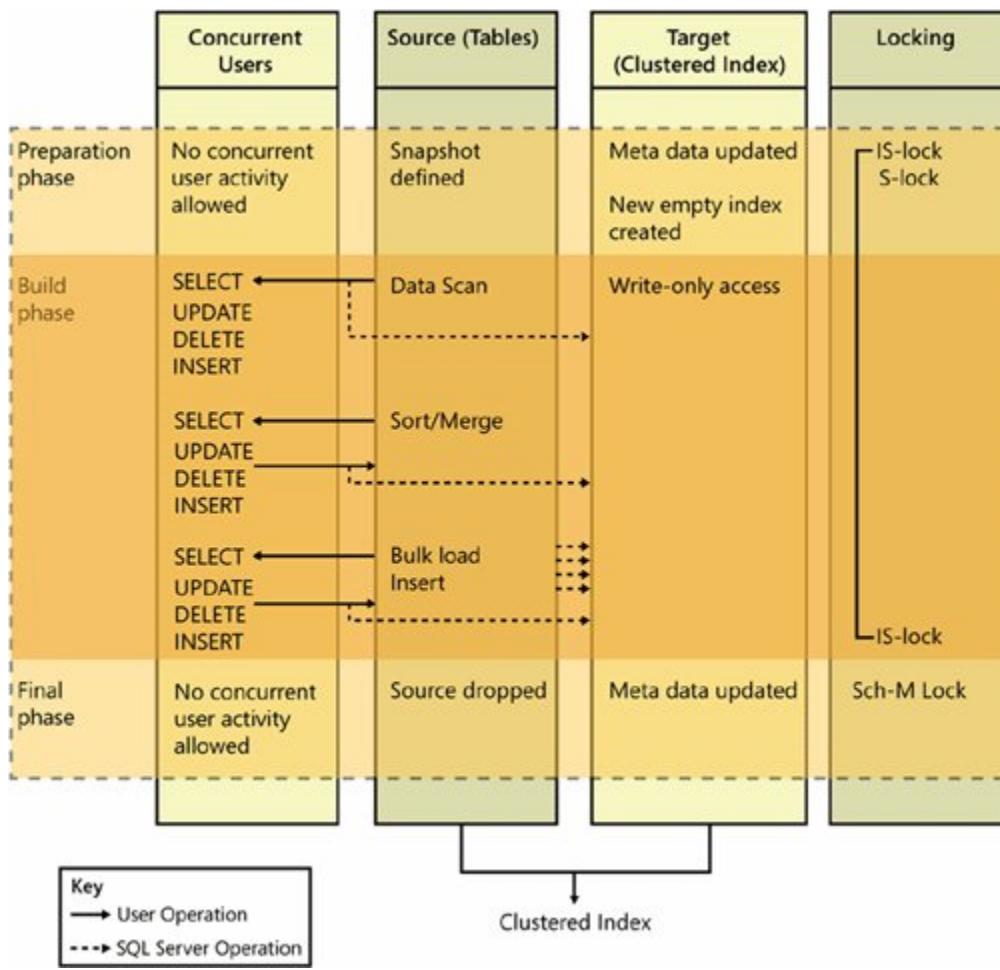
```
ALTER INDEX PK_TransactionHistory_TransactionID
    ON Production.TransactionHistory
REBUILD WITH (ONLINE = ON);
```

The online build works by maintaining two copies of the index simultaneously, the original (source) and the new one (target). The target is used only for writing any changes made while the rebuild is

going on. All reading is done from the source, and modifications are applied to the source as well. SQL Server row-level versioning is used so anyone retrieving information from the index will be able to read consistent data. [Figure 7-10](#) (taken from SQL Server 2005 Books Online) illustrates the source and target, and it shows three phases that the build process goes through. For each phase, the figure describes what kind of access is allowed, what is happening in the source and target tables, and what locks are applied.

Figure 7-10. The structures and phases of online index building

[[View full size image](#)]



The actual processes might differ slightly depending on whether the index is being built initially or being rebuilt, and whether the index is clustered or nonclustered. Following are details about what SQL Server does after the index create or rebuild command is issued.

Here are the steps involved in rebuilding a nonclustered index:

1. A shared lock is taken on the index, which prevents any data modification queries, and an Intent-Shared lock is taken on the table.
2. The index is created with the same structures as the original and marked as write-only.
3. The Shared lock is released on the index, leaving only the Intent-Shared lock on the table.
4. A versioned scan (discussed in detail in [Chapter 8](#)) is started on the original index, which means modifications made during the scan will be ignored. The scanned data is copied to the target.
5. All subsequent modifications will write to both the source and the target. Reads will use only the source.
6. The scan of the source and copy to the target continues while normal operations are performed. SQL Server uses a proprietary method for reconciling obvious problems such as a record being deleted before the scan has inserted it into the new index.
7. The scan completes.
8. A Schema-Modification lock, the strictest of all types of locks, is taken to make the source completely unavailable.
9. The source is dropped, metadata is updated, and the target is made to be read-write.
10. The Schema-Modification lock is released.

Building a new nonclustered index involves exactly the same steps, except there is no target index so the versioned scan is done on the base table. Write operations need to maintain only the target index, rather than both indexes.

A clustered index rebuild works exactly like a nonclustered rebuild, as long as there is no schema change (a change of index keys or uniqueness property).

For a build of new clustered index, or a rebuild of a clustered index with a schema change, there are a few more differences. First, an intermediate mapping index is used to translate between the source and target physical structures. Additionally, all existing nonclustered indexes are rebuilt one at a time after a new base table has been built. For example, creating a clustered index on a heap with two nonclustered indexes involves the following steps:

- 1.** Create a new write-only clustered index.
- 2.** Create a new nonclustered index based on the new clustered index.
- 3.** Create another new nonclustered index based on the new clustered index.
- 4.** Drop the heap and the two original nonclustered indexes.

Before the operation is completed, SQL Server will be maintaining six structures at once. Online index building is not really considered to be a performance enhancement because an index can actually be built faster offline, and all these structures do not need to be maintained simultaneously. Online index building is an availability feature you can rebuild indexes to remove all fragmentation or re-establish a fillfactor even if your data must be fully available 24/7.

Using Indexes

This chapter has primarily been concerned with the structure of indexes and the nuances of creating and managing them. Hopefully, you're aware of at least some of the places where indexes can be useful in your SQL Server applications. I will list a few of the situations in which you can benefit greatly from indexes; the details of how SQL Server decides which indexes to use, and how you can decide which indexes are the best to build for your queries, are explained in *Inside Microsoft SQL Server 2005: Tuning and Optimization*.

Looking for Rows

The most straightforward use of an index is to help SQL Server find one or more rows in a table that satisfy a certain condition. For example, if you're looking for all the customers who live in a particular state (for example, your WHERE clause is WHERE state = 'WI'), you can use an index on the state column to find those rows more quickly. SQL Server can traverse the index from the root, comparing index key values to 'WI' to determine whether 'WI' exists in the table and then find the data associated with that value.

Joining

A typical join tries to find all the rows in one table that match rows in another table. Of course, you can have joins that aren't looking for exact matching, but you're looking for some sort of relationship between tables, and equality is by far the most common relationship. A query plan for a join frequently starts with one of the tables, finds the rows that match the search conditions, and then

uses the join key in the qualifying rows to find matches in the other table. An index on the join column in the second table can be used to quickly find the rows that match.

Sorting

A clustered index stores the data logically in sorted order. The data pages are linked together in order of the clustering keys. If you have a query to ORDER BY the clustered keys or by the first column of a composite clustered key, SQL Server does not have to perform a sort operation to return the data in sorted order it can just follow the page linkage in the data pages. There are also cases where SQL Server can use the ordering that exists in the leaf level of nonclustered index to avoid having to actually perform a sort operation.

Grouping

One way that SQL Server can perform a GROUP BY operation is by first sorting the data by the grouping column. For example, if you want to find out how many customers live in each state, you can write a query with a GROUP BY state clause. A clustered index on state will have all the rows with the same value for state in logical sequence, so the grouping operations can be done very quickly.

Maintaining Uniqueness

Creating a unique index (or defining a PRIMARY KEY or UNIQUE constraint that builds a unique index) is by far the most efficient method of guaranteeing that no duplicate values are entered into a column. By traversing an index tree to determine where a new row

should go, SQL Server can detect within a few page reads that a row already has that value. Unlike all the other uses of indexes described in this section, using unique indexes to maintain uniqueness isn't just one option among others. Although SQL Server might not always traverse a particular unique index to determine where to try to insert a new row, it will always use the existence of a unique index to verify whether a new set of data is acceptable.

Summary

In this chapter, I showed you how SQL Server indexes organize the data on disk and help you access your data more quickly than if no indexes existed. Indexes are organized as B-trees, which means you will always traverse through the same number of index levels when you traverse from the root to any leaf page. To use an index to find a single row of data, SQL Server never has to read more pages than there are levels in an appropriate index.

You also learned about all the options available when you create an index, how to determine the amount of space an index takes up, and how to predict the size of an index that doesn't have any data yet.

We looked at the internal structure of various kinds of index structures, and we looked at the metadata for tables and indexes on a single partition and on multiple partitions. I described what happens to these structures as data modifications operations are carried out.

Indexes can become fragmented as many data modifications are applied. When you want to defragment your indexes, you have several methods to choose from, some of which allow the index to continue to be used by other operations while the defragmentation operation is going on.

Chapter 8. Locking and Concurrency

In this chapter:

<u>Concurrency Models</u>	<u>332</u>
<u>Transaction Processing</u>	<u>332</u>
<u>Locking</u>	<u>340</u>
<u>Lock Compatibility</u>	<u>364</u>
<u>Internal Locking Architecture</u>	<u>365</u>
<u>Bound Connections</u>	<u>372</u>
<u>Row-Level Locking vs. Page-Level Locking</u>	<u>374</u>
<u>Row Versioning</u>	<u>381</u>
<u>Other Features That Use Row Versioning</u>	<u>404</u>
<u>Controlling Locking</u>	<u>407</u>

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'll 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'll use the terms *modifying*, *changing*, or *writing* to describe the impact of using the INSERT, UPDATE, or DELETE statements on your data.

In general, database systems can take two approaches to managing concurrent data access: optimistic or pessimistic. Microsoft SQL Server 2005 supports both approaches. Pessimistic concurrency was the only concurrency model available before SQL Server 2005. In 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'll look at the five possible isolation levels in SQL Server 2005, as well as the internals of how SQL Server controls concurrent access using each model. We'll look at how to control the isolation level, and we'll look at the metadata that shows you what SQL Server is

doing. As you'll see, optimistic concurrency is supported by the use of a new technology called *row versioning*; at the end of this chapter, I'll tell you about additional SQL Server 2005 features that use row versioning.

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 will likely be 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 as long as the transaction that performed the write operation is active, no other process can read the modified data.

The mechanics of transaction control, along with discussions of the ANSI transaction properties and transaction isolation levels, can be found in *Inside Microsoft SQL Server 2005: T-SQL Programming*. I will go into detail about the transaction isolation levels, and the types of locks in the various isolation levels, which will overlap somewhat with the T-SQL programming volume. I will not, however, address the programming aspects of working with transactions in this volume, other than to describe the difference between an implicit and an explicit transaction.

An *implicit transaction* is any individual INSERT, UPDATE, or DELETE statement. You can also consider SELECT statements to be implicit transactions, although there are no records for SELECT statements written to the transaction log. No matter how many rows are affected, the statement must exhibit all the ACID properties of a transaction, which I'll 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. Most of the examples I present use explicit transactions because it is the only way to show SQL Server state 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*: atomicity, consistency, isolation, and durability. In addition to guaranteeing that explicit multi-statement 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 will essentially disappear from the database; that is, funds will be 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 will mysteriously increase in value without a corresponding customer cash deposit or account transfer. Because of SQL Server's atomicity feature, 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 will be 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, SQL Server's durability property 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 midst of a transaction before the transaction is committed, the entire transaction is rolled back to when the system was 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.

Transactions always support all four of the ACID properties. A transaction might exhibit several additional behaviors as well. 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 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** This behavior occurs 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 completely overwrite the first update. For example, suppose that two clerks in a receiving room are receiving parts and adding the new shipments to the inventory database. ClerkA and ClerkB both receive shipments of widgets. They both check the current inventory and see that 25 widgets are currently in stock. ClerkA's shipment has 50 widgets, so he adds 50 to 25 and updates the current value to 75. ClerkB'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 ClerkA processed. ClerkA's update is lost.

Lost updates is the only one of these behaviors that you will probably want to avoid in all cases.

- **Dirty reads** This behavior occurs 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 ClerkA 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 ClerkA then realizes that the widgets are defective and sends them back to the manufacturer, the salesperson will have 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 it's 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.

- **Non-repeatable reads** This behavior is also called *inconsistent analysis*. A read is non-repeatable if a process might get different values when reading the same resource 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 that day, 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 ClerkA received more widgets between the manager's first and second inquiries, the total will be different each time and the reads are non-repeatable.
- **Phantoms** This behavior occurs 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 walk-through a clerk with fewer than 10 widgets returned from a break but was not spotted by the manager, that clerk will not be 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 four 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 2005 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'll 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'll mention the locks or row versions that support that level, but keep in mind that locking and row versioning will be discussed in detail later in the chapter.

Uncommitted Read

In Uncommitted Read isolation, all the behaviors described previously except lost updates are possible. Your queries can read uncommitted data, and both non-repeatable reads and phantoms are possible. Uncommitted read 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 will be able to read data

that another process has modified but not yet committed. Although this scenario isn't usually the ideal option, with Uncommitted Read 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 Uncommitted Read, 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 Uncommitted Read? 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 trade-off 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 2005 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 will be referring to both variations of this isolation level. I'll refer to the pessimistic implementation as Read Committed (locking), and I'll 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 will be 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 will not have changed. In other words, issuing the same query twice within a transaction will not pick up any changes to data values made by another user's transaction. However, the Repeatable Read isolation level does allow phantom rows to appear.

Preventing non-repeatable 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 "hang" 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 whole. You can read about `LOCK_TIMEOUT` in SQL Server Books Online.

Snapshot

Snapshot isolation 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'll see the details in the section on row versioning.) 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 8-1](#) shows an example of two simultaneous transactions. If they run in parallel, they will 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 8-1. Two Simultaneous Transactions That Cannot Be Run Serially

Time	Transaction 1	Transaction 2

Time Transaction 1

1

```
USE pubs  
DECLARE @price money  
BEGIN TRAN
```

Transaction 2

2

```
SELECT @price =  
    price  
FROM titles  
WHERE title_id =  
    'BU1032'
```

```
SELECT @price =  
    price  
FROM titles  
WHERE title_id =  
    'PS7777'
```

Time	Transaction 1	Transaction 2
3	<pre>UPDATE titles SET price = @price WHERE title_id = 'PS7777'</pre>	<pre>UPDATE titles SET price = @price WHERE title_id = 'BU1032'</pre>
3	<pre>COMMIT TRAN</pre>	<pre>COMMIT TRAN</pre>

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 will not have been added in the interim. In other words, phantoms will 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, non-repeatable 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 completion of the transaction. 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 will still be 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 Code.) If there is no index on that column, Serializable isolation requires a table lock. I'll 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. In the upcoming discussion of the Snapshot isolation level, I'll show you a case where transactions are not serializable.

More Info



For an interesting critique of the formal definitions of the ANSI isolation levels, see the book's companion content, which contains a technical report called "A Critique of ANSI SQL Isolation Levels" (published by the Microsoft Research Center).

[Table 8-2](#) 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 8-2. Behaviors Allowed in Each Isolation Level

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom	Concurrency Control
Read Uncommitted	Yes	Yes	Yes	Pessimistic
Read Committed	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 multi-user 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 will always block readers and writers, and readers can block writers. In the optimistic model, the only blocking that occurs is that writers will 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 datadata 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, 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 DMV `sys.dm_os_tasks` shows a status of SPINLOOP for any task that is currently using a spinlock.

Lock Types for User Data

We'll examine four aspects of locking user data. First we'll look at the mode of locking (the type of lock). I already mentioned shared, exclusive, and update locks, and I'll go into more detail about these modes as well as others. Next we'll 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, 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. We'll see a chart illustrating the lock compatibility matrix at the end of this section.

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'll see when we discuss lock compatibility, 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 needs to search the table to find the resource that will 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 8-6](#) 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.) As 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 escalate 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'll 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 can set the table option called *table lock on bulk load* to true, and then any bulk copy IN or BULK INSERT operation will automatically request a bulk update lock. Requesting this special bulk update table lock does not necessarily mean it will be 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 2005 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 are operating at the Repeatable Read transaction isolation level and 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 will acquire a shared table lock. (If the table has only a few rows, SQL Server will acquire individual row or key locks.) The UPDATE statement will then acquire a single exclusive key lock to perform the update of a single row, and the X lock at the key level will mean an IX lock at the page and table level. The table will then show SIX when viewed through *sys.dm_tran_locks*. Similarly, SIU will occur when a process has a shared lock on a table and an update lock on a row of that table, and UIX will occur when a process has an update lock on the table and an exclusive lock on a row.

[Table 8-3](#) shows most of the lock modes, as well as the abbreviations used in *sys.dm_tran_locks*.

Table 8-3. 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 (unless the process is set to the Read Uncommitted isolation level).
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.
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.

Abbreviation	Lock Mode	Description
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

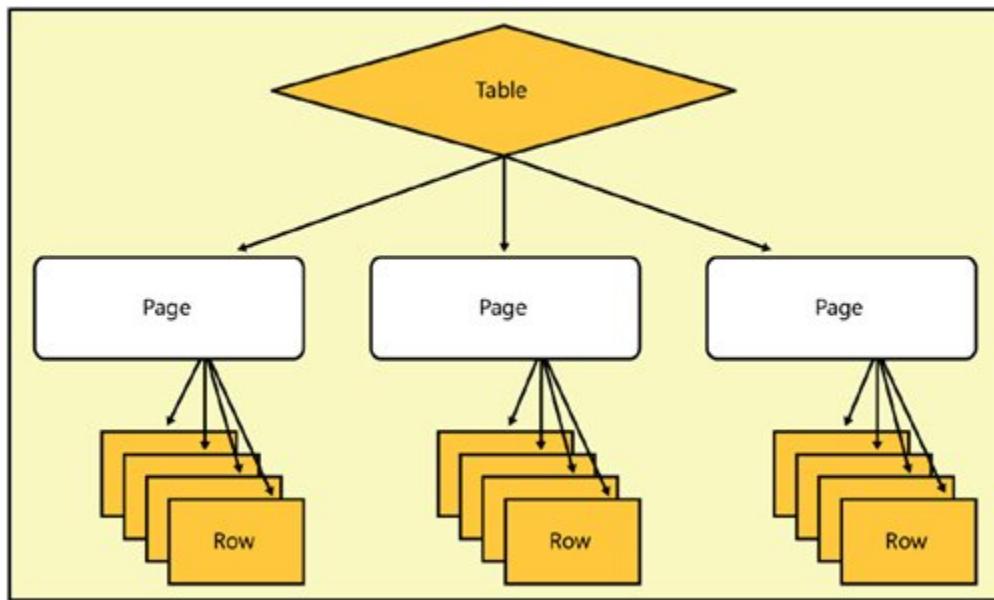
Additional lock modes called key-range locks are taken only in the Serializable isolation level for locking ranges of data. Most lock modes can apply to almost any lock resource. For example, shared and exclusive locks can be taken on a table, a page, a row or a key. Because key-range locks can only be taken on keys, I'll describe the details of key-range locks in the upcoming 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. It also locks index keys and ranges of index keys. [Figure 8-1](#) shows the possible lock levels in a table. 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.

