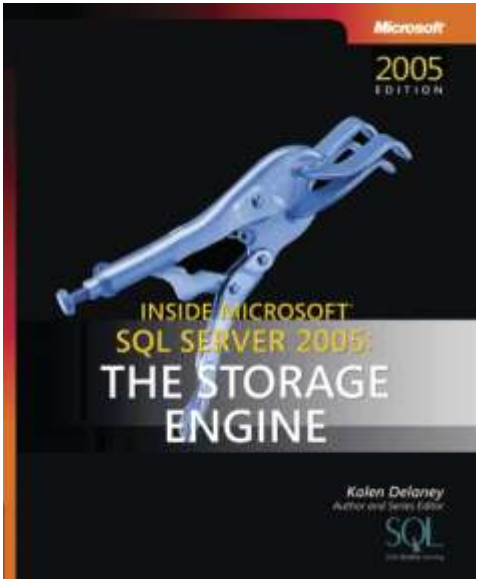

Inside Microsoft® SQL Server™ 2005: The Storage Engine

Kalen Delaney
(Solid Quality Learning)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/7436.aspx

9780735621053
Publication Date: October 2006

Microsoft Press

Table of Contents

Chapter 2

SQL Server 2005 Architecture

In this chapter:

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

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

Components of the SQL Server Engine

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

The protocol layer receives the request and translates it into a form that the relational engine can work with, and it also takes the final results of any queries, status messages, or error messages and translates them into a form the client can understand before sending them back to the client. The relational engine layer accepts SQL batches and determines what to do with them. For Transact-SQL queries and programming constructs, it parses, compiles, and optimizes the request and oversees the process of executing the batch. As the batch is executed, if data is needed, a request for that data is passed to the storage engine. The storage engine manages all data access, both through transaction-based commands and bulk operations such as backup, bulk insert, and certain DBCC (Database Consistency Checker) commands. The SQLOS layer handles activities that are normally considered to be operating system responsibilities, such as thread management (scheduling), synchronization primitives, deadlock detection, and memory management, including the buffer pool.

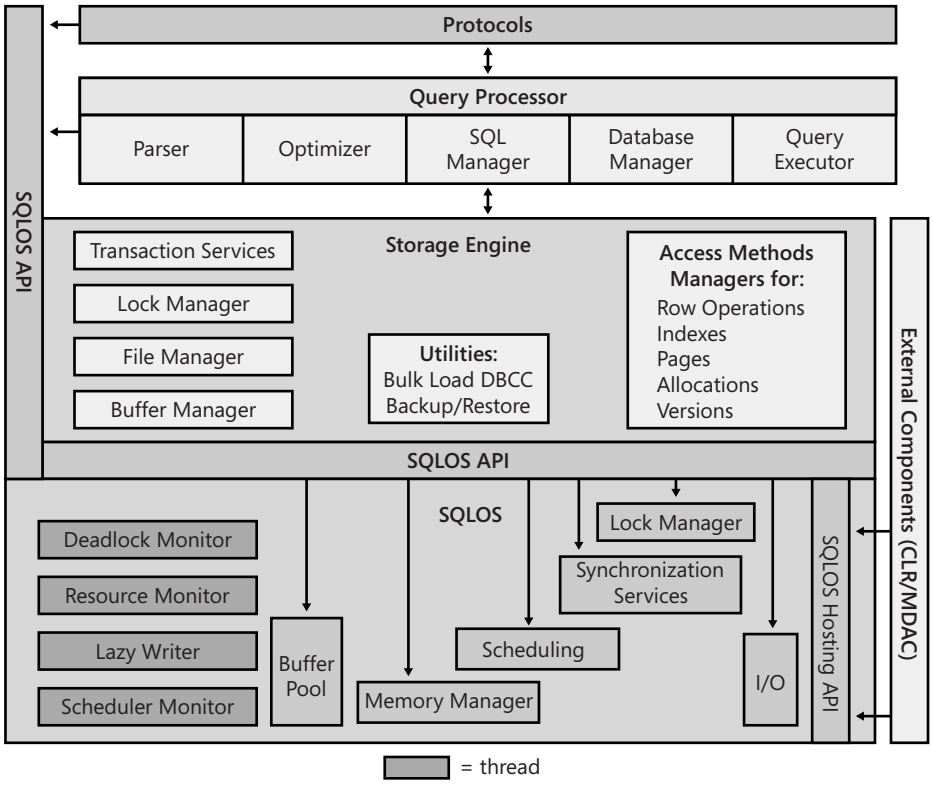

Figure 2-1 The major components of the SQL Server database engine

Observing Engine Behavior

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

The DMVs and DMFs aren't based on real tables stored in database files but are based on internal server structures, some of which I'll discuss in this chapter. I'll discuss further details about the DMVs and DMFs in various places in this book, where the contents of one or more of the objects can illuminate the topics being discussed. The objects are separated into several categories based on the functional area of the information they expose. They are all in the sys schema and have a name that starts with *dm_*, followed by a code indicating the area of the server with which the object deals. The main categories I'll address are:

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

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

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

Protocols

When an application communicates with the SQL Server Database Engine, the application programming interfaces (APIs) exposed by the protocol layer formats the communication

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

SQL Server can be configured to support multiple protocols simultaneously, coming from different clients. Each client connects to SQL Server with a single protocol. If the client program does not know which protocols SQL Server is listening on, you can configure the client to attempt multiple protocols sequentially. In Chapter 3, I'll discuss how you can configure your machine to use one or more of the available protocols. The following protocols are available:

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

Tabular Data Stream Endpoints

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

The Relational Engine

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

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

The Command Parser

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

The Query Optimizer

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

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

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

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

After normalization and optimization are completed, the normalized tree produced by those processes is compiled into the execution plan, which is actually a data structure. Each command included in it specifies exactly which table will be affected, which indexes will be used (if any), which security checks must be made, and which criteria (such as equality to a specified value) must evaluate to TRUE for selection. This execution plan might be considerably more complex than is immediately apparent. In addition to the actual commands, the execution plan includes all the steps necessary to ensure that constraints are checked. Steps for calling a trigger are slightly different from those for verifying constraints. If a trigger is included for the action being taken, a call to the procedure that comprises the trigger is appended. If the trigger is an *instead-of* trigger, the call to the trigger's plan replaces the actual data modification command. For *after* triggers, the trigger's plan is branched to right after the plan for the modification statement that fired the trigger, before that modification is committed. The specific steps for the trigger are not compiled into the execution plan, unlike those for constraint verification.

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

The SQL Manager

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

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

The Database Manager

The database manager handles access to the metadata needed for query compilation and optimization, making it clear that none of these separate modules can be run completely separately from the others. The metadata is stored as data and is managed by the storage engine, but metadata elements such as the datatypes of columns and the available indexes on a table must be available during the query compilation and optimization phase, before actual query execution starts.

The Query Executor

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

The Storage Engine

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

Access Methods

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

is employed not only for queries (selects) but also for qualified updates and deletes (for example, UPDATE with a WHERE clause) and for any data modification operations that need to modify index entries.

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

The row operations code retrieves, modifies, and performs operations on individual rows. It performs an operation within a row, such as "retrieve column 2" or "write this value to column 3." As a result of the work performed by the access methods code, as well as by the lock and transaction management components (discussed shortly), the row is found and appropriately locked as part of a transaction. After formatting or modifying a row in memory, the row operations code inserts or deletes a row. There are special operations that the row operations code needs to handle if the data is a Large Object (LOB) datatype—text, image, or ntext—or if the row is too large to fit on a single page and needs to be stored as overflow data. We'll look at the different types of page storage structures in Chapter 6.

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

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

SQL Server uses eight types of disk pages: data pages, LOB pages, index pages, Page Free Space (PFS) pages, Global Allocation Map and Shared Global Allocation Map (GAM and SGAM) pages, Index Allocation Map (IAM) pages, Bulk Changed Map (BCM) pages, and Differential Changed Map (DCM) pages. All user data is stored on data or LOB pages, and all index rows are stored on index pages. PFS pages keep track of which pages in a database are available to hold new data. Allocation pages (GAMs, SGAMs, and IAMs) keep track of the other pages. They contain no database rows and are used only internally. Bulk Changed Map pages and Differential Changed Map pages are used to make backup and recovery more efficient. I'll explain these types of pages in Chapter 6 and Chapter 7.

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

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

Transaction Services

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

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

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

The transaction management component also coordinates with the locking code regarding when locks can be released, based on the isolation level in effect. It also coordinates with the versioning code to determine when old versions are no longer needed and can be removed from the version store. The isolation level in which your transaction runs determines how sensitive your application is to changes made by others and consequently how long your transaction must hold locks or maintain versioned data to protect against those changes.

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

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

The behavior of your transactions depends on the isolation level and the concurrency model you are working with. A complete understanding of isolation levels also requires an understanding of locking because the topics are so closely related. The next section gives an overview of locking; you'll find more detailed information on isolation, transactions, and concurrency management in Chapter 8.

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

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

Other Operations

Also included in the storage engine are components for controlling utilities such as bulk load, DBCC commands, and backup and restore operations. There is a component to control sorting operations and one to physically manage the files and backup devices on disk. These components are discussed in Chapter 4. The log manager makes sure that log records are written in a manner to guarantee transaction durability and recoverability; I'll go into detail about the transaction log in Chapter 5.

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

The SQLOS

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

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

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

NUMA Architecture

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

The main benefit of NUMA is scalability, which has definite limits when you use symmetric multiprocessing (SMP) architecture. With SMP, all memory access is posted to the same shared memory bus. This works fine for a relatively small number of CPUs, but problems appear when you have many CPUs competing for access to the shared memory bus. The trend in hardware has been to have more than one system bus, each serving a small set of processors. NUMA limits the number of CPUs on any one memory bus. Each group of processors has its own memory and possibly its own I/O channels. However, each CPU can access memory associated with other groups in a coherent way, and I'll discuss this a bit more later in the chapter. Each group is called a *NUMA node* and the nodes are connected to each other by means of a high speed interconnection. The number of CPUs within a NUMA node depends on the hardware vendor. It is faster to access local memory than the memory associated with other NUMA nodes. This is the reason for the name *Non-Uniform Memory Access*. Figure 2-2 shows a NUMA node with four CPUs.

