Chapter 5

Databases and Database Files

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

A SQL Server database:

- Is a collection of many objects, such as tables, views, stored procedures, and constraints. The theoretical limit is $2^{31} - 1$ (more than 2 billion) objects. Typically, the number of objects ranges from hundreds to tens of thousands.
- Is owned by a single user account but can contain objects owned by other users.
- Has its own set of system tables that catalog the definition of the database.
- Maintains its own set of user accounts and security.
- Is the primary unit of recovery and maintains logical consistency among objects in the database. (For example, primary and foreign key relationships always refer to other tables within the same database, not to other databases.)

- Has its own transaction log and manages the transactions within the database.
- Can participate in two-phase commit transactions with other SQL Server databases on the same server or different servers.
- Can span multiple disk drives and operating system files.
- Can range in size from 1 MB to a theoretical limit of 1,048,516 TB.
- Can grow and shrink, either automatically or by command.
- Can have objects joined in queries with objects from other databases in the same SQL Server installation or on linked servers.
- 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.)
- Is conceptually similar to but richer than the ANSI SQL-schema concept (discussed later in this chapter).

A SQL Server database is *not*:

- Synonymous with an entire SQL Server installation.
- A single SQL Server table.
- A specific operating system file.

A database itself isn't synonymous with an operating system file, but a database 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 afterwards, using the ALTER DATABASE command.

SPECIAL SYSTEM DATABASES

A new SQL Server 2000 installation automatically includes six databases: *master*, *model*, *tempdb*, *pubs*, *Northwind*, and *msdb*.

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, usage, systemwide configuration settings, login accounts, the existence of other databases, and the existence of other SQL servers (for distributed operations). The *master* database is absolutely critical to your system, so be sure to 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 activities, 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.

tempdb

The temporary database, *tempdb*, is a workspace. SQL Server's *tempdb* database is unique among all other 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 to hold intermediate results created internally by SQL Server during query processing and sorting, and for the materialization of static cursors and the keys of keyset cursors. Operations within *tempdb* are logged so that transactions on temporary tables 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. 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 only enough information for rolling back transactions in *tempdb* was a new feature in SQL Server 7 and can potentially increase the performance of INSERT statements to make them up to four times faster than inserts in other (fully logged) databases.

All users have the privileges to create and use private and global temporary tables that reside in *tempdb*. (Private and global table names have # and ## prefixes, respectively, which I'll discuss in more detail in Chapter 6.) 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.

pubs

The *pubs* database is a sample database used extensively by much of the SQL Server documentation and in this book. You can safely delete it if you like, but it consumes only 2 MB of space, so unless you're scrounging for a few more megabytes of disk space, I recommend leaving it. This database is admittedly fairly simple, but that's a feature, not a drawback. The *pubs* database provides good examples without a lot of peripheral issues to obscure the central points. Another nice feature of *pubs* is that it's available to everyone in the SQL Server community, which makes it easy to use to illustrate examples without requiring the audience to understand the underlying tables or install some new database to try out your examples. As you become more skilled with SQL Server, chances are you'll find yourself using this database in examples for developers or users.

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 a script in the \Install subdirectory (located right under the SQL Server installation directory). In SQL Query Analyzer, open the file named Instpubs.sql and execute it. You do need to make sure that 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 documentation dealing with APIs uses *Northwind*, as do some of the newer examples in the SQL Server documentation. It's a bit more complex than *pubs*, and at almost 4 MB, slightly larger. I'll be using it in this book to illustrate some concepts that aren't easily demonstrated using *pubs*. As with *pubs*, you can safely delete *Northwind* if you like, although the disk space it takes up is extremely small compared to what you'll be using for your real data. I recommend leaving *Northwind* there.

The *Northwind* database can be rebuilt just like the *pubs* database, by running a script located in the \Install subdirectory. The file is called Instnwnd.sql.

msdb

The *msdb* database is used by the SQL Server Agent service, which performs scheduled activities such as backups and replication tasks. In general, other than performing backups and maintenance on this database, you should 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 the SQL Server Enterprise Manager tools, so you usually don't need to access these tables directly. Think of the *msdb* tables as another form of system tables: just as you should 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.

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, to physical devices such as tape drives, or even to named pipes. 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 2000 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, the name of a primary data file has the extension MDF.
- **Secondary data files** A database can have zero or more secondary data files. By convention, the name of 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: a logical filename, a physical filename, an initial size, a maximum size, and a growth increment. The properties of each file, along with other information about the file, are noted in the *sysfiles* table (shown in Table 5-1), which contains one row for each file used by a database.

Column Name	*Description*
fileid	The file identification number (unique for each database).
groupid	The filegroup identification number.
size	The size of the file (in 8-KB pages).
maxsize	The maximum file size (in 8-KB pages). A value of 0 indicates no growth, and a value of –1 indicates that the file should grow until the disk is full.
growth	The growth size of the database. A value of 0 indicates no growth. Can be either the number of pages or a percentage of the file size, depending on the value of *status*.
status	0x1 = Default device (unused in SQL Server 2000). 0x2 = Disk file. 0x40 = Log device. 0x80 = File has been written to since last backup. 0x4000 = Device created implicitly by CREATE DATABASE 0x8000 = Device created during database creation. 0x100000 = Growth is in percentage, not pages.
name	The logical name of the file.
filename	The name of the physical device, including the full path of the file.

Table 5-1. *The* sysfiles *table.*

CREATING A DATABASE

The easiest way to create a database is to use SQL Server Enterprise Manager, which provides a graphical front end to Transact-SQL commands and stored procedures that actually create the database and set its properties. Figure 5-1 shows the SQL Server Enterprise Manager Database Properties dialog box, which represents the Transact-SQL CREATE DATABASE command for creating a new user database. Only someone with the *sysadmin* role or a user who's been granted CREATE DATABASE permission by someone with the *sysadmin* role can issue the CREATE DATABASE command.