Figure 8-1. Levels of granularity for SQL Server locks on a table

[[View full size image](#)]



The `sys.dm_tran_locks` view keeps track of each lock and contains the resource 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.

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 Adventureworks;
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, any 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'll give you more details about the data in the *sys.dm_tran_locks* view later in this chapter, but for now, I'll point out 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:

[[View full width](#)]

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 an *allocation_unit_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 *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 will be blocked. The *sys.dm_tran_locks* view would show 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 will 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'll discuss lock compatibility in detail later in this chapter.

Key Locks

SQL Server 2005 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 will see key locks acquired. If the table is a heap, you might see key locks for the non-clustered 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 AdventureWorks 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 2005 uses a separate lock mode, called *key-range locks*, 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 AdventureWorks database includes an index on the *LastName* column in the *Person.Contact* table. Assume we are in TRANSACTION ISOLATION LEVEL SERIALIZABLE and we issue this SELECT statement:

```
SELECT * FROM Person.Contact  
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 8-4](#).

Table 8-4. Types of Key-Range Locks

Abbreviation Description

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
Rangeln-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
Rangeln-S	Conversion lock created by S and Rangeln_Null lock
Rangeln-U	Conversion lock created by U and Rangeln_Null lock
Rangeln-X	Conversion of X and Rangeln_Null lock
RangeX-S	Conversion of Rangeln_Null and RangeS_S lock
RangeX-U	Conversion of Rangeln_Null and RangeS_U lock

Many of these lock modes are very rare or transient, so you not often see them 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 escalated 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 will have 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'll 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 kilobytes (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` will have 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 will request an exclusive database lock, and if any other processes have a shared lock on the database, the request will block. Generally, you don't need to be concerned with extent or database locks, but you'll see them if you are perusing *sys.dm_tran_locks*.

You might occasionally see locks on HOBT and ALLOCATION_UNIT resources. Although all table and index structures are based on HOBTs and 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 done just as in versions prior to SQL Server 2005, and no deferred operation takes place. 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 will see ALLOCATION_UNIT locks in *sys.dm_tran_locks*. You can also look in *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 will occur after the transaction commits.

Application Locks

The method used by SQL Server to store information about locking and to check for incompatible locks is very straightforward and extensible. SQL Server knows nothing about the object it is locking. It works only with strings representing the resources, without knowing the actual structure of the item. If two processes are trying to obtain incompatible locks on the same resource, blocking will occur.

Application locks allow you to take advantage of the supplied mechanisms for detecting blocking and deadlocking situations, and you can choose to lock anything you like. These lock resources are called *application locks*. To define an application lock, you specify a name for the resource you are locking, a mode, an owner (or scope) of the lock, a timeout, and a *Database Principal ID* (which is a user, role, or application role that can have permissions in a database). Unlike SQL Server's own lock resources, such as tables and pages, application locks must be specifically requested. Only users meeting one of the following criteria can execute the `sp_getapplock` procedure:

- User name is *dbo*
- User name is in the *db_owner* role
- User name is the specified Database Principal ID (if the ID is an individual name)
- User name is in the specified Database Principal ID role (if the ID is a role)

The default Database Principal ID, when calling `sp_getapplock`, is *public*.

Note



When the term is used with application locks, a lock owner has nothing to do with the user requesting or holding the lock. Instead, you can think of it as the scope of the lock. We'll discuss lock owners shortly.

If two resources have the same name in the same database and have the same Database Principal ID, they are considered to be the same resource and are subject to blocking. For application locks, the *lock owner* can be either the session or the transaction. Two requests for locks on the same resource can be granted if the modes of the locks requested are compatible. The locks are checked for compatibility using the same compatibility matrix used for SQL Serversupplied locks.

For example, suppose that you have a stored procedure that only one user at a time should execute. Anyone in the *ProcUserRole* database role can lock that procedure by using the *sp_getapplock* procedure to acquire a special lock, which indicates to other processes that someone is using this procedure. When the procedure is complete, you can use *sp_releaseapplock* to release the lock:

```
EXEC sp_getapplock 'ProcLock', 'Exclusive',
'session', 'ProcUserRole'
EXEC MySpecialProc <parameter list>
EXEC sp_releaseapplock 'ProcLock', 'session'
```

Until the lock is released using `sp_releaseapplock`, or until the session terminates, no other session can execute this procedure if it follows this protocol and uses `sp_getapplock` to request rights on the resource called *ProcLock* before trying to execute the procedure. SQL Server doesn't know what the resource *ProcLock* means. You can use any identifier you choose. SQL Server just adds a row to the `sys.dm_tran_locks` view when an application lock is requested, and it uses the resource name and Database Principal ID to compare against other requested locks. Note that the procedure itself is not really locked. If another user or application doesn't know that this is a special procedure and tries to execute *MySpecialProc* without acquiring the application lock, SQL Server will not prevent the session from executing the procedure.

The resource name used in these procedures can be any identifier up to 255 characters long; however, only the first 32 characters will be shown in the `resource_description` string in `sys.dm_tran_locks`. The possible modes of the lock, which is used to check compatibility with other requests for this same resource, are Shared, Update, Exclusive, IntentExclusive, and Intent-Shared. There is no default; you must specify a mode. The possible values for lock owner, the third parameter, are `transaction` (the default) or `session`. A lock with an owner of `transaction` must be acquired with a user-defined transaction, and it is automatically released at the end of the transaction without any need to call `sp_releaseapplock`. A lock with an owner of `session` is released automatically only when the session disconnects.

Here's an example. Let's request the lock shown previously and then look at the `sys.dm_tran_locks` view:

```
EXEC sp_getapplock 'ProcLock', 'Exclusive',  
'session';  
GO
```

```
SELECT resource_type, resource_description,
       resource_associated_entity_id,
       request_mode, request_status
  FROM sys.dm_tran_locks
 WHERE resource_type = 'APPLICATION'
```

Here are my results:

[\[View full width\]](#)

resource_type	resource_description	resource_associated_entity_id	request_mode	request_status
APPLICATION	0:[ProcLock]:(8e14701f)	0		
X			GRANT	

Note that no database entity is associated with this lock because SQL Server does not connect it to any resource actually in the database.

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 8-5](#) 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 8-5. Lockable Resources in SQL Server

Resource_Type	Resource_Description	Example
DATABASE	None; the database is always indicated in the <i>resource_database_ID</i> column for every locked resource.	
OBJECT	The object ID (which can be any database object, not necessarily a table) is reported in the <i>resource_associated_entity_id</i> column.	69575286

Resource_Type	Resource_Description	Example
EXTENT	<i>File number:page number of the first page of the extent.</i>	1:96
PAGE	<i>File number:page number of the actual table or index page.</i>	1:104
KEY	A hashed value derived from all the key components and the locator. For a non-clustered index on a heap, where columns <i>c1</i> and <i>c2</i> are indexed, the hash will contain contributions from <i>c1</i> , <i>c2</i> , and the RID.	ac0001a10a00
ROW	<i>File number:page number:slot number of the actual row.</i>	1:161:3
APPLICATION	A concatenation of the database principal with access to this lock, the first 32 characters of the name given to the lock, and a hashed value derived from the full name given to the lock.	0:[ProcLock]: (8e14701f)

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'll see that you can distinguish between these types of locks by the value in the lock mode column.

Other possible resources listed in the *resource_type* column are HOBT and ALLOCATION_UNIT, which I mentioned earlier. These can be identified by the value in the *resource_associated_entity_id* column, which I'll discuss in the next section. A final 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 will be *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 HoBT 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

HoBT ID is provided for resource types PAGE, KEY, RID, and HOBT.

There is no simple function to convert a HoBT 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 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 *object_name* function applies only to the current database, this code is filtered to only return 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'll 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.hobt_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'll discuss query hints for locking briefly later in this chapter. A full discussion of hints is in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

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 *req_ownertype* value of CURSOR must be explicitly requested 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 2005, locks owned by a session must also be explicitly requested 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 explicitly released.

SQL Server 2000 makes much heavier use of session-owned locks; in SQL Server 2005, transaction_workspace locks are used instead. 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 and bound session (discussed later in this chapter), 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'll 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 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 requesting process. 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 made only brief reference 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 8-6](#) lists all the subtypes for resource types other than METADATA.

Table 8-6. Subtype Resources

Resource Type	Resource Subtypes	Description
DATABASE	BULKOP_BACKUP_DB	Used for synchronization of database backups with bulk operations.

Resource Type	Resource Subtypes	Description
	BULKOP_BACKUP_LOG	Used for synchronization of database log backups with bulk operations.
	DDL	Used to synchronize 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.

Resource Type	Resource Subtypes	Description
	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 Snapshot isolation levels.
	ALLOCATION_UNITPAGE_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 2005 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 2005, the only type of resource request tracked in *sys.dm_tran_locks* is for a LOCK. Future versions will include other types of resources that can be requested.
- ***request_status*** Status can be one of three values: GRANTED, 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 will be 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 (DTC) 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 sub-thread 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 I described previously.

Example 1: SELECT with Default Isolation Level

SQL BATCH

```
USE AdventureWorks;
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	AdventureWorks	n/a		NULL
DATABASE		S	GRANT	
60	AdventureWorks	sysrowsets		NULL
OBJECT		Sch-S	GRANT	
60	AdventureWorks	DBlocks		NULL
OBJECT		IS	GRANT	

There are no locks on the data in the *Production.Product* table because the batch was doing 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, an OBJECT lock on the view's schema, and an OBJECT lock on the rowset.

Example 2: SELECT with Repeatable Read Isolation Level

SQL BATCH

```
USE AdventureWorks;
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	AdventureWorks	Product	NULL				

OBJECT		IS	GRANT
54 AdventureWorks	Product	1	
PAGE 1:16897		IS	GRANT
54 AdventureWorks (6b00b8eeda30)	Product S	1 GRANT	KEY
(6a00dd896688)	Product S	1 GRANT	KEY
54 AdventureWorks (9502d56a217e)	Product S	3 GRANT	KEY
54 AdventureWorks PAGE 1:1767	Product	3 IS	GRANT
54 AdventureWorks (9602945b3a67)	Product S	3 GRANT	KEY

This time, I filtered out the database lock and the locks on the view and the rowset, just to keep 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 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.) 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 AdventureWorks;
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	AdventureWorks	Product	NULL	OBJECT		IS	GRANT
54	AdventureWorks	Product	1	1:16897	IS	GRANT	PAGE
54	AdventureWorks	Product	1	(6b00b8eeda30)	S	GRANT	KEY
54	AdventureWorks	Product	1	(6a00dd896688)	S	GRANT	KEY
54	AdventureWorks	Product	3	(9502d56a217e)	Ranges-S	GRANT	KEY
54	AdventureWorks	Product	3	1:1767	IS	GRANT	PAGE
54	AdventureWorks	Product	3	(23027a50f6db)	Ranges-S	GRANT	KEY
54	AdventureWorks	Product	3	(9602945b3a67)	Ranges-S	GRANT	KEY

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* (*IndId* = 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 between *Racing Socks* and the previous key, *Pinch Bolt*, and nothing between *Racing Socks* and the next key, *Rear Brakes*, could be inserted. For example, we could not insert a product with the name *Portkey* or *Racing Tights*.)

Example 4: Update Operations

SQL BATCH

```
USE AdventureWorks;
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	AdventureWorks	Product	NULL	
OBJECT			IX	GRANT
54	AdventureWorks	Product	1	PAGE
1:16897		IX	GRANT	
54	AdventureWorks	Product	1	KEY
(6b00b8eeda30)		X	GRANT	
54	AdventureWorks	Product	1	KEY
(6a00dd896688)		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 escalated to X locks when the actual update is done, 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 AdventureWorks;
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	AdventureWorks	Product	NULL	OBJECT		IX	GRANT
54	AdventureWorks	Product	1	PAGE	1:16897	IX	GRANT
54	AdventureWorks	Product	1	(6a00dd896688)	X	GRANT	KEY
54	AdventureWorks	Product	1	(6b00b8eeda30)	X	GRANT	KEY
54	AdventureWorks	Product	3	(9502d56a217e)	RangeS-U	GRANT	KEY
54	AdventureWorks	Product	3				

PAGE	1:1767	IU	GRANT
54	AdventureWorks	Product	3
(23027a50f6db)		RangeS-U	GRANT
54	AdventureWorks	Product	3
(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 share 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 (*IndId* = 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 Level Not Using an Index

SQL BATCH

```
USE AdventureWorks;
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	AdventureWorks	Product	NULL	
OBJECT		IX	GRANT	
54	AdventureWorks	Product	1	KEY
(7900ac71caca)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	KEY
(6100dc0e675f)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	KEY
(5700a1a9278a)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16898		IU	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16899		IU	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16896		IU	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16897		IX	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16900		IU	GRANT	
54	AdventureWorks	Product	1	PAGE
1:16901		IU	GRANT	
54	AdventureWorks	Product	1	KEY
(5600c4ce9b32)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	KEY
(7300c89177a5)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	KEY
(7f00702ea1ef)		RangeS-U	GRANT	
54	AdventureWorks	Product	1	KEY

(6b00b8eeda30)	RangeX-X	GRANT		
54 AdventureWorks Product	1		KEY	
(c500b9eaac9c)	RangeX-X	GRANT		
54 AdventureWorks Product	1		KEY	
(c6005745198e)	RangeX-X	GRANT		
54 AdventureWorks 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 (*IndId* = 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 escalated to the RangeX-X lock. You can see all the RangeX-X locks, but not all the RangeS-U locks are shown because there are 504 rows in the table.

Example 7: Creating a Table

SQL BATCH

```
USE AdventureWorks;
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	Adventureworks	n/a	NULL	
DATABASE		NULL	GRANT	
54	Adventureworks	n/a	NULL	
DATABASE		NULL	GRANT	
54	Adventureworks	n/a	NULL	
DATABASE		S	GRANT	
54	Adventureworks	n/a	NULL	
METADATA	user_type_id = 258	Sch-S	GRANT	
54	Adventureworks	n/a	NULL	
METADATA	data_space_id = 1	Sch-S	GRANT	
54	Adventureworks	n/a	NULL	
DATABASE		S	GRANT	
54	Adventureworks	n/a	NULL	
METADATA	\$seq_type = 0, objec	Sch-M	GRANT	
54	Adventureworks	n/a	NULL	
METADATA	user_type_id = 260	Sch-S	GRANT	
54	Adventureworks	sysrowsetcol	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	sysrowsets	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	sysallocunit	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	syshobtcolumn	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	syshobts	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	sysserefs	NULL	
OBJECT		IX	GRANT	
54	Adventureworks	sysschobjs	NULL	
OBJECT		IX	GRANT	

54	AdventureWorks	syscolpars	NULL	
OBJECT			IX	GRANT
54	AdventureWorks	sysidxstats	NULL	
OBJECT			IX	GRANT
54	AdventureWorks	sysrowsetcol 1		KEY
(15004f6b3486)		X	GRANT	
54	AdventureWorks	sysrowsetcol 1		KEY
(0a00862c4e8e)		X	GRANT	
54	AdventureWorks	sysrowsets	1	
(000000aaec7b)		X	GRANT	KEY
54	AdventureWorks	sysallocunit	1	
(00001f2dcf47)		X	GRANT	KEY
54	AdventureWorks	syshobtcolum	1	
(1900f7d4e2cc)		X	GRANT	KEY
54	AdventureWorks	syshobts	1	
(000000aaec7b)		X	GRANT	KEY
54	AdventureWorks	NULL	NULL	RID
1:6707:1		X	GRANT	
54	AdventureWorks	DBlocks	NULL	
OBJECT			IS	GRANT
54	AdventureWorks	newProducts	NULL	
OBJECT			Sch-M	GRANT
54	AdventureWorks	sysserefs	1	
(010025fabf73)		X	GRANT	KEY
54	AdventureWorks	sysschobjs	1	
(3b0042322c99)		X	GRANT	KEY
54	AdventureWorks	syscolpars	1	
(4200c1eb801c)		X	GRANT	KEY
54	AdventureWorks	syscolpars	1	
(4e00092bfbc3)		X	GRANT	KEY
54	AdventureWorks	sysidxstats	1	
(3b0006e110a6)		X	GRANT	KEY
54	AdventureWorks	sysschobjs	2	
(9202706f3e6c)		X	GRANT	KEY
54	AdventureWorks	syscolpars	2	
(6c0151be80af)		X	GRANT	KEY

54	AdventureWorks	syscolpars	2	KEY
	(2c03557a0b9d)	X	GRANT	
54	AdventureWorks	sysidxstats	2	KEY
	(3c00f3332a43)	X	GRANT	
54	AdventureWorks	sysschobjs	3	KEY
	(9202d42ddd4d)	X	GRANT	
54	AdventureWorks	sysschobjs	4	KEY
	(3c0040d00163)	X	GRANT	
54	AdventureWorks	newProducts	0	PAGE
1:6707		X	GRANT	
54	AdventureWorks	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. Also notice the schema modification (Sch-M) lock and other metadata locks on the new table.

The final example will look 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 AdventureWorks;
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	AdventureWorks	newProducts	NULL	
	OBJECT		IX	GRANT
54	AdventureWorks	newProducts	0	PAGE
1:6708		IX	GRANT	
54	AdventureWorks	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 criterion 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 8-2](#) summarizes the compatibility of locks in SQL Server 2005. 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.

Figure 8-2. SQL Server lock compatibility matrix

[[View full size image](#)]

	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	C	N	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	C	N	N	C	C
U	N	N	C	N	C	N	C	N	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	N	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	N	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	C	N	C	N	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	N	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	N	C	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	I	C	C	N	N	N	N	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C

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 / indicates an invalid combination that could never occur. All the / 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.

