

Microsoft SQL Server 2012 Internals

**FREE
SAMPLER**



 Professional

Kalen Delaney

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



9 780735 658561

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

Programming/Microsoft SQL Server

Microsoft Press

Celebrating 30 years!

Want to read more?

Microsoft Press books are now available through [O'Reilly Media](#). You can [buy this book](#) in print and or ebook format, along with the complete Microsoft Press product line.

Buy 2 books, get the 3rd FREE!

Use discount code: OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the [Android Marketplace](#), and [Amazon.com](#)



O'REILLY®

Spreading the knowledge of innovators

[oreilly.com](#)

Microsoft SQL Server 2012 Internals

Kalen Delaney

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

Copyright © 2013 by Kalen Delaney

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

ISBN: 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.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

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

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

Acquisitions 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

	<i>Introduction</i>	<i>xix</i>
CHAPTER 1	SQL Server 2012 architecture and configuration	1
CHAPTER 2	The SQLOS	35
CHAPTER 3	Databases and database files	99
CHAPTER 4	Special databases	139
CHAPTER 5	Logging and recovery	171
CHAPTER 6	Table storage	203
CHAPTER 7	Indexes: internals and management	297
CHAPTER 8	Special storage	381
CHAPTER 9	Special indexes	457
CHAPTER 10	Query execution	513
CHAPTER 11	The Query Optimizer	611
CHAPTER 12	Plan caching and recompilation	703
CHAPTER 13	Transactions and concurrency	765
CHAPTER 14	DBCC internals	837
	<i>Index</i>	<i>903</i>

Contents

<i>Introduction</i>	<i>xix</i>
Chapter 1 SQL Server 2012 architecture and configuration	1
SQL Server editions	1
SQL Server installation and tools	2
SQL Server metadata	3
Compatibility views	3
Catalog views	4
Dynamic Management Objects	6
Other metadata	7
Components of the SQL Server engine	10
Protocols	11
Query processor	12
The storage engine	14
SQL Server 2012 configuration	17
Using SQL Server Configuration Manager	18
Managing services	19
SQL Server system configuration	21
Operating system configuration	21
Trace flags	23
SQL Server configuration settings	24
Conclusion	33

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Chapter 2	The SQLOS	35
	NUMA architecture.	36
	The scheduler.	37
	Understanding SQL Server schedulers.	38
	Binding schedulers to CPUs	41
	Observing scheduler internals.	42
	Understanding the Dedicated Administrator Connection (DAC)	45
	Memory	47
	The buffer pool and the data cache.	47
	Column store object pool.	48
	Access to in-memory data pages	48
	Page management in the data cache.	48
	The free buffer list and the lazywriter	49
	Checkpoints.	50
	Memory management in other caches	52
	The Memory Broker.	54
	Memory sizing	54
	Buffer pool sizing.	55
	SQL Server Resource Governor.	61
	Resource Governor overview.	61
	Resource Governor controls	70
	Resource Governor metadata	71
	Extended Events	73
	Extended Events architecture	73
	Event execution life cycle	73
	Core concepts.	75
	Extended Events DDL and querying.	83
	Extended Events UI	86
	Conclusion	97
Chapter 3	Databases and database files	99
	Working with sample databases	100
	AdventureWorks.	100
	pubs	101

Northwind	101
Understanding database files	101
Creating a database.	104
Using CREATE DATABASE: an example.	106
Expanding or shrinking a database	106
Automatic file expansion	106
Manual file expansion	107
Fast file initialization	107
Automatic shrinkage	108
Manual shrinkage.	108
Using database filegroups.	109
The default filegroup.	110
A FILEGROUP CREATION example	112
Filestream filegroups	113
Altering a database.	114
ALTER DATABASE examples	115
Databases under the hood.	116
Space allocation	116
Setting database options.	119
State options.	122
Cursor options	125
Auto options.	125
SQL options.	126
Database recovery options.	128
Other database options	129
Understanding database security	129
Database access	130
Database security.	132
Databases vs. schemas	133
Principals and schemas	133
Default schemas	134
Moving or copying a database.	134
Detaching and reattaching a database	135
Backing up and restoring a database.	136