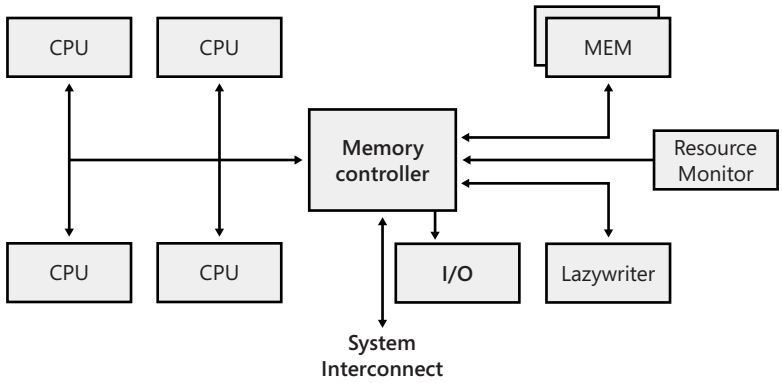

Figure 2-2 A NUMA node with four CPUs

SQL Server 2005 allows you to subdivide one or more physical NUMA nodes into smaller NUMA nodes, referred to as *software NUMA* or *soft-NUMA*. You typically use *soft-NUMA* when you have many CPUs and do not have hardware NUMA because soft-NUMA allows only for the subdividing of CPUs but not memory. You can also use soft-NUMA to subdivide hardware

NUMA nodes into groups of fewer CPUs than is provided by the hardware NUMA. Your soft-NUMA nodes can also be configured to listen on its own port.

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

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

The Scheduler

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

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

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

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

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

configured Max Worker Threads and the number of schedulers, and each scheduler is responsible for creating or destroying workers as needed. A worker cannot move from one scheduler to another, but as workers are destroyed and created, it can appear as if workers are moving between schedulers.

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

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

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

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

In SQL Server 2005, a SPID is no longer bound to a particular scheduler. Each SPID has a preferred scheduler, which is the scheduler that most recently processed a request from the SPID. The SPID is initially assigned to the scheduler with the lowest load. (You can get some insight into the load on each scheduler by looking at the load_factor column in the DMV *dm_os_schedulers*.) However, when subsequent requests are sent from the same SPID, if another scheduler has a load factor that is less than a certain percentage of the average of all the scheduler's load factor, the new task is given to the scheduler with the smallest load factor. There is a restriction that all tasks for one SPID must be processed by schedulers on the same NUMA node. The exception to this restriction is when a query is being executed as a parallel query across multiple CPUs. The optimizer can decide to use more CPUs that are available on the NUMA node processing the query, so other CPUs (and other schedulers) can be used.

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

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

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

on any CPU on that node. However, if you have set the Affinity Mask Configuration option, each scheduler on each node will be fixed to run on a particular CPU.

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

```
SELECT session_id,
    CONVERT (varchar(10), t1.status) AS status,
    CONVERT (varchar(20), t1.command) AS command,
    CONVERT (varchar(15), t2.state) AS worker_state
FROM sys.dm_exec_requests AS t1 JOIN sys.dm_os_workers AS t2
ON  t2.task_address = t1.task_address
WHERE command = 'RESOURCE MONITOR'
```

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

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

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

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

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

affinity mask to 255 (bit string 11111111), all the CPUs will be used, just like with the default. However, with the affinity mask set, the eight CPUS will be bound to the eight schedulers.

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

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

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

- **parent_node_id** The ID of the node that the scheduler belongs to, also known as the *parent node*. This represents a NUMA node.
- **scheduler_id** The ID of the scheduler. All schedulers that are used to run regular queries have IDs of less than 255. Those with IDs greater than or equal to 255, such as the dedicated administrator connection scheduler, are used internally by SQL Server.
- **cpu_id** The ID of the CPU with which this scheduler is associated. If SQL Server is configured to run with affinity, the value is the ID of the CPU on which the scheduler is supposed to run. If the affinity mask has not been specified, the cpu_id will be 255.
- **is_online** If SQL Server is configured to use only some of the available processors on the server, this can mean that some schedulers are mapped to processors that are not in the affinity mask. If that is the case, this column returns 0. This means the scheduler is not being used to process queries or batches.

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

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

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

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

- **started_by_sqlservr** Indicates the thread initiator. A 1 means that SQL Server started the thread and 0 means that another component, such as an extended procedure from within SQL Server, started the thread.

- **creation_time** The time when this thread was created.
- **stack_bytes_used** The number of bytes that are actively being used on the thread.
- **Affinity** The CPU mask on which this thread is supposed to be running. This depends on the value in the sp_configure "affinity mask."
- **Locale** The cached locale LCID for the thread.

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

- **task_state** The state of the task. The value can be one of the following:
  - PENDING: Waiting for a worker thread
  - RUNNABLE: Runnable but waiting to receive a quantum
  - RUNNING: Currently running on the scheduler
  - SUSPENDED: Has a worker but is waiting for an event
  - DONE: Completed
  - SPINLOOP: Processing a spinlock, as when waiting for a signal
- **context_switches_count** The number of scheduler context switches that this task has completed.
- **pending_io_count** The number of physical I/Os performed by this task.
- **pending_io_byte_count** The total byte count of I/Os performed by this task.
- **pending_io_byte_average** The average byte count of I/Os performed by this task.
- **scheduler_id** The ID of the parent scheduler. This is a handle to the scheduler information for this task.
- **session_id** The ID of the session associated with the task.

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

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

- **blocking_exec_context_id** The ID of the execution context of the blocking task.
- **resource_description** The description of the resource that is being consumed.

The Dedicated Administrator Connection

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

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

Access via the DAC must be specially requested. You can also connect to the DAC using the command-line tool SQLCMD, by using the */A* flag. This method of connection is recommended because it uses fewer resources than the graphical interface method but offers more functionality than other command-line tools, such as osql. Through SQL Server Management Studio, you can specify that you want to connect using DAC by preceding the name of your SQL Server with *ADMIN:* in the Connection dialog box.

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

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

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

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

- By default, the DAC is available only locally. However, an administrator can configure SQL Server to allow remote connection by using the configuration option called Remote Admin Connections.
- The user logon to connect via the DAC must be a member of SYSADMIN server role.

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

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

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

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

Memory

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

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

The Buffer Pool and the Data Cache

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

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

Access to In-Memory Data Pages

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

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

Note Finding a data page might require that multiple buffers be accessed via the hash buckets chain (linked list). The hash function attempts to uniformly distribute the dbid-fileno-pageno values throughout the available hash buckets. The number of hash buckets is set internally by SQL Server and depends on the total size of the buffer pool.

Managing Pages in the Data Cache

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

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

Every buffer in the data cache has a header that contains information about the last two times the page was referenced and some status information, including whether the page is dirty (has been changed since it was read in to disk). The reference information is used to implement the page replacement policy for the data cache pages, which uses an algorithm called LRU-K.[1] This algorithm is a great improvement over a strict LRU (Least Recently Used) replacement policy, which has no knowledge of how recently a page was used. It is also an improvement over an LFU (Least Frequently Used) policy involving reference counters because it requires far fewer adjustments by the engine and much less bookkeeping overhead. An LRU-K algorithm keeps track of the last K times a page was referenced and can differentiate between types of pages, such as index and data pages, with different levels of frequency. Its can actually simulate the effect of assigning pages to different buffer pools of specifically tuned sizes. SQL Server 2005 uses a K value of 2, so it keeps track of the two most recent accesses of each buffer page.

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

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

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

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

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

When SQL Server uses memory dynamically, it must constantly be aware of the amount of free memory. The lazywriter for each node queries the system periodically to determine the amount of free physical memory available. The lazywriter expands or shrinks the data cache to keep the operating system's free physical memory at 5 megabytes (MB) plus or minus 200 KB to prevent paging. If the operating system has less than 5 MB free, the lazywriter releases memory to the operating system instead of adding it to the free list. If more than 5 MB of physical memory is free, the lazywriter recommits memory to the buffer pool by adding it to the free list. The lazywriter recommits memory to the buffer pool only when it repopulates the free list; a server at rest does not grow its buffer pool.

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

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

Checkpoints

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

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

Checkpoints are triggered when:

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

option to influence checkpointing frequency, balancing the time to recover vs. any impact on run-time performance. If you're interested in tracing how often checkpoints actually occur, you can start SQL Server with trace flag 3502, which writes information to the SQL Server error log every time a checkpoint occurs.

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

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

Managing Memory in Other Caches

Buffer pool memory that isn't used for the data cache is used for other types of caches, primarily the procedure cache, which actually holds plans for all types of queries, not just procedure plans. The page replacement policy, and the mechanism by which freeable pages are searched for, is quite a bit different than for the data cache.

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

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

Memory management in the stores takes into account both global and local memory management policies. Global policies consider the total memory on the system and enable the running of the clock algorithm across all the caches. Local policies involve looking at one store or cache in isolation and making sure it is not using a disproportionate amount of memory.

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

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

The Memory Broker

Because memory is needed by so many components in SQL Server, and to make sure each component uses memory efficiently, Microsoft introduced a Memory Broker late in the development cycle for SQL Server 2005. The Memory Broker's job is to analyze the behavior of SQL Server with respect to memory consumption and to improve dynamic memory distribution. The Memory Broker is a centralized mechanism that dynamically distributes memory between the buffer pool, the query executor, the query optimizer, and all the various

caches, and it attempts to adapt its distribution algorithm for different types of workloads. You can think of the Memory Broker as a control mechanism with a feedback loop. It monitors memory demand and consumption by component, and it uses the information it gathers to calculate the optimal memory distribution across all components. It can broadcast this information to the component, which then uses the information to adapt its memory usage. You can monitor Memory Broker behavior by querying the Memory Broker ring buffer:

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

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

Sizing Memory

When we talk about SQL Server memory, we're actually talking about more than just the buffer pool. SQL Server memory is actually organized into three sections, and the buffer pool is usually the largest and most frequently used. The buffer pool is used as a set of 8-KB buffers, so any memory that is needed in chunks larger than 8 KB is managed separately. The DMV called *sys.dm_os_memory_clerks* has a column called multi_pages_kb that shows how much space is used by a memory component outside the buffer pool:

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

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

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

Sizing the Buffer Pool

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

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

On a 64-bit platform, the AWE Enabled configuration option is present, but its setting is ignored. However, the Windows policy Lock Pages in Memory option is available, although it is disabled by default. This policy determines which accounts can make use of a Windows feature to keep data in physical memory, preventing the system from paging the data to virtual memory on disk. It is recommended that you enable this policy on a 62-bit system.

On 32-bit operating systems, you will have to enable Lock Pages in Memory policy when using AWE. It is recommended that you don't enable the Lock Pages in Memory policy if you are not using AWE. Although SQL Server will ignore this option when AWE is not enabled, other processes on the system may be impacted.

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

Table 2-1 shows the possible memory configurations for various editions of SQL Server 2005.

Table 2-1 SQL Server 2005 Memory Configurations

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

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

Manager object. There is also a DMV called *sys.dm_os_sys_info* that contains one row of general-purpose SQL Server configuration information, including the following columns:

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

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

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

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

Observing Memory Internals

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

sys.dm_os_memory_clerks This view returns one row per memory clerk that is currently active in the instance of SQL Server. You can think of a clerk as an accounting unit. Each store described earlier is a clerk, but some clerks are not stores, such as those for the CLR and for full-text search. The following query returns a list of all the types of clerks:

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

Interesting columns include the following:

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

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

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

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

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

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

- **clock_hand** The type of clock hand, either external or internal. Remember that there are two clock hands for every store.
- **clock_status** The status of the clock hand: suspended or running. A clock hand runs when a corresponding policy kicks in.

