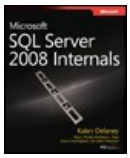

Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 3: Databases and Database Files

Kalen Delaney

Overview

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

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

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

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

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

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

System Databases

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

master

The *master* database is composed of system tables that keep track of the server installation as a whole and all other databases that are subsequently created. Although every database has a set of system catalogs that maintain information about objects that the database contains, the *master* database has system catalogs that keep information about disk

space, file allocations and usage, system-wide configuration settings, endpoints, login accounts, databases on the current instance, and the existence of other servers running SQL Server (for distributed operations).

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

model

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

tempdb

The *tempdb* database is used as a workspace. It is unique among SQL Server databases because it's re-created—not recovered—every time SQL Server is restarted. It's used for temporary tables explicitly created by users, for worktables that hold intermediate results created internally by SQL Server during query processing and sorting, for maintaining row versions used in snapshot isolation and certain other operations, and for materializing static cursors and the keys of keyset cursors. Because the *tempdb* database is re-created, any objects or permissions that you create in the database are lost the next time you start your SQL Server instance. An alternative is to create the object in the *model* database, from which *tempdb* is copied. (Keep in mind that any objects that you create in the *model* database also are added to any new databases you create subsequently. If you want objects to exist only in *tempdb,* you can create a startup stored procedure that creates the objects every time your SQL Server instance starts.)

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

The Resource Database

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

You can see the files in your default *binn* directory by using Microsoft Windows Explorer. My data directory is at C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\ MSSQL\Binn; I can see a file called mssqlsystemresource.mdf, which is 60.2 MB in size, and mssqlsystemresource.ldf, which is 0.5 MB. The created and modified date for both of these files is the date that the code for the current build was frozen. It should be the same date that you see when you run *SELECT @@version.* For SQL Server 2008, the RTM build, this is 10.0.1600.22.

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

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

SQL Server treats this new *resource_COPY* database like any other user database, and it does not treat the objects in it

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

msdb

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

Sample Databases

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

AdventureWorks

AdventureWorks actually comprises a family of sample databases that was created by the Microsoft User Education group as an example of what a "real" database might look like. The family includes: *AdventureWorks2008, AdventureWorksDW2008,* and *AdventureWorksLT2008,* as well as their counterparts created for SQL Server 2005: *AdventureWorks, AdventureWorksDW,* and *AdventureWorksLT.* You can download these databases from the Microsoft codeplex site at *http://www.codeplex.com/SqlServerSamples.* The database was designed to showcase SQL Server features, including the organization of objects into different schemas. These databases are based on data needed by the fictitious Adventure Works Cycles company. The *AdventureWorks* and *AdventureWorks2008* databases are designed to support OLTP applications and *AdventureWorksDW* and *AdventureWorksDW2008* are designed to support the business intelligence features of SQL Server and are based on a completely different database architecture. Both designs are highly normalized. Although normalized data and many separate schemas might map closely to a real production database's design, they can make it quite difficult to write and test simple queries and to learn basic SQL.

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

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

pubs

The *pubs* database is a sample database that was used extensively in earlier versions of SQL Server. Many older publications with SQL Server examples assume that you have this database because it was installed automatically on versions of SQL Server prior to SQL Server 2005. You can download a script for building this database from Microsoft's

Web site, and I have also included the script with this book's companion content at *http://www.SQLServerInternals.com/companion.*

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

Northwind

The *Northwind* database is a sample database that was originally developed for use with Microsoft Office Access. Much of the pre–SQL Server 2005 documentation dealing with application programming uses *Northwind*. *Northwind* is a bit more complex than *pubs*, and, at almost 4 MB, it is slightly larger. As with *pubs*, you can download a script from the Microsoft Web site to build it, or you can use the script provided with the companion content. The file is called Instnwnd.sql. In addition, some of the sample scripts for this book use a modified copy of *Northwind* called *Northwind2.*

Database Files

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

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

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

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

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

Table 3-1: The sys.database_files Catalog View

Column	Description
fileid	The file identification number (unique for each database).
file_guid	GUID for the file. NULL = Database was upgraded from an earlier version of SQL Server.
type	File type:
	0 = Rows (includes full-text catalogs upgraded to or created in SQL Server 2008)
	1 = Log
	2 = FILESTREAM
	3 = Reserved for future use