Understanding compatibility levels	137
Conclusion	138

Chapter 4 Special databases 139

System databases	139
Understanding the master database	139
Understanding the model database	140
Introducing the tempdb database	140
Understanding the resource database	140
Understanding the msdb database	141
Moving system databases	142
Moving the master database	143
The tempdb database	144
Objects in tempdb	144
Optimizations in tempdb	146
Best practices	147
tempdb contention	148
tempdb space monitoring	153
Database snapshots	155
Creating a database snapshot	156
Understanding space used by database snapshots	159
Managing your snapshots	161
Partially contained databases	162
Configuring a contained database	162
Creating contained users	163
Understanding database collation changes	166
Detecting uncontained features	168
Conclusion	169

Chapter 5 Logging and recovery 171

Transaction log internals	171
Phases of recovery	174
Page LSNs and recovery	175
Log reading	176

The log cache	177
Changes in log size	178
Understanding virtual log files	178
Maintaining a recoverable log	185
Automatically shrinking the log	187
Viewing the log file size	188
Database backup and restore	188
Understanding the types of backups	189
Understanding recovery models	190
Choosing a backup type	194
Restoring a database	195
Conclusion	201

Chapter 6 Table storage 203

Table creation	203
Naming tables and columns	204
Avoiding reserved keywords	205
Using delimited identifiers	206
Understanding naming conventions	207
Choosing a data type	208
The NULL problem	233
User-defined data types	235
IDENTITY property	237
Sequence object	240
Internal storage	243
The sys.indexes catalog view	244
Data storage metadata	245
Catalog view queries	246
Data pages	248
The structure of data rows	257
How to find a physical page	259
Storage of fixed-length rows	262
Storage of variable-length rows	265
NULLS and variable-length columns	267

Storage of date and time data	270
Storage of sql_variant data	273
Constraints	276
Constraint names and catalog view information	277
Constraint failures in transactions and multiple-row data modifications	278
Altering a table	279
Changing a data type	280
Adding a new column	281
Adding, dropping, disabling, or enabling a constraint	281
Dropping a column	283
Internals of altering tables	283
Heap modification internals	286
Allocation structures	286
Inserting rows	288
Deleting rows	288
Updating rows	292
Conclusion	295

Chapter 7 Indexes: internals and management 297

Overview	298
SQL Server B-tree indexes	299
Example 1: An index with a large key column	300
Example 2: An index with a very narrow key column	301
Tools for analyzing indexes	302
Using the dm_db_index_physical_stats DMV	302
Using sys.dm_db_database_page_allocations	306
Understanding B-tree index structures	308
Clustering key dependency	308
Nonclustered B-tree indexes	311
Constraints and indexes	312
Index creation options	313
IGNORE_DUP_KEY	313
STATISTICS_NORECOMPUTE	314

MAXDOP	314
Index placement.....	314
Physical index structures for B-trees	315
Index row formats	315
Clustered index structures	316
Non-leaf level(s) of a clustered index	317
Analyzing a clustered index structure	317
Nonclustered index structures.....	322
Indexes on computed columns and indexed views	333
SET options	333
Permissible functions.....	334
Schema binding	335
Indexes on computed columns	335
Implementation of a computed column	336
Persisted columns	337
Indexed views	338
Additional requirements.....	338
Creating an indexed view.....	339
Using an indexed view	340
Data modification internals.....	341
Inserting rows	342
Splitting pages	342
Deleting rows	346
Updating rows	354
Table-level vs. index-level data modification	357
Logging	358
Locking.....	358
Fragmentation	359
Managing B-tree index structures	360
Dropping indexes.....	360
Using the ALTER INDEX command	361
Detecting fragmentation	363
Removing fragmentation	364
Rebuilding an index.....	366
Online index building	367

Columnstore indexes	370
Creation of columnstore indexes.....	370
Storage of columnstore indexes	371
Columnstore index metadata	376
Conclusion	380