- **rounds_count** The number of rounds the clock hand has made. All the external clock hands should have the same (or close to the same) value in this column.
- **removed_all_rounds_count** The number of entries removed by the clock hand in all rounds.

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

NUMA and Memory

As mentioned earlier, one major reason for implementing NUMA is to handle large amounts of memory efficiently. As clock speed and the number of processors increase, it becomes increasingly difficult to reduce the memory latency required to use this additional processing power. Large L3 caches can help alleviate part of the problem, but this is only a limited solution. NUMA is the scalable solution of choice. SQL Server 2005 has been designed to take advantage of NUMA-based computers without requiring any application changes. Keep in mind that the NUMA memory nodes are completely dependent on the hardware NUMA configuration. If you define your own soft-NUMA, as discussed earlier, you will not affect the number of NUMA memory nodes. So, for example, if you have an SMP computer with eight CPUs and you create four soft-NUMA nodes with two CPUs each, you will have only one MEMORY node serving all four NUMA nodes. Soft-NUMA does not provide memory to CPU affinity. However, there is a network I/O thread and a lazywriter thread for each NUMA node, either hard or soft.

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

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

In general, the NUMA nodes function largely independently of each other, but that is not always the case. For example, if a worker running on a node N1 needs to access a database

page that is already in node N2's memory, it does so by accessing N2's memory, which is called non-local memory. Note that non-local is not the same as foreign memory.

Read-Ahead

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

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

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

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

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

Final Words

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

Chapter 4

Databases and Database Files

In this chapter:

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

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

- It is a collection of many objects, such as tables, views, stored procedures, and constraints. The technical limit is $2^{31}-1$ (more than 2 billion) objects. The number of objects typically ranges from hundreds to tens of thousands.
- It is owned by a single SQL Server login account.
- It maintains its own set of user accounts, roles, schemas, and security.
- It has its own set of system tables and views to hold the database catalog.
- It is the primary unit of recovery and maintains logical consistency among objects within it. (For example, primary and foreign key relationships always refer to other tables within the same database, not in other databases.)

- It has its own transaction log and manages its own transactions.
- It can span multiple disk drives and operating system files.
- It can range in size from 1 megabyte (MB) to a technical limit of 1,048,516 terabytes.
- It can grow and shrink, either automatically or by command.
- It can have objects joined in queries with objects from other databases in the same SQL Server instance or on linked servers.
- It can have specific options set or disabled. (For example, you can set a database to be read-only or to be a source of published data in replication.)

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

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

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

System Databases

A new SQL Server 2005 installation always includes four databases: *master*, *model*, *tempdb*, and *msdb*. It also contains a fifth, "hidden" database that you will never see using any of the normal SQL commands that list all your databases. This database is referred to as the *resource database*, but its actual name is *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 it contains, the *master* database has system catalogs that keep information about disk space, file allocations and usage, systemwide configuration settings, endpoints, login accounts, databases on the current instance, and the existence of other SQL servers (for distributed operations).

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

model

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

tempdb

The *tempdb* database is used as a workspace. It is unique among SQL Server databases because it's re-created—not recovered—every time SQL Server is restarted. It's used for temporary tables explicitly created by users, for worktables that will hold intermediate results created internally by SQL Server during query processing and sorting, for maintaining row versions used in
snapshot isolation and certain other operations, and for materializing static cursors and the keys of keyset cursors. Because the *tempdb* database is re-created, any objects or permissions that you create in the database will be lost the next time you restart your SQL Server instance. An alternative is to create the object in the *model* database, from which *tempdb* is copied.

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

mssqlsystemresource

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

You can see the files in your default data directory by using Windows Explorer. My data directory is at C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\data; I can see a file called mssqlsystemresource.mdf, which is 38 MB in size, and mssqlsystemresource.ldf, which is 0.5 MB. The created and modified date for both of these files is the day I installed this SQL Server instance, but their last accessed date is today.

If you have a burning need to "see" the contents of *mssqlsystemresource*, a couple of methods are available. The easiest, if you just want to see what's there, is to stop SQL Server, make copies of the two files for the resource database, restart SQL Server, and then attach the

copied files to create a database with a new name. You can do this by using Object Explorer in SQL Server Management Studio or by using the CREATE DATABASE FOR ATTACH syntax to create a clone database, as shown here:

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

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

msdb

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

Sample Databases

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

AdventureWorks

The *AdventureWorks* database was created by the Microsoft User Education group as an example of what a "real" database might look like. It is an optional component that you can choose to install during the installation process. The database was designed to showcase SQL Server 2005 features, in particular the organization of objects into different schemas. The design is also highly normalized. While normalized data and many separate schemas might closely map to a real production database's design, they can make it quite difficult to write and test simple queries and to learn basic SQL.

Database design is not a major focus of this book, so most of my examples will use simple tables that I create; if more than a few rows of data are needed, I'll copy data from one or more *AdventureWorks* tables into tables of my own. It's a good idea to become familiar with the design of the *AdventureWorks* database because many of the examples in Books Online and in white papers published on the Microsoft Web site use data from this database. Note that it is also possible to install an *AdventureWorksDW* database, which includes data and features relevant to a data warehouse as well as SQL Server 2005 data warehousing features. I will not discuss that database in this book.

pubs

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

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

Northwind

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

Database Files

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

SQL Server 2005 allows the following three types of database files:

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

Each database file has five properties that can be specified when you create the file: a logical filename, a physical filename, an initial size, a maximum size, and a growth increment. The value of these properties, along with other information about each file, can be seen through the metadata view *sys.database_files*, which contains one row for each file used by a database. Most of the columns shown in *sys.database_files* are listed in Table 4-1. The columns not mentioned here contain information dealing with transaction log backups relevant to the particular file, and I'll be discussing the transaction log in Chapter 5.

Table 4-1 The sys.database_files View

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

Table 4-1 The sys.database_files View

Column	Description
type_desc	Description of the file type: ROWS LOG 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-kilobyte (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.

Table 4-1 The sys.database_files View

Column	Description
growth	0 = File is 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 SQL Server Management Studio, which provides a graphical front end to the Transact-SQL commands and stored procedures that actually create the database and set its properties. Figure 4-1 shows the New Database dialog box, which represents the Transact-SQL CREATE DATABASE command for creating a new user database. Only someone with the appropriate permissions can create a database, either through Object Explorer or by using the CREATE DATABASE command. This includes anyone in the *sysadmin* role, anyone who has been granted CONTROL or ALTER permission on the server, and any user who has been granted CREATE DATABASE permission by someone with the *sysadmin* or *dbcreator* role.

When you create a new database, SQL Server copies the *model* database. If you have an object that you want created in every subsequent user database, you should create that object in *model* first. You can also use *model* to set default database options in all subsequently created databases. The *model* database includes 47 objects— 41 system tables and 6 objects used for SQL Server Notification Services and Service Broker. You can see these objects by selecting from the system table called *sys.objects*. However, if you run the procedure *sp_help* in the *model* database, it will list 1788 objects. It turns out that most of these objects are not really stored in the *model* database but are accessible through it. In Chapter 6, I'll tell you what the other kinds

of objects are and how you can tell whether an object is really stored in a particular database. Most of the objects you see in *model* will show up when you run *sp_help* in any database, but your user databases will probably have more objects added to this list. The contents of *model* are just the starting point.

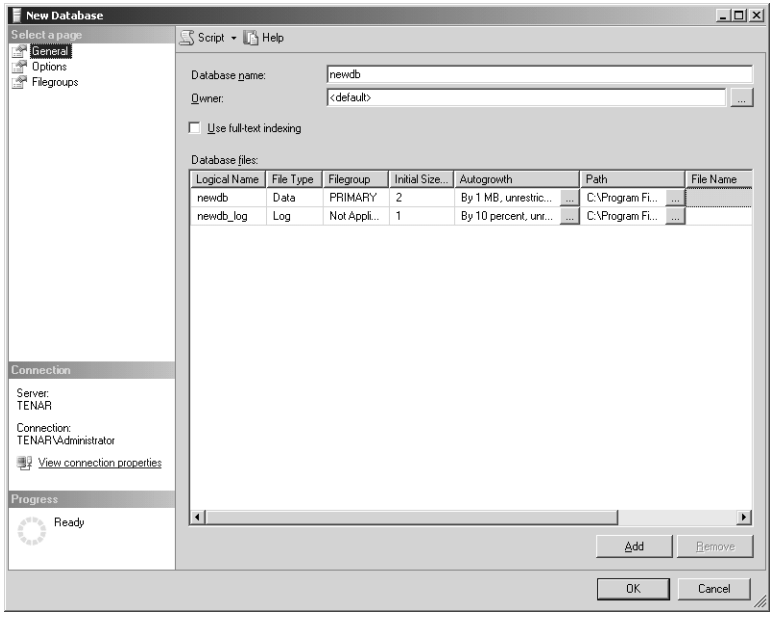

Figure 4-1 The New Database dialog box, where you can create a new database

A new user database must be 1 MB in size or larger, and the primary data file size must be at least as large as the primary data file of the *model* database. (The *model* database only has one file and cannot be altered to add more. So the size of the primary data file and the size of the database are basically the same for *model*.) Almost all the possible arguments to the CREATE DATABASE command have default values, so it's possible to create a database using a simple form of CREATE DATABASE, such as this:

```
CREATE DATABASE newdb;
```

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

The SQL Server login account that created the database is known as the *database owner*, and that information is stored with the information about the database properties in the *master* database. A database can have only one actual owner, who always corresponds to a login name. Any login that uses any database has a user name in that database, which might be the same name as the login name but doesn't have to be. The login that is the owner of a database always has the special user name *dbo* when using the database it owns. I'll discuss database

users later in this chapter when I tell you about the basics of database security. The default size of the data file is the size of the primary data file of the *model* database, and the default size of the log file is 1 MB. Whether the database name, *newdb*, is case sensitive depends on the sort order you chose during setup. If you accepted the default, the name is case insensitive. (Note that the actual command CREATE DATABASE is case insensitive, regardless of the case sensitivity chosen for data.)

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

If the MAXSIZE clause isn't specified for the files, the file will grow until the disk is full. (In other words, the file size is considered unlimited.) You can specify the values for SIZE, MAXSIZE, and FILEGROWTH in units of terabyte, gigabyte (GB), MB (the default), or KB. You can also specify the FILEGROWTH property as a percentage. A value of 0 for FILEGROWTH indicates no growth. If no FILEGROWTH value is specified, the default growth increment for data files is 1 MB. This is a change from SQL Server 2000, where the default growth increment for data files was 10 percent. In SQL Server 2005, the log file FILEGROWTH default is 10 percent, the same as it was in SQL Server 2000.