	4 = Full-text (includes full-text catalogs from versions earlier than SQL Server 2008)
type_desc	Description of the file type:
	ROWS
	LOG
	FILESTREAM
	FULLTEXT
data_space_id	ID of the data space to which this file belongs. Data space is a filegroup.
	0 = Log file.
name	The logical name of the file.
physical_name	Operating-system file name.
state	File state:
	0 = ONLINE
	1 = RESTORING
	2 = RECOVERING
	3 = RECOVERY_PENDING
	4 = SUSPECT
	5 = Reserved for future use
	6 = OFFLINE
	7 = DEFUNCT
state_desc	Description of the file state:
	ONLINE
	RESTORING
	RECOVERING
	RECOVERY_PENDING
	SUSPECT
	OFFLINE
	DEFUNCT
size	Current size of the file, in 8-KB pages.
	0 = Not applicable
	For a database snapshot, size reflects the maximum space that the snapshot can ever use for the file.
max_size	Maximum file size, in 8-KB pages:
	0 = No growth is allowed.
	–1 = File will grow until the disk is full.
	268435456 = Log file will grow to a maximum size of 2 terabytes.
growth	0 = File is a fixed size and will not grow.
	>0 = File will grow automatically.
	If *is_percent_growth* = 0, growth increment is in units of 8-KB pages, rounded to the nearest 64 KB.
	If *is_percent_growth* = 1, growth increment is expressed as a whole number percentage.
is_media_read_only	1 = File is on read-only media.
	0 = File is on read/write media.
is_read_only	1 = File is marked read-only.
	0 = File is marked read/write.
is_sparse	1 = File is a sparse file.
	0 = File is not a sparse file.

	(Sparse files are used with database snapshots, discussed later in this chapter.)
is_percent_growth	See description for *growth* column, above.
is_name_reserved	1 = Dropped file name (name or physical_name) is reusable only after the next log backup. When files are dropped from a database, the logical names stay in a reserved state until the next log backup. This column is relevant only under the full recovery model and the bulk-logged recovery model.

Creating a Database

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

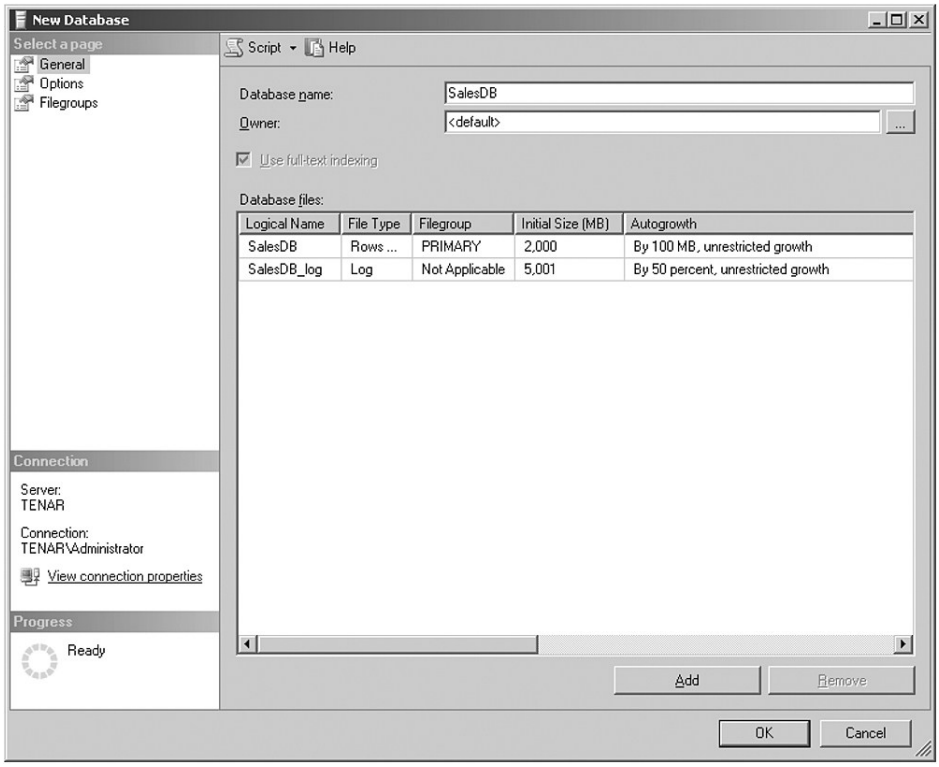

Figure 3-1: The New Database dialog box, where you can create a new database

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

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

```
CREATE DATABASE newdb;
```

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

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

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

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

A CREATE DATABASE Example

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

```
CREATE DATABASE Archive
ON
PRIMARY
( NAME = Arch1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat1.mdf',
SIZE = 100MB,
MAXSIZE = 200MB,
FILEGROWTH = 20MB),
( NAME = Arch2,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archdat2.ndf',
SIZE = 10GB,
MAXSIZE = 50GB,
FILEGROWTH = 250MB)
LOG ON
( NAME = Archlog1,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\archlog1.ldf',
SIZE = 2GB,
MAXSIZE = 10GB,
FILEGROWTH = 100MB);
```

Expanding or Shrinking a Database

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

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

"DBCC Internals."