When you create a new user database, SQL Server copies the *model* database, which—as you just learned—is simply a template database. For example, if you have an object that you would like created in every subsequent user database, create that object in *model* first. (You can also use *model* to set default database options in all subsequently created databases.) The *model* database also includes 19 system tables and 2 system views, which means that every new database also includes these 21 system objects. SQL Server uses these objects for the definition and maintenance of

each database. The two system views are provided for backward compatibility and have no current functionality. They mimic information that was available in earlier versions of SQL Server.

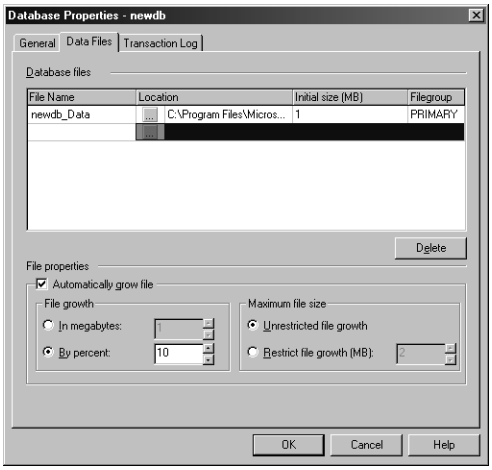

Figure 5-1. *The Database Properties dialog box, which creates a new database called* newdb.

The system objects have names starting with *sys*. If you haven't added any other objects to *model*, these 21 system objects will be the entire contents of a newly created database. Every Transact-SQL command or system stored procedure that creates, alters, or drops a SQL Server object will result in entries being made to system tables.

> **WARNING** Do not directly modify the system tables. You might render your database unusable by doing so. Direct modification is prevented by default: a system administrator must take deliberate action via the *sp_configure* stored procedure to allow system tables to be modified directly. I'll discuss system tables in more detail in Chapter 6.

A new user database must be 1 MB or greater in size, and the primary data file size must be at least as large as the primary data file of the *model* database. Almost all the possible arguments to the CREATE DATABASE command have default values so that 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).

> **NOTE** If you use Enterprise Manager to create a database called *newdb*, the default logical and physical names will be different than if you use the CREATE DATABASE command. Enterprise Manager will give the data file the logical name of *newdb_Data* (instead of just *newdb*), and the physical file will have the name newdb_data.mdf.

The SQL Server login account that created the database is known as the *database owner,* and has the user name DBO when using this database. 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 half a megabyte. 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, a log file is automatically created with a size that is 25 percent of the sum of the sizes of all data files.

For the files, if the MAXSIZE clause isn't specified, the file will grow until the disk is full. (In other words, the file size is considered unlimited.) The values supplied for SIZE, MAXSIZE, and FILEGROWTH can be specified in units of TB, GB, MB (the default), or KB. The FILEGROWTH property can also be specified as a percentage. A value of 0 for FILEGROWTH indicates no growth. If no FILEGROWTH value is specified at all, the default value is 10 percent, and the minimum value is 64 KB.