Chapter 8 Special storage 381

Large object storage	381
Restricted-length large object data (row-overflow data)	382
Unrestricted-length large object data	386
FILESTREAM and FileTable data	394
Enabling FILESTREAM data for SQL Server	395
Creating a FILESTREAM-enabled database	397
Creating a table to hold FILESTREAM data	397
Manipulating FILESTREAM data	399
Exploring metadata with FILESTREAM data	404
Creating a FileTable	406
Considering performance for FILESTREAM data	409
Summarizing FILESTREAM and FileTable	410
Sparse columns	411
Management of sparse columns.....	411
Column sets and sparse column manipulation.....	414
Physical storage	416
Metadata	419
Storage savings with sparse columns.....	420
Data compression	423
Vardecimal.....	423
Row compression.....	424
Page compression	433
Table and index partitioning	444
Partition functions and partition schemes.....	444
Metadata for partitioning.....	446
The sliding window benefits of partitioning	450

Partitioning a columnstore index	452
Conclusion	455

Chapter 9 Special indexes 457

Special indexes vs. ordinary indexes	457
XML indexes.	458
Creating and maintaining XML indexes.	459
Using XQuery in SQL Server: internals	463
Understanding how a query plan uses an XML index.	465
Using secondary XML indexes	468
Working with XML indexes and schema-validated columns	469
Using XML-specific information in query plans	470
Spatial indexes.	471
Purpose of spatial indexes	472
Composition of the spatial index.	475
How a spatial query uses a spatial index	477
How to ensure that your spatial index is being used.	478
Spatial query plans and spatial indexes	479
Nearest neighbor optimization in SQL Server 2012	481
Spatial index diagnostic stored procedures	484
Diagnostics with the SQL Server 2012 spatial functions	491
Full-text indexes	492
Internal tables created by the full-text index	494
Full-text index metadata views	497
Full-text index creation	498
Maintenance of a full-text index	499
Full-text status metadata, configuration, and diagnostic information	500
How a full-text index is used in a query.	501
A full-text query plan	502
Extended event information for full-text queries.	503
Semantic indexes.	505
Conclusion	511

Chapter 10 Query execution 513

Introducing query processing and execution	513
Iterators	513
Properties of iterators	515
Reading query plans.	517
Graphical plans.	517
Text plans.	518
XML plans	518
Estimated vs. actual query plans	518
Query plan display options	520
Analyzing plans.	525
Scans and seeks	526
Seekable predicates and covered columns	528
Bookmark lookup.	531
Joins	533
Aggregations	545
Unions	555
Advanced index operations	560
Subqueries.	566
Parallelism	580
Inserts, updates, and deletes	598
Understanding data warehouses	599
Using columnstore indexes and batch processing	603
Adding new data	607
Hints	609
Conclusion	610

Chapter 11 The Query Optimizer 611

Overview.	611
Understanding the tree format	612
Understanding optimization	613
Search space and heuristics.	614
Rules	614
Properties	614

Storage of alternatives: the Memo	617
Operators	617
Optimizer architecture	624
Before optimization	625
Simplification	625
Trivial plan/auto-parameterization	625
Limitations	627
The Memo: exploring multiple plans efficiently	627
Statistics, cardinality estimation, and costing	630
Statistics design	631
Density/frequency information	634
Filtered statistics	636
String statistics	637
Cardinality estimation details	638
Limitations	642
Costing	643
Index selection	645
Filtered indexes	648
Indexed views	649
Partitioned tables	654
Partition-aligned index views	658
Windowing functions	658
Data warehousing	659
Columnstore indexes	660
Batch mode processing	662
Plan shape	667
Columnstore limitations and workarounds	670
Updates	670
Halloween Protection	674
Split/Sort/Collapse	674
Merge	676
Wide update plans	679
Non-updating updates	681
Sparse column updates	681

Partitioned updates682
Locking685
Partition-level lock escalation686
Distributed query687
Extended indexes689
Plan hinting689
Debugging plan issues691
{HASH ORDER} GROUP692
{MERGE HASH CONCAT} UNION693
FORCE ORDER, {LOOP MERGE HASH} JOIN693
INDEX=<indexname> <indexid>694
FORCESEEK695
FAST <number_rows>695
MAXDOP <N>696
OPTIMIZE FOR696
PARAMETERIZATION {SIMPLE FORCED}698
NOEXPAND699
USE PLAN699
Hotfixes700
Conclusion701