A CREATE DATABASE Example

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

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

Expanding or Shrinking a Database

Databases can be expanded and shrunk automatically or manually. The mechanism for automatic expansion is completely different from the mechanism for automatic shrinkage. Manual expansion is also handled differently than manual shrinkage. Log files have their own rules for growing and shrinking; I'll discuss changes in log file size in Chapter 5.

Warning Shrinking a database or any data file is an extremely resource-intensive operation, and the only reason to do it is if you absolutely must recover disk space.

Automatic File Expansion

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

Allowing SQL Server to grow your data files automatically is no substitute for good capacity planning before you build or populate any tables. Enabling autogrow might prevent some failures due to unexpected increases in data volume, but it can also cause problems. If a data file is full and your autogrow percentage is set to grow by 10 percent, if an application attempts to insert a single row and there is no space, the database might start to grow by a large amount. (Ten percent of 10,000 MB is 1000 MB.) This in itself can take a lot of time if fast file initialization (discussed in the next section) is not being used. The growth might take so long that the client application's timeout value is exceeded, which means the insert query will fail. The query would have failed anyway if autogrow wasn't set, but with autogrow enabled, SQL Server will spend a lot of time trying to grow the file, and you won't be informed of the problem immediately.

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

Manual File Expansion

You can manually expand a database file by using the ALTER DATABASE command to change the SIZE property of one or more of the files. When you alter a database, the new size of a file

must be larger than the current size. To decrease the size of a file, you use the DBCC SHRINKFILE command, which I'll tell you about shortly.

Fast File Initialization

In SQL Server 2005, data files can be initialized instantaneously. This allows for fast execution of the file creation and growth. Instant file initialization adds space to the data file without filling the newly added space with zeros. Instead, the actual disk content is overwritten only as new data is written to the files. Until the data is overwritten, there is always the chance that a hacker using an external file reader tool can see the data that was previously on the disk. Although the SQL Server 2005 documentation describes the instant file initialization feature as an "option," it is not an option within SQL Server. It is actually controlled through a Windows security setting called SE_MANAGE_VOLUME_NAME, which is granted to Windows Administrators by default. (This right can be granted to other Windows users by adding them to the Perform Volume Maintenance Tasks security policy.) If your SQL Server (MSSQLSERVER) service account is in the Windows Administrator role and your SQL Server is running on a Windows XP or Windows 2003 file system, instant file initialization will be used. If you want to make sure your database files are zeroed out as they are created and expanded, you can use traceflag 1806 to always zero the space, as previous SQL Server versions did.

Automatic Shrinkage

The database property *autoshrink* allows a database to shrink automatically. The effect is the same as doing a DBCC SHRINKDATABASE *(dbname, 25)*. This option leaves 25 percent free space in a database after the shrink, and any free space beyond that is returned to the operating system. The thread that performs autoshrink—which always has a session ID (SPID) of 6 in SQL Server 2005 (but there's no guarantee SQL Server will use the same SPID in future versions)—shrinks databases at 30-minute intervals. I'll discuss the DBCC SHRINKDATABASE command in more detail momentarily.

Manual Shrinkage

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

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

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

DBCC SHRINKFILE

DBCC SHRINKFILE allows you to shrink files in the current database. When you specify *target_size*, DBCC SHRINKFILE attempts to shrink the specified file to the specified size in

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

DBCC SHRINKDATABASE

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

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

Because DBCC SHRINKDATABASE shrinks the database on a file-by-file basis, the mechanism used to perform the actual shrinking is the same as that used with DBCC SHRINKFILE. SQL Server first moves pages to the front of files to free up space at the end, and then it releases the appropriate number of freed pages to the operating system.

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

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

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

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

```
DBCC SHRINKDATABASE(pubs, 10);
RESULTS:
DbId   FileId CurrentSize MinimumSize UsedPages   EstimatedPages
------ ------ ----------- ----------- ----------- --------------
5      1      256         80          152         152
5      2      1152        63          1152        56
```

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

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

For the log file (FileId = 2), the only values that really matter are the current size and the minimum size. The other two values are basically meaningless for log files because the current size is always the same as the used pages and because there is really no simple way to estimate how small a log file can be shrunk. Shrinking a log file is very different from shrinking a data file, and understanding how much you can shrink a log file, and what exactly happens when you shrink it, requires an understanding of how the log is used. For this reason, I will postpone the discussion of shrinking log files until Chapter 5.

As the warning at the beginning of this section indicated, shrinking a database or any data files is a resource-intensive operation. If you absolutely need to recover disk space from the database, you should plan the shrink operation carefully and perform it when it will have the least impact on the rest of the system. You should never enable the AUTOSHRINK option, which will shrink *all* the data files at regular intervals and wreak havoc with system performance. Because shrinking data files can move data all around a file, it can also introduce fragmentation, which you then might want to remove. Defragmenting your data files can then have its own impact on productivity because it uses system resources. I'll discuss fragmentation and defragmentation in Chapter 7.

It is possible for shrink operations to be blocked by a transaction that has been enabled for the snapshot isolation level. When this happens, DBCC SHRINKFILE and DBCC

SHRINKDATABASE print out an informational message to the error log every five minutes in the first hour and then every hour after that. SQL Server 2005 also provides progress reporting for the SHRINK commands, available through the *sys.dm_exec_requests* view. I discuss progress reporting in the section on DBCC commands, or you can get the full details from the Books Online page for *sys.dm_exec_requests*.

Using Database Filegroups

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

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

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

- The primary file is always the first file listed when you create a database, and it typically has the file extension .mdf. The one special feature of the primary file is that it has pointers into a table in the master database called *sysfiles1* that contains information about all the files belonging to the database.
- The primary filegroup is always the filegroup that contains the primary file. This filegroup contains the primary data file and any files not put into another specific filegroup. All pages from system tables are always allocated from files in the primary filegroup.

The Default Filegroup

One filegroup always has the property of DEFAULT. Note that DEFAULT is a property of a filegroup, not a name. Only one filegroup in each database can be the default filegroup. By default, the primary filegroup is the also the default filegroup. A database owner can change which filegroup is the default by using the ALTER DATABASE statement. The default filegroup contains the pages for all tables and indexes that aren't placed in a specific filegroup.

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

need 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 2005. Only then will they need to use filegroups. They can then use Object Explorer in SQL Server Management Studio to create the database on multiple filegroups. Then they can right-click on the database name in Object Explore and create a script of the CREATE DATABASE command that includes all the files in their appropriate filegroups. They can save and reuse this script when they need to re-create the database or build a similar database.

Why Use Multiple Files?

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

First, if you need to restore a database from a backup because of a disk crash, the new database must contain the same number of files as the original. For example, if your original database consisted of one large 12-GB file, you would need to restore it to a database with one file of that size. If you don't have another 12-GB drive immediately available, you cannot restore the database! If, however, you originally created the database on several smaller files, you have added flexibility during a restoration. You might be more likely to have several 4-GB drives available than one large 12-GB drive.

Second, spreading the database onto multiple files, even on the same drive, gives you the flexibility of easily moving the database onto separate drives if you modify your hardware configuration in the future.

Objects that have space allocated to them, namely tables and indexes, are created on a particular filegroup. If the filegroup is not specified, they are created on the default filegroup. When you add space to objects stored in a particular filegroup, the data is stored in a *proportional fill* manner, which means that if you have one file in a filegroup with twice as much free space as another, the first file will have two extents (or units of space) allocated from it for each extent allocated from the second file. I'll discuss extents in more detail later in this chapter.

You can also use filegroups to allow backups of parts of the database. Because a table is created on a single filegroup, you can choose to back up just a certain set of critical tables by backing up the filegroups in which you placed those tables. You can also restore individual files or filegroups in two ways. First, you can do a partial restore of a database and restore only a subset of filegroups, which must always include the primary filegroup. The database will be online as soon as the primary filegroup has been restored, but only objects created on the restored filegroups will be available. Partial restore of just a subset of filegroups can be a solution to allow very large databases (VLDBs) to be available within a mandated time

window. Alternatively, if you have a failure of a subset of the disks on which you created your database, you can restore backups of the filegroups on those disks on top of the existing database. This method of restoring also requires that you have log backups, so I'll discuss it in more detail in Chapter 5.

A FILEGROUP CREATION Example

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

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

```
CREATE DATABASE Sales
ON PRIMARY
( NAME = SPri1_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SPri1dat.mdf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
( NAME = SPri2_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SPri2dat.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SG1Fi1dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
( NAME = SGrp1Fi2_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SG1Fi2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SG2Fi1dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql.1\mssql\data\SG2Fi2dt.ndf',
SIZE = 10,
```

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

Altering a Database

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

- Change the name of the database.
- Add one or more new data files to the database. You can optionally put these files in a user-defined filegroup. All files added in a single ALTER DATABASE statement must go in the same filegroup.
- Add one or more new log files to the database.
- Remove a file or a filegroup from the database. You can do this only if the file or filegroup is completely empty. Removing a filegroup removes all the files in it.
- Add a new filegroup to a database. (Adding files to those filegroups must be done in a separate ALTER DATABASE statement.)
- Modify an existing file in one of the following ways:
  - Increase the value of the SIZE property.
  - Change the MAXSIZE or FILEGROWTH property.
  - Change the logical name of a file by specifying a NEWNAME property. The value of NEWNAME is then used as the NAME property for all future references to this file.
  - Change the FILENAME property for files, which can effectively move the files to a new location. (In SQL Server 2000, only files in the *tempdb* database can be moved in this way.) The new name or location doesn't take effect until you restart SQL Server. For *tempdb*, SQL Server automatically creates the files with the new name in the new location; for other databases, you must move the file manually after stopping your SQL Server instance. SQL Server then finds the new file when it restarts.
  - Mark the file as OFFLINE. You should set a file to OFFLINE when the physical file has become corrupted and the file backup is available to use for restoring. (There is also an option to mark the whole database as OFFLINE, which I'll discuss shortly when I talk about database properties.) Marking a file as OFFLINE

is a new feature in SQL Server 2005; it allows you to indicate that you don't want SQL Server to recover that particular file when it is restarted.

- Modify an existing filegroup in one of the following ways:
  - Mark the filegroup as READONLY so that updates to objects in the filegroup aren't allowed. The primary filegroup cannot be made READONLY.
  - Mark the filegroup as READWRITE, which reverses the READONLY property.
  - Mark the filegroup as the default filegroup for the database.
  - Change the name of the filegroup.
- Change one or more database options. (I'll discuss database options later in the chapter.)