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\data\archdat1.mdf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20),
( NAME = Arch2,
FILENAME =
    'c:\program files\microsoft sql server\mssql\data\archdat2.ndf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20)
```

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

EXPANDING AND SHRINKING A DATABASE

Databases can be expanded and shrunk either 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, so I'll discuss changes in log file size in a separate section.

Automatic File Expansion

Expansion of a database 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 is always rounded up to the nearest 64 KB. If the value is specified as a percent, 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.

Manual File Expansion

Manual expansion of a database file is accomplished 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 files, you use the DBCC SHRINKFILE command, which I'll tell you about shortly.

Automatic File 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 server process ID (spid) 6—shrinks databases at 30-minute intervals. I'll discuss the DBCC SHRINKDATABASE command in more detail momentarily.

Manual File Shrinkage

You can manually shrink a database using the following two 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 *target_size* is specified, 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

The DBCC SHRINKDATABASE command shrinks all files in a database. The database can't be made smaller than the *model* database. In addition, the DBCC SHRINKDATABASE command does not allow any file to be shrunk to a size 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 this 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 that SQL Server can shrink this file to leave 25 percent free space. In that case, the file size remains unchanged.

Because DBCC SHRINKDATABASE shrinks the database on a file-by-file basis, the mechanism used to perform the actual shrinking 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:

- The NOTRUNCATE option causes all the freed file space to be retained in the database files. SQL Server only compacts the data by moving it to the front of the file. The default is to release the freed file space to the operating system.
- The TRUNCATEONLY 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.
- The EMPTYFILE 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 and DBCC SHRINKDATABASE commands 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 able to be shrunk to 256 pages of 8 KB each, which is 2 MB. But only 152 pages are used. There could 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. (The minimum size of a file is the smaller of the initial creation size and the size the file has been increased to using the ALTER DATABASE command.)
- 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 down.

CHANGES IN LOG SIZE

No matter how many physical files have been defined for the transaction log, SQL Server always treats the log as one contiguous file. For example, when the DBCC SHRINKDATABASE command determines how much the log can be shrunk, it does not consider the log files separately but determines the shrinkable size based on the entire log.

The transaction log for any database is managed as a set of virtual log files (VLFs) whose size is determined internally by SQL Server based on the total size of all the log files and the growth increment used when enlarging the log. A log always grows in units of entire VLFs and can be shrunk only to a VLF boundary. (Figure 5-2 illustrates a physical log file along with several VLFs.)

A VLF can exist in one of the following three states:

- **Active** The active portion of the log begins at the minimum log sequence number (LSN) representing an active (uncommitted) transaction. The active portion of the log ends at the last LSN written. Any VLFs that contain any part of the active log are considered active VLFs. (Unused space in the physical log is not part of any VLF.)
- **Recoverable** The portion of the log preceding the oldest active transaction is needed only to maintain a sequence of log backups for restoring the database to a former state.
- **Reusable** If transaction log backups are not being maintained or if you have already backed up the log, VLFs prior to the oldest active transaction are not needed and can be reused.

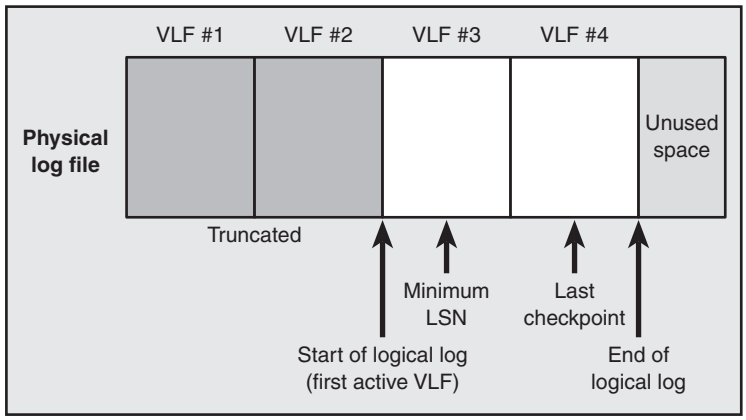

Figure 5-2. *The VLFs that make up a physical log file.*

SQL Server will assume you're not maintaining a sequence of log backups if any of the following are true:

- You have truncated the log using BACKUP LOG WITH NO_LOG or BACKUP LOG WITH TRUNCATE_ONLY.
- You have set the database to truncate the log automatically on a regular basis with the database option *trunc. log on chkpt.* or by setting the recovery mode to SIMPLE.
- You have never taken a full database backup.

In any of these situations, when SQL Server reaches the end of the physical log file, it starts reusing that space in the physical file by circling back to the file's beginning. In effect, SQL Server recycles the space in the log file that is no longer needed for recovery or backup purposes. If a log backup sequence *is* being maintained, the part of the log before the minimum LSN cannot be overwritten until those log records have actually been backed up. After the log backup, SQL Server can circle back to the beginning of the file. Once it has circled back to start writing log records earlier in the log file, the reusable portion of the log is then between the end of the logical log and the active portion of the log. Figure 5-3 depicts this cycle.

You can actually observe this behavior in one of the sample databases, such as *pubs*, as long as you have never made a full backup of the database. If you have never made any modifications to *pubs*, the size of its transaction log file will be just about 0.75 MB. The script on the following page creates a new table in the *pubs* database, inserts three records, and then updates those records 1000 times. Each update is an individual transaction, and each one is written to the transaction log. However, you should

note that the log does not grow at all, even after 3000 update records are written. (If you've already taken a backup of *pubs*, you might want to re-create the database before trying this example. You can do that by running the script *instpubs.sql* in the folder \Program Files\Microsoft SQL Server\MSSQL\install. If your SQL Server is a named instance, you'll need to replace MSSQL with the string MSSQL$<instance_name>.)

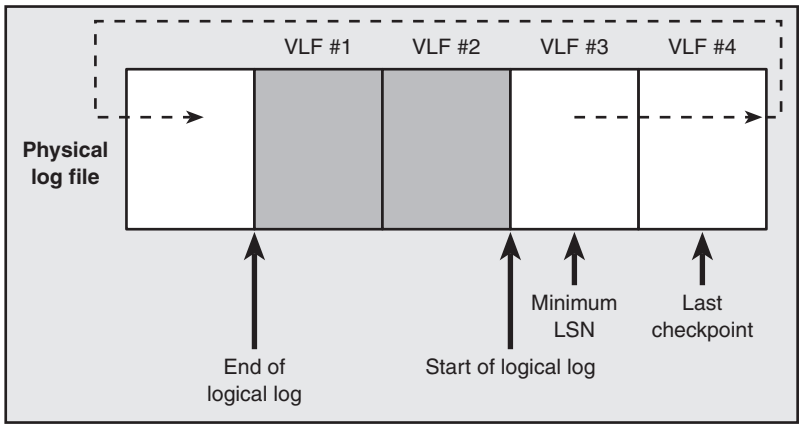

Figure 5-3. *The active portion of the log circling back to the beginning of the physical log file.*

```
CREATE TABLE newtable (a int)
GO
INSERT INTO newtable VALUES (10)
INSERT INTO newtable VALUES (20)
INSERT INTO newtable VALUES (30)
GO
DECLARE @counter int
SET @counter = 1
WHILE @counter < 1000 BEGIN
    UPDATE newtable SET a = a + 1
    SET @counter = @counter + 1
END
```

Now make a backup of the *pubs* database after making sure that the database is not in the SIMPLE recovery mode. I'll discuss recovery modes later in this chapter, but for now, you can just make sure that *pubs* is in the appropriate recovery mode by executing the following command:

```
ALTER DATABASE pubs SET RECOVERY FULL
```

You can use the following statement to make the backup, substituting the path shown with the path to your SQL Server installation:

```
BACKUP DATABASE pubs to disk =
    'c:\Program Files\Microsoft SQL Server\MSSQL\backup\pubs.bak'
```

Run the update script again, starting with the DECLARE statement. You should see that the physical log file has grown to accommodate the log records added. The initial space in the log could not be reused because SQL Server assumed that you were saving that information for backups.

Now you can try to shrink the log back down again. If you issue the following command or if you issue the DBCC SHRINKFILE command for the log file, SQL Server will mark a shrinkpoint in the log but no actual shrinking will take place until log records are freed by either backing up or truncating the log.

```
DBCC SHRINKDATABASE (pubs)
```

You can truncate the log with this statement:

```
BACKUP LOG pubs WITH TRUNCATE_ONLY
```

At this point, you should notice that the physical size of the log file has been reduced. If a log is truncated without any prior shrink command issued, SQL Server marks the space used by the truncated records as available for reuse but it does not change the size of the physical file.

In the previous version of SQL Server, running the preceding commands exactly as specified did not always shrink the physical log file. The cases in which the log file did not shrink happened when the active part of the log was located at the end of the physical file. Physical shrinking can take place only from the end of the log, and the active portion is never shrinkable. To remedy this situation, you had to enter some "dummy" transactions after truncating the log, to force the active part of the log to move around to the beginning of the file. In SQL Server 2000, this process is unnecessary. If a shrink command has already been issued, truncating the log internally generates a series of NO-OP log records that force the active log to move from the physical end of the file. Shrinking happens as soon as the log is no longer needed.

Log Truncation

If a database log backup sequence is not being maintained for a database, you can set the database into log truncate mode by setting the recovery mode to SIMPLE or by using the older mechanism of setting the database option *trunc. log on chkpt.* to TRUE. The log is thus truncated every time it gets "full enough." (I'll explain this in a moment.)

Truncation means that all log records prior to the oldest active transaction are removed. It does not necessarily imply shrinking of the physical log file. In addition, if your database is a publisher in a replication scenario, the oldest open transaction could be a transaction marked for replication that has not yet been replicated.

"Full enough" means that there are more log records than can be redone during system startup in a reasonable amount of time, which is referred to as the "recovery

interval." You can manually change the recovery interval by using the *sp_configure* stored procedure or by using SQL Server Enterprise Manager (right-click the server, select Properties, click the Database Settings tab, and set the value in the Recovery Interval box), but it is recommend that you let SQL Server autotune this value. In most cases, this recovery interval value is set to 1 minute. (SQL Server Enterprise Manager shows zero minutes by default, meaning SQL Server will autotune the value.) SQL Server bases its recovery interval on the estimation that 10 MB worth of transactions can be recovered in 1 minute.

The actual log truncation is invoked by the checkpoint process, which is usually sleeping and is only woken up on demand. Each time a user thread calls the log manager, the log manager checks the size of the log. If the size exceeds the amount of work that can be recovered during the recovery interval, the checkpoint thread is woken up. The checkpoint thread checkpoints the database and then truncates the inactive portion.

In addition, if the log ever gets to 70 percent full, the log manager wakes up the checkpoint thread to force a checkpoint. Growing the log is much more expensive than truncating it, so SQL Server truncates the log whenever it can.

> **NOTE** If the log is in auto truncate mode and the autoshrink option is set, the log will be physically shrunk at regular intervals.

If a database has the autoshrink option on, an autoshrink process kicks in every 30 minutes and determines the size to which the log should be shrunk. The log manager accumulates statistics on the maximum amount of log space used in the 30-minute interval between shrinks. The autoshrink process marks the shrinkpoint of the log as 125 percent of the maximum log space used or the minimum size of the log, whichever is larger. (Minimum size is the creation size of the log or the size to which it has been manually increased or decreased.) The log then shrinks to that size whenever it gets the chance, which is when the log gets truncated or backed up. It's possible to have autoshrink without having the database in auto truncate mode, although there's no way to guarantee that the log will actually shrink. For example, if the log is never backed up, it will never be cleared.

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* 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 its header contains information about all the other files in 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 and 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 accomplish this is to create a database file on a RAID device. Still, there would be no need to use filegroups. At the next level up the sophistication and complexity scale, 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 (DBAs) might decide that they want to have different tables assigned to different drives. Only then will they need to use filegroups. The easiest way to accomplish this goal is to use SQL Server Enterprise Manager to create the database. SQL Server Enterprise Manager will create the necessary filegroups, and the user still doesn't have to learn anything about filegroup syntax. Only the most sophisticated users who want to write scripts that set up databases with multiple filegroups will need to know the underlying details.

WHY USE MULTIPLE FILES?

You might wonder what the reason would be for creating a database on multiple files located on one physical drive. There's 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. Microsoft's internal SAP system uses a SQL Server database created on 12 files. Microsoft has found that this provides the ultimate flexibility. They could separate the files into two groups of six, six groups of two, four groups of three, and so on, which would allow them to experiment with performance enhancements gained as files are spread over different numbers of physical drives.

You can also use filegroups to allow backups of only parts of the database at one time. However, if you create an index in a filegroup that's different from the filegroup the table resides in, you must back up both filegroups (the filegroup containing the table and the filegroup containing the index). If you create more than one index in a filegroup that's different from the filegroups in which the corresponding tables reside, you must immediately back up all filegroups to accommodate these differing filegroups. The BACKUP statement detects all these filegroup situations and communicates to the user the minimum filegroups that must be backed up.

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.

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

(continued)

```
( NAME = SGrp2Fi2_dat,
FILENAME =
    'c:\program files\microsoft sql server\mssql\data\SG2Fi2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
FILENAME =
    'c:\program files\microsoft sql server\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, which you can optionally put in a user-defined filegroup. You must put all files added in a single ALTER DATABASE statement 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.
- Modify an existing file in one of the following ways:
  - Increase the value of the SIZE property.
  - Change the MAXSIZE or FILEGROWTH properties.
  - Change the name of a file by specifying a NEWNAME property. The value given for NEWNAME is then used as the NAME property for all future references to this file.
  - Change the FILENAME property for files only in the *tempdb* database; this change doesn't go into effect until you stop and restart SQL Server. You can change the FILENAME in order to move the *tempdb* files to a new physical location.

- 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 statement ALTER DATABASE.

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\data\t1dat3.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB),
```

(continued)

```
( NAME = 'test1dat4',
FILENAME =
    'c:\program files\microsoft sql server\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 upper 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 either the DBCC SHRINKDATABASE or DBCC SHRINKFILE command, pages are removed starting at the highest page in the database (at the end) and moving toward lower-numbered pages. This ensures that page numbers within a file are always contiguous.

The *master* database contains 50 system tables: the same 19 tables found in user databases, plus 31 tables that are found only in *master*. Many of these tables—including *syslockinfo*, *sysperfinfo*, *syscurconfigs*, and *sysprocesses*—don't physically exist in the *master* database; rather, they're built dynamically each time a user queries them. Twelve of these tables contain information about remote servers; they don't contain any data until you define other servers with which you'll be communicating. The *master* database also contains 26 system views. Twenty of these are INFORMATION_SCHEMA views; I'll discuss them in Chapter 6. Except for the INFORMATION_SCHEMA views, all these system objects, in both the *master* and *user* databases, have names that begin with *sys*. To see a list of the system tables and views that are found in only the *master* database, you can execute the following query:

```
SELECT type, name FROM master..sysobjects
WHERE type IN ('s', 'v') AND name NOT IN
    (SELECT name FROM model..sysobjects)
GO
```

The *master* database also contains nine tables that can be referred to as *pseudo-system tables*. These table names begin with *spt_* and are used as lookup tables by various system procedures, but they aren't true system tables. You should never modify them directly because that might break some of the system procedures. However, deleting them doesn't invalidate the basic integrity of the database, which is what happens if true system tables are altered.