Automatic File Expansion

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

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

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

Manual File Expansion

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

Fast File Initialization

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

Automatic Shrinkage

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

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

Manual Shrinkage

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

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

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

DBCC SHRINKFILE

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

DBCC SHRINKDATABASE

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

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

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

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

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

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

Using Database Filegroups

You can group data files for a database into filegroups for allocation and administration purposes. In some cases, you can improve performance by controlling the placement of data and indexes into specific filegroups on specific drives or

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

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

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

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

The Default Filegroup

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

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

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

Why Use Multiple Files?

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

First, if you need to restore a database from a backup because of a disk crash, the new database must contain the same number of files as the original. For example, if your original database consisted of one large 120-GB file, you would need to restore it to a database with one file of that size. If you don't have another 120-GB drive immediately available, you cannot restore the database. If, however, you originally created the database on several smaller files, you have added flexibility during a restoration. You might be more likely to have several 40-GB drives available than one large 120-GB drive.

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

Objects that have space allocated to them, namely tables and indexes, are created on a particular filegroup. (They can also be created on a partition scheme, which is a collection of filegroups. I'll discuss partitioning and partition schemes in Chapter 7.) If the filegroup (or partition scheme) is not specified, objects are created on the default filegroup. When you add space to objects stored in a particular filegroup, the data is stored in a *proportional fill* manner, which means that if

you have one file in a filegroup with twice as much free space as another, the first file has two extents (or units of space) allocated from it for each extent allocated from the second file. (I'll discuss extents in more detail in the section entitled "Space Allocation," later in this chapter.) It's recommended that you create all of your files to be the same size to avoid the issues of proportional fill.

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

A FILEGROUP CREATION Example

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

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

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

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

Filestream Filegroups

I briefly mentioned filestream storage in Chapter 1, "SQL Server 2008 Architecture and Configuration," when I talked about configuration options. Filestream filegroups can be created when you create a database, just like regular filegroups can be, but you must specify that the filegroup is for filestream data by using the phrase CONTAINS FILESTREAM. Unlike regular filegroups, each filestream filegroup can contain only one file reference, and that file is specified as an operating system folder, not a specific file. The path up to the last folder must exist, and the last folder must not exist. So in my example, the path C:\Data must exist, but the *Reviews_FS* subfolder cannot exist when you execute the *CREATE DATABASE* command. Also unlike regular filegroups, there is no space preallocated to the filegroup and you do not specify size or growth information for the file within the filegroup. The file and filegroup will grow as data is added to tables that have been created with filestream columns:

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

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

Altering a Database

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

- Change the name of the database.
- Add one or more new data files to the database. If you want, you can put these files in a user-defined filegroup. All files added in a single *ALTER DATABASE* command must go in the same filegroup.
- Add one or more new log files to the database.
- Remove a file or a filegroup from the database. You can do this only if the file or filegroup is completely empty. Removing a filegroup removes all the files in it.
- Add a new filegroup to a database. (Adding files to those filegroups must be done in a separate *ALTER DATABASE* command.) Modify an existing file in one of the following ways:
    - Increase the value of the *SIZE* property.
    - Change the *MAXSIZE* or *FILEGROWTH* property.
    - Change the logical name of a file by specifying a *NEWNAME* property. The value of *NEWNAME* is then used as the *NAME* property for all future references to this file.
    - Change the *FILENAME* property for files, which can effectively move the files to a new location. The new name or location doesn't take effect until you restart SQL Server. For *tempdb*, SQL Server automatically creates the files

with the new name in the new location; for other databases, you must move the file manually after stopping your SQL Server instance. SQL Server then finds the new file when it restarts.