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 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 can each hold exclusive page locks simultaneously as 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 outstanding to process A. However, the shared page lock cannot be immediately granted. 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 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

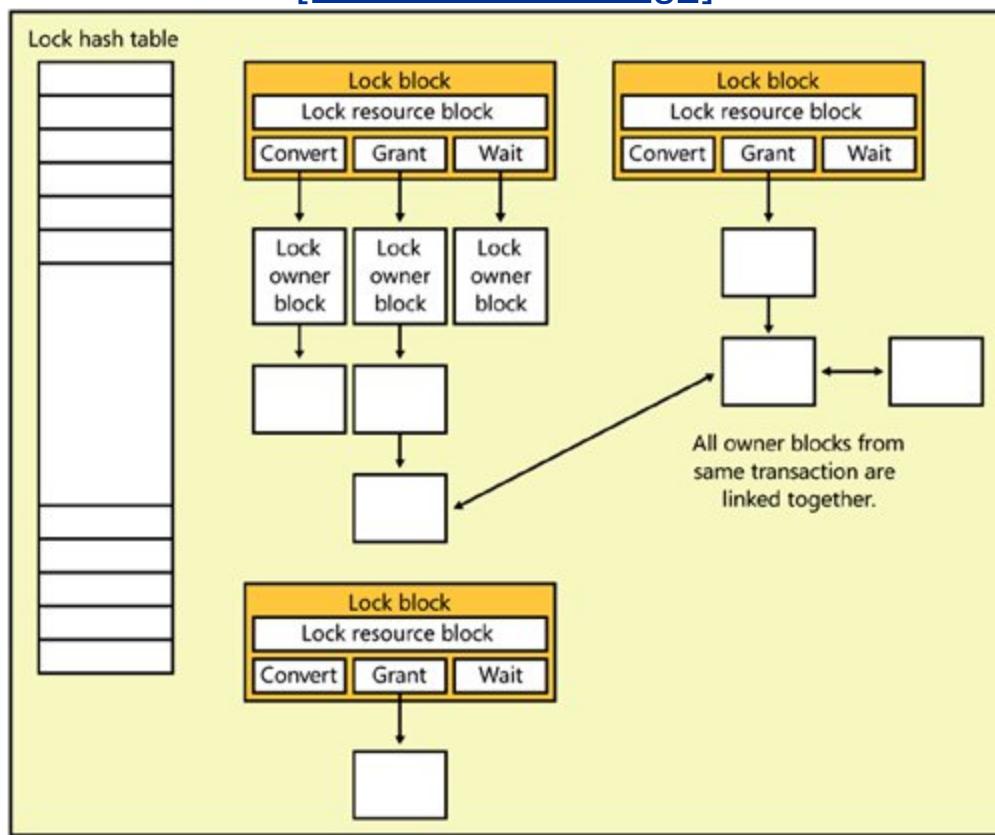
Locks are not on-disk structures. You won't find a lock field directly on a data page or a table header, and the metadata that keeps track of locks is never written to disk. Locks are internal memory structures—they consume part of the memory used for SQL Server. A lock is identified by *lock resource*, 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 8-3](#) shows the general lock architecture.

Figure 8-3. SQL Server locking architecture

[[View full size image](#)]



The number of slots in the hash table is based on the system's physical memory, as shown in [Table 8-7](#). There is an upper limit of 2^{31} slots. An exception is SQLExpress, which always has 2^{11} slots. All instances of SQL Server on the same machine will 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.

Table 8-7. 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

Physical Memory (MB)	Number of Slots	Memory Used
>= 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 is fixed 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 500 lock blocks on a SQL Server 2005 SQLExpress edition and 2500 lock blocks for the other editions. It allocates twice as many ($2 * \# \text{ lock blocks}$) 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. 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 will be 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 will successfully acquire the X lock on partitions 0 to 3, but it will block 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 will 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 will be from the same CPU (and practically always from the same node), which means the spinlock should not spin often.

Lock Blocks

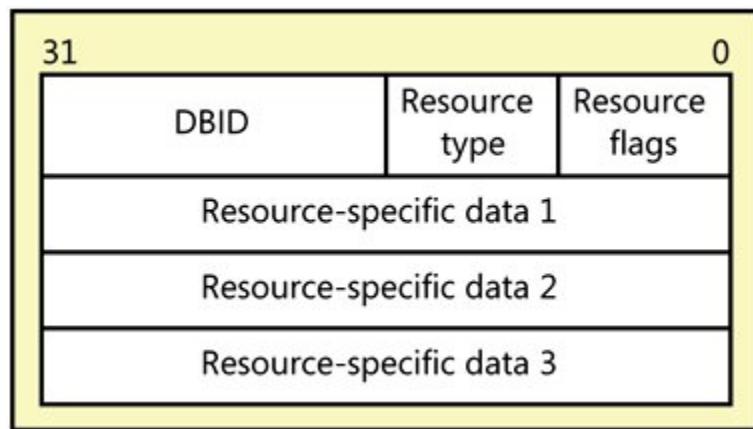
The lock block is the key structure in SQL Server's locking architecture, shown earlier in [Figure 8-3](#). A lock block contains the following information:

- 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 8-4](#). Each "row" in the figure represents 4 bytes, or 32 bits.

Figure 8-4. The structure of a lock resource



The meanings of the fields shown in [Figure 8-4](#) are described in [Table 8-8](#). The value in the *resource type* byte is one of the locking resources described earlier in [Table 8-5](#). The number in parentheses after the resource type is the code number for the resource type (which we'll 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'll describe shortly).

Table 8-8. Fields in the Lock Resource Block

Resource Contents			
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

Resource Contents

Resource Type	Data 1	Data 2	Data 3
Page (6)	Page number		0
Key (7)	Partition ID	Hashed key	
Extent (8)	Extent ID		0
RID (9)	RID		0
Application (10) (For Application an Application resource name resource, only the first 4 bytes of the name are used; the remaining bytes are hashed.)			

Following are some of the possible SR (SubResource) values. If the lock is on a DB 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. 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'll select a single row from the `Person.Contact` table in AdventureWorks using the REPEATABLE READ isolation level, so my shared locks will continue to be held for the duration of the transaction. I'll then look at the `rsc_bin` column in `syslockinfo` for key locks, page locks, and table locks.

```
USE AdventureWorks
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
BEGIN TRAN
SELECT * FROM Person.Contact
WHERE ContactID = 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
0xCDC17312000000000000000000005000500	5
0xA1260000010000000000000000000600	6
0x510000000001F900CE79D52505000700	7

The last two 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 for 0005 for 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 0005, which is the database ID of my *AdventureWorks* 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 1273C1CD, which is 309576141 in decimal. I can translate this to an object name as follows:

```
SELECT object_name(309576141)
```

This shows me the *Contact* 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 000026A1, or 9889 in decimal. So the lock is on file 1, page 9889.

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 010000000051000, which translates to 72057594043236352 in decimal. To see which object this partition belongs to, I can query the *sys.partitions* table:

```
SELECT object_name(object_id)
FROM sys.partitions
WHERE partition_ID = 72057594043236352
```

Again, the result is that this partition is part of the *Contacts* 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 datatype, 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 locked resource.

Bound Connections

Remember that the issue of lock contention applies only between different SQL Server processes. A process holding locks on a resource does not lock itself from the resource only other processes are denied access. However, the same process doesn't always mean the same user or application. It is common for applications to have more than one connection to SQL Server. Although the user and the application are the same, every such connection is treated as an entirely different SQL Server process, and by default no sharing of the "lock space" occurs between connections, even if they belong to the same user and the same application. One connection from the application could select data from a table, and as it retrieves rows, it might decide that one of the rows needs to be updated. If another connection tries to update a row that the first connection still has a shared lock on, the application might block itself. By default, SQL Server does not recognize the two connections from the same application as related in any way. SQL Server 2005 gives us two solutions to this problem, which we'll examine next.

Using Bound Connections

Prior to SQL Server 2005, the solution to avoiding conflicts between multiple connections within an application was to use a feature called *bound connections*, which allows two or more connections to share a lock space and hence not lock each other out. With a bound connection, the first connection asks SQL Server to give out its bind token. The bind token is passed by the application (using a client-side global variable, shared memory, or another method) for use in subsequent connections. The bind token acts as a "magic cookie" so other connections can share the lock space of the original

connection. Locks held by bound connections do not lock each other. (The *sp_getbindtoken* and *sp_bindsession* system stored procedures get and use the bind token.)

Here's an example of using bound connections between two query windows in SQL Server Management Studio. You must be inside of a transaction in order to get a bind token. We don't have a controlling application to declare and store the bind token in a client-side variable, so we have to actually copy it from the first session and paste it into the second. So, in your first query window, you execute this batch:

```
DECLARE @token varchar(255)
BEGIN TRAN
EXEC sp_getbindtoken @token OUTPUT
SELECT @token
GO
```

This should return something like the following:

```
QI9FL\B`QQOW- . GkN3N[6M5--- ._D]
```

Normally, you wouldn't have to look at this messy string; your application would just store it and pass it on without you ever having to see it. But for a quick example using SQL Server Management Studio queries, it's necessary to actually see the value. You use your keyboard or mouse to select the token string that you received and use it in the following batch in a second query window:

```
EXEC sp_bindsession 'QI9FL\B`QQOW- . GkN3N[6M5-- - ._D]'
```

GO

Now go back to the first query window and execute a command that locks some data. Remember that we have already begun a transaction to call `sp_getbindtoken`. You can use something like this:

```
USE AdventureWorks
UPDATE Production.Product
SET ListPrice = 180
WHERE Name = 'Chain';
```

This should exclusively lock a row in the *Products* table. Now go to the second query window and select the locked row:

```
SELECT * FROM Production.Product
WHERE Name = 'Chain';
GO
```

You should be able to see the \$180 price just as if you were part of the same connection as the first query. (A third connection that has not been bound to the first one would block when trying to execute the preceding SELECT statement.) Besides sharing lock space, the bound connection also shares transaction space. You can execute a ROLLBACK TRAN in the second window even though the first window began the transaction. If the first connection tries to then issue a ROLLBACK TRAN, it gets this message:

The transaction active in this session has been committed or aborted by another session.
Msg 3903, Level 16, State 1, Line 1

The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

You can actually see which connections are bound together in the DMV `sys.dm_tran_session_transactions`. Sessions that are bound together have the same `transaction_id` value, and all sessions that used the `sp_bindsession` procedure have a `is_bound` value of 1. Here are the results I get after binding the preceding two connections:

```
SELECT session_id, transaction_id, is_bound  
FROM sys.dm_tran_session_transactions;  
GO
```

RESULTS:

session_id	transaction_id	is_bound
52	13313	0
53	13313	1

There is no technical limit to the number of sessions that can be bound together, so you can have many sessions with the same `transaction_id` value. In practice, you should rarely need to have more than two or three sessions bound together. If a session executes `sp_bindsession` to bind to a different connection, its new binding will override the former binding.

Multiple Active Result Sets

If your goal in using bound connections is to make sure that two connections from the same application can interleave access to the

same data, SQL Server 2005 provides an alternative. In fact, SQL Server Books Online states that *sp_getbindtoken* and *sp_bindsession* will go away in a future version of SQL Server. The recommended alternative is to use a feature called Multiple Active Result Sets, or MARS. This feature allows a single connection from an application to send a statement to SQL Server for execution while a previous statement has still not completely sent its results back to the client. As in the preceding example, you can execute a SELECT that returns a large number of rows, and at some point, the application determines that one of the rows needs updating. Without MARS, the update statement would have to be sent over a separate connection. But with MARS enabled, the application can send the UPDATE command even though the SELECT has not completed. Not all T-SQL statements can be interleaved this way, even with MARS enabled. Only when SELECT, FETCH, and RECEIVE have results pending can another request be sent over the same connection. Any other statements must run to completion before execution can be switched to other requests using MARS.

MARS is not enabled by default; you must be specifically enable it in the connection string through the application. It is primarily a client issue, so a complete discussion of using MARS is beyond the scope of this book. I will return to it briefly, however, when discussing SQL Server 2005 features that use the new row versioning technology.

Row-Level Locking vs. Page-Level Locking

The debate over row-level locking versus page-level locking has been one of those near-religious wars and warrants a few comments here. All major vendors now support row-level locking. Although SQL Server 2005 fully supports row-level locking, in some situations the lock manager will decide not to lock individual rows and will instead lock pages or the whole table. In other cases, many smaller locks will be escalated to a table lock, as I'll discuss in the upcoming section about lock escalation.

Prior to version 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 version 7.0, SQL Server 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 megabytes (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 nonleaf-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 will be 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 stored procedure *sp_indexoption* lets you manually control the unit of locking within an index. It also lets you 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 the *sp_indexoption* setting.) The index options are set for each table or index individually. Two options, *AllowRowLocks* and *AllowPageLocks*, are both set to TRUE initially for every table and index. If both of these options are set to FALSE for a table, only full table locks are allowed.

As mentioned earlier, SQL Server determines at runtime whether to initially lock rows, pages, or the entire table. 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 locks as appropriate. This escalation protects system resourcesit prevents the system from using too much memory for keeping track of locksand increases efficiency. For example, after a query acquires many row locks, the lock level can be escalated to a table lock because it probably makes more sense to acquire and hold a single table lock than to hold many row locks. By escalating the lock, SQL Server acquires a single table lock, and the many row locks are released. This escalation to a table lock 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.

Lock escalation occurs in the following situations:

- The number of locks held by a single statement on one object (index or heap) exceeds a threshold. Currently that threshold is 5000 locks, but it might change in future service packs. The lock escalation will not occur if the locks are spread over multiple objects in the same statementfor example, 2500 locks in one index and 2500 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 non-zero value, memory reserved for locks is statically allocated when SQL Server starts. Escalation will occur 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 to the table level and there are concurrent X locks on the same table held by a different process, the lock escalation attempt will fail. However, SQL Server will continue 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 will release all the row and page locks on the index or the heap.

Note



SQL Server never escalates to page locks, and it is not possible to lock just a single partition of a table or index. The result of a lock escalation is always a full table lock.

Disabling 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 2005 supports disabling lock escalation in two ways, using trace flags.

- "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 will be generated. You should exercise extreme caution when using this trace flag as a poorly designed application can exhaust SQL Server's memory and seriously degrade your SQL Server's performance.
- "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. The trace flags affect the entire instance of SQL Server. In many cases, it is desirable to control the escalation threshold at the object level, perhaps based on the size of the object. The default escalation point of 5,000 locks might be too many locks for a small table and too few locks for a large table.

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. SQL Server 2005 produces more detailed deadlock diagnostics than previous versions. In addition, deadlocks can be detected for more types of resources. In this section, I'll refer mainly to deadlocks acquired due to conflicting locks, although deadlocks can also be detected on worker threads, memory, parallel query resources, and multiple active result sets (MARS) resources.

Note



A simple wait for a lock is not a deadlock. When the process that's holding the lock completes, the waiting process gets the lock. Lock waits are normal, expected, and necessary in multi-user systems.

In SQL Server, two main types of deadlocks can occur: a cycle deadlock and a conversion deadlock. [Figure 8-5](#) 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 SQL Server Management Studio, as follows:

1. Open a query window, and change your database context to the *AdventureWorks* 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:

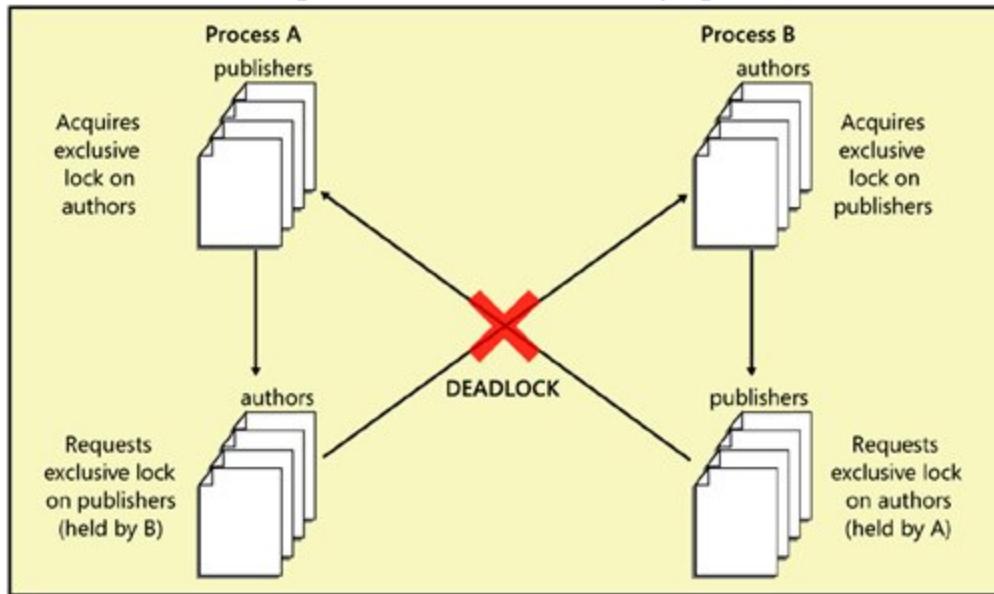
```
UPDATE Production.Product  
SET ListPrice = ListPrice * 0.9  
WHERE ProductID = 922;
```

At this point, a deadlock occurs. The first connection will never get its requested lock on the *PurchaseOrderDetail* table because the second connection will 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 will receive 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 8-5. A cycle deadlock resulting from two processes each holding a resource needed by the other

[\[View full size image\]](#)

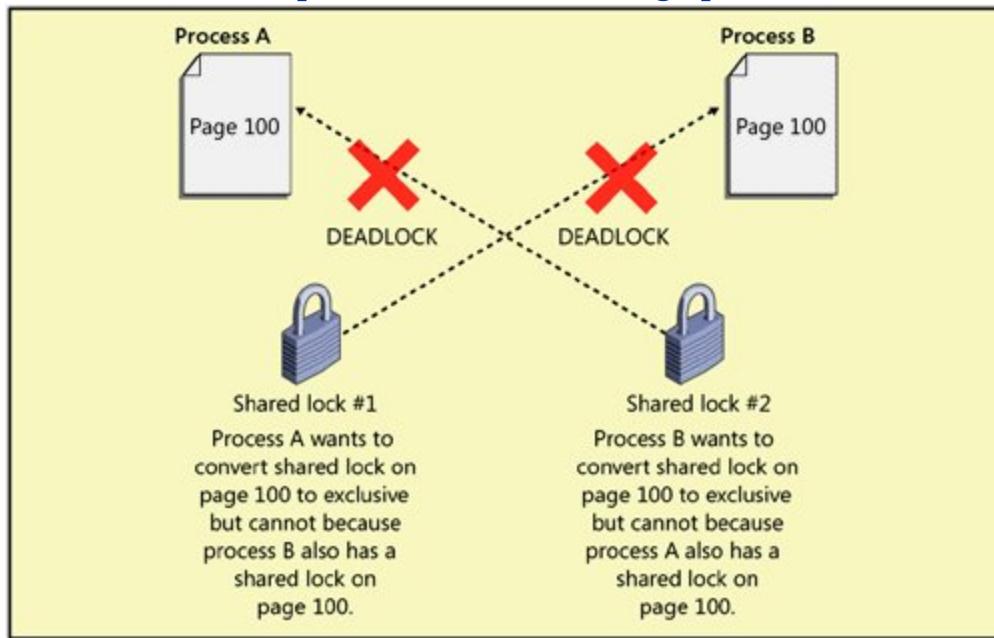


[Figure 8-6](#) shows an example of a conversion deadlock. Process A and process B each hold a shared lock on the same page within a transaction. Each process wants to promote its shared lock to an exclusive lock but cannot do so because of the other process's lock. Again, intervention is required.

Figure 8-6. A conversion deadlock resulting from two processes wanting to promote their locks on the same

resource within a transaction

[[View full size image](#)]



SQL Server automatically detects deadlocks and intervenes through the lock manager, which provides deadlock detection for regular locks. In SQL Server 2005, 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 it can't release until it acquires a lock on *Table1*, the processes will 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 executionrelated resources, and MARS resources. Latches are not involved in deadlock detection because SQL Server uses deadlock-proof algorithms when it acquires latches.

In SQL Server, a separate thread called `LOCK_MONITOR` checks the system for deadlocks every 5 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 5 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 is sent error message 1205. 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. The value `LOW` is equivalent to 5, `NORMAL` is 0, and `HIGH` is 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 8-5](#), the cycle deadlock could have been avoided if the processes had decided on a protocol beforehand for example, if they had decided to always access the *Product* table first and the *PurchaseOrderDetail* table second. Then one of the processes would get the initial exclusive lock on the table being accessed first, and the other process would wait 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 an error 1205 occurs, the application resubmits the batch, which will most likely succeed on a second try. Once one process is killed, its transaction is aborted, and its locks are rolled back, the other process involved in the deadlock can finish its work and release its locks, so the environment will not be conducive to another deadlock.) Although you might not be able to completely avoid deadlocks, 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.

Most of the techniques for troubleshooting and avoiding deadlocks are similar to the techniques for troubleshooting blocking problems in general. This topic will be discussed in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

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. In SQL Server 2005, optimistic concurrency is available after you enable one or both of the new database properties called `READ_COMMITTED_SNAPSHOT` and `ALLOW_SNAPSHOT_ISOLATION`. Exclusive locks are 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 will not block each other. Or, using locking terminology, a process requesting an exclusive lock will not block when the requested resource currently has a shared lock. Conversely, a process requesting a shared lock will 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. When *tempdb* is used to store previous versions of changed rows, it 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 UNCOMMITTED` isolation. If our results must always be based on committed data, we needed to be willing to wait for changes to be committed.

SQL Server 2005 introduces a new isolation level called SNAPSHOT isolation (SI) and a new non-blocking flavor of Read Committed isolation called READ COMMITTED SNAPSHOT isolation (RCSI). These row versioningbased isolations 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 *multi-version 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'll 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

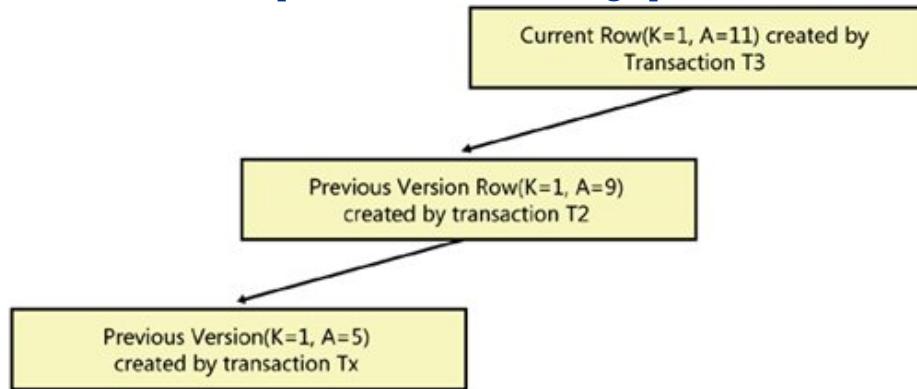
When a row in a table or index is updated, the new row is stamped with the transaction sequence number (XSN) of the transaction that is doing the update. The XSN is a monotonically increasing number that is unique within each SQL Server instance. The concept of XSN is new in SQL Server 2005, and it is not the same as log sequence numbers (LSN), which we discussed in [Chapter 5](#). I'll 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 8-7](#), 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*).

Figure 8-7. Versions of a row

[\[View full size image\]](#)



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 versioningbased 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 will bear this cost, even if there are no current readers of the data. If there are

readers using row versioning, they will have the extra cost of traversing the link pointers to find the appropriate version of the requested 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 will only occur with the full Snapshot isolation (SI), not with the enhanced Read Committed isolation level, RCSI.

Snapshot-Based Isolation Levels

SQL Server 2005 provides two types of snapshot-based isolation, both of which use row versioning to maintain the snapshot. One type, READ COMMITTED SNAPSHOT isolation (RCSI), is enabled with 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 (SI), 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 isolation.

READ COMMITTED SNAPSHOT Isolation

RCSI is a statement-level Snapshot isolation, which means any queries will see the most recent committed values as of the beginning of the statement. For example, let's look at the scenario in [Table 8-9](#). Assume two transactions are running in the AdventureWorks database, which has been

enabled for RCSI, and that before either transaction starts running, the *ListPrice* value of Product 922 is 8.89.

Table 8-9. 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

Time	Transaction 1	Transaction 2
2		<pre>SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 8.89</pre>
3	COMMIT TRAN	
4		<pre>SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 10.00</pre>

Time	Transaction 1	Transaction 2
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 will not block on that lock; it will have 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 will see the new value of the *ListPrice*. This is still READ COMMITTED isolation (just a non-locking 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 8-2](#).

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