When you create a new database using the CREATE DATABASE statement, it is given a unique database ID, or *dbid,* and a new row is inserted in the table *master..sysdatabases* for that database. Only the *master* database contains a *sysdatabases* table. Figure 5-4 on the following page depicts a sample *sysdatabases* table, and Table 5-2 shows its columns.

The rows in *sysdatabases* are updated when a database's ownership or name is changed or when database options are changed. (I'll discuss this in more detail later in the chapter.)

Column	*Information*
name	Name of the database.
dbid	Unique database ID; can be reused when the database is dropped.
sid	System ID of the database creator.
mode	Locking mode; used internally while a database is being created.
status	Bit mask that shows whether a database is read-only, off line, designated for use by a single user only, and so on. Some of the bits can be set by a database owner using the ALTER DATABASE command; others are set internally. (The SQL Server documentation shows most of the possible bit-mask values.)
status2	Another bit mask–like status, with bits indicating additional database options.
crdate	For user databases, the date when the database was created. For *tempdb*, this is the date and time that SQL Server was last started. For other system databases, this date is not really useful. Depending on decisions made during installation, it could be the date Microsoft originally created the database prior to shipping the code, or it could be the date that you installed SQL Server.
reserved	Reserved for future use.
category	Another bit mask–like status. Contains information about whether the database is involved with replication.
cmptlevel	Compatibility level for the database. (I'll discuss this concept briefly at the end of the chapter.)
filename	Operating system path and name of the primary file.
version	Internal version of SQL Server that was used to create the database.

Table 5-2. *Columns of the* sysdatabases *table.*

```
name       dbid   sid   status     crdate                  cmptlevel filename
---------- ------ ----- ---------- ----------------------- --------- -------------------
master     1      0x01  24         2000-04-18 01:51:58.363 80        C:\...\master.mdf
model      3      0x01  1073741840 2000-04-18 02:03:11.240 80        C:\...\model.mdf
msdb       4      0x01  24         2000-04-18 02:03:15.613 80        C:\...\msdbdata.mdf
newdb      7      0x01  16         2000-05-10 13:26:42.013 80        C:\...\newdb.mdf
Northwind  6      0x01  28         2000-04-18 02:03:19.113 80        C:\...\northwind.mdf
pubs       5      0x01  24         2000-05-29 13:18:54.380 80        C:\...\pubs.mdf
tempdb     2      0x01  8          2000-05-28 10:53:39.620 80        C:\...\tempdb.mdf
```

Figure 5-4. *A partial listing of a* sysdatabases *table.*

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 2000 doesn't allocate entire extents to tables with small amounts of data. SQL Server 2000 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. Since there are almost 8000 bytes, or 64,000 bits, available on the page after the header and other overhead are accounted for, 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 is being used as 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 5-3 shows the bit patterns that each extent has set in the GAM and SGAM based on its current use.

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

Table 5-3. *Bit settings in GAM and SGAM pages.*

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.

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 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 map the extents in a database file used by a heap or index. Recall from Chapter 3 that a heap is a table without a clustered index. Each heap or index has one or more IAM pages that record all the extents allocated to the object. A heap or index has at least one IAM for each file on which it has extents. A heap or index can have more than one IAM on a file if the range of the extents exceeds the range that an IAM can record.

An IAM contains a small header, eight page-pointer slots, and a set of bits that map a range of extents onto a file. 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 objects that are 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. *Sysindexes.FirstIAM* points to the first IAM page for an object. All the IAM pages for that object are linked in a chain.

> **NOTE** In a heap, the data pages and the rows within them aren't in any specific order and aren't linked together. The only logical connection between data pages is recorded in the IAM pages.

Once extents have been allocated to an object, SQL Server can use pages in those extents to insert new data. If the data is to be inserted into a B-tree, the location of the new data is based on the ordering within the B-tree. If the data is to be inserted into a heap, the data can be inserted into any available space. PFS pages within a file record whether an individual page has been allocated and the amount of space free on each page. Each PFS page covers 8088 contiguous pages (almost 64 MB). For each page, the PFS has 1 byte recording whether the page is empty, 1–50 percent full, 51–80 percent full, 81–95 percent full, or 96–100 percent full. SQL Server uses this information when it needs to find a page with free space available to hold a newly inserted row. The second page (page 1) of a file is a PFS page, as is every 8088th page thereafter.

There are also two other kinds of special pages within a data file. The seventh page (page 6) is called a DCM (Differential Changed Map) page and keeps track of which extents in a file have been modified since the last full database backup. The eighth page (page 7) of a file is called a BCM (Bulk Changed Map) 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 in the "Backing Up and Restoring a Database" section later in this chapter. Like GAM and SGAM pages, DCM and BCM pages have one bit for each extent in the section of the file they represent. They occur at regular intervals, every 511,230 pages.

SETTING DATABASE OPTIONS

Twenty options can be set for a database to control certain behavior within that database. Most of the options must be set to ON or OFF. By default, all the options that allow only these two values have an initial value of OFF unless they were 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 Enterprise Manager. 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, but that procedure is provided for backward compatibility only. All the options correspond to bits in the *status* and *status2* columns of *sysdatabases*, although those bits can also show states that the database owner can't set directly (such as when the database is in the process of being recovered).