- Mark the file as OFFLINE. You should set a file to OFFLINE when the physical file has become corrupted and the file backup is available to use for restoring. (There is also an option to mark the whole database as OFFLINE, which I'll discuss shortly when I talk about database properties.) Marking a file as OFFLINE allows you to indicate that you don't want SQL Server to recover that particular file when it is restarted. Modify an existing filegroup in one of the following ways:
  - Mark the filegroup as READONLY so that updates to objects in the filegroup aren't allowed. The primary filegroup cannot be made READONLY.
  - Mark the filegroup as READWRITE, which reverses the READONLY property.
  - Mark the filegroup as the default filegroup for the database.
  - Change the name of the filegroup.
- Change one or more database options. (I'll discuss database options later in the chapter.)

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

ALTER DATABASE Examples

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

This example increases the size of a database file:

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

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

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

Databases Under the Hood

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

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

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

Space Allocation

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

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

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

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

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

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

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

Table 3-2: Bit Settings in GAM and SGAM Pages

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

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

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

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

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

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

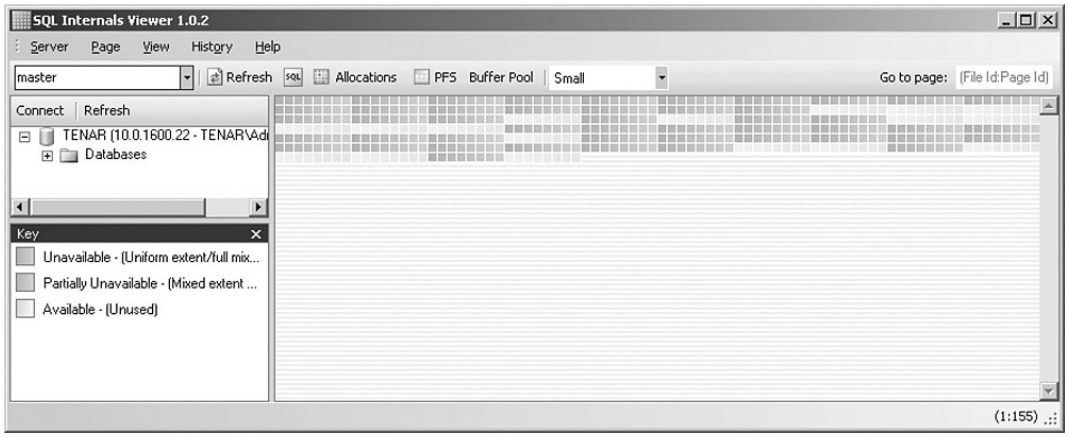

Figure 3-3: SQL Internals Viewer indicating the allocation status of each page

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

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

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

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

there will be additional IAM pages.

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

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

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

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

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

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

Setting Database Options

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

Examining the *sys.databases* catalog view can show you the current values of all the options. The view also contains other useful information, such as database ID, creation date, and the Security ID (SID) of the database owner. The following query retrieves some of the most important columns from *sys.databases* for the four databases that exist on a new default installation of SQL Server:

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

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

```
name   database_id owner create_date             user_access_desc state_desc
------ ----------- ----- ----------------------- ---------------- ----------
master 1           sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE
tempdb 2           sa    2008-04-19 12:02:35.327 MULTI_USER       ONLINE
```

```
model  3          sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE
msdb   4          sa    2008-03-21 01:54:05.240 MULTI_USER       ONLINE
```

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

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

State options

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

Cursor options

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

Auto options

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

Parameterization options

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

SQL options

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

9. RECURSIVE_TRIGGERS { ON | OFF }

Database recovery options

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

External access options

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

Database mirroring options

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

Service Broker options

1. ENABLE_BROKER | DISABLE_BROKER
2. NEW_BROKER
3. ERROR_BROKER_CONVERSATIONS

Change Tracking options

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

Database Encryption options

1. ENCRYPTION {ON | OFF}

Snapshot Isolation options

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

State Options

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

SINGLE_USER | RESTRICTED_USER | MULTI_USER

The three options SINGLE_USER, RESTRICTED_USER, and MULTI_USER describe the user access property of a database. They are mutually exclusive; setting any one of them unsets the others. To set one of these options for your database, you just use the option name. For example, to set the *AdventureWorks2008* database to single-user mode, use the following code:

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

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

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

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

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

OFFLINE | ONLINE | EMERGENCY

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

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

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

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

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

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

metadata pointers. By specifically putting your database in EMERGENCY mode, you are acknowledging that the data might be inconsistent but that you want access to it anyway.