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

ALTER DATABASE Examples

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

This example increases the size of a database file:

```
USE master
GO
ALTER DATABASE Test1
MODIFY FILE
( NAME = 'test1dat3',
SIZE = 20MB);
```

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

```
ALTER DATABASE Test1
ADD FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
ADD FILE
( NAME = 'test1dat3',
FILENAME =
    'c:\program files\microsoft sql server\ mssql.1\mssql\data\t1dat3.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB),
( NAME = 'test1dat4',
FILENAME =
    'c:\program files\microsoft sql server\ mssql.1\mssql\data\t1dat4.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB)
```

```
TO FILEGROUP Test1FG1;
GO
ALTER DATABASE Test1
MODIFY FILEGROUP Test1FG1 DEFAULT;
GO
```

Databases Under the Hood

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

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

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

Space Allocation

The space in a database is used for storing tables and indexes. The space is managed in units called *extents*. An extent is made up of eight logically contiguous pages (or 64 KB of space). To make space allocation more efficient, SQL Server 2005 doesn't allocate entire extents to tables with small amounts of data. SQL Server 2005 has two types of extents:

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

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

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

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

- **Global Allocation Map (GAM) pages** These pages record which extents have been allocated for any type of use. A GAM has a bit for each extent in the interval it covers. If the bit is 0, the corresponding extent is in use; if the bit is 1, the extent is free. Almost 8000 bytes, or 64,000 bits, are available on the page after the header and other overhead are accounted for, so each GAM can cover about 64,000 extents, or almost 4 GB of data. This means that one GAM page exists in a file for every 4 GB of size.
- **Shared Global Allocation Map (SGAM) pages** These pages record which extents are currently used as mixed extents and have at least one unused page. Just like a GAM, each SGAM covers about 64,000 extents, or almost 4 GB of data. The SGAM has a bit for each extent in the interval it covers. If the bit is 1, the extent being used is a mixed extent and has free pages; if the bit is 0, the extent isn't being used as a mixed extent, or it's a mixed extent whose pages are all in use.

Table 4-2 shows the bit patterns that each extent has set in the GAM and SGAM, based on its current use.

Table 4-2 Bit Settings in GAM and SGAM Pages

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

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

SQL Server can quickly locate the GAMs in a file because a GAM is always the third page in any database file (page 2). An SGAM is the fourth page (page 3). Another GAM appears every 511,230 pages after the first GAM on page 2, and another SGAM appears every 511,230 pages after the first SGAM on page 3. Page 0 in any file is the File Header page, and only one exists per file. Page 1 is a Page Free Space (PFS) page (which I'll discuss shortly). In Chapter 6, I'll say more about how individual pages within a table look. For now, because I'm talking about space allocation, I'll examine how to keep track of which pages belong to which tables.

Index Allocation Map (IAM) pages keep track of the extents in a 4-GB section of a database file used by an allocation unit. An allocation unit is a set of pages belonging to a single partition in a table or index and comprises pages of one of three types: pages holding regular in-row data,

pages holding Large Object (LOB) data, or pages holding row-overflow data. I'll discuss these three types of pages, and when each kind is used, in Chapter 6.

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

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

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

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

The internal system view called *sys.system_internals_allocation_units* has a column called *first_iam_page* that points to the first IAM page for an allocation unit. All the IAM pages for that allocation unit are linked in a chain, with each IAM page containing a pointer to the next in the chain. I'll discuss allocation units in more detail in Chapter 6 when I discuss object data storage.

In addition to GAMs, SGAMs, and IAMs, a database file has three other types of special allocation pages. Page Free Space (PFS) pages keep track of how each particular page in a file is used. The second page (page 1) of a file is a PFS page, as is every 8088th page thereafter. I'll talk about them more in Chapter 6. The seventh page (page 6) is called a Differential Changed Map (DCM) page. It keeps track of which extents in a file have been modified since the last full database backup. The eighth page (page 7) is called a Bulk Changed Map (BCM) page and is used when an extent in the file is used in a minimally or bulk-logged operation. I'll tell you more about these two kinds of pages when I talk about the internals of backup and restore operations in Chapter 5. Like GAM and SGAM pages, DCM and BCM pages have 1 bit for each extent in the section of the file they represent. They occur at regular intervals—every 511,230 pages.

Checking Database Consistency

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

In this section, I will discuss the DBCC commands that actually do consistency checking of the database, that is, the validation commands. These commands are the CHECK commands: DBCC CHECKTABLE, DBCC CHECKDB, DBCC CHECKALLOC, DBCC CHECKFILEGROUP, and DBCC CHECKCATALOG. Two others, DBCC CHECKCONSTRAINT and DBCC CHECKIDENT, will be described in Chapter 7, where I'll also discuss some of the table and index maintenance DBCC commands, such as DBCC CLEANTABLE and DBCC UPDATEUSAGE. I will cover DBCC INDEXDEFRAG and its SQL Server 2005 replacement when I cover indexes in Chapter 7.

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

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

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

- **DBCC CHECKALLOC is run on the database.** DBCC CHECKALLOC validates the allocation information maintained in the GAM, SGAM, and IAM pages. You can think of DBCC CHECKALLOC as performing cross-reference checks to verify that every extent that the GAM or SGAM indicates has been allocated really has been allocated, and that any extents not allocated are indicated in the GAM and SGAM as not allocated. DBCC

CHECKALLOC also verifies the IAM chain for each allocation unit, including the consistency of the links between the IAM pages in the chain. Finally, DBCC CHECKALLOC verifies that all extents marked as allocated to the allocation unit really are allocated.

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

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

- **DBCC CHECKCATALOG is run on the database.** DBCC CHECKCATALOG performs more than 50 crosschecks between various metadata tables. You cannot fix errors that it finds by running the DBCC operation with any of the REPAIR options. Prior to SQL Server 2005, DBCC CHECKCATALOG was not included in a DBCC CHECKDB operation and had to be run separately.

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

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

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

Performing Validation Checks

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

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

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

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

The DBCC validation checks can require a significant amount of space because SQL Server needs to temporarily store information about pages and structures that have been observed during the check operation, for cross-checking against pages and structures that are observed later during the DBCC scan. To determine the *tempdb* needs in advance, you can run a DBCC validation check with the ESTIMATEONLY option. For example, if I want to see how much *tempdb* space I might need to run DBCC CHECKDB on the *AdventureWorks* database, I can run the following:

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

Here is the output I receive:

```
Estimated TEMPDB space needed for CHECKALLOC (KB)
-------------------------------------------------
72

Estimated TEMPDB space needed for CHECKTABLES (KB)
--------------------------------------------------
198542
```

Note that even though *AdventureWorks* is considered just a sample database, it can require up to193 MB of *tempdb* space to run to completion. There are several large indexes in *tempdb* that contribute to this large space requirement, and in addition, this value is computed as a

worst-case estimate and assumes there will not be room in memory for any of the sort operations required.

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

```
Date1/24/2006 2:15:52 PM

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

Validation Checks

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

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

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

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

DBCC Repair Options

The validation commands DBCC CHECKDB, DBCC CHECKTABLE, and DBCC CHECKALLOC allow you to indicate whether you want SQL Server to attempt to repair any errors that

might be found. The syntax for the DBCC validation commands (except for DBCC CHECKCATALOG) allows you to specify either REPAIR_ALLOW_DATA_LOSS or REPAIR_REBUILD. Syntactically, you can also specify REPAIR_FAST, but that option is maintained only for backward compatibility, and no repair actions are performed.

Almost all the possible errors that the DBCC command can detect can be repaired. The exceptions are errors found through DBCC CHECKCATALOG and data purity errors found through DBCC CHECKTABLE. When you run DBCC CHECKDB with one of the REPAIR options, SQL Server first runs DBCC CHECKALLOC and repairs what it can, and then it runs DBCC CHECKTABLE on all tables and makes the appropriate repairs on all the tables. The possible repairs for each table are ranked as SQL Server compiles the list of what needs repairing, to make the entire DBCC operation as efficient as possible. In this way, you won't end up, for example, in a situation where an index is being rebuilt, and then a page from table has to be removed, invalidating the work of rebuilding the index.

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

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

You can run the REPAIR options for DBCC inside a user-defined transaction, which means you can perform a ROLLBACK to undo the repairs that have been made. The exception is when you are running the REPAIR options on a database in EMERGENCY mode, which I discuss later in the section on database options. (If a repair in EMERGENCY mode fails, there are no further options except to restore the database from a backup.) Each individual repair in the DBCC operation runs in its own system transaction, which means that if a repair is not possible, it will not affect any of the other repairs, unless subsequent repairs depended on an earlier success repair. If you do run one of the REPAIR options, you can provide a partial safeguard by creating a database snapshot before the repair is initiated, starting a transaction, and then running DBCC with the REPAIR option. Before committing or rolling back, you can

compare the repaired database with the original in the snapshot. If you are not satisfied with the changes made as part of the repair, you can roll back the repair operation.

Progress Reporting

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

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

Progress reporting is available for DBCC CHECKDB, DBCC CHECKTABLE, and DBCC CHECKFILEGROUP. DBCC CHECKALLOC is not included in this list because it is such a fast command there would be no need (and usually no time) to check the progress. The command would be done before you had a chance to select from *sys.dm_exec_requests*. Progress reporting is also available for some of the maintenance commands, such as DBCC SHRINKFILE and DBCC SHRINKDATABASE. SQL Server will also populate the progress report columns when defragmenting an index using ALTER INDEX with the REORG option, because this command is equivalent to DBCC INDEXDEFRAG, which also supports progress reporting.

DBCC Best Practices

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

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

Warning Use REPAIR_ALLOW_DATA_LOSS only as a last resort.

Setting Database Options

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

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

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

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

```
name   database_id owner create_date             user_access_desc state_desc
------ ----------- ----- ----------------------- ---------------- ----------

