Microsoft

Microsoft SQL Server 2012 Internals

Professional

Kalen Delaney

Bob Beauchemin, Conor Cunningham, Jonathan Kehayias, Benjamin Nevarez, Paul S. Randal

Microsoft SQL Server 2012 Internals

Dive deep inside the architecture of SQL Server 2012

Explore the core engine of Microsoft SQL Server 2012—and put that practical knowledge to work. Led by a team of SQL Server experts, you'll learn the skills you need to exploit key architectural features. Go behind the scenes to understand internal operations for creating, expanding, shrinking, and moving databases—whether you're a database developer, architect, or administrator.

Discover how to:

- Dig into SQL Server 2012 architecture and configuration
- Use the right recovery model and control transaction logging
- Reduce query execution time through proper index design
- Track events, from triggers to the Extended Event Engine
- Examine internal structures with database console commands
- Transcend row-size limitations with special storage capabilities
- Choose the right transaction isolation level and concurrency model
- Take control over query plan caching and reuse

Download SQL Server 2012 code samples at:

http://aka.ms/SQL2012Internals/files

About the Author

Kalen Delaney, a Microsoft MVP for SQL Server since 1993, provides advanced SQL Server training to clients worldwide. She is a contributing editor and columnist for *SQL Server Magazine* and the author of several books, including *Microsoft SQL Server 2008 Internals*.

microsoft.com/mspress

ISBN: 978-0-7356-5856-1

90000

9 780735 658561

U.S.A. $61.99
Canada $65.99
[*Recommended*]

Programming/Microsoft SQL Server

Microsoft Press

Celebrating 30 years!

Microsoft

Microsoft SQL Server 2012 Internals

Kalen Delaney
Bob Beauchemin
Conor Cunningham
Jonathan Kehayias
Benjamin Nevarez
Paul S. Randal

ISBN: 978-0-7356-5856-1

1 2 3 4 5 6 7 8 9 LSI 8 7 6 5 4 3

Printed and bound in the United States of America.

Acquisitions and Developmental Editor: Michael Bolinger

Production Editor: Kara Ebrahim

Editorial Production: Box 12 Communications

Technical Reviewers: Benjamin Nevarez and Jonathan Kehayias

Copyeditor: Box 12 Communications

Indexer: Box 12 Communications

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrator: Rebecca Demarest

Contents at a glance

Contents

Introduction

The book you are now holding is the evolutionary successor to the *Inside SQL Server* series, which included *Inside SQL Server 6.5, Inside SQL Server 7, Inside SQL Server 2000*, and *Inside SQL Server 2005* (in four volumes) and the *SQL Server 2008 Internals* book. The name was changed for SQL Server 2008 because the *Inside* series was becoming too unfocused, and the name "Inside" had been usurped by other authors and even other publishers. I needed a title that was much more indicative of what this book is really about.

SQL Server 2012 Internals tells you how SQL Server, Microsoft's flagship relational database product, works. Along with that, I explain how you can use the knowledge of how it works to help you get better performance from the product, but that is a side effect, not the goal. There are dozens of other books on the market that describe tuning and best practices for SQL Server. This one helps you understand why certain tuning practices work the way they do, and it helps you determine your own best practices as you continue to work with SQL Server as a developer, data architect, or DBA.

Who should read this book

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

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

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

Organization of this book

SQL Server 2012 Internals provides detailed information on the way that SQL Server processes your queries and manages your data. It starts with an overview of the architecture of the SQL Server relational database system and then continues looking at aspects of query processing and data storage in 13 additional chapters. The content from the *SQL Server 2008 Internals* book has been enhanced to cover changes and relevant new features of SQL Server 2012. In addition, it contains an entire chapter on the SQLOS, drawn from and enhanced from sections in the previous book, and a whole chapter on system databases, also drawn from and enhanced from content in the *SQL Server 2008* book. There is also a brand new chapter on special indexes, including spatial indexes, XML indexes, fulltext indexes, and semantic indexes. Finally, the chapter on query execution from my *Inside SQL Server 2005: Query Tuning and Optimization* book has been include and updated for SQL Server 2012.

Companion content

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

System requirements

To use the code samples, you'll need Internet access and a system capable of running SQL Server 2012 Enterprise or Developer edition. To get system requirements for SQL Server 2012 and to obtain a trial version, go to *http://www.microsoft.com/en-us/download/details.aspx?id=29066*.

Acknowledgments

As always, a work like this is not an individual effort, and for this current volume, it is truer than ever. I was honored to have five other SQL Server experts join me in writing *SQL Server 2012 Internals*, and I truly could not have written this book alone. I am grateful to Benjamin Nevarez, Paul Randal, Conor Cunningham, Jonathan Kehayias, and Bob Beauchemin for helping to make this book a reality. In addition to my brilliant co-authors, this book could never have seen the light of day without help and encouragement from many other people.

First on my list is you, the reader. Thank you to all of you for reading what I have written. Thank you to those who have taken the time to write to me about what you thought of the book and what else you want to learn about SQL Server. I wish I could answer every question in detail. I appreciate all your input, even when I'm unable to send you a complete reply. One particular reader of one of my previous books, *Inside Microsoft SQL Server 2005: The Storage Engine* (Microsoft Press, 2006), deserves particular thanks. I came to know Ben Nevarez as a very astute reader who found some uncaught errors and subtle inconsistencies and politely and succinctly reported them to me through my website. Ben is now my most valued technical reviewer, and for this new edition, he is also an author!

As usual, the SQL Server team at Microsoft has been awesome. Although Lubor Kollar and Sunil Agarwal were not directly involved in much of the research for this book, I always knew they were there in spirit, and both of them always had an encouraging word whenever I saw them. Kevin Liu volunteered for the daunting task of coordinating my contracts with the SQL team, and always found me the right engineer to talk to when I had specific questions that needed to be answered.

Ryan Stonecipher, Kevin Farlee, Peter Byrne, Srini Acharya, and Susan Price met with me and responded to my (sometimes seemingly endless) emails. Fabricio Voznika, Peter Gvozdjak, Jeff East, Umachandar Jayachandran, Arkadi Brjazovski, Madhan Ramakrishnan, Cipri Clinciu, and Srikumar Rangarajan also offered valuable technical insights and information when responding to my emails. I hope they all know how much I appreciated every piece of information I received.

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

I would like to extend my heartfelt thanks to all of the SQL Server MVPs, but most especially Erland Sommarskog. Erland wrote the section in Chapter 6 on collations just because he thought it was needed, and that someone who has to deal with only the 26 letters of the English alphabet could never do it justice. Also deserving of special mention are Ben Miller, Tibor Karaszi, and John Paul Cook, for all the personal support and encouragement they gave me. Other MVPs who inspired me during the writing of this volume are Hugo Kornelis, Rob Farley, and Allen White. Being a part of the SQL Server MVP team continues to be one of the greatest honors and privileges of my professional life.

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

Most important of all, my family continues to provide the rock-solid foundation I need to do the work that I do. I am truly blessed to have my husband, Dan, my daughter, Melissa, and my three sons, Brendan, Kyle (aka Rickey), and Connor.

—Kalen Delaney

Thanks to Kalen for persisting even when it didn't look like this book would ever be published. I'd also like to thank the chapter reviewers: Joe Sack and Kalen Delaney (yes, she personally reviewed all of it). Thanks to product designers who reviewed the parts in their area of expertise: Ed Katibah, Shankar Pal (who reviewed my original XML index material), and the entire full-text and semantic search team: Mahadevan Venkatraman, Elnata Degefa, Shantanu Kurhekar, Chuan Liu, Ivan Mitic, Todd Porter, and Artak Sukhudyan, who reviewed the final prose, as well as Naveen Garg, Kunal Mukerjee, and others from the original full-text/semantic team. Thank you, one and all.

Special thanks to Mary, without whose encouragement, help, and the ability to provide me the space I needed to work in solitude, I'd never have written anything at all. Finally, special thanks to Ed Katibah, who taught me almost everything I know about spatial concepts and representing spatial data in databases.

—Bob Beauchemin

I'd like to thank my wife Shannon and daughter Savannah for allowing me the late nights and weekend days to complete the work for this book. I could not do it without you both.

—Conor Cunningham

When Kalen asked me to contribute to this book it was a great honor, and I owe a debt of gratitude to her for the opportunity to work on this project. While working on the SQLOS and Extended Events updates for this book, I spent a lot of time discussing changes to the internals of SQLOS with Jerome Halmans at Microsoft. Jerome was also one the primary developers for Extended Events in SQL Server 2008, and has been incredibly gracious in answering my questions for the last four years.

I'd also like to acknowledge my wife, Sarah, and kids, Charlotte and Michael, and their ability and willingness to put up with the late hours at night, spent locked in our home office, as well as the weekends spent sitting on my laptop when there were so many other things we could have been doing. Sarah has spent many nights wondering if I was actually going to make it to bed or not while writing and editing portions of this book. Additional recognition goes out to all of the mentors I've had over the years, the list of which is incredibly long. Without the commitment of SQL Server MVPs like Arnie Rowland, Paul Randal, Aaron Bertrand, Louis Davidson, and countless others, I would have never made it as far as I have with SQL Server.

—Jonathan Kehayias

First of all I would like to thank Kalen for first offering me the opportunity to work as technical reviewer of this and her previous three books and later as co-author of two of the chapters of this book, "Special databases" and "Query execution." It is truly an honor to work on her books as it was her *Inside SQL Server* books, which helped me to learn SQL Server in the first place. It is an honor to be updating Craig Freedman's work as well; his chapter and blog have always been one of my all-time favorites. I also would like to thank Jonathan Kehayias for doing the technical review of my two chapters as he provided invaluable feedback to improve their quality.

Finally, on the personal side, I would like to thank my family: my wife, Rocio, my three boy-scout sons, Diego, Benjamin, and David, and my parents, Guadalupe and Humberto; thanks all for your unconditional support and patience.

—Benjamin Nevarez

By the time we wrote *SQL Server 2008 Internals*, I'd been itching to write a complete description of what DBCC CHECKDB does for many years. When Kalen asked me to write the consistency checking chapter for that book, I jumped at the chance, and for that my sincere thanks go to Kalen. I'm very pleased to have been able to update that chapter for SQL Server 2012 in this book and to add a section on the internals of the shrink functionality as well. I'd like to reaffirm my special gratitude to two people from Microsoft, among the many great folks I worked with there. The first is Ryan Stonecipher, who I hired away from being an Escalation Engineer in SQL Product Support in late 2003 to work with me on DBCC, and who was suddenly thrust into complete ownership of 100K+ lines of DBCC code when I become the team manager two months later. I couldn't have asked for more capable hands to take over my precious DBCC, and I sincerely appreciate the time he took to explore the 2012 changes with me. The second is Bob Ward, who heads up the SQL Product Support team and has been a great friend since my early days at Microsoft. We must have collaborated on many hundreds of cases of corruption over the years and I've yet to meet someone with more drive for solving customer problems and improving SQL Server. Thanks must also go to Steve Lindell, the author of the original online consistency checking code for SQL Server 2000, who spent many hours patiently explaining how it worked in 1999. Finally, I'd like to thank my wife, Kimberly, and our daughters, Katelyn and Kiera, who are the other passions in my life apart from SQL Server.

—Paul Randal

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed at:

http://aka.ms/SQL2012Internals/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

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

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

CHAPTER 1

SQL Server 2012 architecture and configuration

Kalen Delaney

Microsoft SQL Server is Microsoft's premier database management system, and SQL Server 2012 is the most powerful and feature-rich version yet. In addition to the core database engine, which allows you to store and retrieve large volumes of relational data, and the world-class Query Optimizer, which 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 single book could cover all these features in depth. This book, *SQL Server 2012 Internals*, covers only the main features of the core database engine.

This book delves into the details of specific features of the SQL Server Database Engine. This first chapter provides a high-level view of the components of that engine and how they work together. The goal is to help you understand how the topics covered in subsequent chapters fit into the overall operations of the engine.

SQL Server editions

Each version of SQL Server comes in various editions, which you can think of as a subset of the product features, with its own specific pricing and licensing requirements. Although this book doesn't discuss pricing and licensing, some of the information about editions is important because of the features available with each edition. *SQL Server Books Online* describes in detail the editions available and the feature list that each supports, but this section lists the main editions. You can verify what edition you are running with the following query:

```
SELECT SERVERPROPERTY('Edition');
```

You can also inspect a server property known as *EngineEdition*:

```
SELECT SERVERPROPERTY('EngineEdition');
```

The *EngineEdition* property returns a value of 2 through 5 (1 isn't a valid value in versions after SQL Server 2000), which determines what features are available. A value of 3 indicates that your SQL Server edition is either Enterprise, Enterprise Evaluation, or Developer. These three editions have exactly the same features and functionality. If your *EngineEdition* value is 2, your edition is either Standard, Web, or Business Intelligence, and fewer features are available. The features and behaviors discussed in this book are available in one of these two engine editions. The features in Enterprise edition (as well as in Developer and Enterprise Evaluation editions) that aren't in Standard edition generally relate to scalability and high-availability features, but other Enterprise-only features are available, as will be explained. For full details on what is in each edition, see the *SQL Server Books Online* topic, "Features Supported by the Editions of SQL Server 2012."

A value of 4 for *EngineEdition* indicates that your SQL Server edition is Express, which includes SQL Server Express, SQL Server Express with Advanced Services, and SQL Server Express with Tools. None of these versions are discussed specifically.

Finally, a value of 5 for *EngineEdition* indicates that you are running SQL Azure, a version of SQL Server that runs as a cloud-based service. Although many SQL Server applications can access SQL Azure with only minimum modifications because the language features are very similar between SQL Azure and a locally installed SQL Server (called an on-premises SQL Server), almost all the internal details are different. For this reason, much of this book's content is irrelevant for SQL Azure.

A *SERVERPROPERTY* property called *EditionID* allows you to differentiate between the specific editions within each of the different *EngineEdition* values—that is, it allows you to differentiate among Enterprise, Enterprise Evaluation, and Developer editions.

SQL Server installation and tools

Although installation of SQL Server 2012 is usually relatively straightforward, you need to make many decisions during installation, but this chapter doesn't cover all the details of every decision. You need to read the installation details, which are fully documented. Presumably, you already have SQL Server installed and available for use.

Your installation doesn't need to include every single feature because the focus in this book is on the basic SQL Server engine, although you should at least have installed a client tool such as SQL Server Management Studio that you can use for submitting queries. This chapter also refers to options available in the Object Explorer pane of the SQL Server Management Studio.

As of SQL Server 2012, you can install SQL Server on Windows Server 2008 R2 Server Core SP1 (referred to simply as Server Core). The Server Core installation provides a minimal environment for running specific server roles. It reduces the maintenance and management requirements and reduces the attack surface area. However, because Server Core provides no graphical interface capabilities, SQL Server must be installed using the command line and configuration file. Refer to the *Books Online* topic, "Install SQL Server 2012 from the Command Prompt," for details.

Also, because graphical tools aren't available when using Server Core, you can't run SQL Server Management Studio on a Server Core box. Your communications with SQL Server can be through a command line using the SQLCMD tool or by using SQL PowerShell. Alternatively, you can access your SQL Server running on Server Core from another machine on the network that does have the graphical tools available.

SQL Server metadata

SQL Server maintains a set of tables that store information about all objects, data types, constraints, configuration options, and resources available to SQL Server. In SQL Server 2012, these tables are called the *system base tables*. Some of the system base tables exist only in the *master* database and contain system-wide information; others exist in every database (including *master*) and contain information about the objects and resources belonging to that particular database. Beginning with SQL Server 2005, the system base tables aren't 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. If you log on as a system administrator and select from the catalog view called *sys.objects* (discussed shortly), you can see the names of all the system tables. For example, the following query returns 74 rows of output on my SQL Server 2012 instance:

```
USE master;
SELECT name FROM sys.objects
WHERE type_desc = 'SYSTEM_TABLE';
```

But even as a system administrator, if you try to select data from one of the tables returned by the preceding query, you get a 208 error, indicating that the object name is invalid. The only way to see the data in the system base tables is to make a connection using the dedicated administrator connection (DAC), which Chapter 2, "The SQLOS," explains in the section titled "The scheduler." Keep in mind that the system base tables are used for internal purposes only within the Database Engine and aren't intended for general use. They are subject to change, and compatibility isn't guaranteed. In SQL Server 2012, three types of system metadata objects are intended for general use: Compatibility Views, Catalog Views, and Dynamic Management Objects.

Compatibility views

Although you could see data in the system tables in versions of SQL Server before 2005, you weren't encouraged to do so. Nevertheless, many people used 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 system base tables, you would have to use the DAC to utilize your homegrown tools when using SQL Server 2005 or later. However, you still might be disappointed. Many names and much of the content of the SQL Server 2000 system tables have changed, so any code that used them is completely unusable even with the DAC. The DAC is intended only for emergency access, and no support is provided for any other use of it. To save you from this problem, SQL Server 2005 and later versions offer 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 are created in a hidden resource database that Chapter 4, "Special databases," covers.

Some compatibility views have names that might be quite familiar to you, such as *sysobjects*, *sysindexes*, *sysusers*, and *sysdatabases*. Others, such as *sysmembers* and *sysmessages*, might be less familiar. For compatibility reasons, SQL Server 2012 provides views that have the same names as many of the system tables in SQL Server 2000, as well as the same column names, which means that any code that uses the SQL Server 2000 system tables won't break. However, when you select from these views, you're not guaranteed to get exactly the same results that you get from the corresponding tables in SQL Server 2000. The compatibility views also don't contain any metadata related to features added after SQL Server 2000, such as partitioning or the Resource Governor. You should consider the compatibility views to be for backward compatibility only; going forward, you should consider using other metadata mechanisms, such as the catalog views discussed in the next section. All these compatibility views will be removed in a future version of SQL Server.

More info You can find a complete list of names and the columns in these views in *SQL Server Books Online*.

SQL Server also provides compatibility views for the SQL Server 2000 pseudotables, such as *sysprocesses* and *syscacheobjects*. Pseudotables aren't based on data stored on disk but are built as needed from internal structures and can be queried exactly as though they are tables. SQL Server 2005 replaced these pseudotables with Dynamic Management Objects, which are discussed shortly.

Catalog views

SQL Server 2005 introduced a set of catalog views as a general interface to the persisted system metadata. All catalog views (as well as the Dynamic Management Objects and compatibility views) are in the *sys* schema, which you must reference by 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, in the *sys* schema is a catalog view called *objects*, so to reference the view, the following can be executed:

```
SELECT * FROM sys.objects;
```

Similarly, catalog views are named *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, try running these two queries on your own and observe the difference:

```
SELECT * FROM sys.databases;
SELECT * FROM sysdatabases;
```

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 isn't required as it is for the catalog views. (That is, you can't 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 many more columns are 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 you want to know which databases are in *simple* recovery model, you can run the following:

```
SELECT name FROM sys.databases
WHERE recovery_model_desc = 'SIMPLE';
```

The catalog views are built on an inheritance model, so you don't have to redefine internally sets of attributes common to many objects. 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 16 additional columns that aren't applicable to all types of objects but are meaningful for tables. Also, 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 the relationship between the base view and the derived views can be summarized 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 as in SQL Server 2000, some metadata appears only in the *master* database and keeps track of system-wide 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 System Tables to 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 isn't available through catalog views but is still available in system tables, in the schema *dbo*. This includes metadata for backup and restore, replication, Database Maintenance Plans, Integration Services, log shipping, and SQL Server Agent.

As views, these metadata objects are based on an underlying Transact-SQL (T-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.tables*, you can execute the following:

```
SELECT object_definition (object_id('sys.tables'));
```

If you execute this *SELECT* statement, notice that the definition of *sys.tables* references several completely undocumented system objects. On the other hand, some system object definitions refer only to documented objects. For example, the definition of the compatibility view *syscacheobjects* refers only to three fully documented Dynamic Management Objects (one view, *sys.dm_exec_cached_plans*, and two functions, *sys.dm_exec_sql_text* and *sys.dm_exec_plan_attributes*).

Dynamic Management Objects

Metadata with names starting with *sys.dm_*, such as the just-mentioned *sys.dm_exec_cached_plans*, are considered Dynamic Management Objects. Although Dynamic Management Objects include both views and functions, they are usually referred to by the abbreviation DMV.

DMVs allow developers and database administrators to observe much of the internal behavior of SQL Server. You can access them as though they reside in the *sys* schema, which exists in every SQL Server database, but they aren't real objects. They are similar to the pseudotables used in SQL Server 2000 for observing the active processes (*sysprocesses*) or the contents of the plan cache (*syscacheobjects*).

Note A one-to-one correspondence doesn't always occur between SQL Server 2000 pseudotables and Dynamic Management Objects. For example, for SQL Server 2012 to retrieve most of the information available in *sysprocesses*, you must access three Dynamic Management Objects: *sys.dm_exec_connections*, *sys.dm_exec_sessions*, and *sys.dm_exec_requests*. Even with these three objects, information is still available in the old *sysprocesses* pseudotable that's not available in any of the new metadata.

The pseudotables in SQL Server 2000 don't provide any tracking of detailed resource usage and can't always be used to detect resource problems or state changes. Some DMVs allow tracking of detailed resource history, and you can directly query and join more than 175 such objects with T-SQL *SELECT* statements. The DMVs 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 aren't based on real tables stored in database files but are based on internal server structures, some of which are discussed in Chapter 2. More details about DMVs are discussed throughout 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 are:

- **dm_exec_*** This category 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_*** This category contains low-level system information such as memory and scheduling. For example, *sys.dm_os_schedulers* is a DMV that returns one row per scheduler. It's used primarily to monitor the condition of a scheduler or to identify runaway tasks.

- **dm_tran_*** This category 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 pseudotable *syslockinfo* in SQL Server 2000.
- **dm_logpool*** This category contains details about log pools used to manage SQL Server 2012's log cache, a new feature added to make log records more easily retrievable when needed by features such as AlwaysOn. (The new log-caching behavior is used whether or not you're using the AlwaysOn features. Logging and log management are discussed in Chapter 5, "Logging and recovery.")
- **dm_io_*** This category 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_*** This category contains details about databases and database objects such as indexes. For example, the *sys.dm_db_index_physical_stats* function 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 2012 also has dynamic management objects for many of its functional components, including objects for monitoring full-text search catalogs, service broker, replication, availability groups, transparent data encryption, Extended Events, and the Common Language Runtime (CLR).

Other metadata

Although catalog views are the recommended interface for accessing the SQL Server 2012 catalog, other tools are also available.

Information schema views

Information schema views, introduced in SQL Server 7.0, were the original system table-independent view of the SQL Server metadata. The information schema views included in SQL Server 2012 comply with the SQL-92 standard, and all these views are in a schema called *INFORMATION_SCHEMA*. Some 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 show only objects compatible with the SQL-92 standard. This means no information schema view exists for certain features, such as indexes, which aren't defined in the standard. (Indexes are an implementation detail.) If your code doesn't need to be strictly portable, or if you need metadata about nonstandard features such as indexes, filegroups, the CLR, and SQL Server Service Broker, using the Microsoft-supplied catalog views is suggested. Most 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 subsequent versions. Property functions provide individual values for many SQL Server objects as well as 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. The following property functions are available in SQL Server 2012:

- *SERVERPROPERTY*
- *COLUMNPROPERTY*
- *DATABASEPROPERTYEX*
- *INDEXPROPERTY*
- *INDEXKEY_PROPERTY*
- *OBJECTPROPERTY*
- *OBJECTPROPERTYEX*
- *SQL_VARIANT_PROPERTY*
- *FILEPROPERTY*
- *FILEGROUPPROPERTY*
- *FULLTEXTCATALOGPROPERTY*
- *FULLTEXTSERVICEPROPERTY*
- *TYPEPROPERTY*
- *CONNECTIONPROPERTY*
- *ASSEMBLYPROPERTY*

The only way to find out what the possible property values are for the various functions is to check *SQL Server Books Online*.

You also can see some information returned by the property functions by using the catalog views. For example, the *DATABASEPROPERTYEX* function has a *Recovery* property that returns the recovery model of a database. To view the recovery model of a single database, you can use the following property function:

```
SELECT DATABASEPROPERTYEX('msdb', 'Recovery');
```

To view the recovery models of all your databases, you can use the *sys.databases* view:

```
SELECT name, recovery_model, recovery_model_desc
FROM sys.databases;
```

Note Columns with names ending in *_desc* are known as the "friendly name" columns, and they are always paired with another column that is much more compact, but cryptic. In this case, the *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.

In addition to the property functions, the system functions include functions that are merely shortcuts for catalog view access. For example, to find out the database ID for the *AdventureWorks2012* database, you 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 = 'AdventureWorks2012';

SELECT DB_ID('AdventureWorks2012');
```

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 over these procedures: You have control over how much of the metadata you see because you can query the views as though they were tables. With the system stored procedures, you have to accept the data that it returns. Some procedures allow parameters, but they are very limited. For the *sp_helpdb* procedure, for example, you can pass a parameter to see just one database's information or not pass a parameter and see information for all databases. However, if you want to see only databases that the login *sue* owns, or just see databases that are in a lower compatibility level, you can't do so using the supplied stored procedure. Through 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 < 110;
```

Metadata wrap-up

Figure 1-1 shows the multiple layers of metadata available in SQL Server 2012, with the lowest layer being the system base tables (the actual catalog). Any interface that accesses the information contained in the system base tables is subject to the metadata security policies. For SQL Server 2012, that means that no users can see any metadata that they don't need to see or to which they haven't specifically been granted permissions. (The few exceptions are very minor.) "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 interfaces to the system metadata are 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; thinking conceptually of them as another layer on top of the catalog view interface is useful.

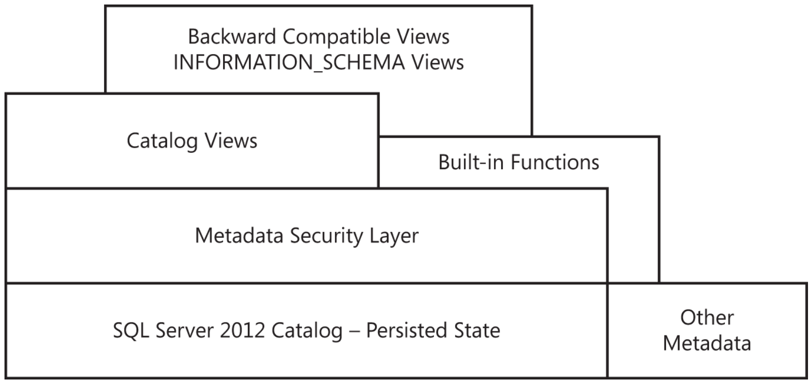
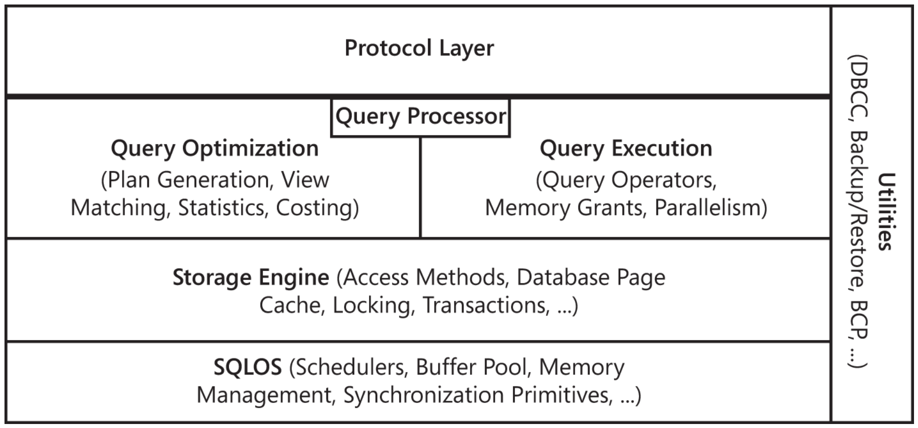


FIGURE 1-1 Layers of metadata in SQL Server 2012.