```
ALTER DATABASE AdventureWorks
    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 will block. (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 4](#).

```
ALTER DATABASE AdventureWorks  
SET READ_COMMITTED_SNAPSHOT ON WITH NO_WAIT
```

If there are any users in the database, the preceding ALTER will fail with the following error:

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

You can also specify one of the ROLLBACK termination options, to basically kill 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 AdventureWorks  
    SET ALLOW_SNAPSHOT_ISOLATION ON;
```

When altering the database to allow SI, a user in the database will not necessarily block the command from completing. However, if there is an active transaction in the database, the ALTER will be 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 will 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 this is the message:

```
Msg 3956, Level 16, State 1, Line 1  
Snapshot isolation transaction failed to start in  
database 'Adventureworks' 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 will be complete. The database will not be 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 data read will be the most recent committed version 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 transaction accesses any data in the database.

As an example of SI, let's look at a scenario similar to the one in [Table 8-9](#). [Table 8-10](#) shows activities in a database with ALLOW_SNAPSHOT_ISOLATION set to ON. Assume two transactions are

running in the *AdventureWorks* database and that before either transaction starts, the *ListPrice* value of Product 922 is 8.89.

Table 8-10. A SELECT Running in a SNAPSHOT Transaction

Time	Transaction 1	Transaction 2
1	BEGIN TRAN	
2	UPDATE Production.Product SET ListPrice = 10.00 WHERE ProductID = 922;	SET TRANSACTION ISOLATION LEVEL SNAPSHOT

Time	Transaction 1	Transaction 2
3		BEGIN TRAN
4		<pre>SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 8.89 -- This is the beginning of -- the transaction</pre>
5	COMMIT TRAN	

Time	Transaction 1	Transaction 2
6		<pre>SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 8.99 -- Return the committed -- value as of the beginning -- of the transaction</pre>
7		COMMIT TRAN

Time	Transaction 1	Transaction 2
		<pre>SELECT ListPrice FROM Production.Product WHERE ProductID = 922; -- SQL Server returns 10.00</pre>

Event though Transaction 1 has committed, Transaction 2 continues to return the initial value it read of 8.99, until Transaction 2 completes. Only after Transaction 2 is done will 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 8-11](#) summarizes what each state means.

**Table 8-11. 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-based isolations levels are not allowed. Database versioning state is initially set to OFF during recovery. (A new feature of SQL Server 2005 is that the database is partially available during the UNDO phase of 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 under Snapshot isolation cannot start.

SNAPSHOT Isolation State	Description
ON	SI is enabled. New snapshot transactions can start in this database. Existing snapshot transactions (in another Snapshot-enabled database) 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 8-12](#) summarizes what each state means.

**Table 8-12. Possible Values for the Database Option
READ_COMMITTED_SNAPSHOT**

READ_COMMITTED_SNAPSHOT State	Description
0	READ_COMMITTED_SNAPSHOT is disabled. Database versioning state is initially set to 0 during recovery. If READ_COMMITTED_SNAPSHOT was enabled in the database being recovered, after recovery the READ_COMMITTED_SNAPSHOT state is set to 1.
1	READ_COMMITTED_SNAPSHOT is enabled. Any query with READ COMMITTED isolation will execute in the non-blocking 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 8-13](#) illustrates two processes attempting to update the *Quantity* value of the same row in the *ProductInventory* table in the *AdventureWorks* database. Two clerks have each received shipments of *ProductID* 872 and are trying up update the inventory. The *AdventureWorks* database has *ALLOW_SNAPSHOT_ISOLATION* set to ON, and before either transaction starts, the *Quantity* value of Product 872 is 324.

Table 8-13. An Update Conflict in SNAPSHOT Isolation

Time	Transaction 1	Transaction 2
1		SET TRANSACTION ISOLATION LEVEL SNAPSHOT
2		BEGIN TRAN

Time Transaction 1

3

Transaction 2

```
SELECT Quantity  
FROM  
Production.ProductInventory  
WHERE ProductID = 872;  
-- SQL Server returns 324  
-- This is the beginning of  
-- the transaction
```

4

```
BEGIN TRAN  
UPDATE  
Production.ProductInvent  
SET Quantity=Quantity +  
200  
WHERE ProductID = 872;  
-- Quantity is now 524
```

Time Transaction 1

5

Transaction 2

```
UPDATE  
Production.ProductInventory  
SET Quantity=Quantity + 300  
WHERE ProductID = 872;  
-- Process will block
```

6 COMMIT TRAN

7 Process will receive 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 will continue 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 will not allow that.

When Transaction 2 first tries to do 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 'AdventureWorks' 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 8-13](#) 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 8-13](#), Transaction 2 could use UPDLOCK on its initial SELECT as follows:

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

The UPDLOCK hint will force 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 will block. It is not using SI, so it will not be able to see the previous value of *Quantity*. Transaction 2 can perform its update because Transaction 1 is blocked, and it will commit. Transaction 1 can then perform its update on the new value of *Quantity*, and neither update will be 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. Certain Data Definition Language (DDL) statements are therefore 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 will affect the creation of these objects. [Table 8-14](#) shows an example of a snapshot transaction that includes both CREATE TABLE and CREATE INDEX.

Table 8-14. DDL Inside a SNAPSHOT Transaction

Time	Transaction 1	Transaction 2
1	SET TRANSACTION ISOLATION LEVEL SNAPSHOT	
2	BEGIN TRAN	

Time	Transaction 1	Transaction 2
3	<pre>SELECT count(*) FROM Production.Product -- This is the beginning of -- the transaction</pre>	
4		BEGIN TRAN

Time	Transaction 1	Transaction 2
5	<pre>CREATE TABLE NewProducts (<column definitions>) -- This DDL is legal</pre>	<pre>INSERT Production.Product VALUES (9999,) -- A new row is insert into -- the Product table</pre>
6		<pre>COMMIT TRAN</pre>

Time	Transaction 1	Transaction 2
7	<pre> CREATE INDEX PriceIndex ON Production.Product (ListPrice) -- This DDL will generate an -- error </pre>	

The CREATE TABLE statement will succeed 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 that snapshot that Transaction 1 is using, and SQL Server would generate an error instead of creating the index.

Another aspect of concurrent DDL to consider is what happens when a statement outside of 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 8-15](#) shows an example.

Table 8-15. Concurrent DDL Outside of 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

Time	Transaction 1	Transaction 2
4		<pre>BEGIN TRAN ALTER TABLE Purchasing.Vendor ADD notes varchar(1000); COMMIT TRAN</pre>
5	<pre>SELECT TOP 10 * FROM Production.Product;</pre> <p>-- Succeeds -- The ALTER to a different -- table does not affect -- this transaction</p>	

Time	Transaction 1	Transaction 2
6		<pre>BEGIN TRAN ALTER TABLE Production.Product ADD LowestPrice money; COMMIT TRAN</pre>
7	<pre>SELECT TOP 10 * FROM Production.Product; -- ERROR</pre>	

For the preceding situation, in Transaction 1, the repeated SELECT statements should always return the same data. 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 to the third SELECT. SQL Server will generate this error:

Msg 3961, Level 16, State 1, Line 1
Snapshot isolation transaction failed in database
'AdventureWorks' 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 will generate this error, even if there is no possibility of anomalies. For example, if Transaction 1 issues a SELECT count(*), which would not be affected by the ALTER TABLE, SQL Server will still generate error 3961.

Summary of Snapshot-Based Isolation levels

SI and RCSI are similar in the sense that they are based on 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 8-16](#) lists both the similarities and the differences between the two types of snapshot-based isolation.

**Table 8-16. SNAPSHOT vs. READ COMMITTED
SNAPSHOT Isolation**

SNAPSHOT Isolation	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 the database administrator (DBA) to turn on versioning for one particular application, such as big reporting snapshot transactions, and turn off versioning after the reporting transaction has started to prevent new snapshot transactions from starting. Turning on SI state 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 an X lock on the database. All users must be kicked out of a database to enable this option.

SNAPSHOT Isolation

READ COMMITTED SNAPSHOT Isolation

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 the table from each database can have its own individual setting. One table might get its data from the version store, and the other table will be 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.

SNAPSHOT Isolation

READ COMMITTED SNAPSHOT Isolation

When a database is recovered after a server crash, shut down, restored, attached, or made ONLINE, all versioning history for that database is lost. If database versioning state is ON, we can allow new snapshot transactions to access the database, but we must prevent previous snapshot transactions from accessing the database. Those previous transactions are interested in a point in time before the database recovers.

N/A. This is an object-level option; it is not at the transaction level.

SNAPSHOT Isolation

READ COMMITTED SNAPSHOT Isolation

If the database is in the IN_TRANSITION_TO_ON state, ALTER DATABASE SET ALLOW_SNAPSHOT_ISOLATION OFF will wait for about 6 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 is because changing the database versioning state requires a U lock on the database, which is compatible with regular users of the database who get an S lock but not compatible with another DBA who already has a U lock to change the state of the database.

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.

N/A. This option can be enabled only when there is no other active session in the database.

Similar.

SNAPSHOT Isolation

READ COMMITTED SNAPSHOT Isolation

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 versioning state will be rolled back and set to the previous one.

N/A.

You cannot use ALTER DATABASE to change database versioning state inside a user transaction.

Similar.

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.

SNAPSHOT Isolation

READ COMMITTED SNAPSHOT Isolation

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*.

You can turn this option ON for *msdb*.

A query in an 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.

SI can result in update conflicts that might cause a rollback or abort the transaction.

Similar, except that there are no implications for *tempdb*.

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 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.

There is no possibility of update conflicts.

The Version Store

As soon as a SQL Server 2005 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 as long as there are snapshot queries that might need them.

SQL Server 2005 provides several dynamic management views (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'll 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'll 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 *AdventureWorks* 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 *AdventureWorks*.

```
USE AdventureWorks
SELECT * INTO NewProduct
FROM Production.Product;
GO
ALTER DATABASE ADVENTUREWORKS SET
ALLOW_SNAPSHOT_ISOLATION ON;
GO
SELECT name, snapshot_isolation_state_desc,
       is_read_committed_snapshot_on
FROM sys.databases
WHERE name= 'Adventureworks';
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 there are no snapshot transactions that need to read those version. 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 *Product* table. The previous version of each row, prior to the update, has been written to *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 8-7](#), 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'll 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 *AdventureWorks* database, which already has *ALLOW_SNAPSHOT_ISOLATION* enabled. I then find the page number using *DBCC IND* (it is page 6709) 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 (AdventureWorks, 'T1', -1) -- page 6709
DBCC TRACEON (3604)
DBCC PAGE('AdventureWorks', 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 will be written to the version store and the XSN will be reflected in the row versioning info.

```
UPDATE T1 SET T1name = '2222222222' where T1ID = 'A';
GO
DBCC PAGE('AdventureWorks', 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...A222222222.
00000010: 00fc1804 00000100 0100590d 00000000 †
.....Y.....
```

As mentioned, if your database is enabled for one of the snapshot-based isolation levels, every new row will have an additional 14 bytes added to it, whether or not that row is ever actually involved in versioning. Every row updated will also have the 14 bytes added to it, if they aren't already part of the row, and the update will be 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 7](#), and they're cleaned up as part of the versioning cleanup process, as I'll discuss shortly. Here's an example of a ghost record under versioning:

```
DELETE T1 WHERE T1ID = 'B'  
DBCC PAGE('AdventureWorks', 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 00000000|||||  
N.....!.....
```

The record header indicates that this row is a GHOST VERSION and that it contains versioning information. The actual data, however, is not on the row, but the XSN is, so that snapshot transactions will know when this row was deleted and whether they should access the older version of it in their snapshot. The DMV `sys.dm_db_index_physical_stats` that I discussed in [Chapter 7](#) 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 done 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 will increase the size of the files. If the disk full gets so full that the files cannot grow, SQL Server will stop generating versions. If that happens, a snapshot query will fail 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 2005 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), *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 SQL Server is restarted. Transactions that do not generate version rows and do not use snapshot scans will 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 will generate a new *transaction_id* value and you'll 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 will be 1 if the current session has explicitly set TRANSACTION ISOLATION LEVEL to SNAPSHOT.)
- ***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'll now create a simple versioning scenario to illustrate how the values in the snapshot metadata get updated. This will not be a complete overview, but it should get you started in exploring the versioning metadata for your

own queries. I'll use the *AdventureWorks* database, which has `ALLOW_SNAPSHOT_ISOLATION` set to ON, and I'll 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'll call this session Connection 1. Change the session's isolation level and 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'll 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 will generate versions, it is not running in SI, so the *is_snapshot* value is 0. I'll 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 will be almost identical to the first, but there will be 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'll see a new XSN for this transaction (XSN3), and you'll 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'll 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 it.

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 will 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 will not show you the output because it is too wide for the page, but there are many columns here 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, will 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 will take longer because SQL Server will have to 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 2005 and was the only choice in all earlier versions of SQL Server. 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 and that many of your operations need to be performed in READ UNCOMMITTED isolation, 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 will automatically use 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 will have to be rolled back because of an update conflict.
- You have reports that need to be generated based on long-running, multi-statement 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 will not block each other.
- All SELECT operations will 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 will need to perform fewer lock escalations.
- Deadlocks will be 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 negatively affected when long-version chains must be scanned. The older the snapshot, the more time it will take 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 will generate 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 will expend 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]
```


Other Features That Use Row Versioning

Although the motivation behind adding row versioning to SQL Server 2005 was to maintain a version store for optimistic concurrency and to support snapshot-based isolation levels to solve the problem of data writers blocking all readers, other features can also take advantage of this data management technology. Two of these features, Multiple Active Result Sets (MARS) and online index rebuilds, are new in SQL Server 2005; the third is a new way of managing triggers, an existing feature. I discussed online index rebuilding in [Chapter 7](#), so I won't discuss that feature again here.