Executing the *sp_helpdb* stored procedure for a database shows you all the values for the non-Boolean options. For the Boolean options, the procedure lists the options that are ON. The option values are all listed in the status column of the output. In addition, the status column of the *sp_helpdb* output provides the database collation and sort order. The procedure also returns other useful information, such as database size, creation date, and database owner. Executing *sp_helpdb* with no parameters shows information about all the databases in that installation. The following databases exist on a new default installation of SQL Server, and *sp_helpdb* produces this output (although the created dates and sizes can vary):

```
> EXEC sp_helpdb
name      db_size  owner dbid created     status
--------- -------- ----- ---- ----------- ---------------------------------
master    11.94 MB sa    1    Jul 31 2000 Status=ONLINE, Updateability=R...
model      1.13 MB sa    3    Jul 31 2000 Status=ONLINE, Updateability=R...
msdb      13.00 MB sa    4    Jul 31 2000 Status=ONLINE, Updateability=R...
Northwind  3.63 MB sa    6    Jul 31 2000 Status=ONLINE, Updateability=R...
pubs       2.00 MB sa    5    Jul 31 2000 Status=ONLINE, Updateability=R...
tempdb     8.50 MB sa    2    Aug 21 2000 Status=ONLINE, Updateability=R...
```

The database options are divided into five categories: state options, cursor options, auto option, SQL options, and recovery options. Some of the options, in particular 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 a number of these SET options on by default, so applications will act as though the corresponding database option has already been set. (Chapter 6 goes into more detail about the SET options that are set by the ODBC and OLE DB drivers and the interaction with the database options.)

Here is a list of all 20 options, by category. Options listed on a single line are mutually exclusive.

- State options
  SINGLE_USER | RESTRICTED_USER | MULTI_USER
  OFFLINE | ONLINE
  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 }
- 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 }
- Recovery options
  RECOVERY { FULL | BULK_LOGGED | SIMPLE }
  TORN_PAGE_DETECTION { ON | OFF }

The following sections describe all the options except RECOVERY, which I'll discuss in detail later, in the section titled "Backing Up and Restoring a Database."

State Options

The state options control the usability of the database, in terms of who can use the database and for what operations. There are three aspects to usability: the user access state option determines which users can use the database, the status state option determines whether the database is available to anybody for use, and the updateability state option 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 *pubs* database to single user mode, you use the following code:

```
ALTER DATABASE pubs 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." A qualified user is any member of the dbcreator or sysadmin server roles or any member of 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—SQL Server's behavior is determined by the TERMINATION option you specify. I'll discuss termination options shortly.

To determine which user access value is set for a database, you can use the DATABASEPROPERTYEX() function, as shown here:

```
SELECT DATABASEPROPERTYEX('<name of database>', 'UserAccess')
```

In previous versions of SQL Server, database access was controlled using the procedure *sp_dboption* and setting the value of the options *dbo use only* or *single user*. If both of these options had a value of *false*, the database was in MULTI_USER mode. If the option *single user* was set to *true*, the database was in SINGLE_USER mode, and if the option *dbo use only* was set to *true* and *single user* was set to *false*, the database was in a state similar to RESTRICTED_USER. In SQL Server 2000, the user access value is still determined by the same bits in the *sysdatabases.status* field that were used in previous versions to determine *single user* or *dbo use only* mode. If the 12 bit is set (with a value of 4096), the database is in SINGLE_USER mode, and if the 11 bit is set (with a value of 2048), the database is in RESTRICTED_USER mode. In SQL Server 2000, you cannot have both of these bits set at the same time.

OFFLINE | ONLINE

These two options are the status options; the status property of a database is described using one of these option names. They are mutually exclusive. The default is for a database to be 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, the database is closed and shut down cleanly and marked as off line. The database cannot be modified while the database is off line. 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 pubs SET OFFLINE
SELECT DATABASEPROPERTYEX('pubs', 'status')
```

DATABASEPROPERTYEX() could return status values other than OFFLINE and ONLINE, 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. It can have a status of SUSPECT if the recovery process could not be completed for some reason—the most common ones being that one or more of the log files for the database is unavailable or unreadable. Each of the five status values corresponds to a different bit set in the *sysdatabases.status* column.

READ_ONLY | READ_WRITE

These two options are the updateability options; they describe the updateability property of a database. They are mutually exclusive. The default is for a database to be 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 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 examples show how to set a database's updatability value to READ_ONLY and how to determine the updatability of a database:

```
ALTER DATABASE pubs SET READ_ONLY
SELECT DATABASEPROPERTYEX('pubs', 'updateability')
```

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 specify that SQL Server wait for the situation to change, that it generate an error message, or that it terminate the connections of nonqualified users. The termination option determines SQL Server's behavior 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

SQL Server's default behavior 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, unqualified connections are all connections except the one issuing the ALTER DATABASE statement. When the transition is to RESTRICTED_USER mode, unqualified connections are connections 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. Unqualified connections are the same as those described earlier.

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 *pubs* database to SINGLE_USER and generates an error if any other connections to the *pubs* database exist:

```
ALTER DATABASE pubs SET SINGLE_USER WITH NO_WAIT
```

Cursor Options

All of 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. In Chapter 13, I'll discuss Transact-SQL cursors in detail.

CURSOR_CLOSE_ON_COMMIT {ON | OFF}

When this option is set to ON, any open cursors are automatically 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 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 created, the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection.

Auto Options

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

AUTO_CLOSE

When this option is set to ON (the default when SQL Server runs on Windows 98), the database is closed and shut down cleanly when the last user of the database exits, thereby freeing any resources. When a user tries to use the database again, it automatically 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_CREATE_STATISTICS

When this option is set to ON (the default), statistics are automatically created by the SQL Server query optimizer 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_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_UPDATE_STATISTICS

When this option is set to ON (the default), existing statistics are automatically updated if the data in the tables has changed. SQL Server keeps a counter of the modifications that have been 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 much more detail in Chapter 15.

SQL Options

The SQL options control how various SQL statements are interpreted. All of these are Boolean options. Although the default for all of these options is OFF for SQL Server itself, many tools, such as the SQL Query Analyzer, 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. I'll discuss the interaction of the SET options with the database options in Chapter 7.

ANSI_NULL_DEFAULT

When this option is set to ON, columns comply with the ANSI-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 to 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.

Recovery Options

Recovery options control how much recovery can be done on a SQL Server database. The RECOVERY option itself 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 the section titled "Backing Up and Restoring a Database" later in this chapter.