Components of the SQL Server engine

Figure 1-2 shows the general architecture of SQL Server and its four major components: the protocol layer, the query processor (also called the relational engine), 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, some minor omissions and simplifications have been made and certain "helper" modules have been ignored among the subcomponents.)



Protocol Layer
Query Processor
Query Optimization
(Plan Generation, View Matching, Statistics, Costing)
Query Execution
(Query Operators, Memory Grants, Parallelism)
Storage Engine (Access Methods, Database Page Cache, Locking, Transactions, ...)
SQLOS (Schedulers, Buffer Pool, Memory Management, Synchronization Primitives, ...)
Utilities
(DBCC, Backup/Restore, BCP, ...)

FIGURE 1-2 The major components of the SQL Server Database Engine.

The protocol layer receives the request and translates it into a form that the relational engine can work with. 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 query processor accepts T-SQL batches and determines what to do with them. For T-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 Database Console Commands (DBCCs). The SQLOS layer handles activities normally considered to be operating system responsibilities, such as thread management (scheduling), synchronization primitives, deadlock detection, and memory management, including the buffer pool.

The next section looks at the major components of the SQL Server Database Engine in more detail.

Protocols

When an application communicates with the 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*. The SQL Server Network Interface (SNI) protocol layer on both the server and client computers encapsulates the TDS packet inside a standard communication protocol, such as TCP/IP or Named Pipes. On the server side of the communication, the network libraries are part of the Database Engine. On the client side, the network libraries are part of the SQL Native Client. The configuration of the client and the instance of SQL Server determine which protocol is used.

You can configure SQL Server to support multiple protocols simultaneously, coming from different clients. Each client connects to SQL Server with a single protocol. If the client program doesn't know which protocols SQL Server is listening on, you can configure the client to attempt multiple protocols sequentially. The following protocols are available:

- **Shared Memory** The simplest protocol to use, with no configurable settings. Clients using the Shared Memory protocol can connect to only a SQL Server instance running on the same computer, so this protocol isn't 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 can't 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 network computer).
- **TCP/IP** The most widely used protocol over the Internet. TCP/IP can communicate across interconnected computer networks 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.

Tabular Data Stream endpoints

SQL Server 2012 also allows you to create a TDS endpoint, so that SQL Server listens on an additional TCP port. During setup, SQL Server automatically creates an endpoint for each of the three 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 can't be used. An additional endpoint is created for the DAC, which only members of the *sysadmin* fixed server role can use. (Chapter 2 discusses the DAC in more detail.)

Query processor

As mentioned earlier, the query processor is also called the relational engine. It includes the SQL Server components that determine exactly what your query needs to do and the best way to do it. In Figure 1-2, the query processor is shown as two primary components: Query Optimization and query execution. This layer also includes components for parsing and binding (not shown in the figure). 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. Chapter 11, "The Query Optimizer," discusses the Query Optimizer in great detail; this section gives you just a high-level overview of the Query Optimizer as well as of the other components of the query processor.

The query processor also manages query execution as it requests data from the storage engine and processes the results returned. Communication between the query processor and the storage engine is generally in terms of Object Linking and Embedding (OLE) DB rowsets. (Rowset is the OLE DB term for a result set.)

Parsing and binding components

The parser processes T-SQL language events sent to SQL Server. It checks for proper syntax and spelling of keywords. After a query is parsed, a binding component performs name resolution to convert the object names into their unique object ID values. After the parsing and binding is done, the command is converted into an internal format that can be operated on. This internal format is known as a *query tree*. If the syntax is incorrect or an object name can't be resolved, an error is immediately raised that identifies where the error occurred. However, other types of error messages can't be explicit about the exact source line that caused the error. Because only parsing and binding components 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 and prepares it for optimization. Statements that can't be optimized, such as flow-of-control and Data Definition Language (DDL) commands, are compiled into an internal form. Optimizable statements are marked as such and then passed to the Query

Optimizer. The Query Optimizer is concerned mainly with the Data Manipulation Language (DML) statements *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *MERGE*, which can be processed in more than one way; the Query Optimizer determines which of the many possible ways is best. It compiles an entire command batch and optimizes queries that are optimizable. 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 Query Optimizer normalizes a query, it *optimizes* it, which means that it determines a plan for executing that query. Query optimization is cost-based; the Query 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 Query 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*. (Chapter 11 discusses statistics in detail.) Based on the available information, the Query Optimizer considers the various access methods and processing strategies that it could use to resolve a query and chooses the most cost-effective plan.

The Query Optimizer also uses pruning heuristics to ensure that optimizing a query doesn't take longer than required to simply choose a plan and execute it. The Query Optimizer doesn't necessarily perform 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 theoretically never causes 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 it's 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 is affected, which indexes are used (if any), 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 aren't 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 be accessed or expressions to be evaluated. Also, the existence of a trigger can cause many more 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 query executor

The query executor runs the execution plan that the Query Optimizer produced, acting as a dispatcher for all commands in the execution plan. This module goes through each command of the execution plan until the batch is complete. Most commands require interaction with the storage engine to modify or retrieve data and to manage transactions and locking. You can find more information on query execution and execution plans in Chapter 10, "Query execution."

The storage engine

The SQL Server storage engine includes all components involved with the accessing and managing of data in your database. In SQL Server 2012, the storage engine is composed of three main areas: access methods, locking and transaction services, and utility commands.

Access methods

When SQL Server needs to locate data, it calls the access methods code, which sets up and requests scans of data pages and index pages and prepares the OLE DB rowsets to return to the relational engine. Similarly, when data is to be inserted, the access methods code can receive an OLE DB rowset from the client. The access methods code contains components to open a table, retrieve qualified data, and update data. It doesn't actually retrieve the pages; instead, 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 used not only for *SELECT* statements but also for qualified *UPDATE* and *DELETE* statements (for example, *UPDATE* with a *WHERE* clause) and for any data modification operations that need to modify index entries. The following sections discuss some types of access methods.

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 structures—namely, 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. The row operations code needs to handle special operations if the data is a large object (LOB) data type—*text*, *image*, or *ntext*—or if the row is too large to fit on a single page and needs to be stored as overflow data. Chapter 6, "Table storage"; Chapter 7, "Indexes: internals and management"; and Chapter 8, "Special storage," look at the different types of data-storage structures.

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. (A very small tree might not have intermediate-level pages.) A B-tree groups records with 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 that the search for any particular 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 for each database and monitors 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 KB disk pages spread across one or more physical files. (Chapter 3, "Databases and database files," goes into more detail about the physical organization of databases.)

SQL Server uses 13 types of disk pages. The ones this book discusses are data pages, two types of Large Object (LOB) pages, row-overflow 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, Minimally Logged (ML) pages, and Differential Changed Map (DIFF) pages. Another type, File Header pages, won't be discussed in this book.

All user data is stored on data, LOB, or row-overflow pages. Index rows are stored on index pages, but indexes can also store information on LOB and row-overflow 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. BCM and DCM pages are used to make backup and recovery more efficient. Chapter 5 explains these page types in more detail.

Versioning operations Another type of data access, which was added to the product in SQL Server 2005, is access through the version store. Row versioning allows SQL Server to maintain older versions of changed rows. The row-versioning technology in SQL Server supports snapshot isolation as well as other features of SQL Server 2012, including online index builds and triggers, and the versioning operations code maintains row versions for whatever purpose they are needed.

Chapters 3, 4, 6, 7, and 8 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. Also, 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 one millisecond after the commit was acknowledged. Transactions must adhere to four properties, called the ACID properties: *atomicity*, *consistency*, *isolation*, and *durability*. Chapter 13, "Transactions and concurrency," covers all four properties in a section on 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 possible the ability to always roll back work in progress or roll forward committed work that hasn't yet been applied to the data pages. Write-ahead logging ensures that the record of each transaction's changes is 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, topics discussed later in this book; this section looks just briefly at transactions themselves.

The transaction management component delineates the boundaries of statements that must be grouped to form an operation. It handles transactions that cross databases within the same SQL Server instance and 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. Also, 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* that you designate 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.

Concurrency models SQL Server 2012 supports two concurrency models for guaranteeing the ACID properties of transactions:

- **Pessimistic concurrency** This model guarantees correctness and consistency by locking data so that it can't be changed. Every version of SQL Server prior to SQL Server 2005 used this currency model exclusively; it's the default in both SQL Server 2005 and later versions.
- **Optimistic currency** SQL Server 2005 introduced 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 don't block writers and writers don't block readers, but writers still block writers. The cost of these non-blocking operations must be considered. To support optimistic concurrency, SQL Server needs to spend more time managing the version store. Administrators also have to pay close attention to the *tempdb* database and plan for the extra maintenance it requires.

Five isolation-level semantics are available in SQL Server 2012. Three of them support only pessimistic concurrency: Read Uncommitted, 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 10.

Locking operations Locking is a crucial function of a multiuser 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 don't block writers and writers don't 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 occurs 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.

Note Concurrency management, whether 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 13 is devoted to the subject, so this chapter won't go into further detail here.

Other operations

Also included in the storage engine are components for controlling utilities such as bulk-load, DBCC commands, full-text index population and management, and backup and restore operations. Chapter 14, "DBCC internals," covers DBCC in detail. The log manager makes sure that log records are written in a manner to guarantee transaction durability and recoverability. Chapter 5 goes into detail about the transaction log and its role in backup-and-restore operations.

SQL Server 2012 configuration

This half of the chapter looks at the options for controlling how SQL Server 2012 behaves. Some options might not mean much until you've read about the relevant components later in the book, but you can always come back and reread this section. One main method of controlling the behavior of the Database Engine is to adjust configuration option settings, but you can configure behavior in a few other ways as well. First, look at using SQL Server Configuration Manager to control network protocols and SQL Server–related services. Then, look at other machine settings that can affect the behavior of SQL Server. Finally, you can examine some specific configuration options for controlling server-wide settings in SQL Server.

Using SQL Server Configuration Manager

Configuration Manager is a tool for managing the services associated with SQL Server, configuring the network protocols used, and managing the network connectivity configuration from client computers connecting to SQL Server. Configuration Manager is accessed by selecting All Programs on the Windows Start menu, and then selecting Microsoft SQL Server 2012 | Configuration Tools | SQL Server Configuration Manager.

Configuring network protocols

A specific protocol must be enabled on both the client and the 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 TCP/IP) should already be installed on the client and the server. Network protocols are typically installed during Windows setup; they aren’t part of SQL Server setup. A SQL Server network library doesn’t 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 Client Tools Connectivity setup. The SQL Native Client is a standalone data-access application programming interface (API) used for both OLE DB and Open Database Connectivity (ODBC). If the SQL Native Client is available, you can configure any network protocol for use with a particular client connecting to SQL Server. 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 earlier in this chapter, it’s 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;
```

Implementing a default network configuration

The network protocols used to communicate with SQL Server 2012 from another computer aren’t 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 because it can be used to connect to the Database Engine only from a client application on the same computer, its usefulness is limited.

TCP/IP connectivity to SQL Server 2012 is disabled for new installations of the Developer, Evaluation, and SQL Express editions. OLE DB applications connecting with MDAC 2.8 can’t connect to the default instance on a local server using “.” (period), “(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 aren't 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 1-1 describes the default network configuration settings.

TABLE 1-1 SQL Server 2012 default network configuration settings

SQL Server edition	Type of installation	Shared memory	TCP/IP	Named pipes
Enterprise	New	Enabled	Enabled	Disabled (available only locally)
Enterprise (clustered)	New	Enabled	Enabled	Enabled
Developer	New	Enabled	Disabled	Disabled (available only locally)
Standard	New	Enabled	Enabled	Disabled (available only locally)
Workgroup	New	Enabled	Enabled	Disabled (available only locally)
Evaluation	New	Enabled	Disabled	Disabled (available only locally)
Web	New	Enabled	Enabled	Disabled (available only locally)
SQL Server Express	New	Enabled	Disabled	Disabled (available only locally)
All editions	Upgrade or side-by-side installation	Enabled	Settings preserved from the previous installation	Settings preserved from the previous installation

Managing services

You can use Configuration Manager to start, pause, resume, or stop SQL Server–related services. The services available depend on the specific components of SQL Server you have installed, but you should always have 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, rather than the Windows service management tools, is the preferred tool for changing service properties. 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 tool automatically makes additional configurations, such as setting permissions in the Windows Registry so that the new account can read the SQL Server settings. Password changes using Configuration Manager take effect immediately without requiring you to restart the service.

SQL Server Browser

One related service that deserves special attention is the SQL Server Browser service, particularly important if you have named instances of SQL Server running on a machine. SQL Server Browser listens for requests to access SQL Server resources and provides information about the various SQL Server instances installed on the computer where the Browser service is running.

Prior to SQL Server 2000, only one installation of SQL Server could be on a machine at one time, and the concept of an "instance" really didn't exist. SQL Server always listened for incoming requests on port 1433, but any port can be used by only one connection at a time. When SQL Server 2000 introduced support for multiple instances of SQL Server, a new protocol called *SQL Server Resolution Protocol (SSRP)* was developed to listen on UDP port 1434. This listener could reply to clients with the names of installed SQL Server instances, along with the port numbers or named pipes used by the instance. SQL Server 2005 replaced SSRP with the SQL Server Browser service, which is still used in SQL Server 2012.

If the SQL Server Browser service isn't running on a computer, you can't connect to SQL Server on that machine unless you provide the correct port number. Specifically, if the SQL Server Browser service isn't running, the following connections won't work:

- Connecting to a named instance without providing the port number or pipe
- Using the DAC to connect to a named instance or the default instance if it isn't using TCP/IP port 1433
- Enumerating servers in SQL Server Management Studio

You are recommended to have the Browser service set to start automatically on any machine on which SQL Server will be accessed using a network connection.

SQL Server system configuration

You can configure the machine that SQL Server runs on, as well as the Database Engine itself, in several ways and through various interfaces. First, look at some operating system–level settings that can affect the behavior of SQL Server. Next, you can see some SQL Server options that can affect behavior that aren't especially considered to be configuration options. Finally, you can examine the configuration options for controlling the behavior of SQL Server 2012, which are set primarily using a stored procedure interface called *sp_configure*.

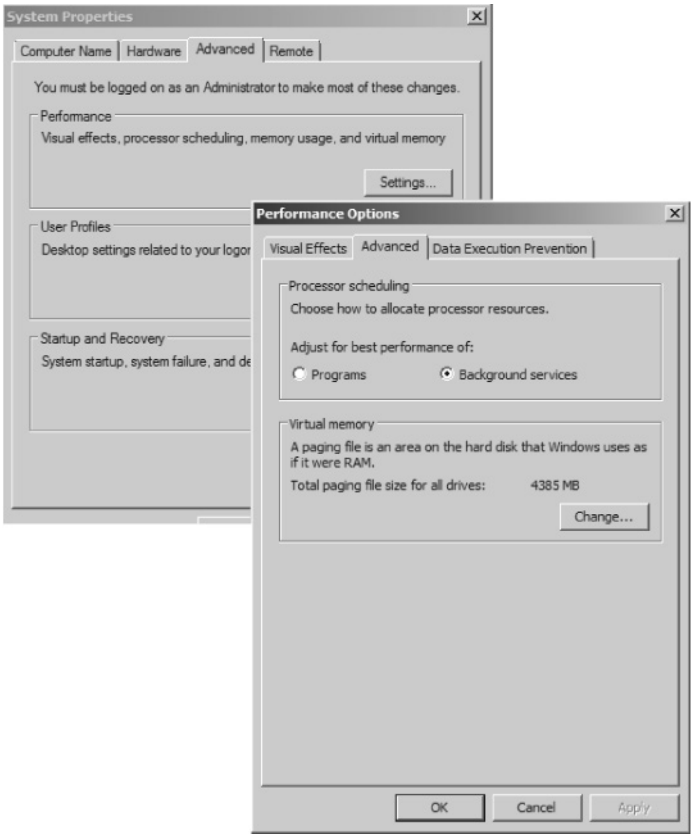
Operating system configuration

For your SQL Server to run well, it must be running on a tuned operating system on a machine that has been properly configured to run SQL Server. Although discussing operating system and hardware configuration and tuning is beyond the scope of this book, a few issues are very straightforward but can have a major effect on the performance of SQL Server.

Task management

The Windows operating system schedules all threads in the system for execution. Each thread of every process has a priority, and the operating system executes the next available thread with the highest priority. By default, it gives active applications a higher priority, but this priority setting might not be appropriate for a server application running in the background, such as SQL Server 2012. To remedy this situation, the SQL Server installation program modifies the priority setting to eliminate the favoring of foreground applications.

Periodically double-checking this priority setting is a good idea, in case someone has set it back. You'll need to use the Advanced tab in the Performance Options dialog box. If you're using Windows Server 2008 or Windows 7, click the Start menu, right-click Computer, and choose Properties. In the System information screen, select Advanced System Settings from the list on the left to open the System Properties sheet. Click the Settings button in the Performance section and then select the Advanced tab. Figure 1-3 shows the Performance Options dialog box.


System Properties

Computer Name | Hardware | Advanced | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles
Desktop settings related to your logor

Startup and Recovery
System startup, system failure, and de

Performance Options

Visual Effects | Advanced | Data Execution Prevention

Processor scheduling
Choose how to allocate processor resources.

Adjust for best performance of:
Programs [✓ Background services]

Virtual memory
A paging file is an area on the hard disk that Windows uses as if it were RAM.
Total paging file size for all drives: 4385 MB

Change...

OK | Cancel | Apply

FIGURE 1-3 Configuration of priority for background services.

The first set of options specifies how to allocate processor resources, and you can adjust for the best performance of programs or background services. Select Background Services so that all programs (background and foreground) receive equal processor resources. If you plan to connect to SQL Server 2012 from a local client—that is, a client running on the same computer as the server—you can use this setting to improve processing time.

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, a 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, such a small amount of page-file activity will occur that the file's location is irrelevant.

Nonessential services

You should disable any services that you don't need. In Windows Server 2008, you can click the Start menu, right-click Computer, and choose Manage. Expand the Services and Applications node in the Computer Management tool, and click Services. In the right-hand pane is a list of all 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 runs SQL Server as a domain controller, the group's file or print server, the Web server, or the Dynamic Host Configuration Protocol (DHCP) server.

Connectivity

You should run only the network protocols that you actually need for connectivity. You can use the SQL Server Configuration Manager to disable unneeded protocols, as described earlier in this chapter.

Firewall setting

Improper firewall settings are another system configuration issue that can inhibit SQL Server connectivity across your network. Firewall systems help prevent unauthorized access to computer resources and are usually desirable, but to access an instance of SQL Server through a firewall, you'll need to configure the firewall on the computer running SQL Server to allow access. Many firewall systems are available, and you'll need to check the documentation for your system for the exact details of how to configure it. In general, you need to follow these steps:

1. Configure the SQL Server instance to use a specific TCP/IP port. Your default SQL Server uses port 1433 by default, but you can change that. Named instances use dynamic ports by default, but you can also change that through the SQL Server Configuration Manager.
2. Configure your firewall to allow access to the specific port for authorized users or computers.
3. As an alternative to configuring SQL Server to listen on a specific port and then opening that port, you can list the SQL Server executable (Sqlservr.exe) and the SQL Browser executable (Sqlbrowser.exe) when requiring a connection to named instances as exceptions to the blocked programs. You can use this method when you want to continue to use dynamic ports.

Trace flags

SQL Server Books Online lists only 17 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. Many dozens, if not hundreds, of trace flags exist, but 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 toggle trace flags on or off by using the *DBCC TRACEON* and *DBCC TRACEOFF* commands 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 *SQL Server Books Online*.) Trace flags enabled with *DBCC TRACEON* are valid only for a single connection unless you specified an additional parameter of *–1*, in which case they are 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 specific ones are discussed with topics they are related to.

Caution Because trace flags change the way SQL Server behaves, they can actually cause trouble if used inappropriately. Trace flags aren't harmless features that you can experiment with just to see what happens, especially 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 whether 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 dynamically adjusts 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 designing, indexing, query tuning, and other such activities, which is covered 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 2012 has 69 server configuration options that you can query, using the catalog view *sys.configurations*.

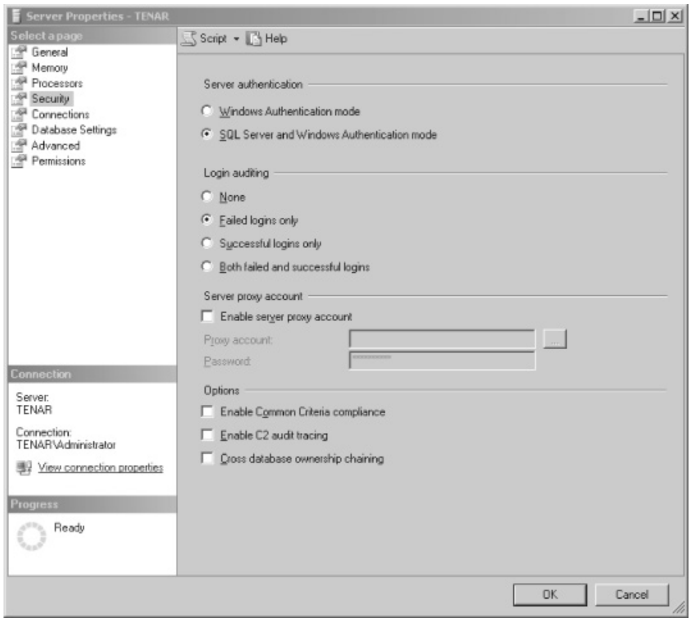
You should change configuration options only when you have a clear reason for doing so and closely monitor the effects of each change to determine whether the change improved or degraded performance. Always make and monitor changes one at a time. The server-wide options discussed here can be changed in several ways. All of them can be set via the *sp_configure* system stored procedure. However, of the 69 options, all but 17 are considered advanced options and aren't manageable by default using *sp_configure*. You first need to change the *show advanced options* setting to be *1*:

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

To see which options are advanced, you can query the *sys.configurations* view and examine a column called *is_advanced*, which lets you see which options are considered advanced:

```
SELECT * FROM sys.configurations
WHERE is_advanced = 1;
GO
```

You also can set many configuration options from the Server Properties sheet in the Object Explorer pane of SQL Server Management Studio, but you can't see or change all configuration settings from just one dialog box or window. Most of the options that you can change from the Server Properties sheet are controlled from one of the property pages that you reach by right-clicking the name of your SQL Server instance in Management Studio. You can see the list of property pages in Figure 1-4.



Server authentication
- Windows Authentication mode
- [✓ SQL Server and Windows Authentication mode]

Login auditing
- None
- [✓ Failed logins only]
- Successful logins only
- Both failed and successful logins

Server proxy account
- Enable server proxy account
- Proxy account:
- Password: **********

Options
- Enable Common Criteria compliance
- Enable C2 audit tracing
- Cross database ownership chaining

Connection
Server:
TENAR
Connection:
TENAR\Administrator
View connection properties

Progress
Ready

FIGURE 1-4 List of server property pages in SQL Server Management Studio.

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 the recommended range. Dynamic changes take effect immediately on reconfiguration, but others don't take effect until the server is restarted. If after running *RECONFIGURE* an option's *run_value* and *config_value* as displayed by *sp_configure* are different, or if the *value* and *value_in_use* in *sys.configurations* are different, you must restart the SQL Server service for the new value to take effect. You can use the *sys.configurations* view to determine which options are dynamic:

```
SELECT * FROM sys.configurations
WHERE is_dynamic = 1;
GO
```

This chapter doesn't look at every configuration option here—only the most interesting ones or ones related to SQL Server performance. In most cases, options that you shouldn't change are discussed. Some of these are resource settings that relate to performance only in that they consume memory, but if they are configured too high, they can rob a system of memory and degrade performance. Configuration settings are grouped by functionality. Keep in mind that SQL Server sets almost all these options automatically, and your applications can work well without you ever looking at them.

Memory options

Memory management involves a lot more than can be described in a few short paragraphs, and for the most part, you can do little to control how SQL Server uses the available memory. Chapter 2 goes into a lot more detail on how SQL Server manages memory, so this section will mention just the configuration options that deal directly with memory usage.

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 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 is actually the largest value that can be stored in the integer field of the underlying system table. It's not related to the actual resources of your system.

The Min Server Memory option doesn't force SQL Server to acquire a minimum amount of memory at startup. Memory is allocated on demand based on the database workload. However, when the Min Server Memory threshold is reached, SQL Server does not release memory if it would be left with less than that amount. To ensure that each instance has allocated memory at least equal to the Min Server Memory value, therefore, you might consider executing a database server load shortly after startup. During normal server activity, the memory available per instance varies, but each instance never has less than the Min Server Memory value.

Set working set size This configuration option is from earlier versions and has been deprecated. It's ignored in SQL Server 2012, even though you don't receive an error message when you try to use this value.

User connections SQL Server 2012 dynamically adjusts the number of simultaneous connections to the server if this configuration setting is left at its default of 0. Even if you set this value to a different number, SQL Server doesn't actually allocate the full amount of memory needed for each user connection until a user actually connects. When SQL Server starts, it allocates an array of pointers with as many entries as the configured value for User Connections.

If you must use this option, don't set the value too high because each connection takes approximately 28 KB of overhead whether or not 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 can't connect until another connection becomes available. (The exception is the

DAC connection, which can always be used.) Keep in mind that the User Connections value isn't 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 This configuration option is a setting from earlier versions and has been deprecated. SQL Server 2012 ignores this setting, even though you don't receive an error message when you try to use this value.

Scheduling options

As you will see in detail in Chapter 2, SQL Server 2012 has a special algorithm for scheduling user processes using the SQLOS, which manages one scheduler per logical processor and ensures that only one process can run on a scheduler at any specific time. The SQLOS manages the assignment of user connections to workers to keep the number of users per 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.

Lightweight Pooling By default, SQL Server operates in thread mode, which means that the workers processing SQL Server requests are threads. As 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 can yield a minor performance advantage, particularly when you have eight or more CPUs and all 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 offsets any benefit of operating in fiber mode.

If you're running in an environment that uses a high percentage of total CPU resources, and if System Monitor shows a lot of context switching, setting Lightweight Pooling to 1 might yield some performance benefit.

Max Worker Threads SQL Server uses the operating system's thread services by keeping a pool of workers (threads or fibers) that take requests from the queue. It attempts to divide the worker threads evenly 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. Having 100 or fewer users means having usually as many worker threads as active users (not just connected users who are idle). With more users, having fewer worker threads than active users often makes sense. 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 is configured by SQL Server, based on the number of processors and machine architecture. For example, for a four-way 32-bit machine running SQL Server, the default is 256 workers. This doesn't 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 now below 256. If, for example, this setting is configured to 256 and the highest number of simultaneously executing commands is 125, the actual number of workers won't exceed 125. It might be even smaller 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 won't be greater than 100.

Table 1-2 lists the default number of workers, considering your machine architecture and number of processors. (Note that Microsoft recommends 1,024 as the maximum for 32-bit operating systems.)

TABLE 1-2 Default settings for Max Worker Threads

CPU	32-bit computer	64-bit computer
Up to 4 processors	256	512
8 processors	288	576
16 processors	352	704
32 processors	480	960

Even systems that handle 5,000 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 the disk read behavior of SQL Server. All 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; it controls how frequently the checkpoint process writes to disk.

Recovery interval This option can be configured automatically. SQL Server setup sets it to 0, which means autoconfiguration. In SQL Server 2012, this means that the recovery time should be less than one minute.

This option lets database administrators 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 doesn't 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 used primarily for read operations won't have many checkpoints issued. To avoid excessive checkpoints, SQL Server tries to ensure that the value set for the recovery interval is the minimum amount of time between successive checkpoints.

SQL Server provides a new feature called *indirect checkpoints* that allow the configuration of checkpoint frequency at the database level using a database option called *TARGET_RECOVERY_TIME*. Chapter 3 discusses this option, and Chapter 5 discusses both the server-wide option and the database setting in the sections about checkpoints and recovery. As you'll see, 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 are still written to the disk in a timely manner. For this reason, you should keep the checkpoint options at their default values.

Affinity I/O Mask and Affinity64 I/O Mask These two options control the 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 bitmasks means that the corresponding processor is used only for I/O operations.

You'll probably never need to set these options. However, if you do decide to use them, perhaps just for testing purposes, you should do so with the Affinity Mask or Affinity64 Mask option and make sure that the bit sets don't overlap. You should thus have one of the following combinations of settings: 0 for both Affinity I/O Mask and Affinity Mask for a CPU, 1 for the Affinity I/O Mask option and 0 for Affinity Mask, or 0 for Affinity I/O Mask and 1 for Affinity Mask.

Backup Compression DEFAULT SQL Server 2008 added Backup Compression as a new feature, and for backward compatibility the default value is 0, meaning that backups aren't compressed. Although only Enterprise edition instances can create a compressed backup, any edition of SQL Server 2012 can restore a compressed backup. When Backup Compression is enabled, the compression is performed on the server before writing, so it can greatly reduce the size of the backups and the I/O required to write the backups to the external device. The amount of space reduction depends on many factors, including the following.

- **The type of data in the backup** For example, character data compresses more than other types of data.
- **Whether the data is encrypted** Encrypted data compresses significantly less than equivalent unencrypted data. If transparent data encryption is used to encrypt an entire database, compressing backups might not reduce their size by much, if at all.

After the backup is performed, you can inspect the *backupset* table in the *msdb* database to determine the compression ratio, using a statement like the following:

```
SELECT backup_size/compressed_backup_size FROM msdb..backupset;
```

Although compressed backups can use significantly fewer I/O resources, it also can significantly increase CPU usage when performing the compression. This additional load can affect other operations occurring concurrently. To minimize this effect, you can consider using the Resource Governor to create a workload group for sessions performing backups and assign the group to a resource pool with a limit on its maximum CPU utilization.

The configured value is the instance-wide default for Backup Compression, but it can be overridden for a particular backup operation by specifying *WITH COMPRESSION* or *WITH NO_COMPRESSION*. You can use compression for any type of backup: full, log, differential, or partial (file or filegroup).

Note The algorithm used for compressing backups varies greatly from the database compression algorithms. Backup Compression uses an algorithm very similar to zip, where it's just looking for patterns in the data. Chapter 8 discusses data compression.

Filestream access level This option integrates the Database Engine with your NTFS file system by storing binary large object (BLOB) data as files on the file system and allowing you to access this data either using T-SQL or Win32 file system interfaces to provide streaming access to the data. Filestream uses the Windows system cache for caching file data to help reduce any effect that filestream data might have on SQL Server performance. The SQL Server buffer pool isn't used so that filestream doesn't reduce the memory available for query processing.

Before setting this configuration option to indicate the access level for filestream data, you must enable Filestream externally using the SQL Server Configuration Manager (if you haven't enabled Filestream during SQL Server setup). In the SQL Server Configuration Manager, right-click the name of the SQL Server service and choose Properties. The properties sheet has a separate tab for Filestream options. You must check the top box to enable Filestream for T-SQL access, and then you can choose to enable Filestream for file I/O streaming if you want.

After enabling Filestream for your SQL Server instance, you then set the configuration value. The following values are allowed:

- ***0 Disables FILESTREAM*** support for this instance
- ***1 Enables FILESTREAM*** for T-SQL access
- ***2 Enables FILESTREAM*** for T-SQL and Win32 streaming access

Databases that store filestream data must have a special filestream filegroup. Chapter 3 discusses filegroups; Chapter 8 provides more details about filestream storage.

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 that it gets is determined partly by the this option. This option is relevant for sort operations that you specifically request using an *ORDER BY* clause; 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 very dynamically, 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 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 This option controls how long a query that needs additional memory waits if that memory isn't available. A setting of –1 means that the query waits 25 times the estimated execution time of the query, but it always waits at least 25 seconds with this setting. A value of 0 or more specifies the number of seconds that a query waits. 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 adding 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 aren't affected.

Blocked Process Threshold This option allows administrators 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 create an Extended Events session to capture events of type *blocked_process_report*. As long as a resource stays blocked on a deadlock-detectable resource, the event is raised every time the deadlock monitor checks for a deadlock. (Chapter 2 discusses Extended Events.)

Alternatively, you can use event notifications to send information about events to a service broker service. You also can use event notifications, which execute asynchronously, to perform an action inside a SQL Server 2012 instance in response to events, with very little consumption of memory resources. Because event notifications execute asynchronously, these actions don't 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 doesn't apply to the sorting that takes place during index creation. Another option, Index Create Memory, lets you allocate a specific amount of memory (in kilobytes) for index creation.

Query Governor Cost Limit You can use this option to specify the maximum number of seconds that a query can run. If you specify a non-zero, non-negative 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.

Note that the value set for seconds isn't clock-based. It corresponds to seconds on a specific hardware configuration used during product development, and the actual limit might be higher or lower on your machine.

Max Degree Of Parallelism and Cost Threshold For Parallelism SQL Server 2012 lets you run certain kinds of complex queries simultaneously on two or more processors. The queries must lend themselves to being executed in sections; the following is an example:

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

If the charge table has 1 million rows and 10 different values for *category*, SQL Server can split the rows into groups and have only a subset of them 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 Query 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. (As in the preceding section, this isn't an exact clock-based number of seconds.) If the cheapest serial plan costs more than this configured threshold, a parallel plan is produced based on assumptions about how many processors and how much memory will actually be available at runtime. 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, 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 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 running on multiple CPUs has one row for each thread, as follows:

```
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 affect the whole server.

Miscellaneous options Most of the other configuration options that haven't been mentioned 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. A Boolean option allows use of the Common Language Runtime (CLR) in programming SQL Server objects; it is off (0) by default.

A few configuration options deal with programming issues, which this book doesn't cover. These options include ones for dealing with recursive and nested triggers, cursors, and accessing objects across databases.

Conclusion

This chapter looked at the general workings of the SQL Server engine, including the key components and functional areas that make up the engine. By necessity, the chapter has been simplified somewhat, but the information should provide some insight into the roles and responsibilities of the major components in SQL Server and the interrelationships among components.

This chapter also covered the primary tools for changing the behavior of SQL Server. The primary means of changing the behavior is by using configuration options, so you saw the options that can have the biggest impact on SQL Server behavior, especially its performance. To really know when changing the behavior is a good idea, you must understand how and why SQL Server works the way it does. This chapter has laid the groundwork for you to make informed decisions about configuration changes.

CHAPTER 8

Special storage

Kalen Delaney

Earlier chapters discussed the storage of "regular rows" for both data and index information. (Chapter 7, "Indexes: internals and management," also looked at a completely different way of storing indexes: using columnstores, which aren't stored in rows at all.) Chapter 6, "Table storage," explained that regular rows are in a format called *FixedVar*. SQL Server provides ways of storing data in another format, called Column Descriptor (CD). It also can store special values in either the *FixedVar* or CD format that don't fit on the regular 8 KB pages.

This chapter describes data that exceeds the typical row size limitations and is stored as either row-overflow or Large Object (LOB) data. You'll learn about two additional methods for storing data on the actual data pages, introduced in Microsoft SQL Server 2008: one that uses a new type of complex column with a regular data row (sparse columns), and one that uses the new CD format (compressed data). This chapter also discusses FILESTREAM data, a feature introduced in SQL Server 2008 that allows you to access data from operating system files as though it were part of your relational tables, and FileTables, a new feature in SQL Server 2012 that allows you to create a table containing both FILESTREAM data and Windows file attribute metadata.

Finally, this chapter covers the ability of SQL Server to separate data into partitions. Although this doesn't change the data format in the rows or on the pages, it does change the metadata that keeps track of what space is allocated to which objects.

Large object storage

SQL Server 2012 has two special formats for storing data that doesn't fit on the regular 8 KB data page. These formats allow you to store rows that exceed the maximum row size of 8,060 bytes. As discussed in Chapter 6, 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, the error message that you get if you try to create a table with more bytes than the allowable maximum is very specific. 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 here:

```
USE testdb;
GO
CREATE TABLE dbo.bigrows_fixed
(   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 wants to store with the row itself. An additional 2 bytes is used for the row-offset information at the end of the page, but those bytes aren't included in this total.

Restricted-length large object data (row-overflow data)

One way to exceed this size limit of 8,060 bytes is to use variable-length columns because for variable-length data, SQL Server 2005 and later versions can store the columns in special row-overflow pages, as long as all the fixed-length columns fit into the regular in-row size limit. So you need to look at a table with all variable-length columns. Notice that although my example uses all *varchar* columns, columns of other data types can 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 whose maximum defined length is much longer than 8,060 bytes:

```
USE testdb;
CREATE TABLE dbo.bigrows
  (a varchar(3000),
   b varchar(3000),
   c varchar(3000),
   d varchar(3000) );
```

In fact, if you ran this *CREATE TABLE* statement in SQL Server 7.0, you would get an error, and the table wouldn't be created. In SQL Server 2000, the table was created, but you got a warning that inserts or updates might fail if the row size exceeds the maximum.

With SQL Server 2005 and later, not only could the preceding *dbo.bigrows* table be created, but you also could insert a row with column sizes that add up to more than 8,060 bytes with a simple *INSERT*:

```
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 following allocation query from Chapter 5, "Logging and recovery":

```
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 that 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 *sys.dm_db_database_page_allocations* function and see the four pages individually:

```
SELECT allocated_page_file_id as PageFID, allocated_page_page_id as PagePID,
       object_id as ObjectID, partition_id AS PartitionID,
       allocation_unit_type_desc as AU_type, page_type as PageType
FROM sys.dm_db_database_page_allocations
       (db_id('testdb'), object_id('bigrows'), null, null, 'DETAILED');
```

You should see the four rows, one for each page, looking similar to the following:

```
PageFID PagePID     ObjectID    PartitionID AU_type             PageType
------- ----------- ----------- ----------- ------------------- -----------
1       303         1653580929  1           IN_ROW_DATA         10
1       302         1653580929  1           IN_ROW_DATA         1
1       297         1653580929  1           ROW_OVERFLOW_DATA   10
1       296         1653580929  1           ROW_OVERFLOW_DATA   3
```

Of course, your actual ID values will be different, but the *AU*-type and *PageType* values should be the same, and you should have four rows returned indicating four pages belong to the *bigrows* table. Two pages are for the row-overflow data, and two are for the in-row data. As you saw in Chapter 7, the *PageType* values have the following meanings.

- *PageType* = 1, Data page
- *PageType* = 2, Index page
- *PageType* = 3, LOB or row-overflow page, *TEXT_MIXED*
- *PageType* = 4, LOB or row-overflow page, *TEXT_DATA*
- *PageType* = 10, IAM page

You learn more about the different types of LOB pages in the next section, “Unrestricted-length large object data.”

You can see 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 *sys.dm_db_database_page_allocations*, you could then look at the page contents with *DBCC PAGE*. On the data page for the in-row data, you 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 1:302 in the preceding output), notice that it isn’t necessarily the fourth column in the column order that is stored off the row. (I won’t show you the entire contents of the rows because the single row fills almost the entire page.) Look at the in-row data page using *DBCC PAGE* and notice the column with *e*, the column with *g*, and the column with *h*. The column with *f* has moved to the new row. In the place of that column, you can see the bytes shown here:

```
65020000 00010000 00c37f00 00340800 00280100 00010000 0067
```

Included are the last byte with *e* (ASCII code hexadecimal 65) and the first byte with *g* (ASCII code hexadecimal 67), and in between are 24 other bytes (boldfaced). Bytes 16 through 23 (the 17th through the 24th bytes) of those 24 bytes are treated as an 8-byte numeric value: 2801000001000000 (bold italic). You need to reverse the byte order and break it into a 2-byte hex value for the slot number, a 2-byte hex value for the file number, and a 4-byte hex value for the page number. So the slot number is 0x0000 for slot 0 because this overflowing column is the first (and only) data on the row-overflow page. You have 0x0001 (or 1) for the file number and 0x00000128 (or 296) for the page number. You saw these the same file and page numbers when using *sys.dm_db_database_page_allocations*.

Table 8-1 describes the first 16 bytes in the row.

TABLE 8-1 The first 16 bytes of a row-overflow pointer

Bytes	Hex value	Decimal value	Meaning
0	0x02	2	Type of special field: 1 = LOB2 = overflow
1–2	0x0000	0	Level in the B-tree (always 0 for overflow)
3	0x00	0	Unused
4–7	0x00000001	1	Sequence: a value used by optimistic concurrency control for cursors that increases every time a LOB or overflow column is updated
8–11	0x00007fc3	32707	Timestamp: a random value used by *DBCC CHECKTABLE* that remains unchanged during the lifetime of each LOB or overflow column
12–15	0x00000834	2100	Length

SQL Server stores variable-length columns on row-overflow pages only under certain conditions. The determining factor is the row length itself. How full the regular page is into which SQL Server is trying to insert the new row doesn’t matter; SQL Server constructs the row as usual and stores some of its columns on overflow pages only if the row itself needs more than 8,060 bytes.

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 can’t 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 the page on which SQL Server is trying to insert the row has no room, regular page-splitting algorithms (described 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 as follows:

```
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);
```

Substituting *hugerows* for *bigrows* for the allocation query shown earlier yields the following results:

```
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
```

The output shows four pages for the row-overflow information, one for the row-overflow 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 isn't unlimited, although it is quite large. A table is limited to 1,024 columns, which can be exceeded when you are using sparse columns, as discussed later in this chapter. However, another limit is reached before that. When a column must 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, which you saw in the *DBCC* output earlier as 24 bytes, and the row still needs 2 bytes in the column-offset array for each variable-length column, whether or not the variable-length column is stored in the row. So 308 turns out to be the maximum number of overflowing columns you 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 many large columns on row-overflow pages doesn't mean that doing so is always a good idea. 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 fit completely on your data pages and you have row-overflow data only occasionally. By 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, if the decrease is just a few bytes, SQL Server doesn't bother checking. Only when a column stored in a row-overflow page is reduced by more than 1,000 bytes does 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 and reduces the row size to 7,900 bytes, which should all fit on one data page:

```
UPDATE bigrows
SET a = replicate('a', 1600);
```

However, if you rerun the allocation query, you'll still see two row-overflow pages: one for the row-overflow data and one for the IAM page. Now reduce the size of the first column by more than 1,000 bytes and rerun the allocation query:

```
UPDATE bigrows
SET a = 'aaaaa';
```

You should see only three pages for the table now, because there is no longer a row-overflow data page. The IAM page for the row-overflow data pages hasn't been removed, but you no longer have a data page for row-overflow data.

Keep in mind that row-overflow data storage applies only to columns of variable-length data, which are defined as no longer than the usual variable-length maximum of 8,000 bytes per column. Also, 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 8,060 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 LOB (*text*, *image*, or *ntext*) columns or the *MAX* data types.

Unrestricted-length large object data

If a table contains the deprecated LOB data types (*text*, *ntext*, or *image* types), by default the actual data isn't 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 allows you to change the storage mechanism by setting a table option to allow LOB data to be stored in the data row itself if it is small enough. Note that no database or server setting is available to control storing small LOB columns on the data pages; it's managed as a table option.

The 16-byte pointer points 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 storing 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. (Even more specifically, one *text* or *image* page can hold only *text* or *image* data from a single partition of a table, which should become clear when partitioning metadata is discussed at the end of this chapter.)

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 varies slightly depending on whether the amount of data is less than or more than 32 KB. (See Figure 8-1 for the general structure.) B-trees were discussed in detail when describing indexes in Chapter 7.

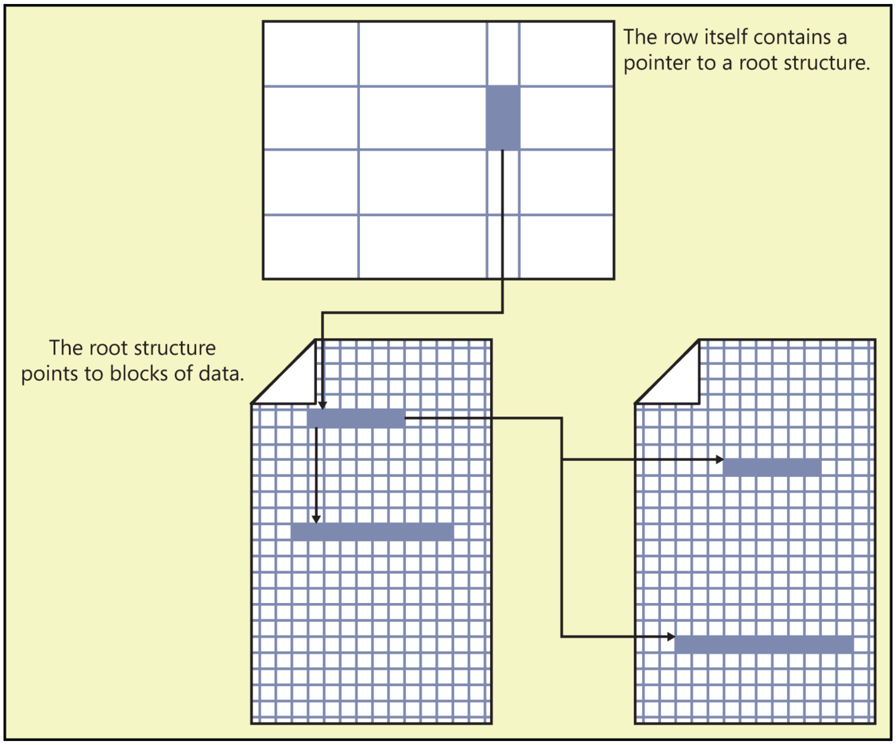

The row itself contains a pointer to a root structure.
The root structure points to blocks of data.

FIGURE 8-1 A text column pointing to a B-tree that contains the blocks of data.

Note Although the acronym LOB can be expanded to mean "large object," these two terms will be used in this chapter to mean two different things. LOB is used only when referring to the data using the special storage format shown in Figure 8-1. The term *large object* is used when referring to any method for storing data that might be too large for a regular data page. This includes row-overflow columns, the actual LOB data types, the *MAX* data types, and FILESTREAM data.

If the amount of LOB 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. Although the data for LOB columns is arranged logically in a B-tree, both the root node and the individual blocks of data are spread physically 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.

SQL Server can store the LOB root and the actual LOB data on two different types of pages. One of these, referred to as *TEXT_MIXED*, allows LOB data from multiple rows to share the same pages. However, when your text data gets larger than about 40 KB, SQL Server starts devoting whole pages to a single LOB value. These pages are referred to as *TEXT_DATA* pages.

You can see this behavior by creating a table with a *text* column, inserting a value of less than 40 KB and then one greater than 40 KB, and finally examining information returned by *sys.dm_db_database_page_allocations* (see Listing 8-1).

LISTING 8-1 Storing LOB data on two types of pages

```
IF OBJECT_ID('textdata') IS NOT NULL
    DROP TABLE textdata;
GO
CREATE TABLE textdata
 (bigcol text);
GO
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 38000);
GO
SELECT allocated_page_file_id as PageFID, allocated_page_page_id as PagePID,
       object_id as ObjectID, partition_id AS PartitionID,
       allocation_unit_type_desc as AU_type, page_type as PageType
FROM sys.dm_db_database_page_allocations(db_id('testdb'), object_id('textdata'),
                                          null, null, 'DETAILED');

GO
```

```
INSERT INTO textdata
   SELECT REPLICATE(convert(varchar(MAX), 'a'), 41000);
GO
SELECT allocated_page_file_id as PageFID, allocated_page_page_id as PagePID,
       object_id as ObjectID, partition_id AS PartitionID,
       allocation_unit_type_desc as AU_type, page_type as PageType
FROM sys.dm_db_database_page_allocations(db_id('testdb'), object_id('textdata'),
                                         null, null, 'DETAILED');
GO
```

The *INSERT* statements in Listing 8-1 convert a string value into the data type *varchar(MAX)* because this is the only way to generate a string value longer than 8,000 bytes. (The next section discusses *varchar(MAX)* in more detail.) The first time you select from *sys.dm_db_database_page_allocations*, you should have *PageType* values of 1, 3, and 10. The second time after data greater than 40 KB in size is inserted, you should also see *PageType* values of 4. *PageType* 3 indicates a *TEXT_MIXED* page, and *PageType* 4 indicates a *TEXT_DATA* page.

Storing LOB data in the data row

If you store all your LOB data type values outside your regular data pages, SQL Server needs to perform additional page reads every time you access that data, just as 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'* (including the single quotation marks) 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*:

```
EXEC sp_tableoption employee, 'text in row', 500;
```

Notice that the value is in bytes, not characters. For *ntext* data, each character needs 2 bytes so that any *ntext* column is stored in the data row if it's less than or equal to 250 characters. When 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 isn't *'ON'*. If the data in the LOB field is more than the specified maximum, the row holds 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. (If you specify the option *'ON'* instead of a specific number, SQL Server assumes the default value of 256 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 inspect the *sys.tables* catalog view as follows:

```
SELECT name, text_in_row_limit
FROM sys.tables
WHERE name = 'employee';
```

This *text_in_row_limit* value indicates the maximum number of bytes allowed for storing LOB data in a data row. If a 0 is returned, the *text in row* option is disabled.

Now create a table very similar to the one that looks at row structures, but change the *varchar(250)* column to the *text* data type. You'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 use the allocation query to find the basic information for this table and look at the *sys.dm_db_database_page_allocations* information for this table (see Listing 8-2).

LISTING 8-2 Finding basic information for the HasText table

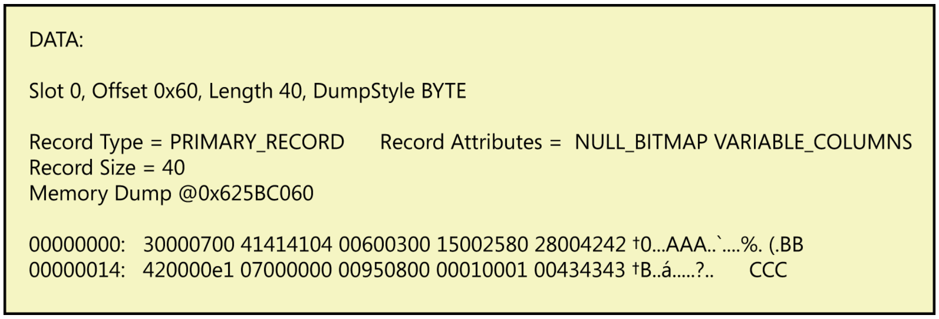
```
SELECT convert(char(7), object_name(object_id))  AS name,
     partition_id, partition_number AS pnum,  rows,
     allocation_unit_id AS au_id, convert(char(17), type_desc) as page_type_desc,
    total_pages AS pages
FROM sys.partitions p  JOIN sys.allocation_units a
   ON p.partition_id = a.container_id
WHERE object_id=object_id('dbo.HasText');

SELECT allocated_page_file_id as PageFID, allocated_page_page_id as PagePID,
       object_id as ObjectID, partition_id AS PartitionID, allocation_unit_type_desc as AU_Type,
       page_type as PageType
FROM sys.dm_db_database_page_allocations(db_id('testdb'),
       object_id('textdata'), null, null, 'DETAILED')

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        AU_Type         PageType
-------  -------  --------   -----------------  --------------  --------
1        2197     133575514  72057594039435264  LOB data        3
1        2198     133575514  72057594039435264  LOB data        10
1        2199     133575514  72057594039435264  In-row data     1
1        2200     133575514  72057594039435264  In-row data     10
```

You can see 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 2199, and the LOB data is on page 2197. The following output shows the data section from running *DBCC PAGE* on page 2199. The row structure is very similar to the row structure shown in Chapter 6, in Figure 6-6, except for the text field itself. Bytes 21 to 36 are the 16-byte text pointer, and you can see the value 9508 starting at offset 29. When the bytes are reversed, it becomes 0x0895, or 2197 decimal, which is the page containing the text data, as you saw in the output in Listing 8-2.



```
DATA:

Slot 0, Offset 0x60, Length 40, DumpStyle BYTE

Record Type = PRIMARY_RECORD      Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 40
Memory Dump @0x625BC060

00000000:   30000700 41414104 00600300 15002580 28004242 †0...AAA..`....%. (.BB
00000014:   420000e1 07000000 00950800 00010001 00434343 †B..á.....?..      CCC
```

FIGURE 8-2 A row containing a text pointer.

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 doesn't force the text data to be moved into the row. You have to update the text value to actually force the data movement:

```
UPDATE HasText
SET col3 = REPLICATE('Z', 250);
```

If you run *DBCC PAGE* on the original data page, notice that the text column of 250 *z*'s is now in the data row, and that the row is practically identical in structure to the row containing *varchar* data that you saw in Figure 6-6.

Note Although enabling *text in row* doesn't 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 be sure not to 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 dealing with the situation in which *text in row* is enabled but the LOB is longer than the maximum configured length for some rows. If you change the maximum length for *text in row* to 50 for the *HasText* table you'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, setting the limit to a smaller value is different from 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 is stored completely in the data row. Second, if the LOB data doesn't fit, the information stored in the data row itself isn't simply the 16-byte pointer, as it would be if *text in row* were turned off. Instead, for LOB data that doesn't fit in the defined size, the row contains a root structure for a B-tree that points to chunks of the LOB data. As long as the *text in row* option isn't *'OFF'* (or 0), SQL Server never stores the simple 16-byte LOB pointer in the row. It stores 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 the *text in row* limit), and the meaning of the bytes is similar to the meaning of the 24 bytes in the row-overflow pointer. The main difference is that no length is stored in bytes 12–15. Instead, bytes 12–23 constitute a link to a chunk of LOB data on a separate page. If multiple LOB chucks are accessed via the root, multiple sets of 12 bytes can be here, each pointing to LOB data on a separate page.

As indicated earlier, when you first enable *text in row*, no data movement occurs until the 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 the LOB data that was stored outside the row, the LOB data isn't moved onto the row automatically. You must update the actual LOB data first.

Keep in mind that even if the amount of LOB data is less than the limit, the data isn't necessarily 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. Also, 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 checks 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* can't always be manipulated with the usual data-manipulation commands and, in many cases, you 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 to deal with your large data columns by using different operators than you used for your shorter columns. Starting with version 2005, SQL Server provides a solution that gives you the best of both worlds, as you'll see in the next section.

Storing MAX-length data

SQL Server 2005 and later versions give you the option of defining a variable-length field with 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 use one of these types to define a column, variable, or parameter. 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 is treated as though it were one of the regular variable-length data types, including possibly overflowing onto row-overflow pages. However, if the *varchar(MAX)* column does need to spill off the page, the extra pages required are considered LOB pages and show the *IAM_chain_type* LOB when examined using *DBCC IND*. If the actual length is greater than 8,000 bytes, SQL Server stores and treats the value exactly as though 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.

The size of values specified with *MAX* can reach the maximum size supported by LOB data, which is currently 2 GB. By using the *MAX* specifier, however, you are indicating that the maximum size should be the maximum the system supports. If you upgrade a table with a *varchar(MAX)* column to a future version of SQL Server, the MAX length becomes whatever the new maximum is in the new version.

Tip Because the *MAX* data types can store LOB data as well as regular row data, you are recommended to use these data types in future development in place of the *text*, *ntext*, or *image* types, which Microsoft has indicated will be removed in a future version.

Appending data into a LOB column

In the storage engine, each LOB column is broken into fragments of a maximum size of 8,040 bytes each. When you append data to a large object, SQL Server finds the append point and looks at the current fragment where the new data will be added. It calculates the size of the new fragment (including the newly appended data). If the size is more than 8,040 bytes, SQL Server allocates new large object pages until a fragment is left that is less than 8,040 bytes, and then it finds a page that has enough space for the remaining bytes.

When SQL Server allocates pages for LOB data, it has two allocation strategies:

- For data that is less than 64 KB in size, it randomly allocates a page. This page comes from an extent that is part of the large object IAM, but the pages aren't guaranteed to be continuous.
- For data that is more than 64 KB in size, it uses an append-only page allocator that allocates one extent at a time and writes the pages continuously in the extent.

From a performance standpoint, writing fragments of 64 KB at a time is beneficial. Allocating 1 MB in advance might be beneficial if you know that the size will be 1 MB, but you also need to take into account the space required for the transaction log. If you a create a 1 MB fragment first with any random contents, SQL Server logs the 1 MB, and then all the changes are logged as well. When you perform large object data updates, no new pages need to be allocated, but the changes still need to be logged.

As long as the large object values are small, they can be in the data page. In this case, some preallocation might be a good idea so that the large object data doesn't become too fragmented. A general recommendation might be that if the amount of data to be inserted into a large object column in a single operation is relatively small, you should insert a large object value of the final expected value, and then replace substrings of that initial value as needed. For larger sizes, try to append or insert in chunks of 8 * 8,040 bytes. This way, a whole extent is allocated each time, and 8,040 bytes are stored on each page.

If you do find that your large object data is becoming fragmented, you can use *ALTER INDEX REORGANIZE* to defragment that data. In fact, this option (*WITH LOB_COMPACTION*) is on by default, so you just need to make sure that you don't set it to *'OFF'*.

FILESTREAM and FileTable data

Although the flexible methods that SQL Server uses to store large object data in the database give you many advantages over data stored in the file system, they also have many disadvantages. Some of the benefits of storing large objects in your database include the following.

- Transactional consistency of your large object data can be guaranteed.
- Your backup and restore operations include the large object data, allowing you integrated, point-in-time recovery of your large objects.
- All data can be stored using a single storage and query environment.

Some of the disadvantages of storing large objects in your database include the following.

- Large objects can require a very large number of buffers in cache.
- The upper limit on the size of any large object value is 2 GB.
- Updating large objects can cause extensive database fragmentation.
- Database files can become extremely large.
- Read or write streaming operations from *varchar(MAX)* and *varbinary(MAX)* columns are significantly slower than streaming from NTFS files.

SQL Server allows you to manage file system objects as though they were part of your database to provide the benefits of having large objects in the database while minimizing the disadvantages. The data stored in the file system can be FILESTREAM or FileTable data. As you start evaluating whether FILESTREAM or FileTable data is beneficial for your applications, consider both the benefits and the drawbacks. Some benefits of both FILESTREAM and FileTable data include the following.

- The large object data is stored in the file system but rooted in the database as a 48-byte file pointer value in the column that contains the FILESTREAM data.
- The large object data is accessible through both Transact-SQL (T-SQL) and the NTFS streaming APIs, which can provide great performance benefits.
- The large object size is limited only by the NTFS volume size, not the old 2 GB limit for large object data stored within a database.

FILESTREAM data has the following additional benefits.

- The large object data is kept transactionally consistent with structured data.
- Databases containing FILESTREAM data can participate in the SQL Server 2012 AlwaysOn availability groups.

FileTable data has these additional benefits.

- The data is available through any Wind32 application without any modification of the application.
- FileTables allow you to support a hierarchy of directories and files.

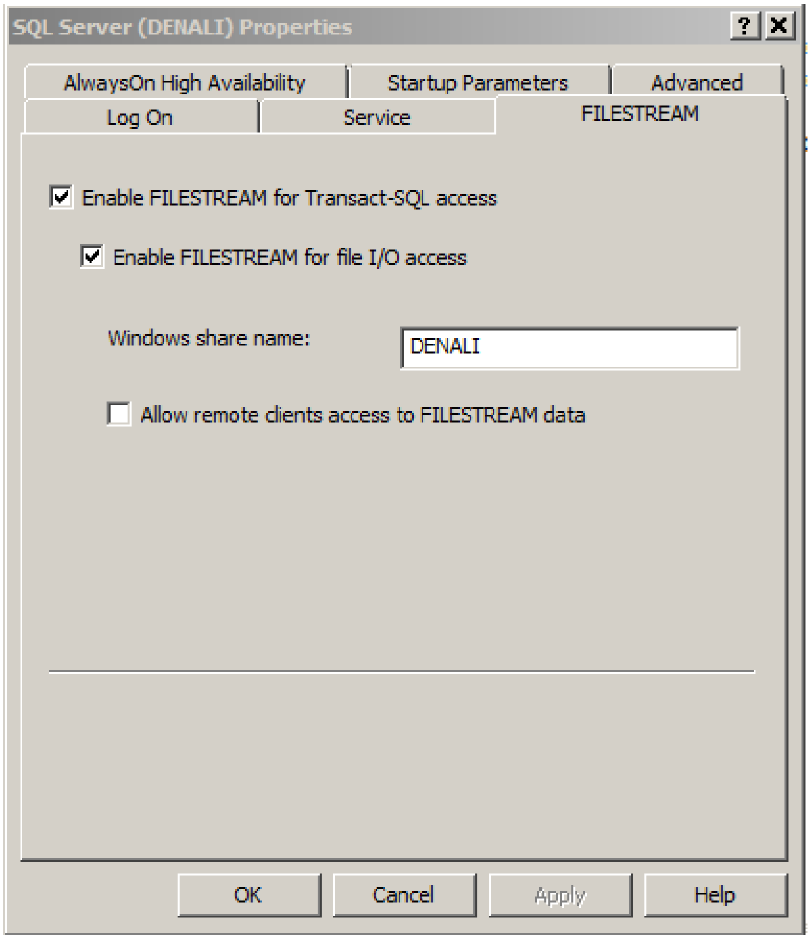
Some drawbacks of using FILESTREAM or FileTable data include the following.

- Database snapshots can't include the FILESTREAM filegroups, so the FILESTREAM data is unavailable. A *SELECT* statement in a database snapshot that requests a FILESTREAM column generates an error.
- SQL Server can't encrypt FILESTREAM data natively.

Because the FileTable feature is built on top of the FILESTREAM technology, I'll first tell you about FILESTREAM (which was introduced in SQL Server 2008) and then about what has been added in SQL Server 2012 to enable FileTables.

Enabling FILESTREAM data for SQL Server

The capability to access FILESTREAM data must be enabled both outside and inside your SQL Server instance, as mentioned in Chapter 1 when discussing configuration. Through the SQL Server Configuration Manager, you must enable T-SQL access to FILESTREAM data, and if that has been enabled, you can also enable file I/O streaming access. If file I/O streaming access is allowed, you can allow remote clients to have access to the streaming data if you want. When the SQL Server Configuration Manager is opened, make sure that you have selected SQL Server Services in the left pane. In the right pane, right-click the SQL instance that you want to configure and select Properties from the drop-down menu. The Properties sheet has six tabs, including one labeled *FILESTREAM*. You can see the details of the FILESTREAM tab of the SQL Server Properties sheet in Figure 8-3.



☑ Enable FILESTREAM for Transact-SQL access

☑ Enable FILESTREAM for file I/O access

Windows share name: DENALI

☐ Allow remote clients access to FILESTREAM data

FIGURE 8-3 Configuring a SQL Server instance to allow FILESTREAM access.

After the server instance is configured, you need to use *sp_configure* to set your SQL Server instance to the level of FILESTREAM access that you require. Three values are possible.

- 0 (the default) means that no FILESTREAM access is allowed.
- 1 means that you can use T-SQL to access FILESTREAM data.
- 2 means that you can use both T-SQL and the Win32 API for FILESTREAM access.

As with all configuration options, don't forget to run the *RECONFIGURE* command after changing a setting:

```
EXEC sp_configure 'filestream access level', 2; RECONFIGURE;
```

Creating a FILESTREAM-enabled database

To store FILESTREAM data, a database must have at least one filegroup that was created to allow FILESTREAM data. When creating a database, a filegroup that allows FILESTREAM data is specified differently from a filegroup containing row data in several different ways.

- The path specified for the FILESTREAM filegroup must exist only up to the last folder name. The last folder name must not exist but is created when SQL Server creates the database.
- The *size* and *filegrowth* properties don't apply to FILESTREAM filegroups.
- If no FILESTREAM-containing filegroup is specified as *DEFAULT*, the first FILESTREAM-containing filegroup listed is the default. (Therefore, you have one default filegroup for row data and one default filegroup for FILESTREAM data.)

Look at the following code, which creates a database called *MyFilestreamDB* with two FILESTREAM-containing filegroups. The path c:\Data2 must exist, but it must not contain either the *filestream1* or the *filestream2* folders:

```
CREATE DATABASE  MyFilestreamDB  ON  PRIMARY
     (NAME = N'Rowdata1', FILENAME = N'c:\data\Rowdata1.mdf' , SIZE = 2304KB ,
      MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB ),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM DEFAULT( NAME = FSData1,
    FILENAME = 'c:\Data2\FileStreamGroup1'),
FILEGROUP FileStreamGroup2 CONTAINS FILESTREAM ( NAME = FSData2,
    FILENAME = 'c:\Data2\FileStreamGroup2')
LOG ON
    (NAME = N'FSDBLOG', FILENAME = N'c:\data\FSDB_log.ldf' , SIZE = 1024KB ,
     MAXSIZE = 2048GB , FILEGROWTH = 10%);
```

When the preceding *MyFilestreamDB* database is created, SQL Server creates the two folders, *FileStreamGroup1* and *FileStreamGroup2*, in the C:\Data2 directory. These folders are referred to as the FILESTREAM *containers*. Initially, each container contains an empty folder called $FSLOG and a header file called filestream.hdr. As tables are created to use FILESTREAM space in a container, a folder for each partition or each table containing FILESTREAM data is created in the container.

An existing database can be altered to have a FILESTREAM filegroup added, and then a subsequent *ALTER DATABASE* command can add a file to the FILESTREAM filegroup. Note that you can't add FILESTREAM filegroups to the *master*, *model*, and *tempdb* databases.

Creating a table to hold FILESTREAM data

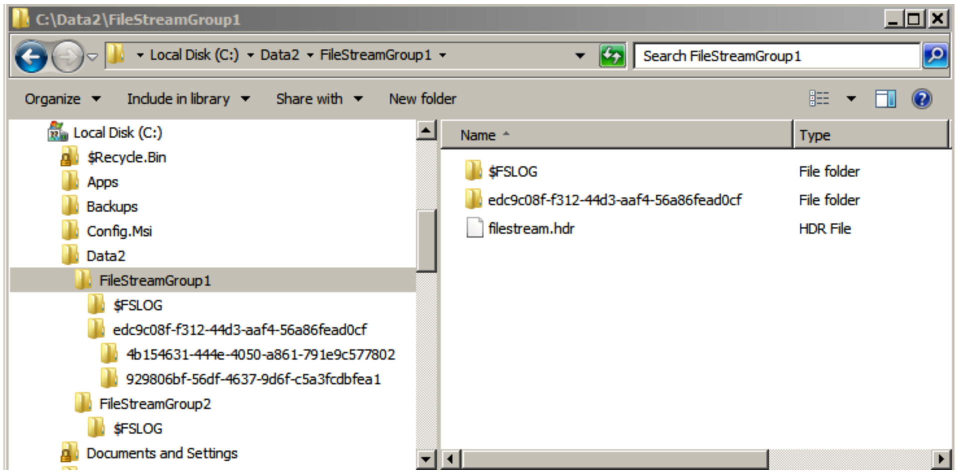
To specify that a column is to contain FILESTREAM data, it must be defined as type *varbinary(MAX)* with a *FILESTREAM* attribute. The database containing the table must have at least one filegroup defined for FILESTREAM. Your table creation statement can specify which filegroup its FILESTREAM data is stored in, and if none is specified, the default FILESTREAM filegroup is used. Finally, any table that has FILESTREAM columns must have a column of the *uniqueidentifier* data type with the *ROWGUIDCOL* attribute specified. This column must not allow NULL values and must be guaranteed to be unique by specifying either the UNIQUE or PRIMARY KEY single-column constraint. The

ROWGUIDCOL column acts as a key that the *FILESTREAM* agent can use to locate the actual row in the table to check permissions, obtain the physical path to the file, and possibly lock the row if required.

Now look at the files that are created within the container. When created in the *MyFilestreamDB* database, the following table adds several folders to the *FileStreamGroup1* container:

```
CREATE TABLE MyFilestreamDB.dbo.Records
(
        [Id] [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        [SerialNumber] INTEGER UNIQUE,
        [Chart_Primary] VARBINARY(MAX) FILESTREAM NULL,
        [Chart_Secondary] VARBINARY(MAX) FILESTREAM NULL)
FILESTREAM_ON FileStreamGroup1;
```

Because this table is created on *FileStreamGroup1*, the container located at C:\Data2\FileStreamGroup1 is used. One subfolder is created within *FileStreamGroup1* for each table or partition created in the *FileStreamGroup1* filegroup, and those filenames are GUIDs. Each file has a subfolder for each column within the table or partition which holds FILESTREAM data, and again, the names of those subfolders are GUIDs. Figure 8-4 shows the structure of the files on my disk right after the *MyFilestreamDB.dbo.Records* table is created. The *FileStreamGroup2* folder has only the $FSLOG subfolder, and no subfolders for any tables. The *FileStreamGroup1* folder has a GUID-named subfolder for the *dbo.Records* table and, within that, a GUID-named subfolder for each of the two *FILESTREAM* columns in the table. No files exist except for the original filestream.hdr file. Files aren't added until FILESTREAM data is actually inserted into the table.



Local Disk (C:)
- $Recycle.Bin
- Apps
- Backups
- Config.Msi
- Data2
  - FileStreamGroup1
    - $FSLOG
    - edc9c08f-f312-44d3-aaf4-56a86fead0cf
      - 4b154631-444e-4050-a861-791e9c577802
      - 929806bf-56df-4637-9d6f-c5a3fcdbfea1
  - FileStreamGroup2
    - $FSLOG
- Documents and Settings

Name	Type
$FSLOG	File folder
edc9c08f-f312-44d3-aaf4-56a86fead0cf	File folder
filestream.hdr	HDR File

FIGURE 8-4 The operating system file structure after creating a table with two FILESTREAM data columns.

Warning When the table is dropped, the folders, subfolders, and files they contain are *not* removed from the file system immediately. Instead, they are removed by a Garbage Collection thread, which fires regularly as well as when the SQL Server service stops and restarts. You can delete the files manually, but be careful: You might delete folders for a column or table that still exists in the database, even while the database is online. Subsequent access to that table generates an error message containing the text "Path not found."

You might think that SQL Server would prevent any file that is part of the database from being deleted. However, to absolutely prevent the file deletions, SQL Server has to hold open file handles for every single file in all the FILESTREAM containers for the entire database, and for large tables that wouldn't be practical.

Manipulating FILESTREAM data

FILESTREAM data can be manipulated with either T-SQL or the Win32 API. When using T-SQL, you can process the data exactly as though it were *varbinary(MAX)*. Using the Win32 API requires that you first obtain the file path and current transaction context. You can then open a WIN32 handle and use it to read and write the large object data. All the examples in this section use T-SQL; you can get the details of Win32 manipulation from *SQL Server Books Online*.

As you add data to the table, files are added to the subfolders for each column. *INSERT* operations that fail with a runtime error (for example, due to a uniqueness violation) still create a file for each FILESTREAM column in the row. Although the row is never accessible, it still uses file system space.

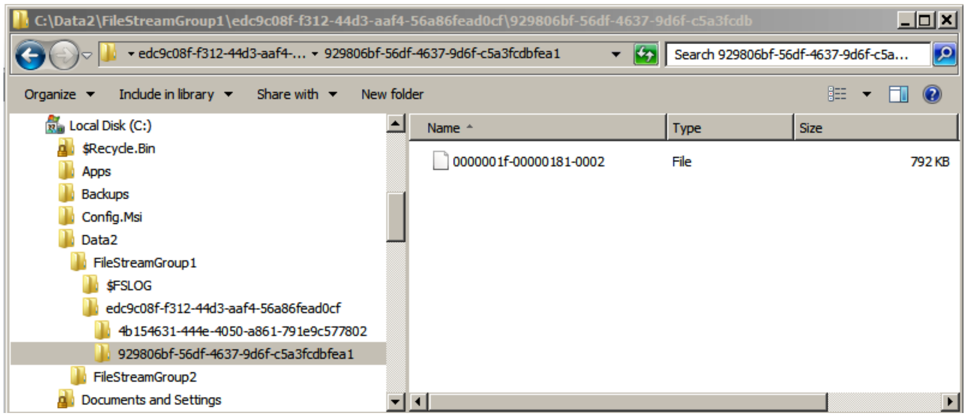
Inserting FILESTREAM data

You can insert data by using regular T-SQL *INSERT* statements. You must insert FILESTREAM data by using the *varbinary(MAX)* data type but can convert any string data in the *INSERT* statement. The following statement adds one row to the *dbo.Records* table, created earlier with two *FILESTREAM* columns. The first *FILESTREAM* column gets a 90,000-byte character string converted to *varbinary(MAX)*, and the second *FILESTREAM* column gets an empty binary string.

```
USE MyFileStreamDB
INSERT INTO dbo.Records
    SELECT newid (), 24,
      CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'), 10000)
                         AS varbinary(max)),
      0x;
```

First, the nine-character string *Base Data* to *varchar(MAX)* is converted because a regular string value can't be more than 8,000 bytes. The *REPLICATE* function returns the same data type as its first parameter, so that first parameter should be unambiguously a large object. Replicating the 9-byte string 10,000 times results in a 90,000-byte string, which is then converted to *varbinary(MAX)*. Notice that a value of 0x is an empty binary string, which isn't the same as a NULL. Every row that has a non-NULL value in a *FILESTREAM* column has a file, even for zero-length values.

Figure 8-5 shows what your file system should look like after running the preceding code to create a database with two FILESTREAM containers and create a table with two *FILESTREAM* columns, and then inserting one row into that table. In the left pane, you can see the two FILESTREAM containers, *FileStreamGroup1* and *FileStreamGroup2*.


C:\Data2\FileStreamGroup1\edc9c08f-f312-44d3-aaf4-56a86fead0cf\929806bf-56df-4637-9d6f-c5a3fcdb
edc9c08f-f312-44d3-aaf4-... ▸ 929806bf-56df-4637-9d6f-c5a3fcdbfea1
Search 929806bf-56df-4637-9d6f-c5a...
Organize
Include in library
Share with
New folder
Local Disk (C:)
$Recycle.Bin
Apps
Backups
Config.Msi
Data2
FileStreamGroup1
$FSLOG
edc9c08f-f312-44d3-aaf4-56a86fead0cf
4b154631-444e-4050-a861-791e9c577802
929806bf-56df-4637-9d6f-c5a3fcdbfea1
FileStreamGroup2
Documents and Settings

Name	Type	Size
0000001f-00000181-0002	File	792 KB

FIGURE 8-5 The operating system file structure after inserting FILESTREAM data.

The *FileStreamGroup1* container has a folder with a GUID name for the *dbo.Records* table that I created, and that folder container has two folders, with GUID names, for the two columns in that table. The right pane shows the file containing the actual data inserted into one of the columns.

Updating FILESTREAM data

Updates to FILESTREAM data are always performed as a *DELETE* followed by an *INSERT*, so you see a new row in the directory for the column(s) updated. Also, the T-SQL "chunked update," specified with the *.WRITE* clause, isn't supported. So any update to FILESTREAM data results into SQL Server creating a new copy of the FILESTREAM data file. I recommend that you use file-system streaming access for manipulation (both inserts and updates) of your FILESTREAM data.

When a FILESTREAM value is set to NULL, the FILESTREAM file associated with that value is deleted when the Garbage Collection thread runs. (Garbage collection is discussed later in this chapter.) The Garbage Collection thread also cleans up old versions of the FILESTREAM files after an *UPDATE* creates a new file.

Deleting FILESTREAM data

When a row is deleted through the use of a *DELETE* or a *TRUNCATE TABLE* statement, any FILESTREAM file associated with the row is deleted. However, deletion of the file isn't synchronous with row deletion. The file is deleted by the *FILESTREAM* Garbage Collection thread. This is also true for *DELETE*s that are generated as part of an *UPDATE*, as mentioned in the preceding section paragraph.

Note The *OUTPUT* clause for data manipulation operations (*INSERT*, *UPDATE*, *DELETE*, and *MERGE*) is supported in the same way it is for column modifications. However, you need to be careful if you are using the *OUTPUT* clause to insert into a table with a *varbinary(MAX)* column instead without the *FILESTREAM* specifier. If the FILESTREAM data is larger than 2 GB, the insert of FILESTREAM data into the table can result in a runtime error.

Manipulating FILESTREAM data and transactions

FILESTREAM data manipulation is fully transactional. However, you need to be aware that when you are manipulating FILESTREAM data, not all isolation levels are supported. Also, some isolation levels are supported for T-SQL access but not for file-system access. Table 8-2 indicates which isolation levels are available in which access mode.

TABLE 8-2 Isolation levels supported with FILESTREAM data manipulation

Isolation level	T-SQL access	File-system access
Read uncommitted	Supported	Not supported
Read committed	Supported	Supported
Repeatable read	Supported	Not supported
Serializable	Supported	Not supported
Read committed snapshot	Supported	Supported
Snapshot	Supported	Supported

If two processes trying to access the same *FILESTREAM* datafile are in incompatible modes, the file-system APIs fail with an *ERROR_SHARING_VIOLATION* message rather than just block, as would happen when using T-SQL. As with all data access, readers and writers within the same transaction can never get a conflict on the same file but unlike non-FILESTREAM access, two write operations within the same transaction can end up conflicting with each other when accessing the same file, unless the file handle has been previously closed. You can read much more about transactions, isolation levels, and conflicts in Chapter 13, "Transactions and concurrency."

Logging FILESTREAM changes

As mentioned previously, each *FILESTREAM* filegroup has a $FSLOG folder that keeps track of all FILESTREAM activity that touches that filegroup. The data in this folder is used when you perform transaction log backup and restore operations in the database (which include the *FILESTREAM* filegroup) and also during the recovery process.

The $FSLOG folder primarily keeps track of new information added to the FILESTREAM filegroup. A file gets added to the log folder to reflect each of the following.

- A new table containing FILESTREAM data is created.
- A *FILESTREAM* column is defined.

- A new row is inserted containing non-NULL data in the *FILESTREAM* column.
- A *FILESTREAM* value is updated.
- A *COMMIT* occurs.

Here are some examples.

- If you create a table containing two *FILESTREAM* columns, four files are added to the $FSLOG folder—one for the table, two for the columns, and one for the implied *COMMIT*.
- If you insert a single row containing FILESTREAM data in an autocommit transaction, two files are added to the $FSLOG folder—one for the *INSERT* and one for the *COMMIT*.
- If you insert five rows in an explicit transaction, six files are added to the $FSLOG folder.

Files aren't added to the $FSLOG folder when data is deleted or when a table is truncated or dropped. However, the SQL Server transaction log keeps track of these operations, and a new metadata table contains information about the removed data.

Using garbage collection for FILESTREAM data

The FILESTREAM data can be viewed as serving as the live user data, as well as the log of changes to that data, and as row versions for snapshot operations (discussed in Chapter 13). SQL Server needs to make sure that the FILESTREAM data files aren't removed if they might possibly be needed for any backup or recovery needs. In particular, for log backups, all new FILESTREAM content must be backed up because the transaction log doesn't contain the actual FILESTREAM data, and only the FILESTREAM data has the redo information for the actual FILESTREAM contents. In general, if your database isn't in the SIMPLE recovery mode, you need to back up the log twice before the Garbage Collector can remove unneeded data files from your *FILESTREAM* folders.

Consider this example: You can start with a clean slate by dropping and re-creating the *MyFilestreamDB* database. A *DROP DATABASE* statement immediately removes all the folders and files because now doing any subsequent log backups isn't possible. The script in Listing 8-3 recreates the database and creates a table with just a single *FILESTREAM* column. Finally, the script inserts three rows into the table and backs up the database. If you inspect the *FileStreamGroup1* container, you see that the folder for the columns contains three files for the three rows.

LISTING 8-3 Dropping and recreating a database

```
USE master;
GO
DROP DATABASE MyFilestreamDB;
GO
CREATE DATABASE  MyFilestreamDB  ON  PRIMARY
     (NAME = N'Rowdata1', FILENAME = N'c:\data\Rowdata1.mdf' , SIZE = 2304KB ,
      MAXSIZE = UNLIMITED, FILEGROWTH = 1024KB ),

FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM DEFAULT( NAME = FSData1,
    FILENAME = 'c:\Data2\FileStreamGroup1'),
FILEGROUP FileStreamGroup2 CONTAINS FILESTREAM ( NAME = FSData2,
    FILENAME = 'c:\Data2\FileStreamGroup2')
 LOG ON
    (NAME = N'FSDBLOG', FILENAME = N'c:\data\FSDB_log.ldf' , SIZE = 1024KB ,
     MAXSIZE = 2048GB , FILEGROWTH = 10%);
GO
USE MyFilestreamDB;
GO
CREATE TABLE dbo.Records
(
        Id [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
        SerialNumber INTEGER UNIQUE,
        Chart_Primary VARBINARY(MAX) FILESTREAM NULL
)
FILESTREAM_ON FileStreamGroup1;
GO
INSERT INTO dbo.Records
    VALUES (newid(), 1,
              CAST (REPLICATE (CONVERT(varchar(MAX), 'Base Data'),
                      10000) as varbinary(max))),
           (newid(), 2,
               CAST (REPLICATE (CONVERT(varchar(MAX), 'New Data'),
                       10000) as   varbinary(max))),
           (newid(), 3, 0x);
GO
BACKUP DATABASE MyFileStreamDB to disk = 'C:\backups\FBDB.bak';
GO
```

Now delete one of the rows, as follows:

```
DELETE dbo.Records
WHERE SerialNumber = 2;
GO
```

Now inspect the files on disk, and you still see three files.

Back up the log and run a checkpoint. Note that on a real system, enough changes would probably be made to your data that your database's log would get full enough to trigger an automatic *CHECKPOINT*. However, during testing, when you aren't putting much into the log at all, you have to force the *CHECKPOINT*:

```
BACKUP LOG  MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

Now if you check the FILESTREAM data files, you still see three rows. Wait five seconds for garbage collection, and you'll still see three rows. You need to back up the log and then force another *CHECKPOINT*:

```
BACKUP LOG  MyFileStreamDB to disk = 'C:\backups\FBDB_log.bak';
CHECKPOINT;
```

Now within a few seconds, you should see one of the files disappear. The reason you need to back up the log twice before the physical file is available for garbage collection is to make sure that the file space isn't reused by other FILESTREAM operations while it still might be needed for restore purposes.

You can run some additional tests of your own. For example, if you try dropping the *dbo.Records* table, notice that you again have to perform two log backups and *CHECKPOINT*s before SQL Server removes the folders for the table and the column.

Note SQL Server 2012 provides a new procedure called *sp_filestream_force_garbage_collection*, which forces the garbage collection of unneeded FILESTREAM files. If the garbage collection of unneeded files seems to be delayed frequently, and you want to force removal of these files, you could schedule regularly executions of this procedure. The procedure takes database name as a parameter and optionally takes a logical name of a FILESTREAM container as a second parameter. If the first parameter is missing, the current database is assumed. If the second parameter is missing, garbage collection is performed on all FILESTREAM containers in the database.

Exploring metadata with FILESTREAM data

Within your SQL Server tables, the storage required for FILESTREAM isn't particularly complex. In the row itself, each FILESTREAM column contains a file pointer that is 48 bytes in size. Even if you look at a data page with the *DBCC PAGE* command, not much more information about the file is available. However, SQL Server does provide a new function to translate the file pointer to a path name. The function is actually a method applied to the column name in the table. So the following code returns a UNC name for the file containing the actual column's data in the row inserted previously:

```
SELECT Chart_Primary, Chart_Primary.PathName()
FROM dbo.Records
WHERE SerialNumber = 3;
GO
```

The UNC value returned looks like this:

```
\\<server_name>\<share_name>\v1\<db_name>\<object_schema>\<table_name>\<column_name>\<GUID>
```

Keep in mind the following points about using the *PathName* function.

- The function name is case-sensitive, even on a server that's not case-sensitive, so it always must be entered as *PathName*.
- The default *share_name* is the service name for your SQL Server instance (so for the default instance, it is MSSQLSERVER). By using the SQL Server Configuration Manager, you can right-click your SQL Server instance and choose Properties. The *FILESTREAM* tab of the SQL Server Properties sheet allows you to change the *share_name* to another value of your choosing.
- The *PathName* function can take an optional parameter of 0, 1, or 2, with 0 being the default. The parameter controls only how the *server_name* value is returned; all other values in the UNC string are unaffected. Table 8-3 shows the meanings of the different values.