Triggers and Row Versioning

Triggers have been a part of SQL Server since the earliest version, and they were the only feature in those earlier versions that offered any type of historical (or versioned) data. One special feature of triggers is the ability to access pseudo-tables called *deleted* and *inserted*. If the trigger is a DELETE trigger, the *deleted* table contains all the rows that were deleted by the operation that caused the trigger to fire. If the trigger is an INSERT trigger, the *inserted* table contains all the rows that were inserted by the operation that caused the trigger to fire. If the trigger is an UPDATE trigger, the *deleted* table contains the old version of all the data rows changed by the UPDATE statement that caused the trigger to fire and the *inserted* table contains all the new versions. Previous versions of SQL Server populated these pseudo-tables by scanning the transaction log looking for all the log records in the current transaction that changed the table to which the trigger was tied.

In SQL Server 2005, the *deleted* and *inserted* tables are materialized using row versioning. When data modification

operations are performed on a table that has a relevant trigger defined, the changes to the table are versioned, regardless of whether a snapshot-based isolation level has been enabled. When the trigger needs to access the *deleted* table, it retrieves the data from the version store. New data, whether from an UPDATE or an INSERT, is accessible through the *inserted* table. When a trigger scans *inserted*, it looks for the most recent versions of the rows.

This short example will show you that the version store is used even if the database is not otherwise enabled for row versioning. The following code creates a copy of the *HumanResources.Department* table and then creates two triggers on the new *Department* table. All the trigger does is return a single row containing the number of rows in the version store and the size of all the row versions in the version store. I have included the statement to turn off row versioning in the *AdventureWorks* database to confirm that the use of the version store by triggers doesn't depend on any database option.

```
-- Turn off the snapshot options
ALTER DATABASE Adventureworks SET
ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE Adventureworks SET
READ_COMMITTED_SNAPSHOT OFF;
GO
-- Make a copy of the Department table (15 rows)
USE Adventureworks
SELECT *
INTO Department
FROM HumanResources.Department;
GO
-- Create two triggers, one for UPDATE and one for
DELETE
CREATE TRIGGER upd_Department
ON Department
FOR UPDATE
```

```
AS
SELECT count(*) AS NumRows,
       (sum(record_length_first_part_in_bytes) +
        sum(record_length_second_part_in_bytes))/8060. AS
Version_store_Pages
FROM sys.dm_tran_version_store;
GO
CREATE TRIGGER del_Department
ON Department
FOR DELETE
AS
SELECT count(*) AS NumRows,
       (sum(record_length_first_part_in_bytes) +
        sum(record_length_second_part_in_bytes))/8060. AS
Version_store_Pages
FROM sys.dm_tran_version_store;
GO
```

Now update a single row in the *Department* table and notice the count of rows in the version store. There should be one row for the inserted table and one row for the deleted table.

```
UPDATE Department
SET ModifiedDate = getdate()
WHERE DepartmentID = 11;
```

Now delete a single row in the *Department* table and notice the count of rows in the version store. There should be only one row for the deleted table.

```
DELETE Department  
WHERE DepartmentID = 12
```

Because *tempdb* is used for the version store, applications that make heavy use of triggers in SQL Server 2000 must be aware of potentially increased demands on *tempdb* after an upgrade to SQL Server 2005.

MARS and Row Versioning

Although MARS is a client-side feature of SQL Server 2005, its implementation relies on the version store, which is very much a server-side feature. I discussed MARS briefly earlier in this chapter as an alternative to bound connections, and I'll tell you a bit more about how this feature uses row versioning.

In earlier versions of SQL Server, database applications could not maintain multiple active statements on a connection when using default result sets. The application had to process or cancel all result sets from one batch before it could execute any other batch on that connection. With SQL Server 2005, an application can specify an attribute in the connection string to allow the application to have more than one pending request per connectionin particular, to have more than one active default result set per connection.

If two batches are submitted under a MARS connection, one of them containing a SELECT statement and the other containing an UPDATE statement, the UPDATE can begin execution before the SELECT has processed its entire result set. However, the UPDATE statement must run to completion before the SELECT statement can make progress, and all changes made by the UPDATE will be versioned. If both statements are running under the same

transaction, any changes made by the UPDATE statement after the SELECT statement has started execution are not visible to the SELECT because the SELECT will access the older version of the required data rows. Let's look at an example. [Table 8-17](#) shows two batches sent from the same application, using the same MARS-enabled connection. Most of the statements are just pseudocode because the requests are not really sent in Transact-SQL but are sent to SQL Server using the client API.

Table 8-17. Two Batches Sent on the Same MARS-Enabled Connection

Time	Batch 1	Batch 2
1	BEGIN TRAN	

Time Batch 1

2

```
SELECT_
FROM MyBigTable
<return ROW1>
<return ROW2>
<return ROW3>
<return ROW4>
-- return only some
of
-- results
```

Batch 2

Time	Batch 1	Batch 2
3		<pre>UPDATE ROW5 in MyBigTable -- There is no blocking even -- though Batch 1 may have a -- S lock on the data, -- because it is the same -- transaction -- The updated row is -- versioned</pre>

Time	Batch 1	Batch 2
------	---------	---------

4

```
<return ROW5>
-- Batch 1 will get
ROW5
-- from the version
store
-- as it was before
Batch 2's
-- update
```

Time Batch 1

5

Batch 2

```
DELETE ROW6 FROM
MyBigTable
-- Again, there will
be no
-- blocking
-- The deleted row
is
--      versioned
```

Time	Batch 1	Batch 2
6	<return ROW5> -- Batch 1 will get ROW6 -- even though it has been -- deleted by Batch 2 <return remaining rows>	

The example shows that row versioning is used for data modification operations submitted on the same connection as a currently active SELECT. The SELECT will read the versioned rows so it will have a consistent snapshot of the data as it was when it started reading. Any other connections submitted by the same application or by a completely different user will see the data as modified by Batch 2.

Controlling Locking

The SQL Server query optimizer usually chooses the correct isolation level, including the 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 will have an impact on the locks that will held, the conflicts that will 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

Transact-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*. 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 in order to obtain the desired concurrency behavior. A more complete discussion of how locking hints can be used to tune queries and applications can be found in *Inside Microsoft SQL Server 2005: Query Tuning and Optimization*.

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 BEGIN TRAN/COMMIT (or ROLLBACK) TRAN blocks. The lock hint syntax is as follows:

```
SELECT select_list
```

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

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

```
UPDATE 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. In SQL Server 2005, it is recommended that all hints be specified using WITH.

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

- **HOLDLOCK** This is equivalent to the following SERIALIZABLE hint. This option 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 takes update page locks instead of shared page 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 takes a shared lock on the table even if page locks would be taken otherwise. This option is useful when you know you'll escalate to a table lock or if you need to get a complete snapshot of a table. You can use this option with HOLDLOCK if you want the table lock held until the end of the transaction block (REPEATABLE READ). 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 keyword takes 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 takes 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 specifies that a shared row lock 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 of 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 READCOMMITTED 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 allows uncommitted, or dirty, reads. Shared locks are not issued by the scan, and the exclusive locks of others are not honored. This hint is equivalent to READUNCOMMITTED.
- **READPAST** This 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 will be 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 automatically canceled by the client 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 5 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 "get around" locking 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 *titles* table:

```
USE AdventureWorks;
BEGIN TRAN;
UPDATE HumanResources.Department
SET ModifiedDate = getdate();
WHERE DepartmentID = 1;
```

2. Open a second connection, and execute the following statements:

```
USE AdventureWorks;
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 timeout for one of the UPDATE statements will still allow the other statement to be processed. You must include error checking 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 AdventureWorks;
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> will return 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 AdventureWorks;
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 hint in SQL Server 2005 because of the availability of row versioning. In fact, any time 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 multiple-user 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 will block users trying to retrieve data. With optimistic concurrency, the locks will be ignored and older committed versions of the data will be 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 available concurrency models.

About the Author

Kalen Delaney is one of the founders of Solid Quality Learning and has been working with SQL Server for 19 years, starting with employment with the Sybase Corporation in October 1987.

After teaching Computer Science at UC Berkeley and Mills College in Oakland, California, Kalen joined Sybase and worked with their Technical Support Group for two years, and then transferred to the Education Department where she taught all the Sybase courses and was the specialist for the advanced course on Performance Tuning and Optimization.



In 1992, Kalen became an independent trainer and consultant, after moving with her family from the San Francisco Bay Area to the beautiful Pacific Northwest. Over the next few years, she worked with both the Microsoft and Sybase corporations to develop courses and to provide internal training for their technical support staff. In 1998, Kalen developed an internal course for Microsoft's product support team to help them learn the internals and new features of SQL Server 7, and she helped develop a similar course for SQL Server 2000. Kalen delivered these courses for many years to various Microsoft offices and partners around the country and

around the world. Today, she offers her own custom course on SQL Server 2005 Architecture, Internals and Query Tuning, which you can find out about on the Solid Quality Learning Web Site:
www.SolidQualityLearning.com.

In 1995, Microsoft awarded Kalen the MVP (Most Valuable Professional) designation for her participation in the public SQL Server help forums on Microsoft's news server (msnews.microsoft.com). She is still a regular participant on the public forums, answering many questions dealing with SQL Server internals and behavior.

Kalen has been writing about SQL Server for over ten years, in addition to courseware development. She wrote a regular monthly column on SQL Server Internals for Pinnacle Publishing's *SQL Server Professional Journal* from October 1995 to September 1998 and has been writing a monthly column for *SQL Server Magazine* since the very first issue. She has also written articles for *Windows IT Pro Magazine* (formerly *Windows 2000 Magazine*) and *MSDN Magazine*.

Kalen has spoken at Microsoft Technical Education Conference (TechEd) on many occasions, and has spoken at several internal Microsoft conferences around the world. In early 1999 she was asked to participate in the original planning committee to form a non-profit, international SQL Server Users' Group. Out of that planning committee, the Professional Association for SQL Server (PASS) was born. She served as the Director of Program Development for the first two years and spoke at the inaugural conference in Chicago in September 1999. She has spoken at every PASS conference since that time.

In 2002, Kalen and five colleagues started their own company, focusing on offering the most advanced SQL Server training in the world. Over the next four years, that company, Solid Quality

Learning, has grown to include more than 40 SQL Server experts from around the world.

Additional Resources for Developers

*Published and Forthcoming Titles on Microsoft® Visual Studio®
2005 and SQL Server™ 2005*

Visual Basic 2005

Microsoft Visual Basic® 2005 Express Edition: Build a Program Now!

Patrice Pelland

0-7356-2213-2

Microsoft Visual Basic 2005 *Step by Step*

Michael Halvorson

0-7356-2131-4

Programming Microsoft Visual Basic 2005: The Language

Francesco Balena

0-7356-2183-7

Visual C# 2005

Microsoft Visual C#® 2005 Express Edition: Build a Program Now!

Patrice Pelland

0-7356-2229-9

Microsoft Visual C# 2005 Step by Step

John Sharp

0-7356-2129-2

Programming Microsoft Visual C# 2005: The Language

Donis Marshall

0-7356-2181-0

Programming Microsoft Visual C# 2005: The Base Class Library

Francesco Balena

0-7356-2308-2

CLR via C#, Second Edition

Jeffrey Richter

0-7356-2163-2

Microsoft .NET Framework 2.0 Poster Pack

Jeffrey Richter

0-7356-2317-1

Web Development

Microsoft Visual Web Developer™ 2005 Express Edition: Build a Web Site Now!

Jim Buyens
0-7356-2212-4

Microsoft ASP.NET 2.0 Step by Step

George Shepherd
0-7356-2201-9

Programming Microsoft ASP.NET 2.0 Core Reference

Dino Esposito
0-7356-2176-4

Programming Microsoft ASP.NET 2.0 Applications Advanced Topics

Dino Esposito
0-7356-2177-2

Developing More-Secure Microsoft ASP.NET 2.0 Applications

Dominick Baier
0-7356-2331-7

Data Access

Microsoft ADO.NET 2.0 Step by Step

Rebecca M. Riordan

0-7356-2164-0

Programming Microsoft ADO.NET 2.0 Core Reference

David Sceppa

0-7356-2206-X

Programming Microsoft ADO.NET 2.0 Applications Advanced Topics

Glenn Johnson

0-7356-2141-1

SQL Server 2005

Microsoft SQL Server 2005 Database Essentials Step by Step

Solid Quality Learning

0-7356-2207-8

Microsoft SQL Server 2005 Applied Techniques Step by Step

Solid Quality Learning

0-7356-2316-3

Microsoft SQL Server 2005 Analysis Services Step by Step

Reed Jacobson, Stacia Misner, and Hitachi Consulting

0-7356-2199-3

Microsoft SQL Server 2005 Reporting Services Step by Step

Stacia Misner Hitachi Consulting

0-7356-2250-7

Programming Microsoft SQL Server 2005

Andrew J. Brust Stephen Forte

0-7356-1923-9

Inside Microsoft SQL Server 2005: The Storage Engine

Kalen Delaney

0-7356-2105-5

Inside Microsoft SQL Server 2005: T-SQL Programming

Itzik Ben-Gan, Dejan Sarka, and Roger Wolter
0-7356-2197-7

Inside Microsoft SQL Server 2005: T-SQL Querying
Itzik Ben-Gan, Lubor Kollar, and Dejan Sarka
0-7356-2313-9

Inside Microsoft SQL Server 2005: Query Tuning and Optimization
Kalen Delaney, et al.
0-7356-2196-9

Other Visual Studio 2005 Topics

Programming Microsoft Windows® Forms

Charles Petzold

0-7356-2153-5

Programming Microsoft Web Forms

Douglas J. Reilly

0-7356-2179-9

Debugging Microsoft .NET 2.0 Applications

John Robbins

0-7356-2202-7

Other Developer Topics

Hunting Security Bugs

Tom Gallagher, Bryan Jeffries and Lawrence Landauer
0-7356-2187-X

Software Estimation: Demystifying the Black Art

Steve McConnell
0-7356-0535-1

The Security Development Lifecycle

Michael Howard Steve Lipner
0-7356-2214-0

Writing Secure Code, Second Edition

Michael Howard David LeBlanc
0-7356-1722-8

Code Complete, Second Edition

Steve McConnell
0-7356-1967-0

Software Requirements, Second Edition

Karl E. Wiegert
0-7356-1879-8

More About Software Requirements: Thorny Issues and Practical Advice

Karl E. Wiegert
0-7356-2267-1

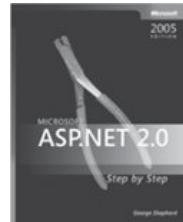
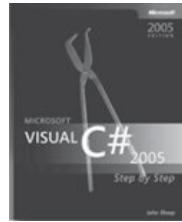
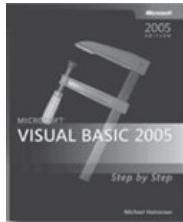
Explore our full line of learning resources at:
microsoft.com/mspress and **microsoft.com/learning**

More Great Developer Resources

Published and Forthcoming Titles from Microsoft Press

Developer Step by Step

- Hands-on tutorial covering fundamental techniques and features



- Practice files on CD

Microsoft®

Visual Basic® 2005

Step by Step

• Prepares and informs new-to-topic programmers

Michael Halvorson

0-7356-2131-4

Microsoft

Visual C#® 2005

Step by Step

John Sharp

0-7356-2129-2

Microsoft

ADO.NET 2.0

Step by Step

Rebecca M. Riordan

0-7356-2164-0

Microsoft

ASP.NET 2.0

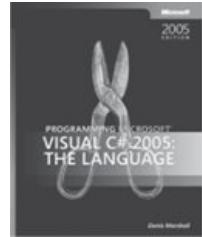
Step by Step

George Shepherd

0-7356-2201-9

Developer Reference

- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology



Programming

**Microsoft
Visual Basic 2005:
The Language**
Francesco Balena
0-7356-2183-7

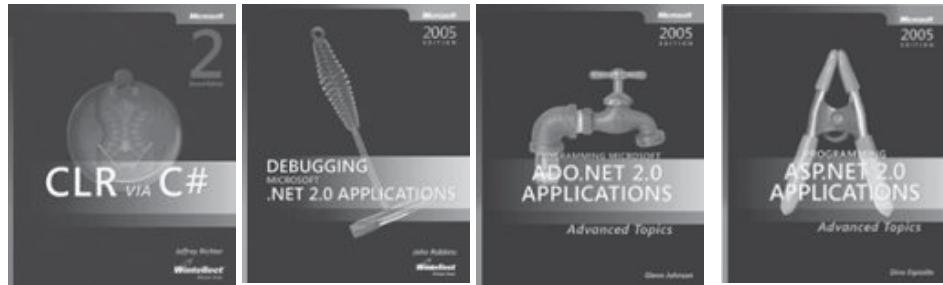
**Programming
Microsoft
Visual C# 2005:
The Language**
Donis Marshall
0-7356-2181-0

**Programming
Microsoft
ADO.NET 2.0
Core Reference**
David Sceppa
0-7356-2206-X

**Programming
Microsoft
ASP.NET 2.0
Core Reference**
Dino Esposito
0-7356-2176-4

Advanced Topics

- Deep coverage of advanced techniques and capabilities



- Extensive, adaptable coding examples

CLR via C#, Second Edition Jeffrey Richter 0-7356-2163-2	Debugging Microsoft .NET 2.0 Applications John Robbins 0-7356-2202-7	Programming Microsoft ADO.NET 2.0 Applications , Advanced Topics Glenn Johnson 0-7356-2141-1	Programming Microsoft ASP.NET 2.0 Applications , Advanced Topics Dino Esposito 0-7356-2177-2
---	---	---	---

See even more titles on our Web site!