Chapter 12 Plan caching and recompilation 703

The plan cache703
Plan cache metadata704
Clearing plan cache704
Caching mechanisms705
Ad hoc query caching706
Optimizing for ad hoc workloads708
Simple parameterization711
Prepared queries717
Compiled objects719
Causes of recompilation722
Plan cache internals732
Cache stores732

Compiled plans.	734
Execution contexts.	734
Plan cache metadata.	735
Cache size management.	740
Costing of cache entries.	743
Objects in plan cache: the big picture.	744
Multiple plans in cache.	746
When to use stored procedures and other caching mechanisms.	747
Troubleshooting plan cache issues.	748
Optimization hints and plan guides.	752
Optimization hints.	752
Purpose of plan guides.	754
Types of plan guides.	755
Managing plan guides.	758
Plan guide considerations.	759
Conclusion.	764

Chapter 13 Transactions and concurrency 765

Concurrency models.	765
Pessimistic concurrency.	766
Optimistic concurrency.	766
Transaction processing.	766
ACID properties.	767
Transaction dependencies.	768
Isolation levels.	770
Locking.	774
Locking basics.	774
Spinlocks.	775
Lock types for user data.	775
Viewing locks.	786
Locking examples.	789
Lock compatibility.	794
Internal locking architecture.	796

Row-level locking vs. page-level locking	803
Lock escalation	804
Deadlocks	806
Row versioning	811
Row versioning details	811
Snapshot-based isolation levels.	813
Choosing a concurrency model.	830
Controlling locking.	832
Lock hints.	832
Conclusion	836

Chapter 14 DBCC internals 837

Shrinking files and databases	837
Data file shrinking	838
Log file shrinking	840
DBCC SHRINKFILE	840
AUTO_SHRINK	841
Consistency checking	841
Getting a consistent view of the database	842
Processing the database efficiently.	845
Performing primitive system catalog consistency checks.	855
Performing allocation consistency checks.	856
Performing per-table logical consistency checks.	860
Processing columns	866
Performing cross-table consistency checks.	881
Understanding DBCC CHECKDB output	885
Reviewing DBCC CHECKDB options	890
Performing database repairs	893
Using consistency-checking commands other than DBCC CHECKDB	898
Conclusion	901

<i>Index</i>	903
------------------------	-----

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.

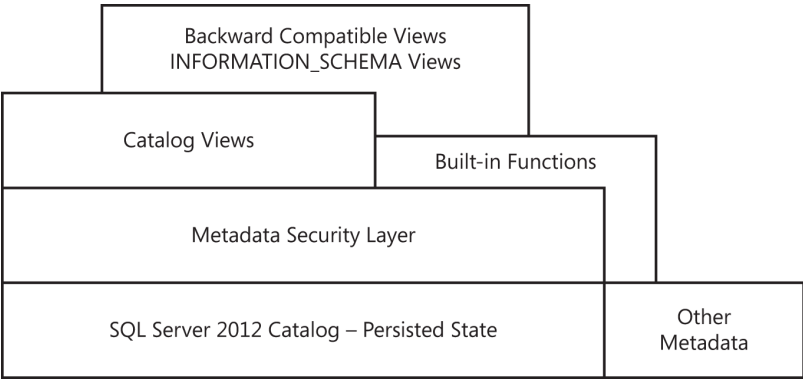


FIGURE 1-1 Layers of metadata in SQL Server 2012.

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.)

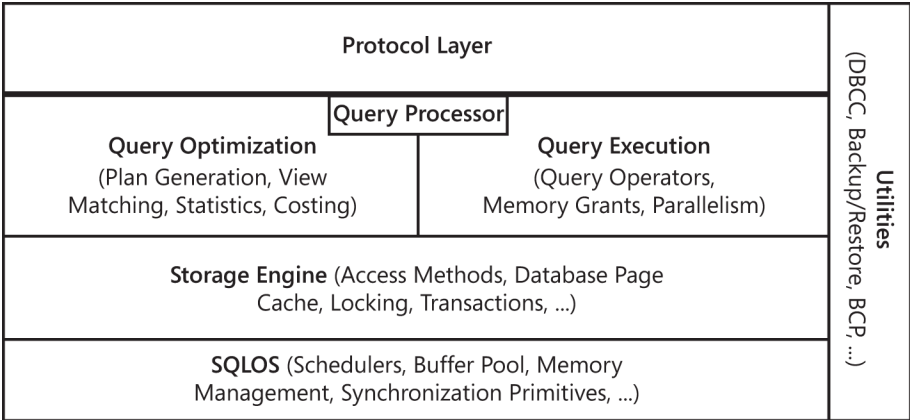


FIGURE 1-2 The major components of the SQL Server Database Engine.