TABLE 8-3 Parameter values for the *PathName* function

Value	Description
0	Returns the server name converted to BIOS format; for example, \\SERVERNAME\MSSQLSERVER\v1\MyFilestream\dbo\Records\Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
1	Returns the server name without conversion; for example, \\ServerName\MSSQLSERVER\v1\MyFilestream\Dbo\Records\Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9
2	Returns the complete server path; for example, \\ServerName.MyDomain.com\MSSQLSERVER\v1\MyFilestream\Dbo\Records\Chart_Primary\A73F19F7-38EA-4AB0-BB89-E6C545DBD3F9

Other metadata gives you information about your FILESTREAM data.

- *sys.database_files* returns a row for each of your FILESTREAM files. These files have a *type* value of 2 and a *type_desc* value of *FILESTREAM*.
- *sys.filegroups* returns a row for each of your FILESTREAM filegroups. These files have a *type* value of FD and a *type_desc* value of *FILESTREAM_DATA_FILEGROUP*.
- *sys.data_spaces* returns one row for each data space, which is either a filegroup or a partition scheme. Filegroups holding FILESTREAM data are indicated by the type *FD*.
- *sys.tables* has a value in the column for *filestream_data_space_id*, which is the data space ID for either the FILESTREAM filegroup or the partition scheme that the FILESTREAM data uses. Tables with no FILESTREAM data have NULL in this column.
- *sys.columns* has a value of 1 in the *is_filestream* column for columns with the *filestream* attribute.

The older metadata, such as the system procedure *sp_helpdb <database_name>* or *sp_help <object_name>*, doesn't show any information about FILESTREAM data.

Earlier, this chapter mentioned that rows or objects that are deleted don't generate files in the $FSLOG folder, but data about the removed data is stored in a system table. No metadata view allows you to see this table; you can observe it only by using the dedicated administrator connection (DAC). You can look in a view called *sys.internal_tables* for an object with *TOMBSTONE* in its name. Then, by using the DAC, you can look at the data inside the *TOMBSTONE* table. If you rerun the preceding script but don't back up the log, you can use the following code:

```
USE MyFilestreamDB;
GO
SELECT name FROM sys.internal_tables
WHERE name like '%tombstone%';

-- I see the table named: filestream_tombstone_2073058421
-- Reconnect using DAC, which puts us in the master database
USE MyFileStreamDB;
GO
SELECT * FROM sys.filestream_tombstone_2073058421;
GO
```

If this table is empty, the login SQL Server and the $FSLOG are in sync, and all unneeded files have been removed from the FILESTREAM containers on disk.

Creating a FileTable

FileTable storage, introduced in SQL Server 2012, allows you to create special user tables that have a predefined schema and to extend the capabilities of FILESTREAM data discussed earlier. The two most important extensions are that FileTables allow full support and compatibility with Win32 applications and support the hierarchical namespace of directories and files. Each row in a FileTable table represents an operating system file or directory in a hierarchical structure and contains attributes about the file or directory, such as created date, modified, date, and the name of the file or directory.

The first step in creating a FileTable is to make sure that the database supports FILESTREAM data with at least one FILESTREAM filegroup. Also, the database option to allow *NON_TRANSACTED_ACCESS* must be set to either *FULL* or *READ_ONLY*, and a directory name must be supplied for use by Windows applications as part of the share name when accessing the FileTable files. These options can be supplied as part of the initial *CREATE DATABASE* operation, or the database can be altered to include these options:

```
ALTER DATABASE MyFilestreamDB
SET FILESTREAM ( NON_TRANSACTED_ACCESS = FULL, DIRECTORY_NAME = N'FileTableData' );
```

By setting *NON_TRANSACTED_ACCESS* to something other than *NONE*, you are enabling FileTable storage within the database. SQL Server 2012 provides a new catalog view, called *sys.database_filestream_options*, to examine each database's readiness for storing FileTable data. The following query selects from that view showing only rows where non-transacted access has been allowed. A value of 0 for *NON_TRANSACTED_ACCESS* means the feature wasn't enabled, 1 means it has been enabled for *READ_ONLY*, and 2 means it has been enabled for *FULL* access.

```
SELECT DB_NAME(database_id) as DBNAME,  non_transacted_access_desc, directory_name
FROM sys.database_filestream_options
WHERE non_transacted_access > 0;
```

Enabling the *NON_TRANSACTED_ACCESS* to the *MyFilestreamDB* as shown in the preceding *ALTER* statement would produce the following results:

```
DB_NAME              non_transacted_access_desc      directory_name
----------------     --------------------------      ----------------
MyFilestreamDB       FULL                            FileTableData
```

After the database is properly configured for FileTable access, you can create a FileTable with a very simple *CREATE TABLE* statement. Because the schema is predefined, all you do is specify a name for your FileTable and a directory name in which all the files stored in this FileTable can be found in the operating system. Optionally, you can also specify a collation for the data in the files stored in the FileTable, but if none is specified, the default database collation is used.

```
CREATE TABLE Documents AS Filetable
      WITH (Filetable_Directory = 'DocumentsData');
```

At this point, you can look at the Windows share that is now available by allowing your database *NON_TRANSACTED_ACCESS*. You can see this by opening Windows Explorer on the server machine and navigating to either \\127.0.0.1 or \\<your machine name>. Along with whatever other shares have been set up on the machine, you should see a share for your SQL Server instance name, or *mssqlserver* if your instance is the default instance. You can open up the share and see a directory named *FiletableData*, and when you open the directory, you should see the table name *DocumentsData*. Figure 8-6 shows you what this structure looks like on my machine, for a SQL Server 2012 instance called denali. (In SQL Server 2008 and 2008 R2, you also saw share names for your SQL Server instances that have been configured to allow FILESTREAM access. However, attempting to access the share directly will result in an error.)

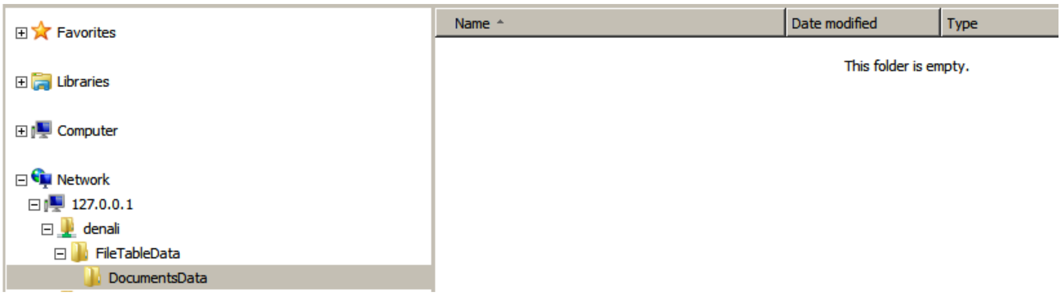
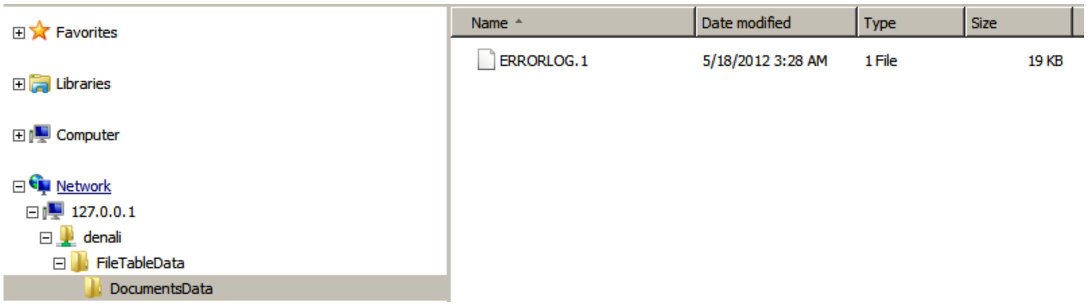


FIGURE 8-6 The FileTable share containing no files.

While exploring the operating system files, you can also revisit the *Data2* folder (which was the default FILESTREAM container for the database), which you saw earlier in the FILESTREAM data discussion, and see that a GUID now exists for one more table containing FILESTREAM data and one FILESTREAM column in that table.

As mentioned, the FileTable data is completely available through the share to all Windows applications, including Windows Explorer. You can copy a file from anywhere on your system and paste it into the share. Figure 8-7 shows the share now containing a copy of one of my errorlog files.

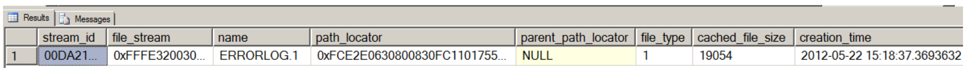


Name	Date modified	Type	Size
ERRORLOG.1	5/18/2012 3:28 AM	1 File	19 KB

FIGURE 8-7 The FileTable share after inserting one file.

When a SQL Server instance is enabled for FILESTREAM access, SQL Server installs a component called an NTFS filter driver. When a Windows application interacts with a FileTable share, the interaction is intercepted by the NTFS filter driver and is redirected to SQL Server to allow these changes that you make to the share, such as inserting a new file, to be reflected in the table inside SQL Server. From SQL Server Management Studio, I can also *SELECT* from my Documents table, and see that it now has one row in it, to reflect the file that was added to it through Windows.

Some of the columns are returned, but the wide output prevents showing all the columns:



	stream_id	file_stream	name	path_locator	parent_path_locator	file_type	cached_file_size	creation_time
1	00DA21...	0xFFFE320030...	ERRORLOG.1	0xFCE2E0630800830FC1101755...	NULL	1	19054	2012-05-22 15:18:37.3693632

Table 8-4 lists all the column names available in the FileTable.

TABLE 8-4 Predefined columns in a FileTable

File attribute name	Type	Description
stream_id	*[uniqueidentifier] rowguidcol*	A unique ID for the FILESTREAM data.
file_stream	*varbinary(max) filestream*	Contains the FILESTREAM data.
Name	*nvarchar(255)*	The file or directory name.
path_locator	*hierarchyid*	The position of this node in the hierarchical FileNamespace. Primary key for the table.
parent_path_locator	*hierarchyid*	The *hierarchyid* of the containing directory. *parent_path_locator* is a persisted computed column.
file_type	*nvarchar(255)*	Represents the type of the file. This column can be used as the *TYPE COLUMN* when you create a full-text index. file_type is a persisted computed column based on the file extension.
cached_file_size	*bigint*	The size in bytes of the FILESTREAM data. *cached_file_size* is a persisted computed column.
creation_time	*datetime2(4) not null*	The date and time that the file was created.
last_write_time	*datetime2(4) not null*	The date and time that the file was last updated.
last_access_time	*datetime2(4) not null*	The date and time that the file was last accessed.
is_directory	*bit not null*	Indicates whether the row represents a directory. This value is calculated automatically, and can't be set.
is_offline	*bit not null*	Offline file attribute.
is_hidden	*bit not null*	Hidden file attribute.
is_readonly	*bit not null*	Read-only file attribute.
is_archive	*bit not null*	Archive attribute.

File attribute name	Type	Description
is_system	*bit* *not null*	System file attribute.
is_temporary	*bit* *not null*	Temporary file attribute.

You can see that many of these column names correspond to attributes that you can see for all your files through Windows Explorer.

Note All application access to your FileTable data is through the FileTable share. The FILESTREAM container is the actual physical storage of the FILESTREAM data, and only database administrators need to be aware of this location. The FILESTREAM containers are backed up when a database containing FILESTREAM data is backed up.

Considering performance for FILESTREAM data

Although a thorough discussion of performance tuning and troubleshooting is beyond the scope of this book, I want to provide you with some basic information about setting up your system to get high performance from FILESTREAM data. Paul Randal, one of the co-authors of this book, has written a white paper on FILESTREAM that you can access on the MSDN site at *http://msdn.microsoft.com/en-us/library/cc949109.aspx*. (This white paper is also available on this book's companion website, *http://www.SQLServerInternals.com/companion*.) This section just briefly mentions some of the main points Paul makes regarding what you can do to get good performance. All these suggestions are explained in much more detail in the white paper.

- Make sure that you're storing the right-sized data in the right way. Jim Gray (et al) published a research paper several years ago titled "To BLOB or Not to BLOB: Large Object Storage in a Database or a Filesystem?" that gives recommendations for when to store data outside the database. To summarize the findings, large object data smaller than 256 KB should be stored in a database, and data that's 1 MB or larger should be stored in the file system. For data between these two values, the answer depends on other factors, and you should test your application thoroughly. The key point here is that you won't get good performance if you store lots of relatively small large objects using FILESTREAM.
- Use an appropriate RAID level for the NTFS volume that hosts the FILESTREAM data container. For example, don't use RAID 5 for a write-intensive workload.
- Use an appropriate disk technology. SCSI is usually faster than SATA/IDE because SCSI drives usually have higher rotational speeds, which help them have lower latency and seek times. However, SCSI drives are also more expensive.
- Whichever disk technology you choose, if it is SATA, ensure that it supports NCQ, and if SCSI, ensure that it supports CTQ. Both of these allow the drives to process multiple, interleaved I/Os concurrently.

- Separate the data containers from each other, and separate the containers from other database data and log files. This avoids contention for the disk heads.
- Defragment the NTFS volume, if needed, before setting up FILESTREAM, and defragment periodically to maintain good scan performance.
- Turn off 8.3 name generation on the NTFS volume by using the command-line *fsutil* utility. This is an order-N algorithm that must check that the new name generated doesn't collide with any existing names in the directory. Note, however, that this slows down insert and update performance a lot.
- Use *fsutil* to turn off tracking of last access time.
- Set the NTFS cluster size appropriately. For larger objects greater than 1 MB in size, use a cluster size of 64 KB to help reduce fragmentation.
- A partial update of FILESTREAM data creates a new file. Batch lots of small updates into one large update to reduce churn.
- When streaming the data back to the client, use a server message block (SMB) buffer size of approximately 60 KB or multiples thereof. This helps keep the buffers from getting overly fragmented, because Transmission Control Protocol/Internet Protocol (TCP/IP) buffers are 64 KB.

Taking these suggestions into consideration and performing thorough testing of your application can give you great performance when working with very large data objects.

Summarizing FILESTREAM and FileTable

When you configure a SQL Server instance to support FILESTREAM storage, you are allowing Windows files to be accessed and manipulated as data in any SQL Server table. When you configure a database with FILESTREAM filegroups to have non-transacted access, you are allowing FileTables to be created in the database. A FileTable builds on the FILESTREAM capability by providing a table whose rows not only contain the Windows file contents but also all the Windows file properties. These files are accessible through the FileTable share and can be manipulated through any Windows application.

Keep in mind that the FileTable structure is predefined and can't be altered. If you want to add your own attributes along with the files in a FileTable, you can create another table with your user-defined attributes and a foreign key reference to the FileTable column *path_locator*, which is the primary key of the FileTable.

Detailed information about how to work with this FileTable data through a Windows application is beyond the scope of this book. The goal of this section was to present some of the storage internals of the FILESTREAM data and FileTables, and well as to give you an idea of how SQL Server works with these constructs.

Sparse columns

This section looks at another special storage format, added in SQL Server 2008. Sparse columns are ordinary columns that have an optimized storage format for NULL values. Sparse columns reduce the space requirements for NULL values, allowing you to have many more columns in your table definition, as long as most of them are NULL. Using sparse columns requires more overhead to store and retrieve non-NULL values.

Sparse columns are intended to be used for tables storing data describing entities with many possible attributes, in which most of the attributes will be NULL for most rows. For example, a content management system such as Microsoft Windows SharePoint Services might need to keep track of many different types of data in a single table. Because different properties apply to different subsets of rows in the table, only a small subset of the columns is populated with values for each row. Another way of looking at this is that for any particular property, only a subset of rows has a value for that property. Sparse columns allow you to store a very large number of possible columns for a single row. For this reason, the Sparse Columns feature is sometimes also referred to as the *wide-table* feature.

Management of sparse columns

You shouldn't consider defining a column as *SPARSE* unless at least 90 percent of the rows in the table are expected to have NULL values for that column. This limit isn't enforced, however, and you can define almost any column as SPARSE. Sparse columns save space on NULL values.

The Sparse Columns feature allows you to have far more columns that you ever could before. The limit is now 30,000 columns in a table, with no more than 1,024 of them being non-sparse. (Computed columns are considered non-sparse.) Obviously, not all 30,000 columns could have values in them. The number of populated columns you can have depends on the bytes of data in the row. Sparse columns optimize the storage size for NULL values, which take no space at all for sparse columns, unlike non-sparse columns, which do need space even for NULLs. (As you saw in Chapter 6, a fixed-length NULL column always uses the whole column width, and a variable-length NULL column uses at least two bytes in the column offset array.)

Although the sparse columns themselves take no space, some fixed overhead is needed to allow for sparse columns in a row. As soon as you define even one column with the *SPARSE* attribute, SQL Server adds a sparse vector to the end of the row. We'll see the actual structure of this sparse vector in the section "Physical storage," later in this chapter, but to start, you should be aware that even with sparse columns, the maximum size of a data row (excluding LOB and row-overflow) remains at 8,060, including overhead bytes. Because the sparse vector includes additional overhead, the maximum number of bytes for the rest of the rows decreases. Also, the size of all fixed-length non-NULL sparse columns in a row is limited to 8,023 bytes.

Creating a table

Creating a table with sparse columns is very straightforward, as you can just add the attribute *SPARSE* to any column of any data type except *text*, *ntext*, *image*, *geography*, *geometry*, *timestamp*, or any user-defined data type. Also, sparse columns can't include the *IDENTITY*, *ROWGUIDCOL*, or *FILESTREAM* attributes. A sparse column can't be part of a clustered index or part of the primary key. Tables containing sparse columns can't be compressed, either at the row level or the page level. (The next section discusses compression in detail.) A few other restrictions are enforced, particularly if you are partitioning a table with sparse columns, so you should check the documentation for full details.

The examples in this section are necessarily very simple because it would be impractical to print code examples with enough columns to make sparse columns really useful. The following example shows the creation of two very similar tables: one that doesn't allow sparse columns and another that does. I attempt to insert the same rows into each table. Because a row allowing sparse columns has a smaller maximum length, it fails when trying to insert a row that the table with no sparse columns has no problem with:

```
USE testdb;
GO
IF OBJECT_ID('test_nosparse') IS NOT NULL
    DROP TABLE test_nosparse;
GO
CREATE TABLE test_nosparse
(
  col1 int,
  col2 char(8000),
  col3 varchar(8000)
);
GO
INSERT INTO test_nosparse
        SELECT null, null, null;
INSERT INTO test_nosparse
        SELECT 1, 'a', 'b';
GO
```

These two rows can be inserted with no error. Now, build the second table:

```
IF OBJECT_ID('test_sparse') IS NOT NULL
    DROP TABLE test_sparse;
GO

CREATE TABLE test_sparse
(
 col1 int SPARSE,
 col2 char(8000) SPARSE,
 col3 varchar(8000) SPARSE
);
GO

INSERT INTO test_sparse
        SELECT NULL, NULL, NULL;
INSERT INTO test_sparse
        SELECT 1, 'a', 'b';
GO
```

The second *INSERT* statement generates the following error:

```
Msg 576, Level 16, State 5, Line 2
Cannot create a row that has sparse data of size 8046 which is greater than the
allowable maximum sparse data size of 8023.
```

Although the second row inserted into the *test_sparse* table looks just like a row that was inserted successfully into the *test_nosparse* table, internally it's not. The total of the sparse columns is 4 bytes for the *int*, plus 8,000 bytes for the *char* and 24 bytes for the row-overflow pointer, which is greater than the 8,023-byte limit.

Altering a table

You can alter tables to convert a non-sparse column into a sparse column, or vice versa. Be careful, however, because if you are altering a very large row in a table with no sparse columns, changing one column to be sparse reduces the number of bytes of data that are allowed on a page. This can result in an error being thrown in cases where an existing column is converted into a sparse column. For example, the following code creates a table with large rows, but the *INSERT* statements, with or without NULLs, are accepted. However, when you try to make one of the columns *SPARSE*—even a relatively small column like the 8-byte *datetime* column—the extra overhead makes the existing rows too large and the *ALTER* fails:

```
IF OBJECT_ID('test_nosparse_alter') IS NOT NULL
    DROP TABLE test_nosparse_alter;
GO
GO
CREATE TABLE test_nosparse_alter
(
c1 int,
c2 char(4020) ,
c3 char(4020) ,
c4 datetime
);
GO
INSERT INTO test_nosparse_alter SELECT NULL, NULL, NULL, NULL;
INSERT INTO test_nosparse_alter SELECT 1, 1, 'b', GETDATE();
GO
ALTER TABLE test_nosparse_alter
  ALTER COLUMN c4 datetime SPARSE;
```

This error is received:

```
Msg 1701, Level 16, State 1, Line 2
Creating or altering table 'test_nosparse_alter' failed because the minimum row size
would be 8075, including 23 bytes of internal overhead. This exceeds the maximum
allowable table row size of 8060 bytes.
```

In general, you can treat sparse columns just like any other column, with only a few restrictions. In addition to the restrictions mentioned earlier on the data types that can't be defined as *SPARSE*, you need to keep in mind the following limitations.

- A sparse column can't have a default value.
- A sparse column can't be bound to a rule.
- Although a computed column can refer to a sparse column, a computed column can't be marked as *SPARSE*.
- A sparse column can't be part of a clustered index or a unique primary key index. However, both persisted and non-persisted computed columns that refer to sparse columns can be part of a clustered key.
- A sparse column can't be used as a partition key of a clustered index or heap. However, a sparse column can be used as the partition key of a nonclustered index.

Except for the requirement that sparse columns can't be part of the clustered index or primary key, building indexes on sparse columns has no other restrictions. However, if you're using sparse columns the way they are intended to be used and the vast majority of your rows have NULL for the sparse columns, any regular index on a sparse column is very inefficient and might have limited usefulness. Sparse columns are really intended to be used with filtered indexes, which are discussed in Chapter 7.

Column sets and sparse column manipulation

If sparse columns are used as intended, only a few columns in each row have values, and your *INSERT* and *UPDATE* statements are relatively straightforward. For *INSERT* statements, you can specify a column list and then specify values only for those few columns in the column list. For *UPDATE* statements, values can be specified for just a few columns in each row. The only time you need to be concerned about how to deal with a potentially very large list of columns is if you are selecting data without listing individual columns—that is, using a *SELECT* *. Good developers know that using *SELECT* * is never a good idea, but SQL Server needs a way of dealing with a result set with potentially thousands (or tens of thousands) of columns. The mechanism to help deal with *SELECT* * is a construct called *COLUMN_SET,* which is an untyped XML representation that combines multiple columns of a table into a structured output. You can think of a *COLUMN_SET* as a nonpersisted computed column because the *COLUMN_SET* isn't physically stored in the table. In this release of SQL Server, the only possible *COLUMN_SET* contains all the sparse columns in the table. Future versions might allow us to define other *COLUMN_SET* variations.

A table can only have one *COLUMN_SET* defined, and when a table has a *COLUMN_SET* defined, *SELECT* * no longer returns individual sparse columns. Instead, it returns an XML fragment containing all the non-NULL values for the sparse columns. For example, the code in Listing 8-4 builds a table containing an identity column, 25 sparse columns, and a column set.

LISTING 8-4 Building a table with an identity column, sparse columns, and a column set

```
USE testdb;
GO
IF EXISTS (SELECT * FROM sys.tables WHERE name = 'lots_of_sparse_columns')
                DROP TABLE lots_of_sparse_columns;
GO
```

```
CREATE TABLE lots_of_sparse_columns
(ID int IDENTITY,
 col1 int SPARSE,
 col2 int SPARSE,
 col3 int SPARSE,
 col4 int SPARSE,
 col5 int SPARSE,
 col6 int SPARSE,
 col7 int SPARSE,
 col8 int SPARSE,
 col9 int SPARSE,
 col10 int SPARSE,
 col11 int SPARSE,
 col12 int SPARSE,
 col13 int SPARSE,
 col14 int SPARSE,
 col15 int SPARSE,
 col16 int SPARSE,
 col17 int SPARSE,
 col18 int SPARSE,
 col19 int SPARSE,
 col20 int SPARSE,
 col21 int SPARSE,
 col22 int SPARSE,
 col23 int SPARSE,
 col24 int SPARSE,
 col25 int SPARSE,
 sparse_column_set XML COLUMN_SET FOR ALL_SPARSE_COLUMNS);
 GO
```

Next, values are inserted into 3 of the 25 columns, specifying individual column names:

```
INSERT INTO lots_of_sparse_columns (col4, col7, col12)  SELECT 4,6,11;
```

You can also insert directly into the *COLUMN_SET*, specifying values for columns in an XML fragment. The capability to update the *COLUMN_SET* is another feature that differentiates *COLUMN_SET*s from computed columns:

```
INSERT INTO lots_of_sparse_columns (sparse_column_set)
                SELECT '<col8>42</col8><col17>0</col17><col22>30000</col22>';
```

Here are my results when I run *SELECT* * from this table:

```
SELECT * FROM lots_of_sparse_columns;
Results:
ID      sparse_column_set
------- ---------------------------------------------------
1       <col4>4</col4><col7>6</col7><col12>11</col12>
2       <col8>42</col8><col17>0</col17><col22>30000</col22>
```

You can still select from individual columns, either instead of or in addition to selecting the entire *COLUMN_SET*. So the following *SELECT* statements are both valid:

```
SELECT ID, col10, col15, col20
    FROM lots_of_sparse_columns;
SELECT *, col11
    FROM lots_of_sparse_columns;
```

Keep the following points in mind if you decide to use sparse columns in your tables.

- When defined, the *COLUMN_SET* can't be altered. To change a *COLUMN_SET*, you must drop and re-create the *COLUMN_SET* column.
- A *COLUMN_SET* can be added to a table that doesn't include any sparse columns. If sparse columns are later added to the table, they appear in the column set.
- A *COLUMN_SET* is optional and isn't required to use sparse columns.
- Constraints or default values can't be defined on a *COLUMN_SET*.
- Distributed queries aren't supported on tables that contain *COLUMN_SET*s.
- Replication doesn't support *COLUMN_SET*s.
- The Change Data Capture feature doesn't support *COLUMN_SET*s.
- A *COLUMN_SET* can't be part of any kind of index. This includes XML indexes, full-text indexes, and indexed views. A *COLUMN_SET* also can't be added as an included column in any index.
- A *COLUMN_SET* can't be used in the filter expression of a filtered index or filtered statistics.
- When a view includes a *COLUMN_SET*, the *COLUMN_SET* appears in the view as an XML column.
- XML data has a size limit of 2 GB. If the combined data of all the non-NULL sparse columns in a row exceeds this limit, the operation produces an error.
- Copying all columns from a table with a *COLUMN_SET* (using either *SELECT * INTO* or *INSERT INTO SELECT* *) doesn't copy the individual sparse columns. Only the *COLUMN_SET*, as data type XML, is copied.

Physical storage

At a high level, you can think of sparse columns as being stored much as they are displayed using the *COLUMN_SET*—that is, as a set of (column-name, value) pairs. So if a particular column has no value, it's not listed and no space at all is required. If a column has a value, not only does SQL Server need to store that value but it also needs to store information about which column has that value. As a result, non-NULL sparse columns take more space than their NULL counterparts. To see the difference graphically, you can compare Tables 8-5 and 8-6.

Table 8-5 represents a table with non-sparse columns. You can see a lot of wasted space when most of the columns are NULL. Table 8-6 shows what the same table looks like if all the columns except the ID are defined as *SPARSE*. All that is stored are the names of all the non-NULL columns and their values.