Explore our full line of learning resources at: microsoft.com/mspress and microsoft.com/learning

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#)
[\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#)

[32-bit and 64-bit editions](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

access methods

[databases](#)

[in-memory data pages](#)

[overview](#)

[row and index operations](#)

[ACID properties](#)

[active virtual log files \(VLFs\)](#)

[active workers count column](#)

[Ad Hoc Distributed Queries configuration option](#)

[adding columns](#)

[adding constraints](#)

[Address Windowing Extensions \(AWE\) memory](#)
[advanced configuration settings](#)
[AdventureWorks database](#) 2nd
[Affinity column](#)
[Affinity I/O Mask configuration option](#) 2nd
[Affinity Mask configuration option](#) 2nd
[Affinity64 I/O Mask configuration option](#) 2nd
[Affinity64 Mask configuration option](#) 2nd
[Agent XPs configuration option](#)
[allocation optimizations in tempdb database](#)
[allocation unit](#)
[LOCATION UNIT locks](#)
[Allow Updates configuration option](#)
[ALTER COLUMN clause](#)
[ALTER DATABASE command](#)
[ALTER DATABASE examples](#)
[ALTER INDEX command](#)
[ALTER TABLE command](#)
[columns](#) 2nd
[constraints](#)
[data types](#)
[overview](#)
[SWITCH option](#)
[triggers](#)
[variations](#)
[altering databases](#)
altering tables
[columns](#) 2nd
[constraints](#)
[data types](#)
[metadata](#)
[methods for](#)

[overview](#)
[rows](#)
[triggers](#)
[analysis recovery phase](#)
[Analysis Services](#)
[ANSI NULL DEFAULT database option](#)
[ANSI NULLS database option](#)
[ANSI PADDING database option](#)
[ANSI WARNINGS database option](#)
[application locks](#)
[approximate numeric values](#)
architecture
[access methods](#)
[access to in-memory data pages](#)
[buffer pool](#)
[checkpoints](#)
[command parser](#)
[components overview](#)
[controlling utilities](#)
[data cache](#)
[database manager](#)
[managing memory in other caches](#)
[managing pages in data cache](#)
[Memory Broker](#)
[memory management overview](#)
[observing database engine behavior](#)
[overview](#)
[protocols](#)
[query executor](#)
[query optimizer](#)
[relational engine](#)
[sizing buffer pools](#)

[sizing memory](#)

[SQL manager](#)

[SQL OS](#)

[See [SQL operating system \(OS\)](#)]

[storage engine](#)

[tabular data stream \(TDS\) endpoints](#)

[transaction services](#)

[ARITHABORT database option](#)

[ASCII character set](#)

[associated entity ID](#)

[atomic transactions](#)

[atomicity in transactions](#)

[attributes](#)

[authentication](#)

[authorization](#)

[auto database options 2nd](#)

[AUTO_CLOSE database option](#)

[AUTO_CREATE_STATISTICS database option](#)

[automatic database file expansion](#)

[automatic database shrinkage](#)

[automatic truncation of virtual log files \(VLFs\)](#)

[AUTO_SHRINK database option](#)

[autoshrink database property](#)

[AUTO_UPDATE_STATISTICS database option](#)

[AWE \(Address Windowing Extensions\) memory](#)

[AWE Enabled configuration option 2nd](#)

[awe_allocated_kb column](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

B-trees

[deleting rows](#)

[ghost records](#)

[overview 2nd](#)

[size](#)

[backing up databases](#) [See also [restoring databases](#)]

[backup types 2nd](#)

[differential backups](#)

[files and filegroups 2nd](#)

[full backups](#)

[logs](#)
[moving databases](#)
[overview](#)
[partial backups](#)
[restart recovery](#)
[restore recovery](#)
[backing up files and filegroups 2nd](#)
[backing up logs](#)
[backup devices](#)
[backup types 2nd](#)
[base tables](#)
best practices
[DBCC commands](#)
[tempdb database](#)
[binary data types](#)
[binary sorting](#)
[binding schedulers to CPUs](#)
[bit data type](#)
[Blocked Process Threshold configuration option 2nd](#)
[blocking_exec_context_id column](#)
[blocking_session_id column](#)
[blocking_task_address column](#)
[Books Online 2nd](#)
[bound connections](#)
[bracketed identifiers](#)
[buckets_avg_length column](#)
[buckets_avg_scan_hit_length column](#)
[buckets_count column](#)
[buckets_in_use_count column](#)
[buckets_max_length column](#)
[buckets_min_length column](#)
[Buffer Manager](#)

buffer pools
[data cache](#)
[memory configurations](#)
[multiple instances on a single computer](#)
[Non-Uniform Memory Access \(NUMA\)](#)
[observing memory internals](#)
[read-ahead memory mechanism](#)
[sizing overview](#)
[Target Memory value](#)
[bulk update locks 2nd](#)
[BULK_LOGGED recovery model](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[C2 Audit Mode configuration option](#)
[caching optimizations in tempdb database](#)
[catalog views 2nd 3rd](#)
[changes in log size](#)
[active VLFs](#)
[automatic shrinking](#)
[automatic truncation of VLFs](#)
[log file size](#)
[maintaining recoverable logs](#)
[multiple physical log files](#)
[observing VLFs](#)

[overview](#)
[recoverable VLFs](#)
[reusable VLFs](#)
[unused VLFs](#)
[VLF overview](#)
[VLF states](#)
[changing data types](#)
[char and varchar \(single-byte characters\)](#)
[character data types](#)
[character sets](#)
[CHECK constraint](#)
[checkpoints](#)
[CHECKSUM database option](#)
[clock hand column](#)
[clock status column](#)
[CLR data types](#)
[CLR Enabled configuration option](#)
clustered indexes
[node rows](#)
[overview](#)
[rows with uniqueifier](#)
[clustering](#)
collation
[pre-installation decisions](#)
[specifying](#)
[column offset array](#)
columns in tables
[adding](#)
[dropping](#)
[overview](#)
[command parser](#)
compatibility

[database levels](#)
[earlier SQL Server versions](#)
[views](#)
[components](#)
[AdventureWorks](#)
[Analysis Services](#)
[Books Online](#)
[database engine 2nd](#)
[Development Tools](#)
[Integration Services](#)
[Notification Services](#)
[overview](#)
[Reporting Services](#)
[selecting for installation](#)
[SQL Enterprise Manager](#)
[SQL Query Analyzer](#)
[SQL Server Database Services](#)
[SQL Server Management Studio 2nd 3rd](#)
[SQLCMD](#)
[Workstation Components](#)
[computed columns in indexes](#)
[CONCAT NULL YIELDS NULL database option](#)

[concurrency \[See also locking\]](#)
[ACID properties](#)
[atomicity in transactions](#)
[consistency in transactions](#)
[durability in transactions](#)
[isolation in transactions](#)
[isolation levels](#)
[models 2nd](#)
[optimistic](#)

[overview](#)

[pessimistic](#)

[Read Committed isolation](#) 2nd

[Repeatable Read isolation](#) 2nd

[row versioning](#) [See [row versioning](#)]

[Serializable isolation](#)

[snapshot isolation](#) [See [snapshot isolation](#)]

[transaction processing overview](#)

[Uncommitted Read isolation](#) 2nd

[Configuration Manager](#) [See also [configurations](#)]

[configuring network protocols](#)

[default network protocols](#)

[managing services](#)

[overview](#)

[configuration settings](#)

[Ad Hoc Distributed Queries option](#)

[advanced](#)

[Affinity I/O Mask option](#) 2nd

[Affinity Mask option](#) 2nd

[Affinity64 I/O Mask option](#) 2nd

[Affinity64 Mask option](#) 2nd

[Agent XPs option](#)

[Allow Updates option](#)

[available options](#)

[AWE Enabled option](#) 2nd

[Blocked Process Threshold option](#) 2nd

[C2 Audit Mode option](#)

[changing](#)

[CLR Enabled option](#)

[Cost Threshold For Parallelism option](#) 2nd

[Cross Db Ownership Chaining option](#)

[Cursor Threshold option](#)
[Database Mail XPs option](#)
[Default Full-Text Language option](#)
[Default Language option](#)
[Default Trace Enabled option](#)
[Disallow Results From Triggers option](#)
[disk I/O options](#)
[Fill Factor option](#)
[Ft Crawl Bandwidth \(max\) option](#)
[FT Crawl Bandwidth \(min\) option](#)
[FT Notify Bandwidth \(max\) option](#)
[FT Notify Bandwidth \(min\) option](#)
[In-Doubt Xact Resolution option](#)
[Index Create Memory option 2nd](#)
[Lightweight Pooling option 2nd](#)
[list of available options](#)
[list of property pages](#)
[Locks option 2nd](#)
[Max Degree Of Parallelism option 2nd](#)
[Max Full-Text Crawl Range option](#)
[Max Server Memory option 2nd](#)
[Max Text Repl Size option](#)
[Max Worker Threads option 2nd](#)
[Media Retention option](#)
[memory options](#)
[Min Memory Per Query option 2nd](#)
[Min Server Memory option 2nd](#)
[Nested Triggers option](#)
[Network Packet Size option](#)
[Ole Automation Procedures option](#)
[Open Objects option](#)
[overview](#)

[PH Timeout option](#)
[Precompute Rank option](#)
[Priority Boost option 2nd](#)
[property pages](#)
[Query Governor Cost Limit option 2nd](#)
[query processing options](#)
[Query Wait option 2nd](#)
[Recovery Interval option 2nd](#)
[Remote Access option](#)
[Remote Admin Connections option](#)
[Remote Login Timeout option](#)
[Remote Proc Trans option](#)
[Remote Query Timeout option](#)
[Scan For Startup Procs option](#)
[scheduling options](#)
[Server Trigger Recursion option](#)
[Set Working Set Size option 2nd](#)
[Show Advanced Options option](#)
[SMO and DMO XPs option](#)
[SQL Mail XPs option](#)
[Transform Noise Words option](#)
[Two Digit Year Cutoff option](#)
[User Connections option 2nd](#)
[User Instance Timeout option](#)
[User Instances Enabled option](#)
[User options](#)
[Web Assistant Procedures option](#)
[xp_cmdshell option](#)
[configurations](#)
[compatibility with earlier SQL Server versions](#)
Configuration Manager [See [Configuration Manager](#)]
[default trace](#)

[network protocols](#)
[nonessential services overview](#)
[resource allocation settings](#) [See [configuration settings](#)]
[system configuration overview](#)
[system paging file location](#)
[task management](#)
[trace flags](#)
[configuring network protocols](#)
[consistency in transactions constraints](#)
[adding catalog views](#)
[compared to CREATE INDEX command](#)
[data integrity](#)
[declarative data integrity](#)
[disabling](#)
[domain integrity](#)
[dropping](#)
[enabling](#)
[entity integrity](#)
[failures](#)
[IDENTITY property](#)
[indexes](#)
[integrity checks](#)
[multiple-statement transactions](#)
[names](#)
[overview](#)
[programmatic data integrity](#)
[transaction errors](#)
[types](#)

[context switches count column](#)
[controlling locking](#)
[conversion locks](#)
[copying databases](#)
[Cost Threshold For Parallelism configuration option 2nd](#)
[cpu_id column](#)
[CREATE DATABASE command 2nd](#)
CREATE INDEX command
[WITH clause](#)
[compared to constraints](#)
[DROP EXISTING option](#)
[FILLFACTOR option](#)
[MAXDOP option](#)
[ONLINE option](#)
[overview](#)
[SORT IN TEMPDB option](#)
[STATISTICS NORECOMPUTE option](#)
[UNIQUE option](#)
[CREATE TABLE command](#)
[creating database snapshots](#)
[creating databases](#)
creating indexes
[constraints](#)
[CREATE INDEX command vs. constraints](#)
[CREATE INDEX options](#)
[included columns](#)
[overview](#)
[placement](#)
[creating tables](#)
[binary data types](#)
[bit data type](#)
[bracketed identifiers](#)

[character data types](#)
[choosing data types](#)
[CLR data types](#)
[cursor data type](#)
[data types overview](#)
[date data types](#)
[delimited identifiers](#)
[Hungarian-style notation](#)
[large object \(LOB\) data types](#)
[naming conventions](#)
[naming tables and columns](#)
[NOT NULL](#)
[NULL](#)
[numeric data types](#)
[overview](#)
[quoted identifiers](#)
[reserved keywords](#)
[rowversion data type](#)
[sql_variant data type](#)
[table data type](#)
[time data types](#)
[uniqueidentifier data type](#)
[user-defined data type \(UDT\)](#)
[creation_time column](#)
[Cross Db Ownership Chaining configuration option](#)
[current_tasks_count column](#)
[current_workers_count column](#)
[cursor data type](#)
[cursor database options 2nd](#)
[Cursor Threshold configuration option](#)
[CURSOR_CLOSE_ON_COMMIT {ON | OFF} database option](#)

CURSOR_DEFAULT {LOCAL | GLOBAL} database option

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[H]

[I]

[J]

[K]

[L]

[M]

[N]

[O]

[P]

[Q]

[R]

[S]

[T]

[U]

[V]

[W]

[X]

[DAC \(dedicated administrator connection\)](#).

data cache

[buffer pool](#)

[managing pages](#)

[Data Definition Language \(DDL\)](#).

[data index pages](#)

[data integrity](#)

data modifications in indexes

[data pages](#)

[deleting rows from B-trees](#)

[deleting rows in heaps](#)

[deleting rows in node levels](#)

[deleting rows overview](#)

[forward pointers](#)

[ghost records](#)

[in-place row updates](#)

[inserting rows](#)

[intermediate pages](#)

[locking](#)

[logging](#)

[moving rows](#)

[not in-place row updates](#)

[overview](#)

[reclaiming pages](#)

[root pages](#)

[splitting pages](#)

[table-level vs. index-level](#)

[updating rows](#)

data pages

data rows [See [data rows](#)]

[DBCC PAGE command](#)
[overview](#)
[page headers](#)
[row offset array](#)
[viewing](#)
[data rows](#)
[fixed-length](#)
[in-row data](#)
[LOB data](#)
[maximum row length](#)
[overview](#)
[row offset array](#)
[row-overflow data](#)
[structure](#)
[variable-length](#)
data storage
[LOB data](#)
[metadata](#)
[sql_variant data](#)
[varchar\(MAX\) data](#)
data types
[approximate numeric values](#)
[binary](#)
[bit](#)
[changing](#)
[character](#)
[choosing](#)
[CLR](#)
[cursor](#)
[date](#)
[datetime values](#)
[decimal](#)

[exact numeric values](#)
[fixed-length](#)
[globally unique identifier \(GUID\)](#)
[image](#)
[integer](#)
[large object \(LOB\) 2nd](#)
[ntext](#)
[numeric](#)
[overview](#)
[precision](#)
[rowversion](#)
[scale](#)
[single-byte characters \(char and varchar\)](#)
[smalldatetime values](#)
[sql_variant 2nd](#)
[storage requirements](#)
[table](#)
[text](#)
[time](#)
[timestamp](#)
[Unicode characters \(nchar and nvarchar\)](#)
[uniqueidentifier](#)
[universal unique identifier \(UUID\)](#)
[user-defined](#)
[varbinary](#)
[varchar\(MAX\) data](#)
[variable-length](#)

[database backups](#) [See also [database restores](#)]
[backup types 2nd](#)
[differential backups](#)
[files and filegroups 2nd](#)

[full backups](#)

[logs](#)

[moving databases](#)

[overview](#)

[partial backups](#)

[restart recovery](#)

[restore recovery](#)

Database Consistency Checker [See [DBCC commands](#)]

Database Console Command

[See [DBCC commands](#)]

[database engine](#)

[access methods](#)

[clustering](#)

[command parser](#)

[components overview](#)

[controlling utilities](#)

[database manager](#)

[full-text search](#)

[observing engine behavior](#)

[overview](#)

[protocols](#)

[query executor](#)

[query optimizer](#)

[query processor](#)

[relational engine](#)

[replication](#)

[SQL manager](#)

SQL OS

[See [SQL operating system \(OS\)](#)]

[storage engine](#)

tabular data stream (TDS) endpoints
transaction services
database files
Database Mail XPs configuration option
database manager
database mirroring options
database options
ANSI NULL DEFAULT
ANSI NULLS
ANSI PADDING
ANSI WARNINGS
ARITHABORT
auto 2nd
AUTO CLOSE
AUTO CREATE STATISTICS
AUTO SHRINK
AUTO UPDATE STATISTICS
CHECKSUM
CONCAT NULL YIELDS NULL
cursor 2nd
CURSOR CLOSE ON COMMIT {ON | OFF}
CURSOR DEFAULT {LOCAL | GLOBAL}
database mirroring
database recovery 2nd
EMERGENCY
emergency mode repair
external access
MULTI USER
NONE (No Page Verify)
NO WAIT
NUMERIC ROUNDABORT
OFFLINE

[ONLINE](#)
[other](#)
[overview](#)
[parameterization](#)
[QUOTED IDENTIFIER](#)
[READ ONLY](#)
[RECURSIVE TRIGGERS](#)
[RESTRICTED USER](#)
[ROLLBACK AFTER integer \[SECONDS\]](#)
[ROLLBACK IMMEDIATE](#)
[Service Broker](#)
[SINGLE USER](#)
[snapshot isolation](#)
[SQL 2nd](#)
[state 2nd](#)
[sys.databases catalog view](#)
[termination options](#)
[TORN PAGE DETECTION](#)
[database owner](#)
[database recovery options 2nd](#)

[database restores](#) [See also [database backups](#), [recovery](#)]
[BULK_LOGGED recovery model](#)
[example](#)
[FULL recovery model](#)
[migrating from earlier SQL Server versions](#)
[moving databases](#)
[overview](#)
[pages](#)
[partial restores](#)
[recovery models](#)
[restoring with standby files](#)

[SIMPLE recovery model](#)
[database security](#)
[authentication](#)
[authorization](#)
[database access](#)
[databases vs. schemas](#)
[default schemas](#)
[login names](#)
[managing](#)
[namespace](#)
[object](#)
[overview](#)
[principal](#)
[securable](#)
[separation of principals and schemas](#)
[SQL Server authentication](#)
[user](#)
[Windows authentication](#)

[Database Services](#) [See also [database engine](#)]
[clustering](#)
[full-text search](#)
[overview](#)
[replication](#)
[database snapshots](#)
[creating](#)
[managing](#)
[overview](#)
[space used by](#)
[database_id column](#)
[databases](#)
[AdventureWorks](#)

ALTER DATABASE examples

altering

backing up [See [backing up databases](#)]

compatibility levels

consistency [See [DBCC commands](#)]

copying

[CREATE DATABASE example](#)

creating

database engine [See [database engine](#)]

database manager

Database Services

[See [Database Services](#)]

DBCC commands

[See [DBCC commands](#)]

detaching and reattaching

expanding

[FILEGROUP CREATION example](#)

filegroups

files

hidden

locking

master 2nd

model [See [model database](#)]

moving

msdb

mssqlsystemresource

Northwind

options [See [database options](#)]

overview 2nd

properties [See [database options](#)]