READ_ONLY | READ_WRITE

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

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

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

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

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

Termination Options

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

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

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

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

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

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

Cursor Options

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

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

Auto Options

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

- **AUTO_CLOSE** When this option is set to ON, the database is closed and shut down cleanly when the last user of the database exits, thereby freeing any resources. All file handles are closed, and all in-memory structures are removed so that the database is not using any memory. When a user tries to use the database again, it reopens. If the database was shut down cleanly, the database isn't initialized (reopened) until a user tries to use the database the next time SQL Server is restarted. The AUTO_CLOSE option is handy for personal SQL Server databases because it allows you to manage database files as normal files. You can move them, copy them to make backups, or even e-mail them to other users. However, you shouldn't use this option for databases accessed by an application that repeatedly makes and breaks connections to SQL Server. The overhead of closing and reopening the database between each connection will hurt performance.
- **AUTO_SHRINK** When this option is set to ON, all of a database's files are candidates for periodic shrinking. Both data files and log files can be automatically shrunk by SQL Server. The only way to free space in the log files so that they can be shrunk is to back up the transaction log or set the recovery model to SIMPLE. The log files shrink at the point that the log is backed up or truncated. This option is never recommended.
- **AUTO_CREATE_STATISTICS** When this option is set to ON (the default), the SQL Server Query Optimizer creates statistics on columns referenced in a query's WHERE or ON clause. Adding statistics improves query performance because the SQL Server Query Optimizer can better determine how to evaluate a query.
- **AUTO_UPDATE_STATISTICS** When this option is set to ON (the default), existing statistics are updated if the data in the tables has changed. SQL Server keeps a counter of the modifications made to a table and uses it to determine when statistics are outdated. When this option is set to OFF, existing statistics are not automatically updated. (They can be updated manually.) Statistics will be discussed in more detail in Chapter 6 and Chapter 8, "The Query Optimizer."

SQL Options

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

- **ANSI_NULL_DEFAULT** When this option is set to ON, columns comply with the ANSI SQL-92 rules for column nullability. That is, if you don't specifically indicate whether a column in a table allows NULL values, NULLs are allowed. When this option is set to OFF, newly created columns do not allow NULLs if no nullability constraint is

specified.

- **ANSI_NULLS** When this option is set to ON, any comparisons with a NULL value result in UNKNOWN, as specified by the ANSI-92 standard. If this option is set to OFF, comparisons of non-Unicode values to NULL result in a value of TRUE if both values being compared are NULL.
- **ANSI_PADDING** When this option is set to ON, strings being compared with each other are set to the same length before the comparison takes place. When this option is OFF, no padding takes place.
- **ANSI_WARNINGS** When this option is set to ON, errors or warnings are issued when conditions such as division by zero or arithmetic overflow occur.
- **ARITHABORT** When this option is set to ON, a query is terminated when an arithmetic overflow or division-by-zero error is encountered during the execution of a query. When this option is OFF, the query returns NULL as the result of the operation.
- **CONCAT_NULL_YIELDS_NULL** When this option is set to ON, concatenating two strings results in a NULL string if either of the strings is NULL. When this option is set to OFF, a NULL string is treated as an empty (zero-length) string for the purposes of concatenation.
- **NUMERIC_ROUNDABORT** When this option is set to ON, an error is generated if an expression will result in loss of precision. When this option is OFF, the result is simply rounded. The setting of ARITHABORT determines the severity of the error. If ARITHABORT is OFF, only a warning is issued and the expression returns a NULL. If ARITHABORT is ON, an error is generated and no result is returned.
- **QUOTED_IDENTIFIER** When this option is set to ON, identifiers such as table and column names can be delimited by double quotation marks, and literals must then be delimited by single quotation marks. All strings delimited by double quotation marks are interpreted as object identifiers. Quoted identifiers don't have to follow the T-SQL rules for identifiers when QUOTED_IDENTIFIER is ON. They can be keywords and can include characters not normally allowed in T-SQL identifiers, such as spaces and dashes. You can't use double quotation marks to delimit literal string expressions; you must use single quotation marks. If a single quotation mark is part of the literal string, it can be represented by two single quotation marks ('). This option must be set to ON if reserved keywords are used for object names in the database. When it is OFF, identifiers can't be in quotation marks and must follow all T-SQL rules for identifiers.
- **RECURSIVE_TRIGGERS** When this option is set to ON, triggers can fire recursively, either directly or indirectly. Indirect recursion occurs when a trigger fires and performs an action that causes a trigger on another table to fire, thereby causing an update to occur on the original table, which causes the original trigger to fire again. For example, an application updates table *T1,* which causes trigger *Trig1* to fire. *Trig1* updates table *T2,* which causes trigger *Trig2* to fire. *Trig2* in turn updates table *T1,* which causes *Trig1* to fire again. Direct recursion occurs when a trigger fires and performs an action that causes the same trigger to fire again. For example, an application updates table *T3,* which causes trigger *Trig3* to fire. *Trig3* updates table *T3* again, which causes trigger *Trig3* to fire again. When this option is OFF (the default), triggers can't be fired recursively.