TABLE 8-5 Representation of a table defined with non-sparse columns, with many NULL values

ID	sc1	sc2	sc3	sc4	sc5	sc6	sc7	sc8	sc9
1	1								9
2		2		4					
3						6	7		
4	1				5				
5				4				8	
6			3						9
7					5		7		
8		2						8	
9			3			6			

TABLE 8-6 Representation of a table defined with sparse columns, with many NULL values

ID	<sparse columns>
1	(sc1,sc9)(1,9)
2	(sc2,sc4)(2,4)
3	(sc6,sc7)(6,7)
4	(sc1,sc5)(1,5)
5	(sc4,sc8)(4,8)
6	(sc3,sc9)(3,9)
7	(sc5,sc7)(5,7)
8	(sc2,sc8)(2,8)
9	(sc3,sc6)(3,6)

SQL Server keeps track of the physical storage of sparse columns with a structure within a row called a *sparse vector*. Sparse vectors are present only in the data records of a base table that has at least one sparse column declared, and each data record of these tables contains a sparse vector. A sparse vector is stored as a special variable-length column at the end of a data record. It's a special system column, and no metadata about this column appears in *sys.columns* or any other view. The sparse vector is stored as the last variable-length column in the row. The only thing after the sparse vector would be versioning information, used primarily with Snapshot isolation, as is discussed in Chapter 13. The NULL bitmap has no bit for the sparse vector column (if a sparse vector exists, it's never NULL), but the count in the row of the number of variable-length columns includes the sparse vector. You might want to revisit Figure 6-5 in Chapter 6 at this time to familiarize yourself with the general structure of data rows.

Table 8-7 lists the meanings of the bytes in the sparse vector.

TABLE 8-7 Bytes in a sparse vector

Name	Number of bytes	Meaning
Complex Column Header	2	Value of 05 indicates that the complex column is a sparse vector.
Sparse Column Count	2	Number of sparse columns.
Column ID Set	2 * the number of sparse columns	Two bytes for the column ID of each column in the table with a value stored in the sparse vector.
Column Offset Table	2 * the number of sparse columns	Two bytes for the offset of the ending position of each sparse column.
Sparse Data	Depends on actual values	Data

Now look at the bytes of a row containing sparse columns. First, build a table containing two sparse columns, and populate it with three rows:

```
USE testdb;
GO
IF OBJECT_ID ('sparse_bits') IS NOT NULL
                 DROP TABLE sparse_bits;
GO
CREATE TABLE sparse_bits
(
c1 int IDENTITY,
c2 varchar(4),
c3 char(4) SPARSE,
c4 varchar(4) SPARSE
);
GO
INSERT INTO sparse_bits SELECT 'aaaa', 'bbbb', 'cccc';
INSERT INTO sparse_bits SELECT 'dddd', null, 'eeee';
INSERT INTO sparse_bits SELECT 'ffff', null, 'gg';
GO
```

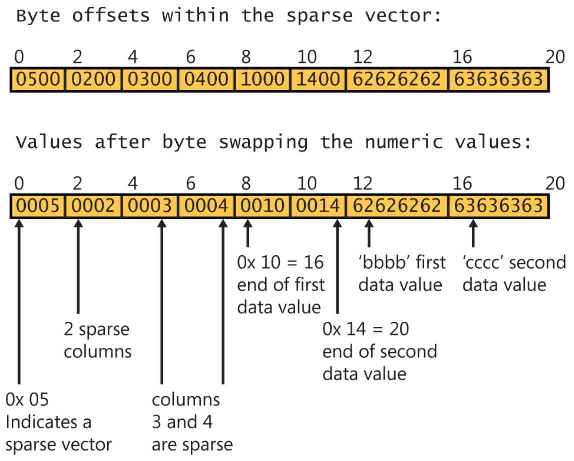
Now you can use *sys.dm_db_database_page_allocations* to find the page number for the data page storing these three rows and then use *DBCC PAGE* to look at the bytes on the page:

```
SELECT allocated_page_file_id as PageFID, allocated_page_page_id as PagePID,
       object_id as ObjectID, partition_id AS PartitionID,
       allocation_unit_type_desc as AU_type, page_type as PageType
FROM sys.dm_db_database_page_allocations
       (db_id('testdb'), object_id('sparse_bits'), null, null, 'DETAILED');
-- The output indicated that the data page for my table was on page 289;
DBCC TRACEON(3604);
DBCC PAGE(testdb, 1, 289, 1);
```

Only the output for the first data row, which is spread over three lines of *DBCC PAGE* output, is shown here:

```
00000000:   30000800 01000000 02000002 00150029 80616161 0..............).aaa
00000014:   61050002 00030004 00100014 00626262 62636363 a             bbbbccc
00000020:   63                                           c
```

The boldfaced bytes are the sparse vector. You can find it easily because it starts right after the last non-sparse variable-length column, which contained *aaaa*, or 61616161, and continues to the end of the row. Figure 8-8 translates the sparse vector according to the meanings from Table 8-7. Don't forget that you need to byte-swap numeric fields before translating. For example, the first two bytes are 05 00, which need to be swapped to get the hex value 0x0005. Then you can convert it to decimal.



Byte offsets within the sparse vector:
0 2 4 6 8 10 12 16 20
0500 0200 0300 0400 1000 1400 62626262 63636363
Values after byte swapping the numeric values:
0 2 4 6 8 10 12 16 20
0005 0002 0003 0004 0010 0014 62626262 63636363
0x 10 = 16 end of first data value
'bbbb' first data value
'cccc' second data value
2 sparse columns
0x 14 = 20 end of second data value
0x 05 Indicates a sparse vector
columns 3 and 4 are sparse

FIGURE 8-8 Interpretation of the actual bytes in a sparse vector.

You can apply the same analysis to the bytes in the other two rows on the page. Here are some things to note:

- No information about columns with NULL values appears in the sparse vector.
- No difference exists in storage between fixed-length and variable-length strings within the sparse vector. However, that doesn't mean you should use the two interchangeably. A sparse *varchar* column that doesn't fit in the 8,060 bytes can be stored as row-overflow data; a sparse *char* column can't be.
- Because only 2 bytes are used to store the number of sparse columns, this sets the limit on the maximum number of sparse columns.
- The 2 bytes for the complex column header indicate that there might be other possibilities for complex columns. At this time, the only other type of complex column that can be stored is one storing a back-pointer, as SQL Server does when it creates a forwarded record. (Chapter 6 briefly discussed forwarded records when discussing updates to heaps.)

Metadata

Very little extra metadata is needed to support sparse columns. The catalog view *sys.columns* contains two columns to keep track of sparse columns in your tables: *is_sparse* and *is_column_set*. Each column has only two possible values, 0 or 1.

Corresponding to these column properties in *sys.columns*, the property function *COLUMNPROPERTY()* also has the following properties related to sparse columns: *IsSparse* and *IsColumnSet*.

To inspect all tables with "sparse" in their name and determine which of their columns are *SPARSE*, which are column sets, and which are neither, you can run the following query:

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column', is_sparse, is_column_set
FROM sys.columns
WHERE OBJECT_NAME(object_id) like '%sparse%';
```

To see just the table and column names for all *COLUMN_SET* columns, you can run the following query:

```
SELECT OBJECT_NAME(object_id) as 'Table', name as 'Column'
FROM sys.columns
WHERE COLUMNPROPERTY(object_id, name, 'IsColumnSet') = 1;
```

Storage savings with sparse columns

The sparse column feature is designed to save you considerable space when most of your values are NULL. In fact, as mentioned earlier, columns that aren't NULL but are defined as *SPARSE* take up more space than if they weren't defined as *SPARSE* because the sparse vector has to store a couple of extra bytes to keep track of them. To start to see the space differences, you can run the script in Listing 8-5, which creates four tables with relatively short, fixed-length columns. Two have sparse columns and two don't. Rows are inserted into each table in a loop, which inserts 100,000 rows. One table with sparse columns is populated with rows with NULL values, and the other is populated with rows that aren't NULL. One table with no sparse columns is populated with rows with NULL values; the other is populated with rows that aren't NULL.

LISTING 8-5 Saving space with sparse columns

```
USE testdb;
GO
SET NOCOUNT ON;
GO

IF OBJECT_ID('sparse_nonulls_size') IS NOT NULL
               DROP TABLE sparse_nonulls_size;
GO
CREATE TABLE sparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF OBJECT_ID('nonsparse_nonulls_size') IS NOT NULL
               DROP TABLE nonsparse_nonulls_size;
GO
GO
CREATE TABLE nonsparse_nonulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
 );
GO
IF OBJECT_ID('sparse_nulls_size') IS NOT NULL
               DROP TABLE sparse_nulls_size;
GO
GO
CREATE TABLE sparse_nulls_size
(col1 int IDENTITY,
 col2 datetime SPARSE,
 col3 char(10) SPARSE
 );
GO
IF OBJECT_ID('nonsparse_nulls_size') IS NOT NULL
               DROP TABLE nonsparse_nulls_size;
GO
GO
CREATE TABLE nonsparse_nulls_size
(col1 int IDENTITY,
 col2 datetime,
 col3 char(10)
 );
GO
DECLARE @num int
SET @num = 1
WHILE @num < 100000
BEGIN
  INSERT INTO sparse_nonulls_size
        SELECT GETDATE(), 'my message';
  INSERT INTO nonsparse_nonulls_size
        SELECT GETDATE(), 'my message';
  INSERT INTO sparse_nulls_size
        SELECT NULL, NULL;
  INSERT INTO nonsparse_nulls_size
        SELECT NULL, NULL;
  SET @num = @num + 1;
END;
GO
```

Now look at the number of pages in each table. The following metadata query looks at the number of data pages in the *sys.allocation_units* view for each of the four tables:

```
SELECT object_name(object_id) as 'table with 100K rows', data_pages
FROM sys.allocation_units au
    JOIN sys.partitions p
       ON p.partition_id = au.container_id
WHERE object_name(object_id) LIKE '%sparse%size';
```

And here are my results:

```
table with 100K rows     data_pages
------------------------ ----------
sparse_nonulls_size      610
nonsparse_nonulls_size   402
sparse_nulls_size        169
nonsparse_nulls_size     402
```

Note that the smallest number of pages is required when the table has NULL sparse columns. If the table has no sparse columns, the space usage is the same whether or not the columns have NULLs because the data was defined as fixed length. This space requirement is more than twice as much as needed for the sparse columns with NULL. The worst case is if the columns have been defined as *SPARSE* but have no NULL values.

Of course, the previous examples are edge cases, where *all* the data is either NULL or non-NULL, and is of all fixed-length data types. So although you can say that sparse columns require more storage space for non-NULL values than is required for identical data that's not declared as *SPARSE*, the actual space savings depends on the data types and the percentage of rows that are NULL. Table 8-8—reprinted from *SQL Server Books Online*—shows the space usage for each data type. The NULL Percentage column indicates what percent of the data must be NULL to achieve a net space savings of 40 percent.

TABLE 8-8 Storage requirements for sparse columns

Data type	Storage bytes when not SPARSE	Storage bytes when SPARSE and bot NULL	NULL percentage
Fixed-length data types			
Bit	0.125	5	98 percent
Tinyint	1	5	86 percent
Smallint	2	6	76 percent
Int	4	8	64 percent
Bigint	8	12	52 percent
Real	4	8	64 percent
Float	8	12	52 percent
smallmoney	4	8	64 percent
Money	8	12	52 percent
smalldatetime	4	8	64 percent
Datetime	8	12	52 percent
uniqueidentifier	16	20	43 percent
Date	3	7	69 percent
Precision-dependent–length data types			
datetime2(0)	6	10	57 percent
datetime2(7)	8	12	52 percent
time(0)	3	7	69 percent
time(7)	5	9	60 percent
datetimetoffset(0)	8	12	52 percent
datetimetoffset (7)	10	14	49 percent
decimal/numeric(1,s)	5	9	60 percent
decimal/numeric(38,s)	17	21	42 percent

Data type	Storage bytes when not SPARSE	Storage bytes when SPARSE and bot NULL	NULL percentage
Data-dependent–length data types			
sql_variant	Varies		
varchar or char	2+avg. data	4+avg. data	60 percent
nvarchar or nchar	2+avg. data	4+avg. data	60 percent
varbinary or binary	2+avg. data	4+avg. data	60 percent
Xml	2+avg. data	4+avg. data	60 percent
hierarchyId	2+avg. data	4+avg. data	60 percent

The general recommendation is that you should consider using sparse columns when you anticipate that they provide a space savings of at least 20 to 40 percent.

Data compression

SQL Server provides the capability of data compression, a feature introduced in SQL Server 2008 and available in the Enterprise edition only. Compression can reduce the size of your tables by exploiting existing inefficiencies in the actual data. These inefficiencies can be grouped into two general categories.

- The first category relates to storage of individual data values when they are stored in columns defined using the maximum possible size. For example, a table might need to define a *quantity* column as *int* because occasionally you could be storing values larger than 32,767, which is the maximum *smallint* value. However, *int* columns always need 4 bytes, and if most of your *quantity* values are less than 100, those values could be stored in *tinyint* columns, which need only 1 byte of storage. The Row Compression feature of SQL Server can compress individual columns of data to use only the actual amount of space required.
- The second type of inefficiency in the data storage occurs when the data on a page contains duplicate values or common prefixes across columns and rows. This inefficiency can be minimized by storing the repeating values only once and then referencing those values from other columns. The Page Compression feature of SQL Server can compress the data on a page by maintaining entries containing common prefixes or repeating values. Note that when you choose to apply page compression to a table or index, SQL Server always also applies Row Compression.

Vardecimal

SQL Server 2005 SP2 introduced a simple form of compression, which could be applied only to columns defined using the *decimal* data type. (Keep in mind that the data type *numeric* is completely equivalent to *decimal*, and anytime I mention *decimal*, it also means *numeric*.) In SQL Server 2005, the option must be enabled at both the database level (using the procedure *sp_db_vardecimal_storage_format*) and at the table level (using the procedure *sp_tableoption*). In SQL Server 2008, SQL Server

2008 R2, and SQL Server 2012, all user databases are enabled automatically for the vardecimal storage format, so vardecimal must be enabled only for individual tables. Like data compression, which this section looks at in detail, the vardecimal storage format is available only in SQL Server Enterprise edition.

In SQL Server 2005, when both of these stored procedures were run, *decimal* data in the tables enabled for vardecimal were stored differently. Rather than be treated as fixed-length data, *decimal* columns are stored in the variable section of the row and use only the number of bytes required. (Chapter 6 looked at the difference between fixed-length data and variable-length data storage.) In addition to all the table partitions that use the vardecimal format for all *decimal* data, all indexes on the table automatically use the vardecimal format.

Decimal data values are defined with a precision of between 1 and 38 and, depending on the defined precision, they use between 5 and 17 bytes. Fixed-length *decimal* data uses the same number of bytes for every row, even if the actual data could fit into far fewer bytes. When a table doesn't use the vardecimal storage format, every entry in the table consumes the same number of bytes for each defined decimal column, even if the value of a row is 0, NULL, or some value that can be expressed in a smaller number of bytes, such as the number 3. When vardecimal storage format is enabled for a table, the *decimal* columns in each row use the minimum amount of space required to store the specified value. Of course, as you saw in Chapter 6, every variable-length column has 2 bytes of additional overhead associated with it, but when storing very small values in a column defined as *decimal* with a large precision, the space saving can more than make up for those additional 2 bytes. For vardecimal storage, both NULLs and zeros are stored as zero-length data and use only the 2 bytes of overhead.

Although SQL Server 2012 supports the vardecimal format, I recommend that you use row compression when you want to reduce the storage space required by your data rows. Both the table option and the database option for enabling vardecimal storage have been deprecated.

Row compression

You can think of row compression as an extension of the vardecimal storage format. In many situations, SQL Server uses more space than needed to store data values, and without the Row Compression feature, the only control you have is to use a variable-length data type. Any fixed-length data types always use the same amount of space in every row of a table, even if space is wasted.

As mentioned earlier, you can declare a column as type *int* because occasionally you might need to store values greater than 32,767. An *int* needs 4 bytes of space, no matter what number is stored, even if the column is NULL. Only character and binary data can be stored in variable-length columns (and, of course, decimal, when that option is enabled). Row compression allows integer values to use only the amount of storage space required, with the minimum being 1 byte. A value of 100 needs only a single byte for storage, and a value of 1,000 needs 2 bytes. The storage engine also includes an optimization that allows zero and NULL to use no storage space for the data itself.

Later, this section provides the details about compressed data storage. Starting in SQL Server 2008 R2, row compression also can compress Unicode data. Rather than each Unicode character always be stored in two bytes, if the character needs only a single byte, it is stored only in a single byte.

Enabling row compression

You can enable compression when creating a table or index, or when using the *ALTER TABLE* or *ALTER INDEX* command. Also, if the table or index is partitioned, you can choose to compress just a subset of the partitions. (You'll look at partitioning later in this chapter.)

The script in Listing 8-6 creates two copies of the *dbo.Employees* table in the *AdventureWorks2012* database. When storing row-compressed data, SQL Server treats values that can be stored in 8 bytes or fewer (that is, short columns) differently than it stores data that needs more than 8 bytes (long columns). For this reason, the script updates one of the rows in the new tables so that none of the columns in that row contains more than 8 bytes. The *Employees_rowcompressed* table is then enabled for row compression, and the *Employees_uncompressed* table is left uncompressed. A metadata query examining pages allocated to each table is executed against each table so that you can compare the sizes before and after row compression.

LISTING 8-6 Comparing two tables to show row compression

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('Employees_uncompressed') IS NOT NULL
               DROP TABLE Employees_uncompressed;
GO
GO
SELECT e.BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_uncompressed

  FROM HumanResources.Employee e
   JOIN Person.Person p
        ON e.BusinessEntityID = p.BusinessEntityID;
GO
UPDATE Employees_uncompressed
SET NationalIDNumber = '1111',
        JobTitle = 'Boss',
        LastName = 'Gato'
WHERE FirstName = 'Ken'
AND LastName = 'Sánchez';
GO
ALTER TABLE dbo.Employees_uncompressed
   ADD CONSTRAINT EmployeeUn_ID
       PRIMARY KEY (BusinessEntityID);
GO
SELECT OBJECT_NAME(object_id) as name,
        rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
        ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_uncompressed');

IF OBJECT_ID('Employees_rowcompressed') IS NOT NULL
               DROP TABLE Employees_rowcompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_rowcompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_rowcompressed
   ADD CONSTRAINT EmployeeR_ID
       PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_rowcompressed
REBUILD WITH (DATA_COMPRESSION = ROW);
GO
SELECT OBJECT_NAME(object_id) as name,
        rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
        ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_rowcompressed');
GO
```

The *dbo.Employees_rowcompressed* table is referred to again later in this section, or you can examine it on your own as the details of compressed row storage are covered.

Now you can start looking at the details of row compression, but keep these points in mind:

- Row compression is available only in SQL Server 2008, SQL Server 2008 R2, and SQL Server 2012 Enterprise and Developer editions.
- Row compression doesn't change the maximum row size of a table or index.
- Row compression can't be enabled on a table with any columns defined as *SPARSE*.
- If a table or index has been partitioned, row compression can be enabled on all the partitions or on a subset of the partitions.

New row format

Chapter 6 looked at the format for storing rows that has been used since SQL Server 7.0 and is still used in SQL Server 2012 if you haven't enabled compression. That format is referred to as the *FixedVar* format because it has a fixed-length data section separate from a variable-length data section. A completely new row format was introduced in SQL Server 2008 for storing compressed rows, and this format is referred to as CD format. The term *CD*, which stands for "column descriptor," refers to every column having description information contained in the row itself.

You might want to re-examine Figure 6-6 in Chapter 6 as a reminder of what the *FixedVar* format looks like and compare it to the new CD format. Figure 8-9 shows an abstraction of the CD format. It's difficult to be as specific as Figure 6-6 is because except for the header, the number of bytes in each region is completely dependent on the data in the row.

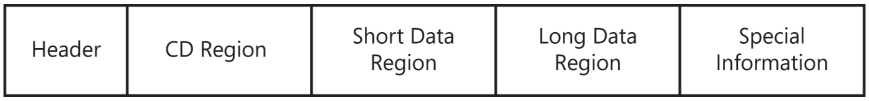


FIGURE 8-9 General structure of a CD record.

Each of these sections is described in detail.

Header The row header is always a single byte and roughly corresponds to what Chapter 6 referred to as Status Bits A. The bits have the following meanings.

- **Bit 0** This bit indicates the type of record; it's 1 for the new CD record format.
- **Bit 1** This bit indicates that the row contains versioning information.
- **Bits 2 through 4** Taken as a 3-bit value, these bits indicate what kind of information is stored in the row. The possible values are as follows:

  000 Primary record

  001 Ghost empty record

  010 Forwarding record

  011 Ghost data record

  100 Forwarded record

  101 Ghost forwarded record

  110 Index record

  111 Ghost index record
- **Bit 5** This bit indicates that the row contains a long data region (with values greater than 8 bytes in length).
- **Bits 6 and 7** These bits are not used in SQL Server 2012.

The CD region The CD region is composed of two parts. The first part is either 1 or 2 bytes, indicating the number of short columns. If the most significant bit of the first byte is set to 0, it's a 1-byte field with a maximum value of 127. If a table has more than 127 columns, the most significant bit is 1, and SQL Server uses 2 bytes to represent the number of columns, which can be up to 32,767.

Following the 1 or 2 bytes for the number of columns is the CD array, which uses 4 bits for each column in the table to represent information about the length of the column. Four bits can have 16 different possible values, but in SQL Server 2012, only 13 of them are used.

- 0 (0x0) indicates that the corresponding column is NULL.
- 1 (0x1) indicates that the corresponding column is a 0-byte short value.
- 2 (0x2) indicates that the corresponding column is a 1-byte short value.
- 3 (0x3) indicates that the corresponding column is a 2-byte short value.
- 4 (0x4) indicates that the corresponding column is a 3-byte short value.
- 5 (0x5) indicates that the corresponding column is a 4-byte short value.

- 6 (0x6) indicates that the corresponding column is a 5-byte short value.
- 7 (0x7) indicates that the corresponding column is a 6-byte short value.
- 8 (0x8) indicates that the corresponding column is a 7-byte short value.
- 9 (0x9) indicates that the corresponding column is an 8-byte short value.
- 10 (0xa) indicates that the corresponding column is long data value and uses no space in the short data region.
- 11 (0xb) is used for columns of type bit with the value of 1. The corresponding column takes no space in the short data region.
- 12 (0xc) indicates that the corresponding column is a 1-byte symbol, representing a value in the page dictionary. (Later, the section "Page compression" talks about the dictionary).

The short data region The short data region doesn't need to store the length of each short data value because that information is available in the CD region. However, if table has hundreds of columns, accessing the last columns can be expensive. To minimize this cost, columns are grouped into clusters of 30 columns each and at the beginning of the short data region is an area called the short data cluster array. Each array entry is a single-byte integer and indicates the sum of the sizes of all the data in the previous cluster in the short data region, so that the value is basically a pointer to the first column of the cluster. The first cluster of short data starts right after the cluster array, so no cluster offset is needed for it. A cluster might not have 30 data columns, however, because only columns with a length less than or equal to 8 bytes are stored in the short data region.

As an example, consider a row with 64 columns, and columns 5, 10, 15, 20, 25, 30, 40, 50, and 60 are long data, and the others are short. The CD region contains the following.

- A single byte containing the value 64, the number of columns in the CD region.
- A CD array of 4 * 64 bits, or 32 bytes, containing information about the length of each column. It has 55 entries with values indicating an actual data length for the short data, and 8 entries of 0xa, indicating long data.

The short data region contains the following.

- A short data cluster offset array containing the two values, each containing the length of a short data cluster. In this example, the first cluster, which is all the short data in the first 30 columns, has a length of 92, so the 92 in the offset array indicates that the second cluster starts 92 bytes after the first. The number of clusters can be calculated as (Number of columns – 1) /30. The maximum value for any entry in the cluster array is 240, if all 30 columns were short data of 8 bytes in length.
- All the short data values.

Figure 8-10 illustrates the CD region and the short data region with sample data for the row described previously. The CD array is shown in its entirety, with a symbol indicating the length of each of the 64 values. (So the depiction of this array can fit on a page of this book, the actual data values aren't shown.) The first cluster has 24 values in the short data region (6 are long values), the second cluster has 27 (3 are long), and the third cluster has the remaining 4 columns (all short).

CD Region		Short Data Region				
Number of columns	CD array --64 4-bit values ('a' indicates long column)	Length of short data in each 30-column cluster (N–1)/30 values		Three clusters of actual data		
N = 64	3285a4358a6543a3456a6666a5463a 254372644a745269277a463495736a 5433	92	106	24 values	27 values	4 values

FIGURE 8-10 The CD region and short data region in a CD record.

To locate the entry for a short column value in the short data region, the short data cluster array is first examined to determine the start address of the containing cluster for the column in the short data region.

The long data region Any data in the row longer than 8 bytes is stored in the long data region. This includes complex columns, which don't contain actual data but instead contain information necessary to locate data stored off the row. This can include large object data and row overflow data pointers. Unlike short data, where the length can be stored simply in the CD array, long data needs an actual offset value to allow SQL Server to determine the location of each value. This offset array looks very similar to the offset array discussed in Chapter 6 for the *FixedVar* records.

The long data region is composed of three parts: an offset array, a long data cluster array, and the long data.

The offset array is composed of the following.

- **A 1-byte header in which currently only the first two bits are used** Bit 0 indicates whether the long data region contains any 2-byte offset values. Currently, this value is always 1, because all offsets are always 2 bytes. Bit 1 indicates whether the long data region contains any complex columns.
- **A 2-byte value indicating the number of offsets to follow** The most significant bit in the first byte of the offset value indicates whether the corresponding entry in the long data region is a complex column. The rest of the bits/bytes in the array entry store the ending offset value for the corresponding entry in the long data region.

Similar to the cluster array for the short data, the long data cluster array is used to limit the cost of finding columns near the end of a long list of columns. It has one entry for each 30-column cluster (except the last one). Because the offset of each long data column is already stored in the offset array, the cluster array just needs to keep track of how many of the long data values are in each cluster. Each value is a 1-byte integer representing the number of long data columns in that cluster. Just as for the short data cluster, the number of entries in the cluster array can be computed as (Number of columns in the table – 1)/30.

Figure 8-11 illustrates the long data region for the row described previously, with 64 columns, nine of which are long. Values for the offsets aren't included for space considerations. The long data cluster array has two entries indicating that six of the values are in the first cluster and two are in the second. The remaining values are in the last cluster.

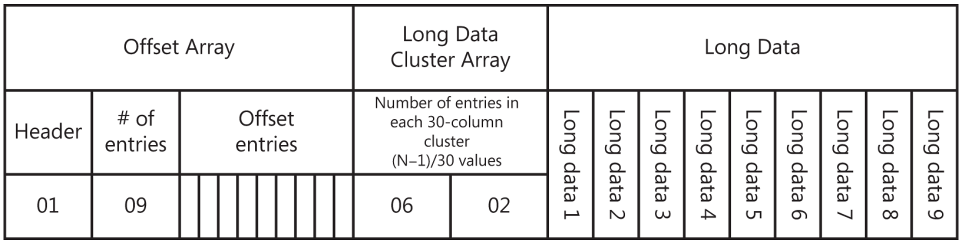
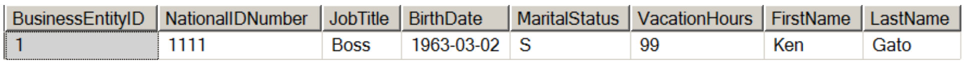


FIGURE 8-11 The long data region of a CD record.

Special information The end of the row contains three optional pieces of information. The existence of any or all of this information is indicated by bits in the 1-byte header at the very beginning of the row.

- **Forwarding pointer** This value is used when a heap contains a forwarding stub that points to a new location to which the original row has been moved. Chapter 6 discussed forwarding pointers. The forwarding pointer contains three header bytes and an 8-byte row ID.
- **Back pointer** This value is used in a row that has been forwarded to indicate the original location of the row. It's stored as an 8-byte Row ID.
- **Versioning info** When a row is modified under one of the snapshot-based isolation levels, SQL Server adds 14 bytes of versioning information to the row. Chapter 13 discusses row versioning and Snapshot isolation.