master 1           sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE

tempdb 2           sa    2006-05-27 12:02:35.327 MULTI_USER       ONLINE

model  3           sa    2003-04-08 09:13:36.390 MULTI_USER       ONLINE

msdb   4           sa    2005-10-14 01:54:05.240 MULTI_USER       ONLINE
```

The *sys.databases* view actually contains both a number and a name for both the *user_access* and *state* information. Selecting all the columns from *sys.databases* would show you that the *user_access_desc* value of MULTI_USER has a corresponding *user_access* value of 0, and the *state_desc* value of ONLINE has a *state* value of 0. Books Online shows the complete list of number and name relationships for the columns in *sys.databases*. These are just two of the database options displayed in the *sys.databases* view. The complete list of database options is divided into seven main categories: state options, cursor options, auto options, parameterization options, SQL options, database recovery options, and external access options. There are also options for specific technologies SQL Server can participate in, including database mirroring, Service Broker activities, and snapshot isolation. Some of the options, particularly the SQL options, have corresponding SET options that you can turn on or off for a particular connection.

Be aware that the ODBC or OLE DB drivers turn on a number of these SET options by default, so applications will act as if the corresponding database option has already been set.

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

State options

SINGLE_USER | RESTRICTED_USER | MULTI_USER

OFFLINE | ONLINE | EMERGENCY

READ_ONLY | READ_WRITE

Cursor options

CURSOR_CLOSE_ON_COMMIT { ON | OFF }

CURSOR_DEFAULT { LOCAL | GLOBAL }

Auto options

AUTO_CLOSE { ON | OFF }

AUTO_CREATE_STATISTICS { ON | OFF }

AUTO_SHRINK { ON | OFF }

AUTO_UPDATE_STATISTICS { ON | OFF }

AUTO_UPDATE_STATISTICS_ASYNC { ON | OFF }

Parameterization options

DATE_CORRELATION_OPTIMIZATION { ON | OFF }

PARAMETERIZATION { SIMPLE | FORCED }

SQL options

ANSI_NULL_DEFAULT { ON | OFF }

ANSI_NULLS { ON | OFF }

ANSI_PADDING { ON | OFF }

ANSI_WARNINGS { ON | OFF }

ARITHABORT { ON | OFF }

CONCAT_NULL_YIELDS_NULL { ON | OFF }

NUMERIC_ROUNDABORT { ON | OFF }

QUOTED_IDENTIFIER { ON | OFF }

RECURSIVE_TRIGGERS { ON | OFF }

Database recovery options

RECOVERY { FULL | BULK_LOGGED | SIMPLE }

TORN_PAGE_DETECTION { ON | OFF }

PAGE_VERIFY { CHECKSUM | TORN_PAGE_DETECTION | NONE }

External access options

DB_CHAINING { ON | OFF }

TRUSTWORTHY { ON | OFF }

Database mirroring options

PARTNER { = 'partner_server'

| FAILOVER

| FORCE_SERVICE_ALLOW_DATA_LOSS

| OFF

| RESUME

| SAFETY { FULL | OFF }

| SUSPEND

| TIMEOUT *integer*

}

WITNESS { = 'witness_server' |

OFF

}

Service Broker options

ENABLE_BROKER | DISABLE_BROKER

NEW_BROKER

ERROR_BROKER_CONVERSATIONS

Snapshot Isolation options

ALLOW_SNAPSHOT_ISOLATION {ON | OFF }

READ_COMMITTED_SNAPSHOT {ON | OFF } [WITH <termination>]

State Options

The state options control who can use the database and for what operations. There are three aspects to usability: The user access state determines which users can use the database, the status state determines whether the database is available to anybody for use, and the updateability state determines what operations can be performed on the database. You control each of these aspects by using the ALTER DATABASE command to enable an option

for the database. None of the state options uses the keywords ON and OFF to control the state value.

SINGLE_USER | RESTRICTED_USER | MULTI_USER

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

```
ALTER DATABASE AdventureWorks SET SINGLE_USER;
```

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

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

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

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

OFFLINE | ONLINE | EMERGENCY

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

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

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

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

As shown in the preceding query, you can determine the current status of a database by examining the *state_desc* column of the *sys.databases* view. This column can return status values other than OFFLINE, ONLINE, and EMERGENCY, but those values are not directly settable using ALTER DATABASE. A database can have the status value RESTORING while it is in the process of being restored from a backup. It can have the status value RECOVERING during a restart of SQL Server. The restore process is done on one database at a time, and until SQL Server has finished restoring a database, the database has a status of RECOVERING. If the recovery process cannot be completed for some reason (most likely because one or more of the log files for the database is unavailable or unreadable), SQL Server gives the database the status of RECOVERY_PENDING. Your databases can also be put into RECOVERY_ PENDING mode if SQL Server runs out of either log or data space during rollback recovery, or if SQL Server runs out of locks or memory during any part of the startup process. I'll go into more detail about the difference between rollback recovery and startup recovery in Chapter 5.

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

Emergency Mode Repair

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

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

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

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

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

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

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

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

You can attempt to set a database that is in EMERGENCY mode into ONLINE mode (if the missing files have been made available, for example), and SQL Server will try to run recovery on the database. If the transition to ONLINE cannot be completed, the database will be left in either RECOVERY_PENDING or SUSPECT status, just like when you first bring up your SQL Server instance and try to recover the database. Once again, you can change the state of the RECOVERY_PENDING database to EMERGENCY mode to allow the data to be read.

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

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

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

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

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

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

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

READ_ONLY | READ_WRITE

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

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

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

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

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

Termination Options

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

connections of unqualified users. The termination option determines the behavior of SQL Server in the following situations:

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

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

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

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

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

Cursor Options

The cursor options control the behavior of server-side cursors that were defined using one of the following Transact-SQL commands for defining and manipulating cursors: DECLARE,

OPEN, FETCH, CLOSE, and DEALLOCATE. Transact-SQL cursors are discussed in detail in *Inside SQL Server 2005: TSQL Programming*.

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

Auto Options

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

- **AUTO_CLOSE** When this option is set to ON, the database is closed and shut down cleanly when the last user of the database exits, thereby freeing any resources. All file handles are closed, and all in-memory structures are removed so that the database is not using any memory. When a user tries to use the database again, it reopens. If the database was shut down cleanly, the database isn't initialized (reopened) until a user tries to use the database the next time SQL Server is restarted. The AUTO_CLOSE option is handy for personal SQL Server databases because it allows you to manage database files as normal files. You can move them, copy them to make backups, or even e-mail them to other users. However, you shouldn't use this option for databases accessed by an application that repeatedly makes and breaks connections to SQL Server. The overhead of closing and reopening the database between each connection will hurt performance.
- **AUTO_SHRINK** When this option is set to ON, all of a database's files are candidates for periodic shrinking. Both data files and log files can be automatically shrunk by SQL Server. The only way to free space in the log files so that they can be shrunk is to back up the transaction log or set the recovery mode to SIMPLE. The log files shrink at the point that the log is backed up or truncated.
- **AUTO_CREATE_STATISTICS** When this option is set to ON (the default), the SQL Server query optimizer creates statistics on columns referenced in a query's WHERE clause.

Adding statistics improves query performance because the SQL Server query optimizer can better determine how to evaluate a query.

- **AUTO_UPDATE_STATISTICS** When this option is set to ON (the default), existing statistics are updated if the data in the tables has changed. SQL Server keeps a counter of the modifications made to a table and uses it to determine when statistics are outdated. When this option is set to OFF, existing statistics are not automatically updated. (They can be updated manually.) I'll discuss statistics in more detail in Chapter 7.

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

SQL Options

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

- **ANSI_NULL_DEFAULT** When this option is set to ON, columns comply with the ANSI SQL-92 rules for column nullability. That is, if you don't specifically indicate whether a column in a table allows NULL values, NULLs are allowed. When this option is set to OFF, newly created columns do not allow NULLs if no nullability constraint is specified.
- **ANSI_NULLS** When this option is set to ON, any comparisons with a NULL value result in UNKNOWN, as specified by the ANSI-92 standard. If this option is set to OFF, comparisons of non-Unicode values to NULL result in a value of TRUE if both values being compared are NULL.
- **ANSI_PADDING** When this option is set to ON, strings being compared with each other are set to the same length before the comparison takes place. When this option is OFF, no padding takes place.
- **ANSI_WARNINGS** When this option is set to ON, errors or warnings are issued when conditions such as division by zero or arithmetic overflow occur.
- **ARITHABORT** When this option is set to ON, a query is terminated when an arithmetic overflow or division-by-zero error is encountered during the execution of a query. When this option is OFF, the query returns NULL as the result of the operation.
- **CONCAT_NULL_YIELDS_NULL** When this option is set to ON, concatenating two strings results in a NULL string if either of the strings is NULL. When this option is set to OFF, a NULL string is treated as an empty (zero-length) string for the purposes of concatenation.

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

Database Recovery Options

The database option RECOVERY (FULL, BULK_LOGGED or SIMPLE) determines how much recovery can be done on a SQL Server database. It also controls how much information is logged and how much of the log is available for backups. I'll cover this option in more detail in Chapter 5.

Two other options also apply to work done when a database is recovered. Setting the TORN_PAGE_DETECTION option to ON or OFF is possible in SQL Server 2005, but that particular option will go away in a future version. The recommended alternative is to set the PAGE_VERIFY option to a value of TORN_PAGE_DETECTION or CHECKSUM. (So TORN_PAGE_DETECTION should now be considered a value, rather the name of an option.)

The PAGE_VERIFY options discover damaged database pages caused by disk I/O path errors, which can cause database corruption problems. The I/O errors themselves are generally

caused by power failures or disk failures that occur when a page is being written to disk.

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

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

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

In SQL Server 2005, the default is CHECKSUM. In SQL Server 2000, TORN_PAGE_DETECTION is the default, and CHECKSUM is not available. If you upgrade a database from SQL Server 2000, the PAGE_VERIFY value will be NONE or TORN_PAGE_DETECTION. If it is TORN_PAGE_DETECTION, you should consider changing it to CHECKSUM. Although TORN_PAGE_DETECTION uses fewer resources, it provides less protection than CHECKSUM.

Other Database Options

Of the four other main categories of database options, I will cover only one category in detail—the external access options. The snapshot isolation options will be discussed in Chapter 8. The Service Broker options are discussed in *Inside SQL Server 2005: TSQL Programming*. The database mirroring options are also beyond the scope of this book. Database mirroring is a new SQL Server 2005 technology that provides more options for high availability and is fully supported as of Service Pack 1. (Microsoft did not fully support database mirroring in the initial RTM release of SQL Server 2005, but mirroring can be enabled in that release by using trace flag 1400 as a startup parameter.) All the details, both internal and external, that you might want to know about database mirroring in SQL Server 2005 can be found in the white paper "Database Mirroring in SQL Server 2005" by Ron Talmage and the Microsoft TechNet article "Database Mirroring Best Practices and Performance Considerations" by Sanjay Mishra, which are both included with the companion content for this book.

Database Snapshots

One of the most interesting new features in SQL Server 2005 is database snapshots, which allows you to create a point-in-time read-only copy of any database. In fact, you can create multiple snapshots of the same source database at different points in time. The actual space needed for each snapshot is typically much less than the space required for the original database because the snapshot only stores pages that have changed, as we'll see shortly.

Database snapshots allow you to do the following:

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

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

Creating a Database Snapshot

The mechanics of snapshot creation are straightforward—you simply specify an option for the CREATE DATABASE command. As of this writing, there is no graphical equivalent through

Object Explorer, so you must use the Transact-SQL syntax. When you create a snapshot, you must include each data file from the source database in the CREATE DATABASE command, with the original logical name and a new physical name. No other properties of the files can be specified, and no log file is used.

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

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

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

The snapshot files contain only the data that has changed from the source. For every file, SQL Server creates a bitmap that is kept in cache, with a bit for each page of the file, indicating whether that page has been copied to the snapshot. Every time a page in the source is updated, SQL Server checks the bitmap for the file to see if the page has already been copied, and if it hasn't, it is copied at that time. This operation is called a copy-on-write operation. Figure 4-2 shows a database with a snapshot that contains 10 percent of the data (one page) from the source.

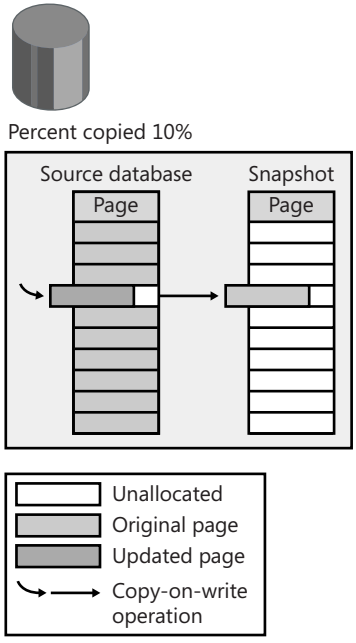

Figure 4-2 A database snapshot that contains one page of data from the source database

When a process reads from the snapshot, it first accesses the bitmap to see whether the page it wants is in the snapshot file or is still the source. Figure 4-3 shows read operations from the same database as in Figure 4-2. Nine of the pages are accessed from the source database, and one is accessed from the snapshot because it has been updated on the source. When a process reads from a snapshot database, no locks are taken no matter what isolation level you are in. This is true whether the page is read from the sparse file or from the source database. This is one of the big advantages of using database snapshots.

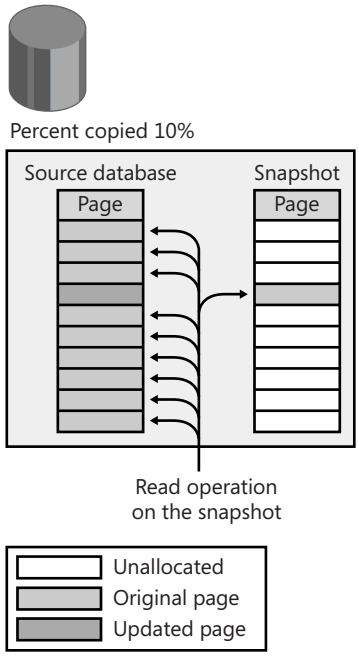

Figure 4-3 Read operations from a database snapshot, reading changed pages from the snapshot and unchanged pages from the source database

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

The snapshot reflects the point in time when the CREATE DATABASE command is issued—that is, when the creation operation commences. SQL Server checkpoints the source database and records a synchronization Log Sequence Number (LSN) in the source database's transaction log. As you'll see in Chapter 5, when I talk about the transaction log, the LSN is a way to determine a specific point in time in a database. SQL Server then runs recovery on the source database so that any uncommitted transactions are rolled back in the snapshot. So, although the sparse file for the snapshot starts out empty, it might not stay that way for long. If transactions are in progress at the time the snapshot is created, the recovery process will undo uncommitted

transactions before the snapshot database is usable, so the snapshot will contain the original versions of any page in the source that contains modified data.

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

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

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

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

Space Used by Database Snapshots

You can find out the number of bytes each sparse file of the snapshot is currently using on disk by looking at the dynamic management function *sys.dm_io_virtual_file_stats*, which returns the current number of bytes in a file in the *size_on_disk_bytes* column. This function takes *database_id* and *file_id* as parameters. The database ID of the snapshot database and the file IDs of each of its sparse files are displayed in the *sys.master_files* catalog view. You can also view the size in Windows Explorer. Right-click on the file name and look at the properties, as shown in Figure 4-4. The Size value is the maximum size, and the size on disk should be the same value that you see using *sys.dm_io_virtual_file_stats*. The maximum size should be about the same size the source database was when the snapshot was created.

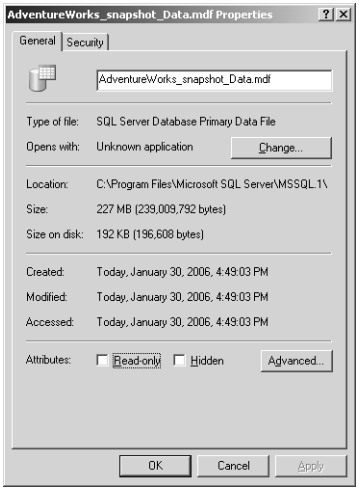

Figure 4-4 The snapshot file's Properties dialog box in Windows Explorer shows the current size of the sparse file as the size on disk.

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

It is possible to over-commit your storage. This means that under normal circumstances, you can have many times more snapshots than you have physical storage for, but if the snapshots grow, the physical volume might run out of space. (Note that this can happen when running online DBCC CHECKDB, and related commands, because you have no control of the placement of the internal snapshot that the command uses—it's placed on the same volume that the files of the parent database reside on. If this happens, the DBCC check will fail.) Once the physical volume runs out of space, the write operations to the source cannot copy the before image of the page to the sparse file. The snapshots that cannot write their pages out are marked as
suspect and are unusable, but the source database continues operating normally. There is no way to "fix" a suspect snapshot; you must drop the snapshot database.

Managing Your Snapshots

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