[pubs](#)

[resource 2nd](#)

restoring [See [restoring databases](#)]

[sample](#)

security

[See [database security](#)]

[shrinking 2nd](#)

snapshots [See [database snapshots](#)]

[space allocation](#)

[system 2nd](#)

tempdb [See [tempdb database](#)]

[data space id column](#)

[date data types](#)

[datetime values](#)

[DBCC commands](#)

[best practices](#)

[DBCC CHECKALLOC](#)

[DBCC CHECKCATALOG](#)

[DBCC CHECKDB](#)

[DBCC CHECKTABLE](#)

[DBCC SHRINKDATABASE](#)

[DBCC SHRINKFILE](#)

[overview](#)

[progress reporting](#)

[REPAIR options](#)

[validation checks](#)

[verifying Service Broker data](#)

[DBCC IND command](#)

[DBCC PAGE command](#)

[DBCC SHOWCONTIG function](#)

[DDL \(Data Definition Language\)](#)

[deadlocks](#)
[decimal data types](#)
[declarative data integrity](#)
[dedicated administrator connection \(DAC\)](#)
[DEFAULT constraint](#)
[default filegroup](#)
[Default Full-Text Language configuration option](#)
[Default Language configuration option](#)
[default network protocols](#)
[default schemas](#)
[default trace](#)
[Default Trace Enabled configuration option](#)
deleting index rows
[B-trees](#)
[ghost records](#)
[heaps](#)
[node levels](#)
[overview](#)
[reclaiming pages](#)
[delimited identifiers](#)
[detaching databases](#)
[Developer Edition](#)
[Development Tools](#)
[differential backups](#)
[dirty reads](#)
[disabling constraints](#)
[disabling indexes](#)
[disabling nonessential services](#)
[disabling triggers](#)
[Disallow Results From Triggers configuration option](#)
[disk I/O configuration options](#)
[disk space requirements](#)

display name

DMFs (dynamic management functions)

DMVs (dynamic management views)

domain integrity

dropping columns

dropping constraints

durability in transactions

dynamic affinity

dynamic management functions (DMFs)

dynamic management views (DMVs)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[editions, multiple](#)

[EMERGENCY database options](#)

[emergency mode repair database option](#)

[EMPTYFILE option](#)

[enabling constraints](#)

[enabling triggers](#)

[Enterprise Edition](#)

[Enterprise Manager](#)

[entities](#)

[entity integrity](#)

[entries_count column](#)

entries in use count column
Evaluation Edition
exact numeric values
examples of locking
creating tables
overview
row locks
SELECT with default isolation level
SELECT with Repeatable Read isolation level
SELECT with Serializable isolation level
Serializable isolation level
Update operations
exclusive locks 2nd
exec context id column
expanding databases
explicit transactions
Express Edition
extents in databases
external access database options
external fragmentation of indexes

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[failures in constraints](#)

[fast file initialization](#)

[fast recovery](#)

[features](#)

[fibers vs. threads](#)

[file and filegroup backups 2nd](#)

[file_guid column](#)

[file_id column](#)

[FILEGROUP CREATION example](#)

[filegroups](#)

[FILEGROWTH file property](#)

fileid column

Fill Factor configuration option

fixed-length data types

fixed-length rows

FOREIGN KEY constraint

forward pointers in indexes

fragmentation of indexes

DBCC SHOWCONTIG function

detecting

external

internal

removing

reports

sys.dm_db_index_physical_stats function

types

Ft Crawl Bandwidth (max) configuration option

FT Crawl Bandwidth (min) configuration option

FT Notify Bandwidth (max) configuration option

FT Notify Bandwidth (min) configuration option

full database backups

FULL recovery model

full-text search

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[GAM \(Global Allocation Map\) pages](#)

[ghost records](#)

[Global Allocation Map \(GAM\) pages](#)

[globally unique identifier \(GUID\)](#)

granularity, lock

[ALLOCATION UNIT locks](#)

[application locks](#)

[associated entity ID](#)

[databases](#)

[extents](#)

[HOBT locks](#)

[key-range locks](#)

[lock resources](#)

[overview](#)

[sys.dm_tran_locks view](#)

[grouping, index](#)

[growth column](#)

[GUID \(globally unique identifier\)](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[hardware requirements](#)

[hash table](#)

[hashing](#)

[Heap Or B-Tree \(hobt\)](#)

[hidden database](#)

[hobt \(Heap Or B-Tree\)](#)

[HOBT locks](#)

[HOLDLOCK keyword](#)

[Hungarian-style notation](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[IAM \(Index Allocation Map\)](#) 2nd

[identifying lock resources](#)

[IDENTITY property](#) 2nd

[image data type](#)

[implicit transactions](#)

[In-Doubt Xact Resolution configuration option](#)

[in-memory data pages](#)

[in-row data pages](#)

[included columns in indexes](#)

[inconsistent analysis](#)

[Index Allocation Map \(IAM\)](#) 2nd

Index Create Memory configuration option 2nd
indexes
actual vs. estimated size
ALTER INDEX command
B-trees 2nd 3rd
changing options
clustered
clustered index node rows
clustered index rows with uniqueifier
computed columns
constraints
CREATE INDEX command vs. constraints
CREATE INDEX options
creating
DBCC IND command
detecting fragmentation
disabling
fragmentation
fragmentation reports
grouping
included columns
index row formats
joining
looking for rows
managing overview
modifying [See modifying indexes]
nonclustered
nonclustered index leaf rows
nonclustered index node rows
online index building
organization
overview 2nd

[partitioning](#)
[placement](#)
[rebuilding 2nd](#)
[removing fragmentation](#)
[reorganizing](#)
[sorting](#)
[space requirements](#)
[special](#)
[structure overview](#)
[sys.system internals allocation units view](#)
[types of fragmentation](#)
[uniqueness](#)
[views 2nd](#)
[information schema views](#)
[initializing data files](#)
[inserting index rows](#)

[installations \[See also upgrades\]](#)

[ASCII character set](#)

[binary sorting](#)

[Books Online](#)

[character sets](#)

[collation](#)

[components \[See components\]](#)

[Developer Edition](#)

[disk space requirements](#)

[editions](#)

[Enterprise Edition](#)

[Evaluation Edition](#)

[Express Edition](#)

[hardware requirements](#)

[initial splash screen](#)

[memory](#)

[multiple instances on single computer](#)

[named instances](#)

[performance considerations](#)

[pre-installation decisions overview](#)

[prerequisites overview](#)

[processors](#)

[release notes](#)

[security](#)

[software requirements](#)

[sort order overview](#)

[sort order semantics](#)

[specifying collation](#)

[Standard Edition](#)

[Upgrade Advisor](#)

[user context](#)

[Workgroup Edition](#)

[instance name](#)

[instant file initialization](#)

[integer data types](#)

[Integration Services](#)

[integrity checks](#)

[intent locks 2nd](#)

[intermediate index pages](#)

[internal fragmentation of indexes](#)

internal locking architecture

[lock blocks](#)

[lock owner blocks](#)

[lock partitioning](#)

[number of slots in hash table](#)

[overview](#)

[syslockinfo table](#)

[internal objects in tempdb database](#)

internal storage of tables

[allocation unit](#)

[data storage metadata](#)

[hobt \(Heap Or B-Tree\)](#)

[in-row data pages](#)

[LOB data pages](#)

[overview](#)

[querying catalog views](#)

[row-overflow data](#)

[row-overflow data pages](#)

[sys.allocation_units view](#)

[sys.indexes view](#)

[sys.partitions view](#)

[internal_object_reserved_page_count column](#)

[is_fiber column](#)

[is_media_read_only column](#)

[is_name_reserved column](#)

[is_online column](#)

[is_percent_growth column](#)

[is_preemptive column](#)

[is_read_only column](#)

[is_sparse column](#)

[isolation in transactions](#)

isolation levels

[overview](#)

[Read Committed isolation 2nd](#)

[Repeatable Read isolation 2nd](#)

snapshot isolation [See [snapshot isolation](#)]

[Uncommitted Read isolation 2nd](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[joining, index](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[key-range locks 2nd](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[large object \(LOB\) data 2nd 3rd](#)

[latches vs. locks](#)

[Lightweight Pooling configuration option 2nd](#)

[load factor column](#)

[LOB \(large object\) data 2nd 3rd](#)

[Locale column](#)

[lock blocks](#)

[lock compatibility](#)

[lock duration](#)

[lock escalation](#)

[lock hints](#)

lock modes

[bulk update locks 2nd](#)

[conversion locks](#)

[exclusive locks 2nd](#)

[intent locks 2nd](#)

[key-range locks](#)

[overview](#)

[schema modification locks](#)

[schema stability locks](#)

[shared locks 2nd](#)

[update locks 2nd](#)

[lock owner blocks](#)

[lock ownership](#)

[lock partitioning](#)

[lock resources](#)

[LOCK_TIMEOUT option](#)

[locking](#) [See also [concurrency](#)]

[ALLOCATION_UNIT locks](#)

[application locks](#)

[associated entity ID](#)

[bound connections](#)

[bulk update locks 2nd](#)

[controlling](#)

[conversion locks](#)

[data modifications in indexes](#)

[databases](#)

[deadlocks](#)

[examples](#)

[exclusive locks 2nd](#)

[extents](#)

[granularity](#)

[HOBT locks](#)
[intent locks 2nd](#)
[internal architecture](#)
[key-range locks 2nd](#)
[latches vs. locks](#)
[lock blocks](#)
[lock compatibility](#)
[lock duration](#)
[lock escalation](#)
[lock hints](#)
[lock modes](#)
[lock owner blocks](#)
[lock ownership](#)
[lock partitioning](#)
[lock resources](#)
[lock types for user data](#)
[LOCK_TIMEOUT option](#)
[Multiple Active Result Sets \(MARS\)](#).
[number of slots in hash table](#)
[overview 2nd](#)
row versioning [See [row versioning](#)]
[row-level vs. page-level](#)
[schema modification locks](#)
[schema stability locks](#)
[shared locks 2nd](#)
[spinlocks](#)
[sys.dm_tran_locks view](#)
[syslockinfo table](#)
[timeouts](#)
[update locks 2nd](#)
[viewing locks](#)
[Locks configuration option 2nd](#)

log backups

log files

backing up databases [See [backing up databases](#)]

changes in log size

[See [changes in log size](#)]

[log sequence numbers \(LSNs\)](#)

[overview 2nd](#)

[reading logs](#)

[recovery phases](#)

restoring databases [See [restoring databases](#)]

[transaction log basics](#)

[log sequence numbers \(LSNs\)](#)

[log size changes](#)

[active VLFs](#)

[automatic shrinking](#)

[automatic truncation of VLFs](#)

[log file size](#)

[maintaining recoverable logs](#)

[multiple physical log files](#)

[observing VLFs](#)

[overview](#)

[recoverable VLFs](#)

[Reusable VLFs](#)

[unused VLFs](#)

[VLF overview](#)

[VLF states](#)

logging

[analysis recovery phase](#)

backing up databases [See [backing up databases](#)]

[Buffer Manager](#)

[fast recovery](#)

[index data modifications](#)
[log sequence numbers \(LSNs\)](#)
log size changes [See [log size changes](#)]
[optimizations in tempdb database](#)
[overview](#)
[phases of recovery](#)
[reading logs](#)
[redo recovery phase](#)
[restart recovery](#)
[restore recovery](#)
restoring databases [See [restoring databases](#)]
[transaction log basics](#)
[undo recovery phase](#)
[write-ahead](#)
[lost updates](#)
[LSNs \(log sequence numbers\)](#).

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[Management Studio 2nd 3rd](#)

[managing database security](#)

[managing database snapshots](#)

managing indexes

[ALTER INDEX command](#)

[changing index options](#)

[detecting fragmentation](#)

[disabling indexes](#)

[fragmentation](#)

[fragmentation reports](#)

[online index building](#)

[overview](#)
[rebuilding indexes](#) 2nd
[removing fragmentation](#)
[reorganizing indexes](#)
[types of fragmentation](#)
[managing memory in other caches](#)
[managing pages in data cache](#)
[managing services](#)
[manual database file expansion](#)
[manual database shrinkage](#)
[MARS \(Multiple Active Result Sets\)](#) 2nd
[master database](#) 2nd
[Max Degree Of Parallelism configuration option](#) 2nd
[Max Full-Text Crawl Range configuration option](#)
[Max Server Memory configuration option](#) 2nd
[MAX specifier](#)
[Max Text Repl Size configuration option](#)
[Max Worker Threads configuration option](#) 2nd
[max_size column](#)
[maximum row length](#)
[MAXSIZE file property](#)
[Media Retention configuration option](#)
[memory](#)
[access to in-memory data pages](#)
[Address Windowing Extensions \(AWE\)](#).
[AWE Enabled option](#)
[buffer pool](#)
[checkpoints](#)
[configuration options overview](#)
[configurations](#)
[data cache](#)
[installation prerequisites](#)

[Locks option](#)
[management overview](#)
[managing memory in other caches](#)
[managing pages in data cache](#)
[Max Server Memory option](#)
[Memory Broker](#)
[Min Server Memory option](#)
[multiple instances on a single computer](#)
[Non-Uniform Memory Access \(NUMA\)](#)
[observing internals](#)
[read-ahead](#)
[Set Working Set Size option](#)
[sizing](#)
[sizing buffer pools overview](#)
[Target Memory value](#)
[User Connections option](#)
[Memory Broker](#)
metadata
[catalog views](#)
[changing](#)
[compatibility views](#)
[data storage](#)
[information schema views](#)
[layers of](#)
[partitioning](#)
[property functions](#)
[system functions](#)
[system stored procedures](#)
[migrations](#)
[Min Memory Per Query configuration option 2nd](#)
[Min Server Memory configuration option 2nd](#)
[mixed extents](#)

[mixed_extent_page_count column](#)
[model database](#)
[creating new databases](#)
[options](#)
[overview](#)
[modifying data types](#)
[modifying databases](#)
[modifying indexes](#)
[data pages](#)
[deleting rows from B-trees](#)
[deleting rows in heaps](#)
[deleting rows in node levels](#)
[deleting rows overview](#)
[forward pointers](#)
[ghost records](#)
[in-place row updates](#)
[inserting rows](#)
[intermediate pages](#)
[locking](#)
[logging](#)
[moving rows](#)
[not in-place row updates](#)
[overview](#)
[reclaiming pages](#)
[root pages](#)
[splitting pages](#)
[table-level vs. index-level modifications](#)
[updating rows](#)
[modifying tables](#)
[columns 2nd](#)
[constraints](#)
[data types](#)

metadata
methods for overview
rows
triggers
monitoring space in tempdb database
moving databases
moving index rows
msdb database
mssqlsystemresource database
multi_pages_in_use_kb column
multi_pages_kb column 2nd
MULTI_USER database option
Multiple Active Result Sets (MARS) 2nd
multiple database files
multiple editions
multiple instances
multiple physical log files
multiple-statement transactions

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[named instances](#)

[Named Pipes protocol](#)

[names of constraints](#)

[namespace](#)

[naming conventions](#)

[nchar and nvarchar \(Unicode characters\)](#)

[Nested Triggers configuration option](#)

[Network Packet Size configuration option](#)

[network protocols](#)

[configuring using Configuration Manager](#)

[default](#)

[system configurations](#)
[New Database dialog box](#)
[new features](#)
[newdb database](#)
[NO_WAIT database option](#)
[NOLOCK keyword](#)
[non-repeatable reads](#)
[Non-Uniform Memory Access \(NUMA\) 2nd 3rd](#)
nonclustered indexes
[leaf rows](#)
[node rows](#)
[overview](#)
[NONE \(No Page Verify\) database option](#)
[nonessential services](#)
[Northwind database](#)
[NOT NULL](#)
[Notification Services](#)
[NOTRUNCATE option](#)
[ntext data type](#)
[NULL](#)
[NUMA \(Non-Uniform Memory Access\) 2nd 3rd](#)
[NUMERIC ROUNDABORT database option](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[Object Explorer](#)

[object vs. securable](#)

objects, system

[catalog views](#)

[compatibility views](#)

[information schema views](#)

[overview](#)

[property functions](#)

[system stored procedures](#)

[observing memory internals](#)

[observing scheduler internals](#)

OFFLINE database options

Ole Automation Procedures configuration option

ONLINE database options

online index building

Open Objects configuration option

optimistic concurrency

optimizations in tempdb database

options, database [See database options]

order of integrity checks

organization of indexes

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[page allocation operations](#)

[page headers](#)

[page linkage in tables](#)

[page LSNs \(log sequence numbers\)](#)

[page-level locking vs. row-level locking](#)

[pages in indexes](#)

[reclaiming](#)

[splitting](#)

[PAGLOCK keyword](#)

[parameterization database options](#)

[parent_node_id column](#)

[partial backups](#)

[partitioning](#)

[catalog views](#)

[functions](#)

[metadata](#)

[overview](#)

[schemes](#)

[SWITCH option to ALTER TABLE command](#)

[pending_disk_io_count column](#)

[pending_io_byte_average column](#)

[pending_io_byte_count column](#)

[pending_io_count column](#)

[performance, sort order](#)

[pessimistic concurrency](#)

[PH Timeout configuration option](#)

[phantoms](#)

[phases of recovery](#)

[physical_name column](#)

[pinning tables](#)

[placement of indexes](#)

[post-upgrade operations](#)

[pre-installation decisions](#) [See also [prerequisites for installation](#)]

[ASCII character set](#)

[binary sorting](#)

[character sets](#)

[collation](#)

[multiple instances on a single computer](#)

[named instances](#)

[overview](#)

[performance considerations](#)

[release notes](#)
[security](#)
[sort order overview](#)
[sort order semantics](#)
[specifying collation](#)
[Upgrade Advisor](#)
[user context](#)
[precision](#)
[Precompute Rank configuration option](#)

[prerequisites for installations](#) [See also [pre-installation decisions](#)]

[Developer Edition](#)

[disk space requirements](#)
[editions](#)

[Enterprise Edition](#)

[Evaluation Edition](#)

[Express Edition](#)

[hardware requirements](#)

[initial splash screen](#)

[memory](#)

[overview](#)

[processors](#)

[software requirements](#)

[Standard Edition](#)

[Workgroup Edition](#)

[primary data files 2nd](#)

[primary filegroup](#)

[PRIMARY KEY constraint 2nd](#)

[principal vs. user](#)

[Priority Boost configuration option 2nd](#)

[processors](#)

[programmatic data integrity](#)
[progress reporting for database consistency](#)
properties, database [See [database options](#)]
[property functions](#)
protocols
[configuring using Configuration Manager](#)
[default](#)
[Named Pipes](#)
[overview](#)
[Shared Memory](#)
[system configurations](#)
[tabular data stream \(TDS\) endpoints](#)
[TCP/IP](#)
[Virtual Interface Adapter \(VIA\)](#).
[pubs database](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[Query Analyzer](#) 2nd