Now look at the actual bytes in two of the rows in the *dbo.Employees_rowcompressed* table created earlier in Listing 8-6. The *DBCC PAGE* command gives additional information about compressed rows and pages. In particular, before the bytes for the row are shown, *DBCC PAGE* displays the CD array. For the first row returned on the first page in the *dbo.Employees_rowcompressed* table, all the columns contain short data. The row has the data values shown here:

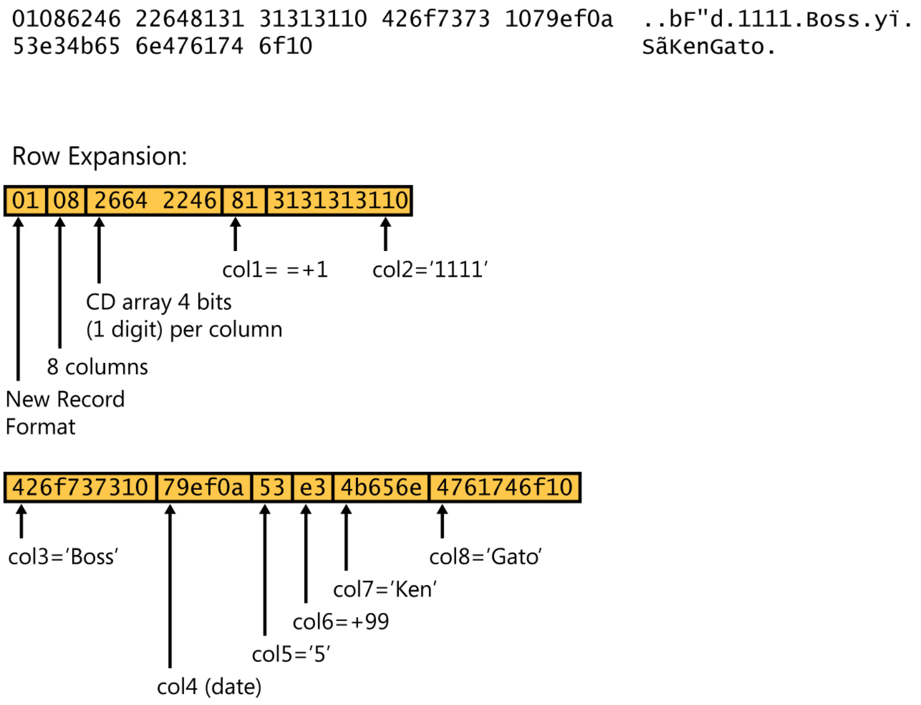


BusinessEntityID	NationalIDNumber	JobTitle	BirthDate	MaritalStatus	VacationHours	FirstName	LastName
1	1111	Boss	1963-03-02	S	99	Ken	Gato

For short data, the CD array contains the actual length of each of the columns, and you can see the following information for the first row in the *DBCC PAGE* output:

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x06 (FIVE_BYTE_SHORT)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x06 (FIVE_BYTE_SHORT)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x06 (FIVE_BYTE_SHORT)
```

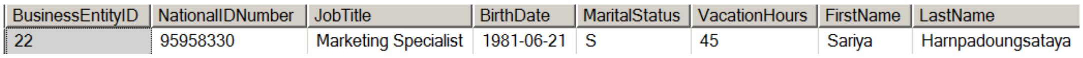
So the first column has a CD code of 0x02, which indicates a 1-byte value, and, as you can see in the data row, is the integer 1. The second column contains a 5-byte value and is the Unicode string 1111. Notice that compressed Unicode strings are always an odd number of bytes. This is how SQL Server determines that the string has actually been compressed, because an uncompressed Unicode string—which needs 2 bytes for each character—will always be an even number of bytes. Because the Unicode string has an even number of characters, SQL Server adds a single byte 0x01 as a terminator. In Figure 8-12, which shows the *DBCC PAGE* output for the row contents, you can see that three strings have the 0x01 terminator to make their length odd: '1111', 'Boss', and 'Gato'. I'll leave it to you to inspect the codes for the remaining columns.



```
01086246 22648131 31313110 426f7373 1079ef0a   ..bF"d.1111.Boss.yï.
53e34b65 6e476174 6f10                         SãKenGato.
```

FIGURE 8-12 A compressed row with eight short data columns.

Now look at a row with some long columns. The 22nd row on the page (Slot 21) has three long columns in the data values shown here:



BusinessEntityID	NationalIDNumber	JobTitle	BirthDate	MaritalStatus	VacationHours	FirstName	LastName
22	95958330	Marketing Specialist	1981-06-21	S	45	Sariya	Harnpadoungsataya

The CD array for this row looks like the following:

```
CD array entry = Column 1 (cluster 0, CD array offset 0): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 2 (cluster 0, CD array offset 0): 0x0a (LONG)
CD array entry = Column 3 (cluster 0, CD array offset 1): 0x0a (LONG)
CD array entry = Column 4 (cluster 0, CD array offset 1): 0x04 (THREE_BYTE_SHORT)
CD array entry = Column 5 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 6 (cluster 0, CD array offset 2): 0x02 (ONE_BYTE_SHORT)
CD array entry = Column 7 (cluster 0, CD array offset 3): 0x08 (SEVEN_BYTE_SHORT)
CD array entry = Column 8 (cluster 0, CD array offset 3): 0x0a (LONG)
```

Figure 8-13 shows the bytes that *DBCC PAGE* returns for this data row. The bytes in the long data region are boldfaced.

```
Record Memory Dump
000000001492A57A:   2108a24a 22a89697 090b53ad 53617269 79611001  !.¢J"¨–– .SSariya..
000000001492A58E:   03000900 1e002f00 39353935 38333330 104d6172  .. .../.95958330.Mar
000000001492A5A2:   6b657469 6e672053 70656369 616c6973 74104861  keting Specialist.Ha
000000001492A5B6:   726e7061 646f756e 67736174 617961             rnpadoungsataya
```

FIGURE 8-13 A compressed row with five short data columns and three long.

Notice the following in the first part of the row, before the long data region.

- The first byte in the row is 0x21, indicating that not only is this row in the new CD record format, but also that the row contains a long data region.
- The second byte indicates eight columns in the table, just as for the first row.
- The following 4 bytes for the CD array has three values of *a*, which indicate long values not included in the short data region.
- The short data values are listed in order after the CD array and are as follows:
  - The *BusinessEntityID* is 1 byte, with the value 0x96, or +22.
  - The *Birthdate* is 3 bytes.
  - The *MaritalStatus* is 1 byte, with the value 0x0053, or 'S'.
  - The *VacationHours* is 1 byte, with the value 0xad, or +45.
  - The *FirstName* is 7 bytes, with the value 53617269796110 or *'Sariya'*.

The Long Data Region Offset Array is 8 bytes long, as follows.

- The first byte is 0x01, which indicates that the row-offset positions are 2 bytes long.
- The second byte is 0x03, which indicates three columns in the long data region.
- The next 6 bytes are the 2-byte offsets for each of the three values. Notice that the offset refers to position the column ends with the Long Data area itself.
    - The first 2-byte offset is 0x0009, which indicates that the first long value is 9 bytes long.
    - The second 2-byte offset is 001e, or 30, which indicates that the second long value ends 21 bytes after the first. The second value is *Marketing Specialist*, which is a 21-byte string.
    - The third 2-byte offset is 0x002f, or 47, which indicates the third value, *Harnpadoungsataya*, ends 17 bytes after the second long value.

Fewer than 30 columns means no Long Data Cluster Array, but the data values are stored immediately after the Long Data Region Offset Array.

Because of space constraints, this chapter won't show you the details of a row with multiple column clusters (that is, more than 30 columns), but you should have enough information to start exploring such rows on your own.

Page compression

In addition to storing rows in a compressed format to minimize the space required, SQL Server 2012 can compress whole pages by isolating and reusing repeating patterns of bytes on the page.

Unlike row compression, page compression is applied only after a page is full, and only if SQL Server determines that compressing the page saves a meaningful amount of space. (You'll find out what that amount is later in this section.) Keep the following points in mind when planning for page compression.

- Page compression is available only in the SQL Server 2008, SQL Server 2008 R2, and SQL Server 2012 Enterprise and Developer editions.
- Page compression always includes row compression—that is, if you enable page compression for a table, row compression is automatically enabled.
- When compressing a B-tree, only the leaf level can be page compressed. For performance reasons, the node levels are left uncompressed.
- If a table or index has been partitioned, page compression can be enabled on all the partitions or on a subset of the partitions.

The code in Listing 8-7 makes another copy of the *dbo.Employees* table and applies page compression to it. It then captures the page location and linkage information from *DBCC IND* for the three tables: *dbo.Employees_uncompressed*, *dbo.Employees_rowcompressed*, and *dbo. Employees_pagecompressed*. The code then uses the captured information to report on the number of data pages in each of the three tables.

LISTING 8-7 Applying page compression to a table

```
USE AdventureWorks2012;
GO
IF OBJECT_ID('Employees_pagecompressed') IS NOT NULL
               DROP TABLE Employees_pagecompressed;
GO
SELECT BusinessEntityID, NationalIDNumber, JobTitle,
        BirthDate, MaritalStatus, VacationHours,
        FirstName, LastName
  INTO Employees_pagecompressed
  FROM dbo.Employees_uncompressed
GO
ALTER TABLE dbo.Employees_pagecompressed
   ADD CONSTRAINT EmployeeP_ID
       PRIMARY KEY (BusinessEntityID);
GO
ALTER TABLE dbo.Employees_pagecompressed
REBUILD WITH (DATA_COMPRESSION = PAGE);
GO
SELECT OBJECT_NAME(object_id) as name,
                 rows, data_pages, data_compression_desc
FROM sys.partitions p JOIN sys.allocation_units au
                ON p.partition_id = au.container_id
WHERE object_id = object_id('dbo.Employees_pagecompressed');
GO

SELECT object_name(object_id) as Table_Name, count(*) as Page_Count
FROM sys.dm_db_database_page_allocations(db_id('AdventureWorks2012'), null, null, null,
'DETAILED')
WHERE object_name(object_id) like ('Employees%compressed')
AND page_type_desc = 'DATA_PAGE'
GROUP BY object_name(object_id);
```

If you run this script, notice in the output that row compression reduced the size of this small table from five pages to three, and then page compression further reduced the size from three pages to two.

SQL Server can perform two different operations to try to compress a page by using common values: *column prefix compression* and *dictionary compression.*

Column prefix compression

As the name implies, column prefix compression works on data columns in the table being compressed, but it looks only at the column values on a single page. For each column, SQL Server chooses a common prefix that can be used to reduce the storage space required for values in that column. The longest value in the column that contains that prefix is chosen as the *anchor value*. Each column is then stored—not as the actual data value, but as a delta from the anchor value. Suppose that you have the following character values in a column of a table to be page-compressed:

```
DEEM
DEE
FFF
DEED
DEE
DAN
```

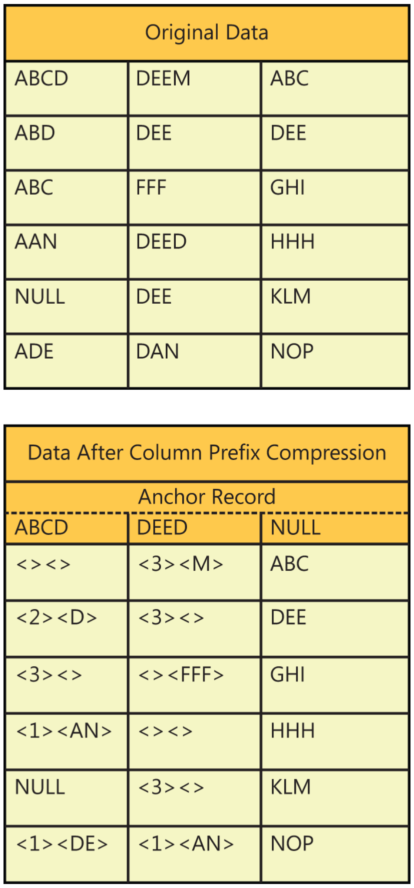
SQL Server might note that DEE is a useful common prefix, so *DEED* is chosen as the anchor value. Each column would be stored as the difference between its value and the anchor value. This difference is stored as a two-part value: the number of characters from the anchor to use and the additional characters to append. So *DEEM* is stored as <3><M>, meaning the value uses the first three characters from the common prefix and appends a single character, *M*, to it. *DEED* is stored as an empty string (but not null) to indicate it matched the prefix exactly. *DEE* is stored as <3>, with the second part empty, because no additional characters can be appended. The list of column values is replaced by the values shown here:

```
DEEM -> <3><M>
DEE -> <3><>
FFF -> <><FFF>
DEED -> <><>
DEE -> <3><>
DAN -> <1><AN>
```

Keep in mind that the compressed row is stored in the CD record format, so the CD array value has a special encoding to indicate the value is actually NULL. If the replacement value is <><>, and the encoding doesn't indicate NULL, the value matches the prefix exactly.

SQL Server applies the prefix detection and value replacement algorithm to every column and creates a new row called an *anchor record* to store the anchor values for each column. If no useful prefix can be found, the value in the anchor record is NULL, and then all the values in the column are stored just as they are.

Figure 8-14 shows an image of six rows in a table page compression, and then shows the six rows after the anchor record has been created and the substitutions have been made for the actual data values.



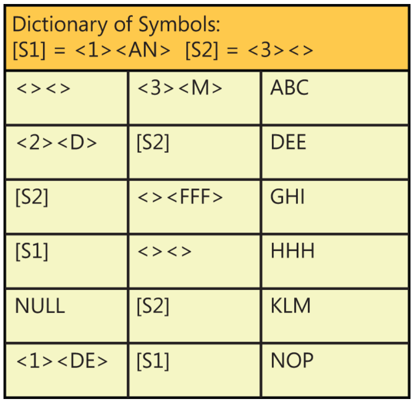
Original Data		
ABCD	DEEM	ABC
ABD	DEE	DEE
ABC	FFF	GHI
AAN	DEED	HHH
NULL	DEE	KLM
ADE	DAN	NOP

Data After Column Prefix Compression		
Anchor Record		
ABCD	DEED	NULL
<><>	<3><M>	ABC
<2><D>	<3><>	DEE
<3><>	<><FFF>	GHI
<1><AN>	<><>	HHH
NULL	<3><>	KLM
<1><DE>	<1><AN>	NOP

FIGURE 8-14 Before and after column prefix compression.

Dictionary compression

After prefix compression is applied to every column individually, the second phase of page compression looks at all values on the page to find duplicates in any column of any row, even if they have been encoded to reflect prefix usage. You can see in the bottom part of Figure 8-16 that two of the values occur multiple times: <3><> occurs three times and <1><AN> occurs twice. The process of detecting duplicate values is data type–agnostic, so values in completely different columns could be the same in their binary representation. For example, a 1-byte character is represented in hex as 0x54, and it would be seen as a duplicate of the 1-byte integer 84, which is also represented in hex as 0x54. The dictionary is stored as a set of symbols, each of which corresponds to a duplicated value on the data page. After the symbols and data values are determined, each occurrence of one of the duplicated values is replaced by the symbol. SQL Server recognizes that the value actually stored in the column is a symbol and not a data value by examining the encoding in the CD array. Values which have been replaced by symbols have a CD array value of 0xc. Figure 8-15 shows the data from Figure 8-14 after replacing the five values with symbols.



Dictionary of Symbols:
[S1] = <1><AN> [S2] = <3><>

<><>	<3><M>	ABC
<2><D>	[S2]	DEE
[S2]	<><FFF>	GHI
[S1]	<><>	HHH
NULL	[S2]	KLM
<1><DE>	[S1]	NOP

FIGURE 8-15 A page compressed with dictionary compression.

Not every page in a compressed table has both an anchor record for prefixes and a dictionary. If no useful prefix values are available, the page might have just a dictionary. If no values repeat often enough that replacing them with symbols saves space, the page might have just an anchor record. Of course, some pages might have neither an anchor record nor a dictionary if the data on the page has no patterns at all.

Physical storage

When a page is compressed, only one main structural change occurs. SQL Server adds a hidden row right after the page header (at byte offset 96, or 0x60) called the compression information (CI) record. Figure 8-16 shows the structure of the CI record.

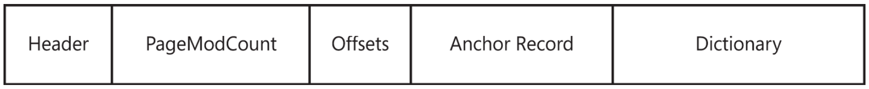


FIGURE 8-16 Structure of a CI record.

The CI record doesn't have an entry in the slot array for the page, but it's always at the same location. Also, a bit in the page header indicates that the page is page-compressed, so SQL Server looks for the CI record. If you use *DBCC PAGE* to dump a page, the page header information contains a value called *m_typeFlagBits*. If this value is 0x80, the page is compressed.

You can run the following script to use the *sys.dm_db_database_page_allocations* function to find the page ID (PID) of the first page and the file ID (FID) of the first page for each of the three tables that you've been exploring. You can use this information to examine the page with *DBCC PAGE*. Notice that only the page for *Employees_pagecompressed* has the *m_typeFlagBits* value set to 0x80.

```
USE AdventureWorks2012;
GO
SELECT object_name(object_id) as Table_Name, allocated_page_file_id as First_Page_FID,
       allocated_page_page_id as First_Page_PID
```

```
FROM sys.dm_db_database_page_allocations(db_id('AdventureWorks2012'),
       null, null, null, 'DETAILED')
WHERE object_name(object_id) like ('Employees%compressed')
AND page_type_desc = 'DATA_PAGE'
AND previous_page_page_id  IS NULL;
```

Using *DBCC PAGE* to look at a page-compressed page does provide information about the contents of the CI record, and you'll look at some of that information after examining what each section means.

Header The header is a 1-byte value keeping track of information about the CI. Bit 0 indicates the version, which in SQL Server 2012 is always 0. Bit 1 indicates whether the CI has an anchor record, and bit 2 indicates whether the CI has a dictionary. The rest of the bits are unused.

PageModCount The *PageModCount* value keeps track of the changes to this particular page and is used when determining whether the compression on the page should be reevaluated, and a new CI record built. The next section, "Page compression analysis," talks more about how this value is used.

Offsets The offsets contain values to help SQL Server find the dictionary. It contains a value indicating the page offset for the end of the anchor record and a value indicating the page offset for the end of the CI record itself.

Anchor Record The anchor record looks exactly like a regular CD record on the page, including the record header, the CD array, and both a short data area and a long data area. The values stored in the data area are the common prefix values for each column, some of which might be NULL.

Dictionary The dictionary area is composed of three sections.

- A 2-byte field containing a numeric value representing the number of entries in the dictionary
- An offset array of 2-byte entries, indicating the end offset of each dictionary entry relative to the start of the dictionary data section
- The actual dictionary data entries

Remember that each dictionary entry is a byte string that is replaced in the regular data rows by a symbol. The symbol is simply an integer value from 0 to N. Also, remember that the byte strings are data type–independent—that is, they are just bytes. After SQL Server determines what recurring values are stored in the dictionary, it sorts the list first by data length, then by data value, and then assigns the symbols in order. So suppose that the values to be stored in the dictionary are these:

```
0x 53 51 4C
0x FF F8
0x DA 15 43 77 64
0x 34 F3 B6 22 CD
0x 12 34 56
```

Table 8-9 shows the sorted dictionary, along with the length and symbol for each entry.

TABLE 8-9 Values in a page compression dictionary

Value	Length	Symbol
0x FF F8	2 bytes	0
0x 12 34 56	3 bytes	1
0x 53 51 4C	3 bytes	2
0x 34 F3 B6 22 CD	4 bytes	3
0x DA 15 43 77 64	4 bytes	4

The dictionary area would then resemble Figure 8-17.

Header	Offsets	Dictionary
5	02 00	0x FF F8
	05 00	0x 12 34 56
	08 00	0x 53 51 4C
	0D 00	0x 34 F3 B6 22 CD
	12 00	0x DA 15 43 77 64

FIGURE 8-17 Dictionary area in a compression information record.

Note that the dictionary never actually stores the symbol values. They are stored only in the data records that need to use the dictionary. Because they are simply integers, they can be used as an index into the offset list to find the appropriate dictionary replacement value. For example, if a row on the page contains the dictionary symbol [2], SQL Server looks in the offset list for the third entry, which in Figure 8-17 ends at offset 0800 from the start of the dictionary. SQL Server then finds the value that ends at that byte, which is 0x 53 51 4C. If this byte string was stored in a *char* or *varchar* column—that is, a single-byte character string—it would correspond to the character string *SQL*.

Earlier in this chapter, you saw that the *DBCC PAGE* output displays the CD array for compressed rows. For compressed pages, *DBCC PAGE* shows the CI record and details about the anchor record within it. Also, with format 3, *DBCC PAGE* shows details about the dictionary entries. When I captured the *DBCC PAGE* in format 3 for the first page of my *Employees_pagecompressed* table and copied it to a Microsoft Office Word document, it needed 384 pages. Needless to say, I won't show you all that output (just copying the CI record information required 10 pages, which is still too much to show in this book). You can explore the output of *DBCC PAGE* for the tables with compressed pages on your own.

Page compression analysis

This section covers some of the details regarding how SQL Server determines whether to compress a page and what values it uses for the anchor record and the dictionary. Row compression is always performed when requested, but page compression depends on the amount of space that can be saved. However, the actual work of compressing the rows has to wait until after page compression is performed. Because both types of page compression—prefix substitution and dictionary symbol substitution—replace the actual data values with encodings, the row can't be compressed until SQL Server determines what encodings will replace the actual data.

When page compression is first enabled for a table or partition, SQL Server goes through every full page to determine the possible space savings. (Any pages that aren't full aren't considered for compression.) This compression analysis actually creates the anchor record, modifies all the columns to reflect the anchor values, and generates the dictionary. Then it compresses each row. If the new compressed page can hold at least five more rows, or 25 percent more rows than the current page (whichever is larger), the compressed page replaces the uncompressed page. If compressing the page doesn't result in this much savings, the compressed page is discarded.

When determining what values to use for the anchor record on a compressed page, SQL Server needs to look at every byte in every row, one column at a time. As it scans the column, it also keeps track of possible dictionary entries that can be used in multiple columns. The anchor record values can be determined for each column in a single pass—that is, by the time all the bytes in all the rows for the first column are examined once, SQL Server has determined the anchor record value for that column or has determined that no anchor record value will save sufficient space.

As SQL Server examines each column, it collects a list of possible dictionary entries. As discussed earlier, the dictionary contains values that occur enough times on the page so that replacing them with a symbol is cost-effective in terms of space. For each possible dictionary entry, SQL Server keeps track of the value, its size, and the count of occurrences. If (size_of_data_value –1) * (count–1) –2 is greater than zero, it means the dictionary replacement saves space, and the value is considered eligible for the dictionary. Because the dictionary symbols are single-byte integers, SQL Server tries can't store more than 255 entries in the dictionary on any page, so if more dictionary entries might be used based on the data on the page, they are sorted by number of occurrences during the analysis, and only the most frequently occurring values are used in the dictionary.

CI record rebuilding

If a table is enabled for either page or row compression, new rows are always compressed before they are inserted into the table. However, the CI record containing the anchor record and the dictionary is rebuilt on an all-or-nothing basis—that is, SQL Server doesn't just add some new entry to the dictionary when new rows are inserted. SQL Server evaluates whether to rebuild the CI record when the page has been changed a sufficient number of times. It keeps track of changes to each page in the *PageModCount* field of the CI record, and that value is updated every time a row is inserted, updated, or deleted.

If a full page is encountered during a data modification operation, SQL Server examines the *PageModCount* value. If the *PageModCount* value is greater than 25 or the value *PageModCount/*

<*number of rows on the page*> is greater than 25 percent, SQL Server applies the compression analysis as it does when it first compresses a page. Only when recompressing the page makes room for at least five more rows (or 25 percent more rows than the current page) does the new compressed page replace the old page.

Page compression in a B-tree and page compression in a heap each have important differences.

Compression of B-tree pages For B-trees, only the leaf level is page compressed. When inserting a new row into a B-tree, if the compressed row fits on the page, it is inserted, and nothing more is done. If it doesn't fit, SQL Server tries to recompress the page, according to the conditions described in the preceding section. A successful recompression means that the CI record changed, so the new row must be recompressed and then SQL Server tries to insert it into the page. Again, if it fits, it is simply inserted; if the new compressed row doesn't fit on the page, even after possibly recompressing the page, the page needs to be split. When splitting a compressed page, the CI record is copied to a new page exactly as is, except that the *PageModCount* value is set to 25. This means that the first time the page gets full, it gets a full analysis to determine whether it should be recompressed. B-tree pages are also checked for possible recompression during index rebuilds (either online or offline) and during shrink operations.

Compression of heap pages Pages in a heap are checked for possible compression only during rebuild and shrink operations. Also, if you drop a clustered index on a table so that it becomes a heap, SQL Server runs compression analysis on any full pages. To make sure that the *RowID* values stay the same, heaps aren't recompressed during typical data modification operations. Although the *PageModCount* value is maintained, SQL Server never tries to recompress a page based on the *PageModCount* value.

Compression metadata

An enormous amount of metadata information relating to data compression doesn't exist. The catalog view *sys.partitions* has a *data_compression* column and a *data_compression_desc* column. The *data_compression* column has possible values of 0, 1, 2, and 3 corresponding to *data_compression_desc* values of *NONE*, *ROW*, *PAGE*, and *COLUMNSTORE*. (Only *ROW* and *PAGE* compression are discussed here.) Keep in mind that although row compression is always performed if enabled, page compression isn't. Even if *sys.partitions* indicates that a table or partition is page compressed, that just means that page compression is enabled. Each page is analyzed individually, and if a page isn't full, or if compression won't save enough space, the page isn't compressed.

You can also inspect the dynamic management function *sys.dm_db_index_operational_stats*. This table-valued function returns the following compression-related columns.

- ***page_compression_attempt_count*** The number of pages evaluated for *PAGE*-level compression for specific partitions of a table, index, or indexed view. This includes pages that weren't compressed because significant savings couldn't be achieved.
- ***page_compression_success_count*** The number of data pages that were compressed by using *PAGE* compression for specific partitions of a table, index, or indexed view.

SQL Server also provides a stored procedure called *sp_estimate_data_compression_savings,* which can give you some idea of whether compression provides a large space savings. This procedure samples up to 5,000 pages of the table and creates an equivalent table with the sampled pages in *tempdb*. Using this temporary table, SQL Server can estimate the new table size for the requested compression state (*NONE*, *ROW*, or *PAGE*). Compression can be evaluated for whole tables or parts of tables, including heaps, clustered indexes, nonclustered indexes, indexed views, and table and index partitions.

Keep in mind that the result is only an estimate and your actual savings can vary widely based on the fill factor and the size of the rows. If the procedure indicates that you can reduce your row size by 40 percent, you might not actually get a 40 percent space savings for the whole table. For example, if you have a row that's 8,000 bytes long and you reduce its size by 40 percent, you still can fit only one row on a data page, and your table still needs the same number of pages.

Running *sp_estimate_data_compression_savings* might yield results that indicate that the table will grow. This can happen when many rows in the table use almost the whole maximum size of the data types, and the addition of the overhead needed for the compression information is more than the savings from compression.

If the table is already compressed, you can use this procedure to estimate the size of the table (or index) if it were to be uncompressed.

Performance issues

The main motivation for compressing your data is to save space with extremely large tables, such as data warehouse fact tables. A second goal is to increase performance when scanning a table for reporting purposes, because far fewer pages need to be read. Keep in mind that compression comes at a cost: You see a tradeoff between the space savings and the extra CPU overhead to compress the data for storage and then uncompress the data when it needs to be used. On a CPU-bound system, you might find that compressing your data can actually slow down your system considerably.

Page compression provides the most benefit for I/O-bound systems, with tables for which the data is written once and then read repeatedly, as in the situations mentioned in the preceding paragraph: data warehousing and reporting. For environments with heavy read and write activity, such as online transaction processing (OLTP) applications, you might want to consider enabling row compression only and avoid the costs of analyzing the pages and rebuilding the CI record. In this case, the CPU overhead is minimal. In fact, row compression is highly optimized so that it's visible only at the storage engine layer. The relational engine (query processor) doesn't need to deal with compressed rows at all. The relational engine sends uncompressed rows to the storage engine, which compresses them if required. When returning rows to the relational engine, the storage engine waits as long as it can before uncompressing them. In the storage engine, comparisons can be done on compressed data, as internal conversions can convert a data type to its compressed form before comparing to data in the table. Also, only columns requested by the relational engine need to be uncompressed, as opposed to uncompressing an entire row.

Compression and logging In general, SQL Server logs only uncompressed data because the log needs to be read in an uncompressed format. This means that logging changes to compressed records has a greater performance impact because each row needs to be uncompressed and decoded (from the anchor record and dictionary) before writing to the log. This is another reason compression gives you more benefit on primarily read-only systems, where logging is minimal.

SQL Server writes compressed data to the log in a few situations. The most common situation is when a page is split. SQL Server writes the compressed rows as it logs the data movement during the split operation.

Compression and the version store Chapter 13 covers the version store during a discussion about Snapshot isolation, but I want to mention briefly here how the version store interacts with compression. SQL Server can write compressed rows to the version store, and the version store processing can traverse older versions in their compressed form. However, the version store doesn't support page compression, so the rows in the version store can't contain encodings of the anchor record prefixes and the page dictionary. So anytime any row from a compressed page needs to be versioned, the page must be uncompressed first.

The version store is used for both varieties of Snapshot isolation (full snapshot and read-committed snapshot) and is also used for storing the before-and-after images of changed data when triggers are fired. (These images are visible in the logical tables *inserted* and *deleted*.) Keep this in mind when evaluating the costs of compression. Snapshot isolation has lots of overhead already, and adding page compression into the mix affects performance even more.

Backup compression

Chapter 1, "SQL Server 2012 architecture and configuration," briefly mentioned backup compression when discussing configuration options. It's worth repeating that the algorithm used for compressing backups is very different than the database compression algorithms discussed in this chapter. Backup compression uses an algorithm very similar to zipping, where it's just looking for patterns in the data. Even after tables and indexes are compressed by using the data compression techniques, they still can be compressed further by using the backup compression algorithms.

Page compression looks only for prefix patterns and can still leave other patterns uncompressed, including common suffixes. Page compression eliminates redundant strings, but in most cases plenty of strings aren't redundant, and string data compresses very well using zip-type algorithms.

Also, a fair amount of space in a database constitutes overhead, such as unallocated slots on pages and unallocated pages in allocated extents. Depending on whether Instant File Initialization was used, and what was on the disk previously if it was, the background data can actually compress very well.

Thus, making a compressed backup of a database that has many compressed tables and indexes can provide additional space savings for the backup set.

Table and index partitioning

As you've already seen when looking at the metadata for table and index storage, partitioning is an integral feature of SQL Server space organization. Figure 6-3 in Chapter 6 illustrated the relationship between tables and indexes, partitions, and allocation units. 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 *sys.dm_db_partition_stats* dynamic management view, which combines information found in *sys.partitions*, *sys.allocation_units* and *sys.indexes*.

A partitioned object is split internally into separate physical units that can be stored in different locations. Partitioning is invisible to the users and programmers, who can use T-SQL code to select from a partitioned table exactly the same way they select from a nonpartitioned 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 was available only by manually creating a view that combines multiple tables—a functionality that's referred to as *partitioned* views. SQL Server 2005 introduced built-in partitioning of tables and indexes, which has many advantages over partitioned views, including improved execution plans and fewer prerequisites for implementation.

This section focuses primarily on physical storage of partitioned objects and the partitioning metadata. Chapter 11, "The Query Optimizer," examines query plans involving partitioned tables and partitioned indexes.

Partition functions and partition schemes

To understand the partitioning metadata, you need a little background into how partitions are defined, using an example based on the SQL Server samples. You can find my *Partition.sql* script on the companion website. 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, *TransactionHistory* has 12 partitions and *TransactionHistoryArchive* has two.

Before you create a partitioned table or index, you must define a partition function, which is used to define the partition boundaries logically. When a partition function is created, you must specify whether the partition should use a *LEFT*-based or *RIGHT*-based boundary point. Simply put, this defines whether the boundary value itself is part of the left-hand or right-hand partition. Another way to consider this is to ask this question: Is it an upper boundary of one partition (in which case it goes to the *LEFT*), or a lower boundary point of the next partition (in which case it goes to the *RIGHT*)? The number of partitions created by a partition function with *n* boundaries will be *n*+1. Here is the partition function being used for this example:

```
CREATE PARTITION FUNCTION [TransactionRangePF1] (datetime)
AS RANGE RIGHT FOR VALUES ('20111001', '20111101', '20111201',
               '20120101', '20120201', '20120301', '20120401',
               '20120501', '20120601', '20120701', '20120801');
```

Notice that the table name isn't mentioned in the function definition because the partition function isn't tied to any particular table. The *TransactionRangePF1* function divides the data into 12 partitions because 11 *datetime* boundaries exist. The keyword *RIGHT* specifies that any value that equals one of the boundary points goes into the partition to the right of the endpoint. So for this function, all values less than October 1, 2011 go in the first partition and values greater than or equal to October 1, 2011 and less than November 1, 2011 go in the second partition. *LEFT* (the default) could also have been specified, in which case the value equal to the endpoint goes 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 is 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 should probably have each partition on its own filegroup. The *CREATE PARTITION SCHEME* command must list at least as many filegroups as partitions, but it can list one more filegroup, which is considered the "next used" filegroup. If the partition function splits, the new boundary point is added in the filegroup used next. If you don't specify an extra filegroup at the time you create the partition scheme, you can alter the partition scheme to set the next-used filegroup before modifying the function.

As you've seen, the listed filegroups don't have to be unique. In fact, if you want to have all the partitions on the same filegroup, as I have here, you can use a shortcut syntax:

```
CREATE PARTITION SCHEME [TransactionsPS1]
AS PARTITION [TransactionRangePF1]
ALL TO ([PRIMARY]);
GO
```

Note that putting all the partitions on the same filegroup is usually done just for the purpose of testing your code.

Additional filegroups are 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 don't 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 ('20110901');
GO

CREATE PARTITION SCHEME [TransactionArchivePS2]
AS PARTITION [TransactionArchivePF2]
TO ([PRIMARY], [PRIMARY]);
GO
```

The 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 *CREATE TABLE* statement. I create a table called *TransactionHIstory* that includes this line as the last part of the *CREATE TABLE* statement as follows:

```
ON [TransactionsPS1] (TransactionDate)
```

The second table, *TransactionHistoryArchive*, is created using the *TransactionsPS2* partitioning scheme. The script then loads data into the two tables, and because the partition scheme has already been defined, each row is placed in the appropriate partition as the data is loaded. After the tables are loaded, you can examine the metadata.

Metadata for partitioning

Figure 8-18 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* catalog views that were discussed earlier in this chapter.

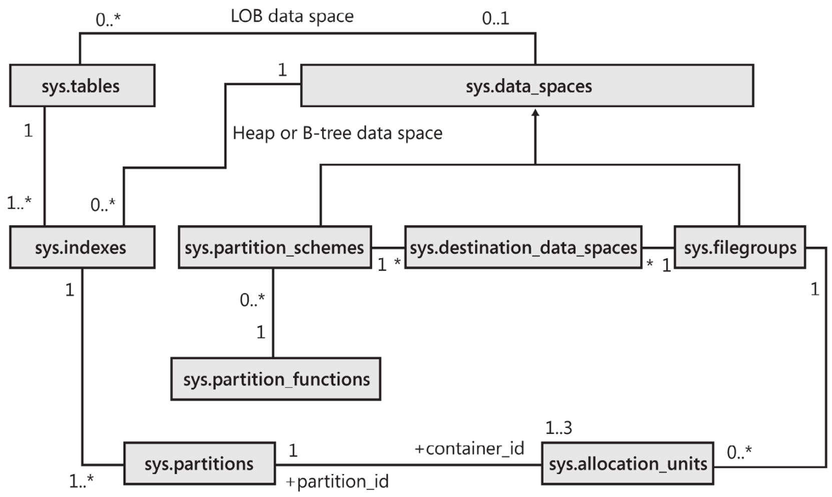


FIGURE 8-18 Catalog views containing metadata for partitioning and data storage.

Some of the queries use the undocumented *sys.system_internals_allocation_units* view instead of *sys.allocation_units* to retrieve page address information. The following are the most relevant columns of each of these views.

- ***sys.data_spaces*** has a primary key called *data_space_id*, which is either a partition ID or a filegroup ID. Each filegroup has one row, and each partition scheme has one row. One column 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 you know that a table is partitioned, you can join it directly 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 *dboTransactionHistoryArchive* 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(dbo.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 not mentioned here, and additional views provide additional 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 8-18 and the view shown in Listing 8-8. 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. I first create a function to return an index name, with an object ID and an index ID given, so that the view can easily return any index names. When selecting from the view, you can add your own *WHERE* clause to find information about just the table you're interested in.

LISTING 8-8 View returning data about each partition of each partitioned table

```
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;
```

```
GO

CREATE VIEW Partition_Info AS
  SELECT OBJECT_NAME(i.object_id) as ObjectName,
    dbo.INDEX_NAME(i.object_id,i.index_id) AS IndexName,
    object_schema_name(i.object_id) as SchemaName,
    p.partition_number as PartitionNumber, fg.name AS FilegroupName, rows as Rows,
    au.total_pages as TotalPages,
    CASE boundary_value_on_right
        WHEN 1 THEN 'less than'
        ELSE 'less than or equal to'
    END as 'Comparison'
    , rv.value as BoundaryValue,
    CASE WHEN ISNULL(rv.value, rv2.value) IS NULL THEN 'N/A'
    ELSE
      CASE
        WHEN boundary_value_on_right = 0 AND rv2.value IS NULL
           THEN 'Greater than or equal to'
        WHEN boundary_value_on_right = 0
           THEN 'Greater than'
        ELSE 'Greater than or equal to' END + ' ' +
           ISNULL(CONVERT(varchar(15), rv2.value), 'Min Value')
                + ' ' +
                +
           CASE boundary_value_on_right
             WHEN 1 THEN 'and less than'
               ELSE 'and less than or equal to'
               END + ' ' +
                + ISNULL(CONVERT(varchar(15), rv.value),
                           'Max Value')
        END as 'TextComparison'
  FROM sys.partitions p
    JOIN sys.indexes i
      ON p.object_id = i.object_id and p.index_id = i.index_id
    LEFT JOIN sys.partition_schemes ps
      ON ps.data_space_id = i.data_space_id
    LEFT 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
    LEFT JOIN sys.partition_range_values rv2
      ON f.function_id = rv2.function_id
          AND p.partition_number - 1= rv2.boundary_id
    LEFT JOIN sys.destination_data_spaces dds
      ON dds.partition_scheme_id = ps.data_space_id
          AND dds.destination_id = p.partition_number
    LEFT JOIN sys.filegroups fg
      ON dds.data_space_id = fg.data_space_id
    JOIN sys.allocation_units au
      ON au.container_id = p.partition_id
WHERE i.index_id <2 AND au.type =1;
```

The *LEFT JOIN* operator 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. *LEFT JOIN* gives the last partition with a boundary value of NULL, which means that the value of the last partition has no upper limit. A derived table 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) is aggregated into a single value. This query uses the preceding view to get information about my *TransactionHistory* table's partitions:

```
SELECT ObjectName, PartitionNumber, Rows, TotalPages, Comparison, BoundaryValue
FROM Partition_Info
WHERE ObjectName = 'TransactionHistory' AND SchemaName = 'dbo'
ORDER BY ObjectName, PartitionNumber ;
```

Here are my results for the *TransactionHistory* object:

Object_Name	Partitionnumber	Rows	Totalpages	Comparison	BoundaryValue
TransactionHistory	1	11155	89	Less than	2011-10-01
TransactionHistory	2	9339	74	Less than	2011-11-01
TransactionHistory	3	10169	81	Less than	2011-12-01
TransactionHistory	4	12181	97	Less than	2012-01-01
TransactionHistory	5	9558	74	Less than	2012-02-01
TransactionHistory	6	10217	81	Less than	2012-03-01
TransactionHistory	7	10703	89	Less than	2012-04-01
TransactionHistory	8	10640	89	Less than	2012-05-01
TransactionHistory	9	12508	90	Less than	2012-06-01
TransactionHistory	10	12585	97	Less than	2012-07-01
TransactionHistory	11	3380	33	Less than	2012-08-01
TransactionHistory	12	1008	17	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. It also contains a few additional columns that aren't shown here, just to keep the output from being too wide. In particular, I didn't return the *FilegroupName* value, because in my example, all the partitions are on the same filegroup. Anytime your partitions are on different filegroups, you most likely will want to see that value for each partition. 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. The next section looks at a metadata query that provides location information.

Note If a partitioned table contains FILESTREAM data, you should partition the FILESTREAM data by using the same partition function as the non-FILESTREAM data. Because the regular data and the FILESTREAM data are on separate filegroups, the FILESTREAM data needs its own partition scheme. However, the partition scheme for the FILESTREAM data can use the same partition function to make sure the same partitioning is used for both FILESTREAM and non-FILESTREAM data.

The sliding window benefits of partitioning

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 mentioned earlier, this isn't intended to be a complete how-to guide to SQL Server 2012 partitioning; instead, it's a description of the internal storage of partitioning information.

Note For a complete description of designing, setting up, and managing partitioned tables and indexes, read Ron Talmage's white paper at *http://msdn.microsoft.com/en-us/library/dd578580.aspx*.

To understand the internals of rearranging partitions, you need to look at 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 of 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. Instead, the metadata is updated in the *sys.partitions* and *sys.system_internals_allocation_units* views to indicate that a particular allocation unit now is part of a partition in a different object. For example, the following query returns information about each allocation unit in the first two partitions of the *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('[dbo].[TransactionHistory]')
   OR object_id=object_id('[dbo].[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*.)

```
name                       rows    pages      first_page      index_id    partition_number
-------------------------  ------- ---------- --------------  ----------- -----------------
TransactionHistory         11155   89        0xD81B00000100  1           1
TransactionHistory         9339    74        0xA82200000100  1           2
TransactionHistoryArchive  89253   633       0x981B00000100  1           1
TransactionHistoryArchive  0       0         0x000000000000  1           2
```

Now you can move one of the partitions. The 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, 2012. You can alter the partition function to put a new boundary point in for September 1, 2012, so the last partition is split. Before doing that, you 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 occurs and all data from the last partition of any tables using this partition scheme is moved to a new allocation unit. Refer to *SQL Server Books Online* for complete details about each of the following commands:

```
ALTER PARTITION SCHEME TransactionsPS1
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionRangePF1()
SPLIT RANGE ('20120901');
GO
```

Next, you can do something similar for the function and partition scheme used by *TransactionHistoryArchive*. In this case, add a new boundary point for October 1, 2011:

```
ALTER PARTITION SCHEME TransactionArchivePS2
NEXT USED [PRIMARY];
GO

ALTER PARTITION FUNCTION TransactionArchivePF2()
SPLIT RANGE ('20111001');
GO
```

Now move all data from *TransactionHistory* with dates earlier than October 1, 2011, to the second partition of *TransactionHistoryArchive*. However, the first partition of *TransactionHistory* technically has no lower limit; it includes everything earlier than October 1, 2011. The second partition of *TransactionHistoryArchive* does have a lower limit, which is the first boundary point, or September 1, 2011. To *SWITCH* a partition from one table to another, you must guarantee that all the data to be moved meets the requirements for the new location, so you need to add a *CHECK* constraint that guarantees that no data in *TransactionHistory* is earlier than September 1, 2011. 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 testing purposes, you could try leaving out the next step that adds the constraint and try just executing the *ALTER TABLE/SWITCH* command. You get an error message. After that, you can add the constraint and run the *ALTER TABLE/SWITCH* command again.)

```
ALTER TABLE [dbo].[TransactionHistory]
ADD CONSTRAINT [CK_TransactionHistory_DateRange]
CHECK ([TransactionDate] >= '20110901');
GO
ALTER TABLE [dbo].[TransactionHistory]
SWITCH PARTITION 1
TO [dbo].[TransactionHistoryArchive] PARTITION 2;
GO
```

Now 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('[dbo].[TransactionHistory]')
   OR object_id=object_id('[dbo].[TransactionHistoryArchive]'))
  AND index_id = 1 AND partition_number <= 2;

RESULTS:
name                 rows    pages      first_page      index_id    partition_number
--------------------  ------- ---------- --------------  ----------- ----------------
TransactionHistory    0       0          0x000000000000  1           1
TransactionHistory    9339    74         0xA82200000100  1           2
TransactionHistoryAr  89253   633        0x981B00000100  1           1
TransactionHistoryAr  11155   89         0xD81B00000100  1           2
```

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 (89), and the same starting page (0xD81B00000100, or file 1, page 7,128). No data was moved; the only change was that the allocation unit starting at file 1, page 7,128 isn't recorded as belonging to the second partition of the *TransactionHistoryArchive* table.

Although my partitioning script created the indexes for the partitioned tables by using the same partition scheme used for the tables themselves, this isn't always necessary. An index for a partitioned table can be partitioned using the same partition scheme or a different one. If you don't 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*.

However, an internal storage component is associated with automatically aligned indexes. As mentioned earlier, if you build an index on a partitioned table and don't 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 isn't 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 isn't necessary because the clustered index already contains all the columns. Another case in which SQL Server doesn't add an included column automatically is when you create a unique index, either clustered or nonclustered. Because unique partitioned indexes require that the partitioning column is contained in the unique key, a unique index for which you haven't explicitly included the partitioning key isn't partitioned automatically.

Partitioning a columnstore index

To end this section, look at an example that combines partitioning with columnstore indexes, which Chapter 7 described. If you still have the *dbo.FactInternetSalesBig* table, you can follow the examples here.

First, create a very simple partition function and partition scheme. The partition function splits the data into five partitions, which eventually are mapped to the *SalesTerritoryKey* column of the big table. Then define a partition scheme that puts all the partitions on the PRIMARY filegroup:

```
USE AdventureWorksDW2012
GO
CREATE PARTITION FUNCTION [PF_TerritoryKey](int) AS RANGE LEFT FOR VALUES (2, 4, 6, 8)
GO
CREATE PARTITION SCHEME [PS_TerritoryKey] AS PARTITION [PF_TerritoryKey] ALL TO ([PRIMARY]);
GO
```

Now you can rebuild the clustered index to use this partitioning scheme, but you should get an error message initially, because if a table has a columnstore index, it must be partitioned aligned with the table. So you have to drop the columnstore index before you can rebuild the clustered index, and then you can rebuild the columnstore index using the same partitioning scheme.

```
DROP INDEX dbo.FactInternetSalesBig.csi_FactInternetSalesBig;
GO
CREATE CLUSTERED INDEX clus_FactInternetSalesBig ON  dbo.FactInternetSalesBig
(SalesTerritoryKey)
ON  PS_TerritoryKey (SalesTerritoryKey)
GO
```

Now you can rebuild the columnstore index on the same partitioning scheme, as shown in Listing 8-9.

LISTING 8-9 Rebuilding the columnstore index on the same partitioning scheme

```
CREATE NONCLUSTERED COLUMNSTORE INDEX csi_FactInternetSalesBig
ON dbo.FactInternetSalesBig (
   ProductKey,
   OrderDateKey,
   DueDateKey,
   ShipDateKey,
   CustomerKey,
   PromotionKey,
   CurrencyKey,
   SalesTerritoryKey,
   SalesOrderNumber,
   SalesOrderLineNumber,
   RevisionNumber,
   OrderQuantity,
   UnitPrice,
   ExtendedAmount,
   UnitPriceDiscountPct,
   DiscountAmount,
   ProductStandardCost,
   TotalProductCost,
   SalesAmount,
   TaxAmt,
   Freight,
   CarrierTrackingNumber,
   CustomerPONumber
) ON PS_TerritoryKey (SalesTerritoryKey)
GO
```

To explore my partitions, you can re-create the *Partition_Info* view from Listing 8-8 in the *AdventureWorksDW2012* database. Then you can determine how many rows are in each partition by looking at just a couple of columns from that view:

```
select PartitionNumber, Rows from Partition_Info
where ObjectName = 'FactInternetSalesBig';
GO
```

Here are my results:

```
PartitionNumber Rows
--------------- --------------------
1               4618240
2               6289920
3               3921408
4               5725696
5               10368512
```

Now that you have a columnstore index, you can also use the metadata view that Chapter 7 explored—namely, *sys.column_store_segments*. The following query groups by column to show you the total number of segments in the table. If you run this query, you'll see 24 rows indicating 24 columns. The index had only 23 columns defined, but because the clustered index wasn't unique, the uniquifier is added as a column. The result of this query also shows 41 total segments:

```
-- GROUP BY COLUMN
SELECT s.column_id,  col_name(ic.object_id, ic.column_id) as column_name,   count(*) as segment_
count
FROM sys.column_store_segments s join sys.partitions p on s.partition_id = p.partition_id
  LEFT JOIN sys.index_columns ic
      ON p.object_id = ic.object_id AND p.index_id = ic.index_id
     AND s.column_id = ic.index_column_id
WHERE object_name(p.object_id) = 'FactInternetSalesBig'
GROUP BY s.column_id,  col_name(ic.object_id, ic.column_id), object_name(p.object_id)
ORDER by 1;
GO
```

Because of the boundary values used, not every partition has exactly the same number of rows, as you saw in the data from the *Partition_Info* view. Each partition could have a different number of segments, and the following query shows how many segments are created for each partition:

```
SELECT   partition_number, count( segment_id) as NumSegments, sum(row_count) as NumRows
FROM sys.column_store_segments s join sys.partitions p on s.partition_id = p.partition_id
   JOIN sys.index_columns ic
      ON p.object_id = ic.object_id AND p.index_id = ic.index_id
     AND s.column_id = ic.index_column_id
WHERE object_name(p.object_id) = 'FactInternetSalesBig' and index_column_id = 2
GROUP BY partition_number WITH ROLLUP;
GO
```

The results show that partition 5 has more than 10 million rows and 12 segments, whereas partitions 1 and 3 each have only six segments. The grand totals produced by the *ROLLUP* clause, show the 41 total segments, and that the total number of rows in the table is 30923776.

```
partition_number NumSegments NumRows
---------------- ----------- -----------
1                6           4618240
2                8           6289920
3                6           3921408
4                9           5725696
5                12          10368512
NULL             41          30923776
```

Conclusion

This chapter looked at how SQL Server stores data that doesn't use the typical *FixedVar* record format and data that doesn't fit into the usual 8 KB data page.

This chapter discussed row-overflow and large object data, which is stored on its own separate pages, and FILESTREAM data, which is stored outside SQL Server, in files in the file system. You also read about FileTables, which allow FILESTREAM data to be accessed and manipulated through SQL Server tables.

Some special storage capabilities in SQL Server 2012 require that you look at row storage in a completely different way. Sparse columns allow you to have very wide tables of up to 30,000 columns, as long as most of those columns are NULL in most rows. Each row in a table containing sparse columns has a special descriptor field that provides information about which columns are non-NULL for that particular row.

This chapter also described the row storage format used with compressed data. Data can be compressed at either the row level or the page level, and the rows and pages themselves describe the data that is contained therein. This type of row format is referred to as the CD format.

Finally, you looked at partitioning of tables and indexes. Although partitioning doesn't really require a special format for your rows and pages, it does require accessing the metadata in a special way.

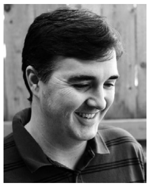
Index

About the authors

KALEN DELANEY (primary author) has been working with Microsoft SQL Server for over 26 years, and she provides advanced SQL Server training to clients around the world. She has been a SQL Server MVP (Most Valuable Professional) since 1992 and has been writing about SQL Server almost as long. Kalen has spoken at dozens of technical conferences, including almost every PASS Community Summit held in the United States since the organization's founding in 1999.

Kalen is a contributing editor and columnist for *SQL Server Magazine* and the author or co-author of many Microsoft Press books on SQL Server, including *Inside Microsoft SQL Server 7*, *Inside Microsoft SQL Server 2000*, *Inside Microsoft SQL Server 2005: The Storage Engine*, *Inside Microsoft SQL Server 2005: Query Tuning and Optimization* and *SQL Server 2008 Internals*. Kalen blogs at *www.sqlblog.com*, and her personal website can be found at *www.SQLServerInternals.com*.

BOB BEAUCHEMIN (author) is a database-centric instructor, course author, writer, conference speaker, application practitioner and architect, and a Developer Skills Partner for SQLskills. Bob has been a Microsoft MVP since 2002. He's written and taught courses on SQL Server and data access worldwide since the mid-1990s, and currently writes and teaches SQLskills' developer and DBA-centric immersion course offerings. He is lead author of the books *A Developer's Guide to SQL Server 2005* and *A First Look at SQL Server 2005 For Developers*, and sole author of *Essential ADO.NET*. He's written numerous Microsoft whitepapers, as well as articles on SQL Server and other databases, database security, ADO.NET, and OLE DB for a number of magazines.



CONOR CUNNINGHAM (author) is principal architect of the SQL Server Core Engine Team, with over 15 years experience building database engines for Microsoft. He specializes in query processing and query optimization, and he designed and/or implemented a number of the query processing features available in SQL Server. Conor holds a number of patents in the field of query optimization, and he has written numerous academic papers on query processing. Conor blogs at "Conor vs. SQL" at *http://blogs.msdn.com/b/conor_cunningham_msft*.

JONATHAN KEHAYIAS (author and technical reviewer) is a Principal Consultant and Trainer for SQLskills, one of the most well-known and respected SQL Server training and consulting companies in the world. Jonathan is a SQL Server MVP and was the youngest person ever to obtain the Microsoft Certified Masters for SQL Server 2008. Jonathan is a performance tuning expert, for both SQL Server and hardware, and has architected complex systems as a developer, business analyst, and DBA. Jonathan also has extensive development (T-SQL, C#, and ASP.Net), hardware and virtualization design expertise, Windows expertise, Active Directory experience, and IIS administration experience.

Jonathan frequently blogs about SQL Server at *http://www.SQLskills.com/blogs/Jonathan*, and can be reached by email at *Jonathan@SQLskills.com*, or on Twitter as *@SQLPoolBoy*. He regularly presents at PASS Summit, SQLBits, SQL Intersections, SQL Saturday events, and local user groups and has remained a top answerer of questions on the MSDN SQL Server Database Engine forum since 2007.

BENJAMIN NEVAREZ (author and technical reviewer) is a database professional based in Los Angeles, California, and author of *Inside the SQL Server Query Optimizer*, published by Red Gate books. He has 20 years of experience with relational databases, and has been working with SQL Server since version 6.5. Benjamin holds a Master's degree in computer science and has been a speaker at many technology conferences, including the PASS Summit and SQL Server Connections. His blog is at *http://benjaminnevarez.com* and can be reached by email at *admin@benjaminnevarez.com* and on twitter at *@BenjaminNevarez*.

PAUL S. RANDAL (author) is the CEO of SQLskills.com, the world-renowned training and consulting company that he runs with his wife Kimberly L. Tripp. He is also a SQL Server MVP and a Microsoft Regional Director. Paul worked at Microsoft for almost nine years, after spending five years at DEC working on the OpenVMS file system. He wrote various DBCC commands for SQL Server 2000 and then rewrote all of DBCC CHECKDB and repair for SQL Server 2005 before moving into management in the SQL team. During SQL Server 2008 development, he was responsible for the entire Storage Engine.

Paul regularly consults and teaches at locations around the world, including the SQLskills Immersion Events on internals, administration, high-availability, disaster recovery, and performance tuning. He also wrote and taught the SQL Server Microsoft Certified Master certification for Microsoft. He is a top-rated presenter at conferences such as the PASS Summit, and owns and manages the SQLintersections conferences. Paul's popular blog is at *www.SQLskills.com/blogs/paul* and he can be reached at *Paul@SQLskills.com* and on Twitter as *@paulrandal*.