TORN_PAGE_DETECTION

When this option is set to ON (the default), 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, 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. If the first sector of a database page is successfully written before the crash, it will appear that the database page on disk was updated even though the operation might not have succeeded. Using battery-backed disk caches can ensure that the data is successfully written to disk or not written at all. In this case, you can set TORN_PAGE_DETECTION to OFF because it isn't needed. If a torn page is detected, the database must be restored from backup because it will be physically inconsistent.

Other Database Considerations

Keep these additional points in mind about databases in SQL Server.

Databases vs. Schemas

The ANSI SQL-92 standard includes the notion of a *schema*—or, more precisely, an *SQL-schema,* which in many ways is similar to SQL Server's database concept. Per the ANSI standard, an SQL-schema is a collection of *descriptors,* each of which is described in the documentation as "a coded description of an SQL object." Basically, a schema is a collection of SQL objects, such as tables, views, and constraints. ANSI SQL-schemas are similar to SQL Server databases.

SQL Server 6.5 introduced support for the ANSI SQL-schema. However, the concept of a database in SQL Server is longstanding and much richer than that of a schema. SQL Server provides more extensive facilities for working with a database

than for working with a schema. SQL Server includes commands, stored procedures, and powerful tools such as SQL Server Enterprise Manager that are designed around the fundamental concept of a database. These tools control backup, restoring, security, enumeration of objects, and configuration; counterparts of these tools don't exist for schemas. The SQL Server implementation of a schema is essentially a check box feature that provides conformance with the ANSI standard; it's not the preferred choice. Generally speaking, you should use databases, not schemas.

Using Removable Media

After you've created a database, you can package it so that you can distribute it via removable media such as CD. This can be useful for distributing large datasets. For example, you might want to put a detailed sales history database on a CD and send a copy to each of your branch offices. Typically, such a database is read-only (because CDs are read-only), although this isn't mandatory.

To create a removable media database, you create the database using the stored procedure *sp_create_removable* instead of the CREATE DATABASE statement. When calling the procedure, you must specify three or more files (one for the system catalog tables, one for the transaction log, and one or more for the user data tables). You must have a separate file for the system tables because when the removable media database is distributed and installed, the system tables will be installed to a writable device so that users can be added, permissions can be granted, and so on. The data itself is likely to remain on the read-only device.

Because removable media devices such as CDs are typically slower than hard drives, you can distribute on removable media a database that will be moved to a hard disk. If you're using a writable removable device, such as an optical drive, be sure that the device and controller are both on the Hardware Compatibility List (HCL). (You can find the HCL at www.microsoft.com/hcl.) I also recommend that you run the hard-disk test discussed in Chapter 4 on any such device. The failure rates of removable media devices are typically higher than those of standard hard disks.

A database can use multiple CDs or removable media devices. However, all media must be available simultaneously. For example, if a database uses three CDs, the system must have three CD drives so that all discs can be available when the database is used.

You can use the *sp_certify_removable* stored procedure to ensure that a database created with the intention of being burned onto a CD or other removable media meets certain restrictions. The main restriction is that the login sa must own the database and all the objects must be owned by the user dbo. There can be no users in the database other than dbo and guest. You can, however, have roles defined in the database, and permissions can be assigned to those roles. The stored procedure *sp_certify_removable* ensures that the database was created properly with the system tables separate from any user tables.

The first time you use a database sent on removable media, you use the *sp_attach_db* stored procedure to see the location of each file. You'll probably want to move the file containing the system tables to a writable disk so that you can create users, stored procedures, and additional permissions. You can keep the data on the removable media if you won't be modifying it. You can subsequently set the OFFLINE database option using ALTER DATABASE to toggle the database's availability. This book's companion CD contains a sample script that creates a database, ensures that it's appropriate for removable media use, and then installs it on your system. However, a database with no tables or data is pretty useless, so in the next chapter you'll learn how to create tables.

Detaching and Reattaching a Database

The ability to detach and reattach databases offers much broader benefits than just allowing the creation of removable databases. You can use the procedures *sp_detach_db* and *sp_attach_db* to move a database to a new physical drive—for example, if you're upgrading your hardware. You can use these stored procedures to make a copy of the database for testing or development purposes or as an alternative to the backup and restore commands.

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 automatically breaks existing connections. Detaching a database ensures that there are no incomplete transactions 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 *sysdatabases* table in the *master* database, and from SQL Server's perspective, it's as if you had dropped the database. The command to detach a database is shown here:

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

> **NOTE** You can also drop the database with the DROP DATABASE command, but using this command is a little more severe. SQL Server makes sure that no one is connected to the database before dropping it, but no check of dirty pages or open transactions is made. Dropping a database also removes the physical files from the operating system, so unless you have a backup, the database is really gone.

The files for a detached database still exist, but the operating system considers them closed files, so they can be copied, moved, or even deleted like any other operating system files. A database that has been detached can be reattached using the stored procedure *sp_attach_db*. This procedure has a few more options than its detaching counterpart. If all the files still exist in their original locations, which would

be the case if you detached the database just so you could copy the files to another server, all you need to specify is the location of the primary file. Remember that the primary file's header information contains the location of all the files belonging to the database. In fact, if some of the files exist in the original locations and only some of them have moved, you must specify only the moved files' locations when you execute the *sp_attach_db* procedure.

Although the documentation says that you should use *sp_attach_db* 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, 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 that the database is consistent. One benefit of using the *sp_detach_db* procedure is that SQL Server will know that the database was cleanly shut down, and the log file does not have to be available to attach the database. SQL 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. Note that this trick for shrinking the log will not work if the database has more than one log file.

Here is the syntax for the *sp_attach_db* procedure:

```
sp_attach_db [ @dbname = ] 'dbname' ,
 [ @filename1 = ] 'filename_n'
 [ ,...16 ]
```

Note that all you need to specify is the current filenames, regardless of whether the current names are the same as the original names. SQL Server will find the specified files and use them when attaching the database. You can even supply a new database name as you're attaching the files. You are limited to specifying up to 16 files for the database. Remember that you have to specify the filenames only if they are not in the original location stored in the header of the primary file. If you have a database for which you must specify more than 16 files, you can use the CREATE DATABASE command and specify the FOR ATTACH option.