```
RESTORE DATABASE AdventureWOrks
FROM SNAPSHOT = AdventureWorks_snapshot;
```

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

Keep in mind these additional considerations regarding database snapshots:

- Snapshots cannot be created for the *model*, *master*, or *tempdb* databases. (Internally, snapshots can be created to run the online DBCC checks on the *master* database, but they cannot be explicitly created.)
- A snapshot inherits the security constraints of its source database, and because it is read-only, you cannot change the permissions.
- If you drop a user from the source database, the user is still in the snapshot.

- Snapshots cannot be backed up or restored, but backing up the source database works normally; it is unaffected by database snapshots.
- Snapshots cannot be attached or detached.
- Full-text indexing is not supported on database snapshots, and full-text catalogs are not propagated from the source database.

The *tempdb* Database

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

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

Every time *tempdb* is re-created, it inherits most database options from the model database. However, the recovery mode is not copied because *tempdb* always uses simple recovery, which will be discussed in detail in Chapter 5. Certain database options cannot be set for *tempdb*, such as OFFLINE, READONLY, and CHECKSUM. You also cannot drop the *tempdb* database.

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

Logging for *tempdb* is also different than for other databases. (Normal logging will be discussed in Chapter 5.) Many people assume that there is no logging in *tempdb*, but this is not true. Operations within *tempdb* are logged so that transactions on temporary objects can be rolled back, but the records in the log contain only enough information to roll back a transaction, not to recover (or redo) it.

As I mentioned earlier, recovery is run on a database as one of the first steps in creating a snapshot. We can't recover *tempdb*, so we cannot create a snapshot of it, and this means we can't run DBCC CHECKDB (or, in fact, most of the DBCC validation commands) in online mode. Another difference with running DBCC in *tempdb* is that SQL Server will skip all allocation and catalog checks. Running DBCC CHECKDB (or CHECKTABLE) in *tempdb* acquires a Shared Table lock on each table as it is checked. (Locking will be discussed in Chapter 8.)

Objects in *tempdb*

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

User Objects

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

Other user objects that need space in *tempdb* include table variables and table-valued functions. The user objects that are created in *tempdb* are in many ways treated just like user objects in any other database. Space must be allocated for them when they are populated, and the metadata needs to be managed. You can see user objects by examining the system catalog views, such as *sys.objects*, and information in the *sys.partitions* and *sys.allocation_units* views will allow you to see how much space is taken up by user objects. I'll discuss these views in Chapter 6.

Internal Objects

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

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

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

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

Sort units are created when a sort operation takes place, and this occurs in many situations in addition to a query containing an ORDER BY clause. SQL Server uses sorting to build an index, and it might use sorting to process queries involving grouping. Certain types of joins might require that SQL Server first sort the data before performing the join. Sort units are created in *tempdb* to hold the data as it is being sorted. SQL Server can also create sort units in user databases in addition to *tempdb*, in particular when creating indexes. As you'll see in Chapter 7, when you create an index, you have the option to do the sort in the current user database or in *tempdb*.

Version Store

The version store supports a new technology in SQL Server 2005 for row-level versioning of data. Older versions of updated rows are kept in *tempdb* in the following situations:

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

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

Optimizations in *tempdb*

Because *tempdb* is so much more heavily used in SQL Server 2005 than in previous versions, you have to take much more care in managing it. The next section will present some best practices and monitoring suggestions. In this section, I'll tell you about some of the internal optimizations in SQL Server that allow *tempdb* to manage objects much more efficiently.

Logging Optimizations

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

Allocation and Caching Optimizations

Many of the allocation optimizations are used in all databases, not just *tempdb*. However, *tempdb* is most likely the database in which the greatest number of new objects are created and dropped during production operations, so the impact on *tempdb* is greater than on user databases. In SQL Server 2005, allocation pages are accessed much more efficiently to determine where free extents are available; you should see far less contention on the allocation pages than in previous versions. SQL Server 2005 also has a more efficient search algorithm for finding an available single page from mixed extents. When a database has multiple files, SQL Server 2005 has a very efficient proportional fill algorithm that allocates space to multiple data files, proportional to the amount of free space available in each file.

Another optimization specific to *tempdb* prevents you from having to allocate any new space for some objects. If a work table is dropped, one Index Allocation Map (IAM) page and one extent are saved (for a total of nine pages), so there is no need to deallocate and then reallocate the space if the same work table needs to be created again. This dropped work table cache is

not very big and has room for only 64 objects. If a work table is truncated internally and the query plan that uses that worktable is still in the plan cache, again the first IAM page and the first extent are saved. For these truncated tables, there is no specific limitation on the number of objects that can be cached; it depends only on the available memory space.

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

For a large table, the entire drop is done as a deferred operation. Deferred drop operations are in fact used in every database as a way to improve overall throughput because a thread does not need to wait for the drop to complete before proceeding with its next task. Like the other allocation optimizations that are available in all databases, the deferred drop probably provides the most benefit in *tempdb*, which is where tables are most likely to be dropped during production operations. A background thread eventually cleans up the space allocated for dropped tables, but until then, the allocated space remains. You can detect this space by looking at the *sys.allocation_units* system view for rows with a *type* value of 0, which indicates a dropped object; you will also see that the column called *container_id* is 0, which indicates that the allocated space does not really belong to any object. We'll look at *sys.allocation_units* and the other system views that keep track of space usage in Chapter 6.

Best Practices

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

To determine the optimum size of your *tempdb*, you must test your own applications with your data volumes, but knowing when and how *tempdb* is used can help you make preliminary estimates. Keep in mind that there is only one *tempdb* for each SQL Server instance, so one badly behaving application can affect all other users in all other applications. In Chapter 7, we'll look at estimating the size of tables and indexes, and we'll talk more about online index building and the *tempdb* space required for that operation. Finally, in Chapter 8, you'll see how to determine the size of the version store. All these factors affect the space needed for your *tempdb*.

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

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

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

tempdb Space Monitoring

Quite a few tools, stored procedures, and system views report on object space usage, as discussed in Chapter 6 and Chapter 7. However, one set of system views reports information only for *tempdb*. The simplest view is *sys.dm_db_file_space_usage*, which returns one row for each file in *tempdb*. It returns the following columns:

- database_id (even though the database ID 2 is the only one used)
- file_id
- unallocated_extent_page_count
- version_store_reserved_page_count
- user_object_reserved_page_count

- internal_object_reserved_page_count
- mixed_extent_page_count

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

Two other system views are similar to each other:

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

Database Security

Security is a huge topic that affects almost every action of every SQL Server user, including administrators and developers, and it deserves an entire book of its own. However, some areas of the SQL Server security framework are crucial to understanding how to work with a database or with any objects in a SQL Server database, and vast changes have been made in the security realm for SQL Server 2005, so I can't leave the topic completely untouched here.

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

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

In this section, I'll discuss the basic issues of database access and then describe the metadata where information on database access is stored. I'll also tell you about the new concept of schemas in SQL Server 2005 and describe how they are used to access objects.

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

- **Securable** Known as an *object* in SQL Server 2000, a *securable* is an entity on which permissions can be granted. Securables include databases, schemas, and objects.
- **Principal** Known as a *user* in SQL Server 2000, a principal is an entity that can access securable objects. A *primary principal* represents a single user (such as a SQL Server or a Windows login); a *secondary principal* represents multiple users (such as a role or a Windows group).

Database Access

Authentication is performed at two different levels in SQL Server. First, anyone who wants to access any SQL Server resource must be authenticated at the server level. SQL Server 2005 security provides two basic methods for authenticating logins: Windows Authentication and SQL Server Authentication. In Windows Authentication, SQL Server login security is integrated directly with Windows security, allowing the operating system to authenticate SQL Server users. In SQL Server Authentication, an administrator creates SQL Server login accounts within SQL Server, and any user connecting to SQL Server must supply a valid SQL Server login name and password.

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

Under Windows 2000 and Windows 2003, SQL Server can use Kerberos to support mutual authentication between the client and the server, as well as to pass a client's security credentials between computers so that work on a remote server can proceed using the credentials of the impersonated client. With Windows 2000 and Windows 2003, SQL Server uses Kerberos and delegation to support Windows Authentication as well as SQL Server Authentication.

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

One advantage of Windows Authentication has always been that it allows SQL Server to take advantage of the security features of the operating system, such as password encryption,

password aging, and minimum and maximum length restrictions on passwords. As of SQL Server 2005, when running on Windows 2003, SQL Server Authentication can also take advantage of the Windows password policies. Take a look at the ALTER LOGIN command in Books Online for the full details. Another change in SQL Server 2005 is that if you choose Windows Authentication during setup, the default SQL Server *sa* login will be disabled. If you switch to Mixed Mode after setup, you can enable the *sa* login using the ALTER LOGIN command. You can change the authentication mode in SQL Server Management Studio by right-clicking on the server name, choosing properties, and then selecting the security page. Under Server authentication, select the new server authentication mode, as shown in Figure 4-5.

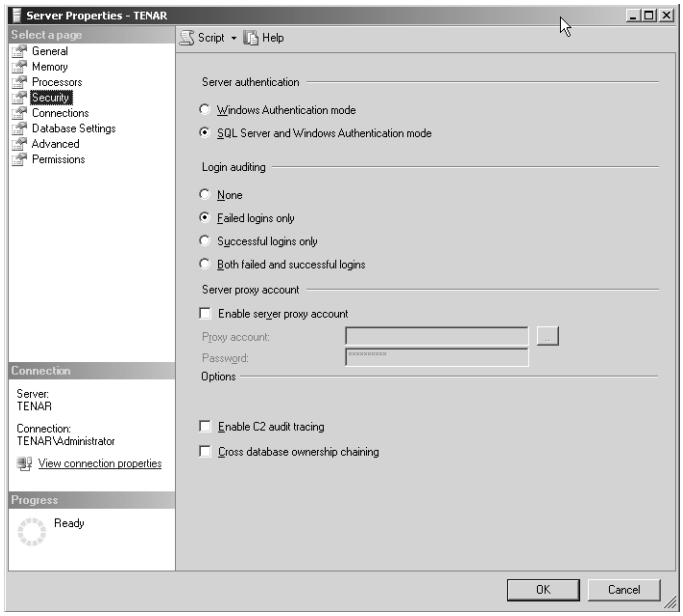

Figure 4-5 Choosing an authentication mode for your SQL Server instance in the Server Properties dialog box

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

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

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

Managing Database Security

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

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

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

In my *AdventureWorks* database, these are the results I get back:

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

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

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

Databases vs. Schemas

In the ANSI SQL-92 standard, a schema is defined as a collection of database objects that are owned by a single user and form a single namespace. A *namespace* is a set of objects that

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

Separation of Principals and Schemas

The previous version, SQL Server 2000, provides a CREATE SCHEMA statement, but it effectively does nothing because there is an implicit relationship between users and schemas that cannot be changed or removed. In fact, the relationship is so close that many users of SQL Server 2000 are unaware that users and schemas are different things. Every user is the owner of a schema that has the same name as the user. If you create a user *sue*, for example, SQL Server 2000 creates a schema called *sue*, which is *sue*'s default schema. Permissions are granted to users, but objects are placed in schemas.

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

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

This table is put in the schema *sue* because that is *sue*'s default schema. If another user wants to retrieve data from this table, he can issue this statement:

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

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

In the new version, SQL Server 2005 breaks apart the linking of users to schemas. Schemas can be owned by either primary or secondary principals. Although every object in a SQL Server 2005 database is owned by a user, we never reference an object by its owner; we reference it by the schema in which it is contained. In addition, a user is never added to a schema; schemas contain objects, not users. For backward compatibility, if you execute the *sp_adduser* or *sp_grantdbaccess* procedure to add a user to a database, SQL Server 2005 creates both a user and a schema, and it makes the schema the default schema for the new user. However, you should get used to using the new DDL CREATE USER and CREATE SCHEMA. When you create a user, you can optionally specify a default schema, but the default for the default schema is the *dbo* schema.

Default Schemas

When you create a new database in SQL Server 2005, several schemas are included in it. These include schemas that correspond to the default user names in SQL Server 2000: dbo, INFORMATION_SCHEMA, and guest. In addition, every database has a schema called sys,

which provides a way to access all the system tables and views. Finally, every predefined database role from SQL Server 2000 has a schema in SQL Server 2005.

Users can be assigned a default schema that might or might not exist when the user is created. A user can have at most one default schema at any time. As mentioned earlier, if no default schema is specified for a user, the default schema for the user is dbo. A user's default schema is used for name resolution during object creation or object reference. This can be both good news and bad news for backward compatibility. The good news is that if you've upgraded a database from SQL Server 2000, which has many objects in the dbo schema, your code can continue to reference those objects without having to specify the schema explicitly. The bad news is that for object creation, SQL Server will try to create the object in the dbo schema rather than in a schema owned by the user creating the table. The user might not have permission to create objects in the dbo schema, even if that is the user's default schema. To avoid confusion, in SQL Server 2005 you should always specify the schema name for all object access as well as object management.

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

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

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

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

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

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

Moving or Copying a Database

You might need to move a database before performing maintenance on your system, after a hardware failure, or when you replace your hardware with a newer, faster system. Copying a

database is a common way to create a secondary development or testing environment. You can move or copy a database by using a technique called "detach and attach" or by backing up the database and restoring it in the new location.

Detaching and Reattaching a Database

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

Here is the command to detach a database:

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

Once the database has been detached, from the perspective of SQL Server it's as if you had dropped the database. No trace of the database remains within the SQL Server instance. If you are planning to reattach the database later, it's a good idea to record the properties of all the files that were part of the database.

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

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

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

Note that only the primary file is required to have a <filespec> entry because the primary file contains information about the location of all the other files. If you'll be attaching existing files with a different path than when the database was first created or last attached, you must have

additional <filespec> entries. In any event, all the data files for the database must be available, whether or not they are specified in the CREATE DATABASE command. If there are multiple log files, they must all be available.

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

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

If your transaction log is rebuilt by attaching the database, using the FOR ATTACH_REBUILD _LOG will break the log backup chain. You should consider making a full backup after performing this operation.

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

Although the documentation says that you should use sp_attach_db or CREATE DATABASE FOR ATTACH only on databases that were previously detached using sp_detach_db, sometimes following this recommendation isn't necessary. If you shut down the SQL Server instance, the files will be closed, just as if you had detached the database. However, you will not be guaranteed that all dirty pages from the database were written to disk before the shutdown. This should not cause a problem when you attach such a database if the log file is available. The log file will have a record of all completed transactions, and a full recovery will be done when the database is attached to make sure the database is consistent. One benefit of using the sp_detach_db procedure is that SQL Server will know that the database was shut down cleanly, and the log file does not have to be available to attach the database. SQL Server will build a new log file for you. This can be a quick way to shrink a log file that has become much larger than you would like, because the new log file that sp_attach_db creates for you will be the minimum 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 in Chapter 5. Most of these issues involve the role of the transaction log in backup-and-restore operations.

Moving System Databases

You might need to move system databases as part of a planned relocation or scheduled maintenance operation. The steps for moving *tempdb*, *model*, and *msdb* are slightly different than for moving the *master* database or the resource database.

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

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

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

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

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

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

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

3. For each file in the database to be moved, use the ALTER DATABASE command with the MODIFY FILE option to specify the new physical location. You can use either SQL Server Management Studio or the SQLCMD utility.

4. Exit SQL Server Management Studio or the SQLCMD utility.
5. Stop the instance of SQL Server.
6. Physically move the file or files to the new location.
7. Restart the instance of SQL Server. For example, run NET START MSSQLSERVER.
8. Verify the change by running the following query:

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

Moving the *master* Database and Resource Database

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

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

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

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