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 concurrency model exclusively; it's the default in both SQL Server 2005 and later versions.
- **Optimistic concurrency** 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.

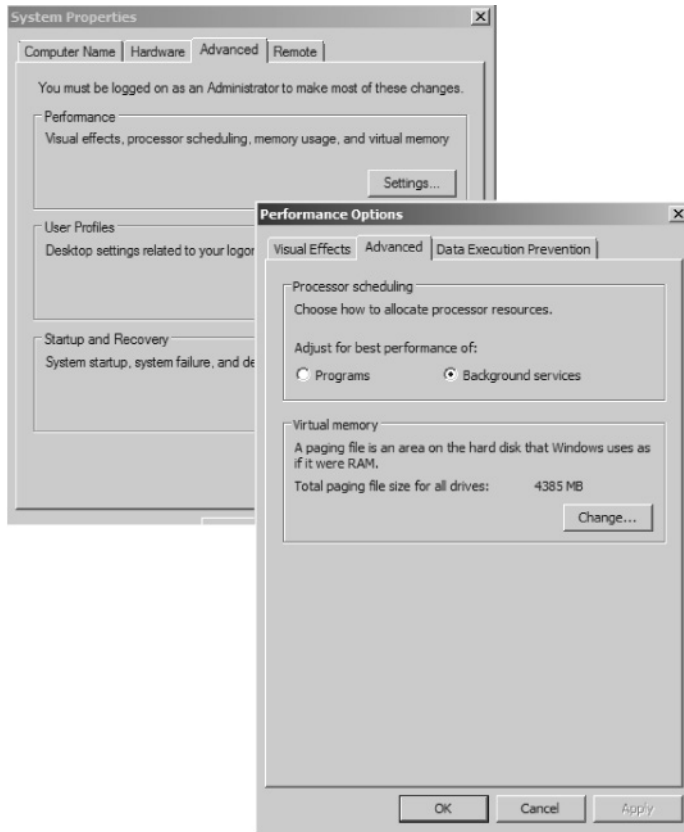


FIGURE 1-3 Configuration of priority for background services.

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.

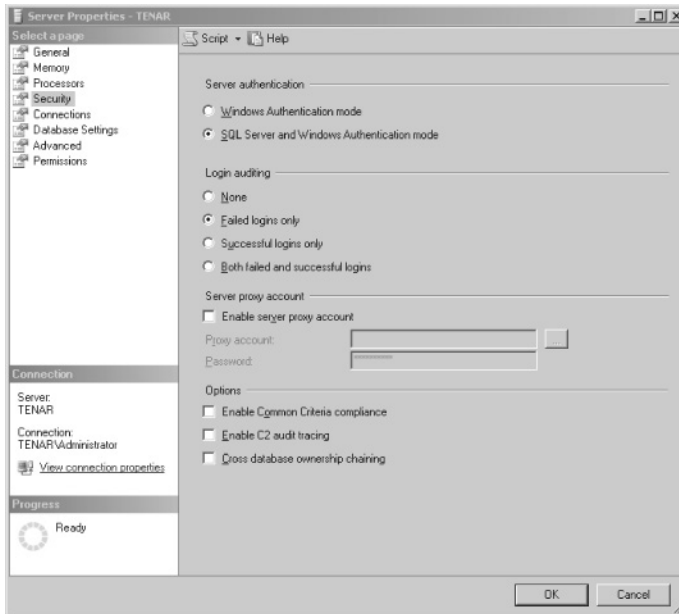


FIGURE 1-4 List of server property pages in SQL Server Management Studio.

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.

Microsoft Press Ebooks—Your bookshelf on your devices!



When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in five, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, Android .apk, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we’ve updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O’Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace), and Amazon.com.

O'REILLY®

Spreading the knowledge of innovators

oreilly.com