Database Recovery Options

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

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

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

- **CHECKSUM** When the PAGE_VERIFY option is set to CHECKSUM, SQL Server calculates a checksum over the contents of each page and stores the value in the page header when a page is written to disk. When the page is read from disk, a checksum is recomputed and compared with the value stored in the page header. If the values do not

match, error message 824 (indicating a checksum failure) is reported.

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

Both checksum and torn page errors generate error message 824, which is written to both the SQL Server error log and the Windows event log. For any page that generates an 824 error when read, SQL Server inserts a row into the system table *suspect_pages* in the *msdb* database. (*SQL Server Books Online* has more information on “Understanding and Managing the suspect _pages Table.”)

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

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

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

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

Other Database Options

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

Database Snapshots

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

Database snapshots allow you to do the following:

- Turn a database mirror into a reporting server. (You cannot read from a database mirror, but you can create a snapshot of the mirror and read from that.)

- Generate reports without blocking or being blocked by production operations.
- Protect against administrative or user errors.

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

Creating a Database Snapshot

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

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

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

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

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

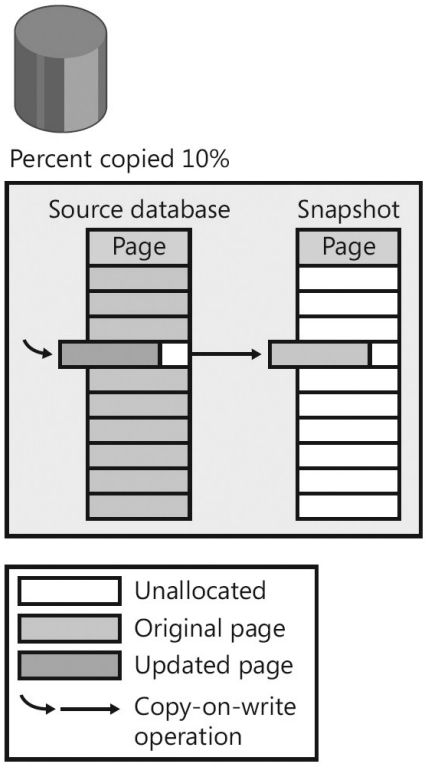

Figure 3-4: A database snapshot that contains one page of data from the source database

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

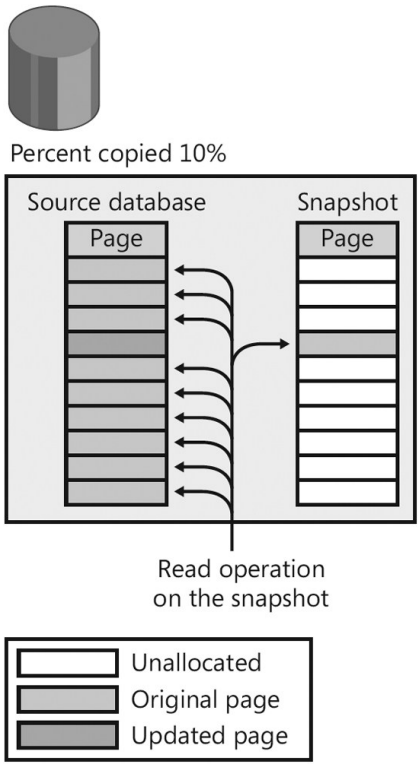

Figure 3-5: Read operations from a database snapshot, reading changed pages from the snapshot and unchanged pages from the source database