[query executor](#)

[Query Governor Cost Limit configuration option](#) 2nd

[query optimizer](#)

query processor

[command parser](#)

[database manager](#)

[overview](#)

[query executor](#)

[query optimizer](#)

[SQL manager](#)

Query Wait configuration option 2nd
querying catalog views
quoted identifiers
QUOTED_IDENTIFIER database option

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[RCSI \(READ COMMITTED SNAPSHOT isolation\) 2nd 3rd](#)

[Read Committed isolation 2nd](#)

[READ COMMITTED SNAPSHOT isolation \(RCSI\) 2nd 3rd](#)

[read-ahead memory mechanism](#)

[READ_ONLY database option](#)

[READCOMMITTED keyword](#)

[READCOMMITTEDLOCK keyword](#)

[READPAST keyword](#)

[READUNCOMMITTED keyword](#)

[reattaching databases](#)

[rebuilding indexes 2nd](#)

[reclaiming index pages](#)
[recoverable virtual log files \(VLFs\)](#).
recovery
[analysis phase](#)
backing up databases [See [backing up databases](#)]
[Buffer Manager](#)
[fast](#)
[log sequence numbers \(LSNs\)](#)
log size changes [See [log size changes](#)]
[overview](#)
[phases](#)
[reading logs](#)
[redo phase](#)
[restart recovery](#)
[restore recovery](#)
restoring databases [See [restoring databases](#)]
[transaction log basics](#)
[undo phase](#)
[write-ahead logging](#)
[Recovery Interval configuration option 2nd](#)
[recovery models](#)
[RECURSIVE TRIGGERS database option](#)
[redo recovery phase](#)
[referential integrity](#)
relational engine
[command parser](#)
[database manager](#)
[overview](#)
[query executor](#)
[query optimizer](#)
[SQL manager](#)
[release notes](#)

[Remote Access configuration option](#)
[Remote Admin Connections configuration option](#)
[Remote Login Timeout configuration option](#)
[Remote Proc Trans configuration option](#)
[Remote Query Timeout configuration option](#)
[removed all rounds count column](#)
[removing fragmentation](#)
[reorganizing indexes](#)
[REPAIR options for database consistency](#)
[Repeatable Read isolation 2nd](#)
[REPEATABLEREAD keyword](#)
[replication](#)
[Reporting Services](#)
[reports, fragmentation](#)
[reserved keywords](#)
[resource allocation for system configurations](#)
[resource database 2nd](#)
[resource address column](#)
[resource description column](#)
[restart recovery 2nd](#)
[restore recovery 2nd](#)

[restoring databases](#) [See also [backing up databases](#),
[recovery](#)]
[BULK_LOGGED recovery model](#)
[example](#)
[FULL recovery model](#)
[migrating from earlier SQL Server versions](#)
[moving databases](#)
[overview](#)
[pages](#)
[partial restores](#)

[recovery models](#)
[restoring with standby files](#)
[SIMPLE recovery model](#)
[restoring files and filegroups](#)
[restoring pages](#)
[RESTRICTED USER database option](#)
[reusable virtual log files \(VLFs\)](#)
[ROLLBACK AFTER integer \[SECONDS\] database option](#)
[ROLLBACK IMMEDIATE database option](#)
[root index pages](#)
[rounds_count column](#)
[row offset array](#)
[row operations](#)
[row versioning](#)
[concurrency models](#)
[Data Definition Language \(DDL\)](#)
[details](#)
[Multiple Active Result Sets \(MARS\)](#)
[overview](#)
[RCSI vs. SI 2nd](#)
[READ COMMITTED SNAPSHOT isolation \(RCSI\)](#)
[snapshot isolation overview](#)
[SNAPSHOT isolation \(SI\)](#)
[snapshot transaction metadata](#)
[sys.databases catalog view](#)
[transaction sequence number \(XSN\)](#)
[triggers](#)
[update conflicts](#)
[version store](#)
[row-level locking vs. page-level locking](#)
[row-overflow data 2nd 3rd](#)
[ROWLOCK keyword](#)

rows, index

[clustered index node rows](#)

[clustered index rows with uniqueifier](#)

[deleting from B-trees](#)

[deleting from heaps](#)

[deleting overview](#)

[deleting rows in node levels](#)

[formats](#)

[forward pointers](#)

[ghost records](#)

[in-place updates](#)

[inserting](#)

[joining](#)

[looking for](#)

[moving](#)

[nonclustered index leaf rows](#)

[nonclustered index node rows](#)

[not in-place updates](#)

[reclaiming pages](#)

[updating](#)

rows, table

[fixed-length](#)

[in-row data](#)

[LOB data](#)

[maximum row length](#)

[overview](#)

[row offset array](#)

[row-overflow data](#)

[structure](#)

[variable-length](#)

[rowversion data type](#)

runnable_tasks_count column

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[sample databases](#)

[scale](#)

[Scan For Startup Procs configuration option](#)

[scheduler](#)

[binding to CPUs](#)

[dedicated administrator connection \(DAC\)](#)

[dynamic affinity](#)

[Non-Uniform Memory Access \(NUMA\)](#)

[observing internals](#)

[overview](#)

[SQL Server schedulers](#)

[SQL Server tasks](#)
[SQL Server workers](#)
[threads vs. fibers](#)
[scheduler_id column](#) 2nd
[scheduling configuration options](#)
[schema modification locks](#)
[schema stability locks](#)
schemas
[compared to databases](#)
[default](#)
[separation of principals and schemas](#)
[secondary data files](#)
[securable vs. object](#)
security
database [See [database security](#)]
[pre-installation decisions](#)
[Serializable isolation](#)
[SERIALIZABLE keyword](#)
[Server Trigger Recursion configuration option](#)
[Service Broker database options](#)
[service name](#)
services
[disabling nonessential](#)
[managing](#)
[session ID \(SPID\)](#)
[session_id column](#)
[SET options](#)
[Set Working Set Size configuration option](#) 2nd
[Shared Global Allocation Map \(SGAM\) pages](#)
[shared locks](#) 2nd
[Shared Memory protocol](#)
[Show Advanced Options configuration option](#)

shrinking databases 2nd
SI (SNAPSHOT isolation) 2nd
SIMPLE recovery model
single-byte characters (char and varchar).
single_pages in use kb column
single_pages kb column 2nd
SINGLE USER database option
size column
sizing buffer pools
memory configurations
multiple instances on a single computer
Non-Uniform Memory Access (NUMA).
observing memory internals
overview
read-ahead memory mechanism
Target Memory value
sizing memory
smalldatetime values
SMO and DMO XPs configuration option
snapshot isolation
behaviors allowed
Data Definition Language (DDL).
database options
overview 2nd
RCSI vs. SI 2nd
READ COMMITTED SNAPSHOT isolation (RCSI).
scope
SNAPSHOT isolation (SI).
snapshot transaction metadata
sys.databases catalog view
types of
update conflicts

[version store](#)

snapshots, database

[creating](#)

[managing](#)

[overview](#)

[space used by](#)

[software requirements](#)

sort orders

[binary sorting](#)

[overview](#)

[performance considerations](#)

[semantics](#)

[specifying collation](#)

[sorting, index](#)

[space allocation in databases](#)

[space monitoring in tempdb database](#)

[space requirements in indexes](#)

special indexes

[COLUMNPROPERTY function](#)

[computed columns](#)

[deterministic vs. nondeterministic functions](#)

[implementation](#)

[overview](#)

[persisted columns](#)

[prerequisites](#)

[schema binding](#)

[SET options](#)

[views 2nd](#)

[SPID \(session ID\)](#)

[spinlocks](#)

[splitting pages in indexes](#)

[SQL database options 2nd](#)

[SQL Enterprise Manager](#)
[SQL Mail XPs configuration option](#)
[SQL manager](#)
[SQL Native Client](#)
[SQL operating system \(OS\)](#)
[binding schedulers to CPUs](#)
[dedicated administrator connection \(DAC\)](#).
[dynamic affinity](#)
[NUMA and schedulers](#)
[NUMA architecture](#)
[observing scheduler internals](#)
[overview](#)
[scheduler overview](#)
[SQL Server schedulers](#)
[SQL Server tasks](#)
[SQL Server workers](#)
[threads vs. fibers](#)
[SQL Query Analyzer 2nd](#)
[SQL Server 2005 Upgrade Advisor](#)
[SQL Server authentication](#)
[SQL Server Books Online 2nd](#)

[SQL Server Configuration Manager](#) [See also
configurations]
[configuring network protocols](#)
[default network protocols](#)
[managing services](#)
[overview](#)

[SQL Server Database Services](#) [See also [database engine](#)]
[clustering](#)
[full-text search](#)

[overview](#)

[replication](#)

[SQL Server Enterprise Manager](#)

[SQL Server Management Studio 2nd 3rd](#)

[SQL Server schedulers](#)

SQL Server storage engine

[access methods overview](#)

[controlling utilities](#)

[locking operations](#)

[overview](#)

[page allocation operations](#)

[row and index operations](#)

[transaction services](#)

[versioning operations](#)

[SQL Server tasks](#)

[SQL Server workers](#)

[SQL-92 reserved keywords](#)

[sql variant data 2nd](#)

SQLOS layer

[binding schedulers to CPUs](#)

[dedicated administrator connection \(DAC\)](#)

[dynamic affinity](#)

[NUMA and schedulers](#)

[NUMA architecture](#)

[observing scheduler internals](#)

[overview](#)

[scheduler overview](#)

[SQL Server schedulers](#)

[SQL Server tasks](#)

[SQL Server workers](#)

[threads vs. fibers](#)

[stack bytes used column](#)

[Standard Edition](#)

[standby_files](#)

[started_by_sqlservr column](#)

[state column](#)

state database options

[EMERGENCY](#)

[emergency_mode repair](#)

[MULTI_USER](#)

[NO_WAIT](#)

[OFFLINE](#)

[ONLINE](#)

[overview 2nd](#)

[READ_ONLY](#)

[RESTRICTED_USER](#)

[ROLLBACK AFTER integer \[SECONDS\]](#)

[ROLLBACK IMMEDIATE](#)

[SINGLE_USER](#)

[termination options](#)

[state_desc column](#)

[statistic_updates_during_upgrades](#)

storage engine

[access_methods_overview](#)

[controlling_utilities](#)

[locking_operations](#)

[overview](#)

[page_allocation_operations](#)

[row_and_index_operations](#)

[transaction_services](#)

[versioning_operations](#)

storage of data

[LOB_data](#)

[metadata](#)

[sql_variant data](#)
[varchar\(MAX\) data](#)
structure of indexes
[clustered index node rows](#)
[clustered index rows with uniqueifier](#)
[DBCC IND command](#)
[index row formats](#)
[nonclustered index leaf rows](#)
[nonclustered index node rows](#)
[overview](#)
[sys.system_internals_allocation_units view](#)
[Surface Area Configuration Tool 2nd](#)
[SWITCH option to ALTER TABLE command](#)
[sys.allocation_units view](#)
[sys.database_files view](#)
[sys.databases_catalog view 2nd](#)
[sys.dm_db_file_space_usage view](#)
[sys.dm_db_index_physical_stats function](#)
[sys.dm_db_session_space_usage view](#)
[sys.dm_db_task_space_usage view](#)
[sys.dm_os_memory_cache_clock_hands view](#)
[sys.dm_os_memory_cache_counters view](#)
[sys.dm_os_memory_cache_hash_tables view](#)
[sys.dm_os_memory_clerks view](#)
[sys.dm_osSchedulers view](#)
[sys.dm_os_tasks view](#)
[sys.dm_os_threads view](#)
[sys.dm_os_waiting_tasks view](#)
[sys.dm_os_workers view](#)
[sys.dm_tran_locks view 2nd](#)
[sys.indexes view](#)
[sys.partitions view](#)

[sys.system_internals_allocation_units view](#)

[syslockinfo table](#)

[system base tables](#)

[system catalogs](#)

[system configurations](#) [See also [configurations](#)]

[compatibility with earlier SQL Server versions](#)

[default trace](#)

[network protocols](#)

[nonessential services](#)

[overview](#)

[resource allocation](#)

[settings](#) [See [configuration settings](#)]

[system paging file location](#)

[task management](#)

[trace flags](#)

[system databases 2nd](#)

[system functions](#)

[system objects](#)

[catalog views](#)

[compatibility views](#)

[information schema views](#)

[overview](#)

[system functions](#)

[system stored procedures](#)

[system paging file location](#)

[system stored procedures](#)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[table data type](#)

[table-level hints](#)

[tables](#)

[adding columns](#)

[catalog views](#)

[changing data types](#)

[column offset array](#)

[compatibility views](#)

[constraint catalog views](#)

[constraint failures](#)

[constraint names](#)

[constraints overview](#)

creating [See [creating tables](#)]

[data pages](#)

[data rows](#)

[data storage metadata](#)

[fixed-length rows](#)

[IDENTITY property](#)

[information schema views](#)

[integrity checks](#)

[internal storage overview](#)

[LOB data](#)

[methods for modifying](#)

[modifying](#)

[modifying constraints](#)

[multiple-statement transactions](#)

[overview](#)

[page linkage](#)

[partitioning](#)

[property functions](#)

[querying catalog views](#)

[row-overflow data](#)

[sql variant data](#)

[sys.indexes view](#)

[system functions](#)

[system objects overview](#)

[system stored procedures](#)

[table-level vs. index-level modifications](#)

[transaction errors](#)

[varchar\(MAX\) data](#)

[variable-length rows](#)

[TABLOCK keyword](#)

[TABLOCKX keyword](#)

[tabular data stream \(TDS\) 2nd](#)
[Target Memory value](#)
[task management for system configurations](#)
[task_state column](#)
[tasks](#)
[TCP/IP protocol](#)
[TDS \(tabular data stream\) 2nd](#)
[tempdb database](#)
[allocation optimizations](#)
[best practices](#)
[caching optimizations](#)
[internal objects](#)
[logging optimizations](#)
[object types](#)
[optimizations](#)
[overview 2nd](#)
[space monitoring](#)
[sys.dm_db_file_space_usage view](#)
[sys.dm_db_session_space_usage view](#)
[sys.dm_db_task_space_usage view](#)
[user objects](#)
[version store](#)
[termination database options](#)
[text data type](#)
[threads vs. fibers](#)
[time data types](#)
[timestamp data type](#)
[TORN PAGE DETECTION database option](#)
[trace flags](#)
[trace, default](#)
[Transact-SQL CREATE DATABASE command](#)
[transaction errors](#)

[transaction logs](#)

backing up databases [See [backing up databases](#)]

changes in log size [See [changes in log size](#)]

[log sequence numbers \(LSNs\)](#)

[overview](#)

[reading logs](#)

[recovery phases](#)

restoring databases [See [restoring databases](#)]

transaction processing

[ACID properties](#)

[atomicity](#)

[consistency](#)

[dirty reads](#)

[durability](#)

[explicit transactions](#)

[implicit transactions](#)

[inconsistent analysis](#)

[isolation](#)

[isolation levels](#)

[lost updates](#)

[non-repeatable reads](#)

[overview](#)

[phantoms](#)

[Read Committed isolation 2nd](#)

[Repeatable Read isolation 2nd](#)

[Serializable isolation](#)

snapshot isolation [See [snapshot isolation](#)]

[Uncommitted Read isolation](#)

[transaction sequence number \(XSN\)](#)

[transaction services](#)

[Transform Noise Words configuration option](#)

[triggers 2nd](#)

TRUNCATEONLY option

tuples

Two Digit Year Cutoff configuration option

type column

type_desc column

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[UDT \(user-defined data type\)](#)

[unallocated_extent_page_count column](#)

[Uncommitted Read isolation 2nd](#)

[undo recovery phase](#)

[Unicode characters \(nchar and nvarchar\)](#)

[uniform extents](#)

[UNIQUE constraint 2nd](#)

[uniqueidentifier data type](#)

[uniqueifiers](#)

[universal unique identifier \(UUID\)](#)

[unused virtual log files \(VLFs\)](#)

[update locks](#) 2nd

updating index rows

[forward pointers](#)

[in-place row updates](#)

[moving rows](#)

[not in-place row updates](#)

[overview](#)

[UPDLOCK keyword](#)

[Upgrade Advisor](#)

[upgrade blockers](#)

[upgrades](#) [See also [installations](#)]

[benefits](#)

[changing database compatibility level](#)

[compared to migrations](#)

components [See [components](#)]

[drawbacks](#)

[internals](#)

[post-upgrade operations](#)

[steps for](#)

[Surface Area Configuration Tool](#)

[updating statistics](#)

[User configuration options](#)

[User Connections configuration option](#) 2nd

[User Instance Timeout configuration option](#)

[User Instances Enabled configuration option](#)

[user objects in tempdb database](#)

[user vs. principal](#)

[user-defined data type \(UDT\)](#)

[user-defined filegroups](#)

[user_object_reserved_page_count column](#)

UUID (universal unique identifier)

Index

[[SYMBOL](#)]

[[A](#)]

[[B](#)]

[[C](#)]

[[D](#)]

[[E](#)]

[[F](#)]

[[G](#)]

[[H](#)]

[[I](#)]

[[J](#)]

[[K](#)]

[[L](#)]

[[M](#)]

[[N](#)]

[[O](#)]

[[P](#)]

[[Q](#)]

[[R](#)]

[[S](#)]

[[T](#)]

[[U](#)]

[[V](#)]

[[W](#)]

[[X](#)]

[validation checks for database consistency](#)

[varbinary data type](#)

[varchar\(MAX\) data](#)

[variable-length data types](#)

[variable-length rows](#)

[VAS \(virtual address space\)](#)

[version compatibility](#)

[version store](#)

[version store in tempdb database](#)

[version_store_reserved_page_count column](#)

[versioning operations](#)

VIA (Virtual Interface Adapter)

viewing data pages

viewing locks

views

catalog 2nd

compatibility

indexes 2nd

information schema

virtual address space (VAS)

Virtual Interface Adapter (VIA)

virtual log files (VLFs)

active

automatic truncation

multiple physical

observing

overview

recoverable

reusable

unused

virtual memory committed kb column

virtual memory reserved kb column

VLFs [See virtual log files (VLFs)]

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[wait_duration_ms column](#)

[wait_type column](#)

[Web Assistant Procedures configuration option](#)

[Windows authentication](#)

[work_queue_count column](#)

[workers](#)

[Workgroup Edition](#)

[Workstation Components](#)

[write-ahead logging](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]

[XLOCK keyword](#)

[xp_cmdshell configuration option](#)

[XSN \(transaction sequence number\)](#)