```
-dE:\SQLData\master.mdf;-
eC:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\LOG\ERRORLOG;
-lE:\SQLData\mastlog.ldf;
-eE:\ SQLData\LOG\ERRORLOG;-
```

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

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

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

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

Compatibility Levels

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

All the examples and explanations in this book assume that you're using a database in 90 compatibility mode, unless otherwise noted. If you find that your SQL statements behave differently than the ones in the book, you should first verify that your database is in 90 compatibility mode by executing this procedure:

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

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

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

Note The compatibility-level options merely provide a transition period while you're upgrading a database or an application to SQL Server 2005. I strongly suggest that you try to change your applications so that compatibility options are not needed. Microsoft doesn't guarantee that these options will continue to work in future versions of SQL Server.

Not all changes in behavior from older versions of SQL Server can be duplicated by changing the compatibility level. For the most part, the differences have to do with whether new keywords and new syntax are recognized, and they do not affect how your queries are processed internally. For example, if you change to compatibility level 80, you don't make the system tables viewable or do away with schemas. But because the word PIVOT is a new reserved keyword in SQL Server 2005 (compatibility level 90), by setting your compatibility level to 80 you can create a table called PIVOT without using any special delimiters—or a table you already have in a SQL Server 2000 database will continue to be accessible if the database stays in 80 compatibility level.

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

Summary

A database is a collection of objects such as tables, views, and stored procedures. Although a typical SQL Server installation has many databases, it always includes the following three: *master*, *model*, and *tempdb*. (An installation usually also includes *msdb,* but that database can be removed.) Every database has its own transaction log; integrity constraints among objects keep a database logically consistent.

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