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

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

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

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

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

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

Space Used by Database Snapshots

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

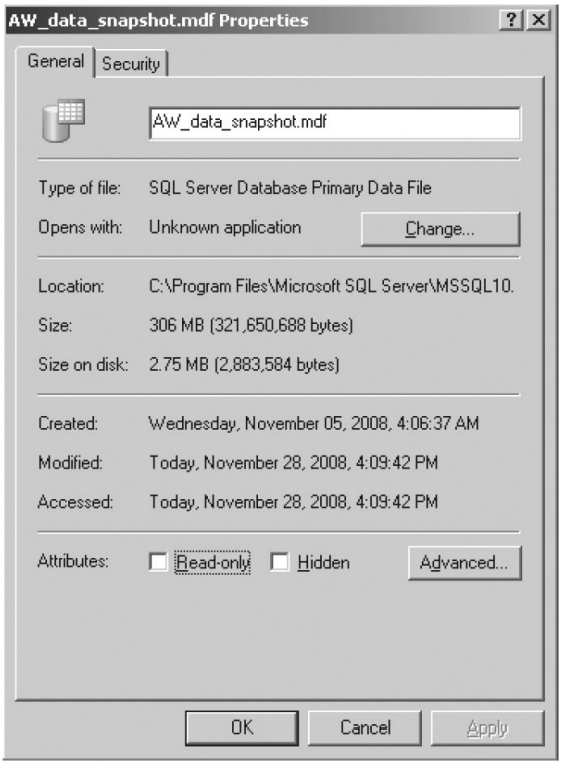

Figure 3-6: The snapshot file's Properties dialog box in Windows Explorer showing the current size of the sparse file as the size on disk

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

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

Managing Your Snapshots

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

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

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

Keep in mind these additional considerations regarding database snapshots:

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

The tempdb Database

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

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

Every time *tempdb* is re-created, it inherits most database options from the model database. However, the recovery model is not copied because *tempdb* always uses simple recovery, which will be discussed in detail in Chapter 4. Certain database options cannot be set for *tempdb*, such as OFFLINE and READONLY. You also cannot drop the *tempdb* database.

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

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

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

Objects in tempdb

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

User Objects

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

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

Internal Objects

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

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

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

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

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

Version Store

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

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

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

Optimizations in tempdb

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

Logging Optimizations

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

Allocation and Caching Optimizations

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

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

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

For a large table, the entire drop is performed as a deferred operation. Deferred drop operations are in fact used in every database as a way to improve overall throughput because a thread does not need to wait for the drop to complete before proceeding with its next task. Like the other allocation optimizations that are available in all databases, the deferred drop probably provides the most benefit in *tempdb*, which is where tables are most likely to be dropped during production operations. A background thread eventually cleans up the space allocated for dropped tables, but until then, the allocated space remains. You can detect this space by looking at the *sys.allocation_units* system view for rows with a *type* value of 0, which indicates a dropped object; you will also see that the column called *container_id* is 0, which indicates that the allocated space does not really belong to any object. I'll look at *sys.allocation_units* and the other system views that keep track of space usage in Chapter 5.

Best Practices

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

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

Database options for *tempdb* should rarely be changed, and some options are not applicable to *tempdb*. In particular, the autoshrink option is ignored in *tempdb*. In any case, shrinking *tempdb* is not recommended unless your workload patterns have changed significantly. If you do need to shrink your *tempdb*, you're probably better off shrinking each file individually. Keep in mind that the files might not be able to shrink if any internal objects or version store pages need to be moved. The best way to shrink *tempdb* is to ALTER the database, change the files' sizes, and then stop and restart SQL Server so *tempdb* is rebuilt to the desired size. You should allow your *tempdb* files to autogrow only as a last resort and only to

prevent errors due to running out of room. You should not rely on autogrow to manage the size of your *tempdb* files. Autogrow causes a delay in processing when you can probably least afford it, although the impact is somewhat less if you use instant file initialization. You should determine the size of *tempdb* through testing and planning so that *tempdb* can start with as much space as it needs and won't have to grow while your applications are running.

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

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

tempdb Space Monitoring

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

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

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

Two other system views are similar to each other:

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

Database Security

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

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

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

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

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

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

Database Access

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

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

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

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

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

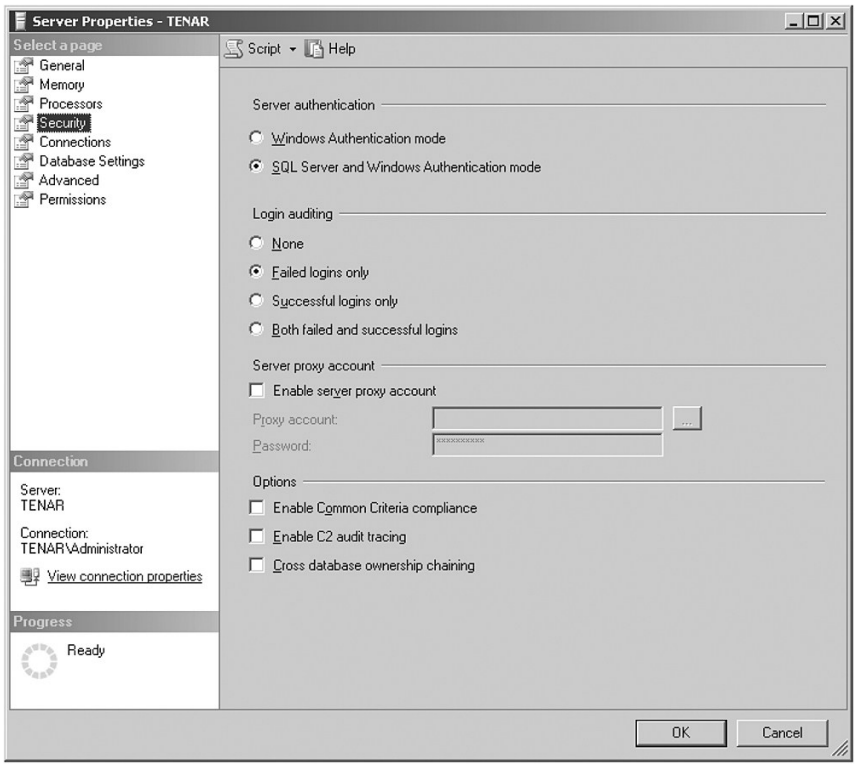

Figure 3-7: Choosing an authentication mode for your SQL Server instance in the Server Properties dialog box

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

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

Managing Database Security

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

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

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

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

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

Note that the login *sue* has the same value for the user name in this database. There is no guarantee that other databases that *sue* has access to will use the same user name. The login name *sa* has the user name *dbo*. This name is a special login that is used by the *sa* login, by all logins in the *sysadmin* role, and by whatever login is listed in *sys.databases* as the owner of the database. Within a database, it is users, not logins, who own objects, and users, not logins, to whom permissions are granted.

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

Databases vs. Schemas

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

Principals and Schemas

Prior to SQL Server 2005, there was a *CREATE SCHEMA* command, but it effectively did nothing because there was an implicit relationship between users and schemas that could be changed or removed. In fact, the relationship was so close that many users of these earlier versions of SQL Server were unaware that users and schemas are different things. Every user was the owner of a schema that has the same name as the user. If you created a user *sue*, for example, SQL Server 2000 created a schema called *sue*, which was *sue*'s default schema.

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

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

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

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

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

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

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

Default Schemas

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

Users can be assigned a default schema that might or might not exist when the user is created. A user can have at most one default schema at any time. As mentioned earlier, if no default schema is specified for a user, the default schema for the user is *dbo.* A user's default schema is used for name resolution during object creation or object reference. This can be both good news and bad news for backward compatibility. The good news is that if you've upgraded a database from SQL

Server 2000, which has many objects in the *dbo* schema, your code can continue to reference those objects without having to specify the schema explicitly. The bad news is that for object creation, SQL Server tries to create the object in the *dbo* schema rather than in a schema owned by the user creating the table. The user might not have permission to create objects in the *dbo* schema, even if that is the user's default schema. To avoid confusion, in SQL Server 2008 you should always specify the schema name for all object access as well as object management.

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

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

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

Moving or Copying a Database

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

Detaching and Reattaching a Database

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

Here is the command to detach a database:

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

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

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

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

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

Note that only the primary file is required to have a <filespec> entry because the primary file contains information about the

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

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

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

If your transaction log is rebuilt by attaching the database, using the FOR ATTACH_REBUILD_ LOG breaks the log backup chain. You should consider making a full backup after performing this operation.

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

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

Backing Up and Restoring a Database

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

Moving System Databases

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

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

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

1. For each file in the database to be moved, use the *ALTER DATABASE* command with the MODIFY FILE option to specify the new physical location.

2. Stop the SQL Server instance.
3. Physically move the files.
4. Restart the SQL Server instance.
5. Verify the change by running the following query:

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

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

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

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

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

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

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

Moving the master Database

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

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

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

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

3. Stop the instance of SQL Server and physically move the files for to the new location.
4. Restart the instance of SQL Server.

5. Verify the file change for the *master* database by running the following query:

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

Compatibility Levels

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

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

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

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

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

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

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

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

Summary

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

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