Compatibility Levels

SQL Server 7 included a tremendous amount of new functionality and changed certain behaviors that existed in earlier versions. SQL Server 2000 has added even more new features. To provide the most complete level of backward compatibility, Microsoft allows you to set the compatibility level of a database to one of four modes: 80, 70,

65, or 60. All newly created databases in SQL Server 2000 have a compatibility level of 80 unless you change the level for the *model* database. A database that has been upgraded (using the Upgrade Wizard) from SQL Server version 6.0 or 6.5 will have its compatibility level set to the SQL Server version under which the database was last used (either 60 or 65). If you upgrade a server for SQL Server 7 to SQL Server 2000, all the databases on that server will have their compatibility level set to 80, although you can force SQL Server 2000 to behave like SQL Server 7 by setting this level to 70.

All the examples and explanations in this book assume that you're using a database that's in 80 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 80 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 three modes:

```
EXEC sp_dbcmptlevel 'database name', compatibility-mode
```

> **NOTE** 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 have no effect on how your queries are processed internally. For a complete list of the behavioral differences between the four modes, see the online documentation for the *sp_dbcmptlevel* procedure.

The compatibility-level options merely provide a transition period while you're upgrading a database or an application to SQL Server 2000. I strongly suggest that you carefully consider your use of this option and make every effort to change your applications so that compatibility options are no longer needed. Microsoft doesn't guarantee that these options will continue to work in future versions of SQL Server.

BACKING UP AND RESTORING A DATABASE

As you're probably aware by now, this book is not intended as a how-to book for database administrators. The Bibliography lists several excellent books that can teach you the mechanics of actually making database backups and restoring and can suggest best practices for setting up a backup-and-restore plan for your organization. Nevertheless, I'd like to discuss some important issues relating to backup and restore processes to help you understand why one backup plan might be better suited to your needs than another.

Types of Backups

No matter how much fault tolerance you have implemented on your database system, it is no replacement for regular backups. Backups can provide a solution to accidental or malicious data modifications, programming errors, and natural disasters (if you store backups in a remote location). If you choose to provide the fastest possible speed for your data files at the cost of fault tolerance, backups provide insurance in case your data files are damaged.

The process of re-creating a database from backups is called restoring. The degree to which you can restore the lost data depends on the type of backup. There are three main types of backups in SQL Server 2000, and a couple of additional variations on those types:

- **Full backup** A full database backup basically copies all the pages from a database onto a backup device, which can be a local or network disk file, a local tape drive, or even a named pipe.
- **Differential backup** A differential backup copies only the extents that were changed since the last full backup was made. The extents are copied onto a specified backup device. SQL Server can quickly tell which extents need to be backed up by examining the bits on the DCM pages for each data file in the database. Each time a full backup is made, all the bits are cleared to 0. When any page in an extent is changed, its corresponding bit in the DCM page is changed to 1.
- **Log backup** In most cases, a log backup copies all the log records that have been written to the transaction log since the last full or log backup was made. However, the exact behavior of the BACKUP LOG command depends on your database's recovery mode setting. I'll discuss recovery modes shortly.

> **NOTE** For full details on the mechanics of defining backup devices, making backups, or scheduling backups to occur at regular intervals, consult SQL Server Books Online or one of the SQL Server administration books listed in the Bibliography.

A full backup can be made while your SQL Server is in use. This is considered a "fuzzy" backup—that is, it is not an exact image of the state of the database at any particular point in time. The backup threads just copy extents, and if other processes need to make changes to those extents while the backup is in progress, they can do so.

To maintain consistency for either a full or a differential backup, SQL Server records the current log sequence number (LSN) at the time the backup starts and then again at the time the backup ends. This allows the backup to also capture the

relevant parts of the log. The relevant part starts with the oldest open transaction at the time of the first recorded LSN and ends with the second recorded LSN.

As mentioned previously, what gets recorded with a log backup depends on the recovery model you are using. So before I talk about log backup in detail, I'll tell you about recovery models.

Recovery Models

As I told you in the section on database options, three values can be set for the RECOVERY option: FULL, BULK_LOGGED, or SIMPLE. The value you choose determines the speed and size of your transaction log backups as well as the degree to which you are at risk for loss of committed transactions in case of media failure.

FULL Recovery Model

The FULL recovery model provides the least risk of losing work in the case of a damaged data file. If a database is in this mode, all operations are fully logged, which means that in addition to logging every row added with the INSERT operation, removed with the DELETE operation, or changed with the UPDATE operation, SQL Server also writes to the transaction log in its entirety every row inserted using a *bcp* or BULK INSERT operation. If you experience a media failure for a database file and need to recover a database that was in FULL recovery mode, and you've been making regular transaction log backups preceded by a full database backup, you can restore to any specified point in time up to the time of the last log backup. In addition, if your log file is available after the failure of a data file, you can restore up to the last transaction committed before the failure. SQL Server 2000 also supports a feature called log marks, which allows you to place reference points in the transaction log. If your database is in FULL recovery mode, you can choose to recover to one of these log marks. I'll talk a bit more about log marks in Chapter 12.

In FULL recovery mode, SQL Server will also fully log CREATE INDEX operations. In SQL Server 2000, when you restore from a transaction log backup that includes index creations, the recovery operation is much faster because the index does not have to be rebuilt—all the index pages have been captured as part of the database backup. In previous versions, SQL Server logged only the fact that an index had been built, so when you restored from a log backup, the entire index would have to be built all over again!

So, FULL recovery mode sounds great, right? As always, there's a tradeoff. The biggest tradeoff is that the size of your transaction log files can be enormous, and therefore it can take substantially longer to make log backups than with any previous release.

BULK_LOGGED Recovery Model

The BULK_LOGGED recovery model allows you to completely restore a database in case of media failure and also gives you the best performance and least log space

usage for certain bulk operations. These bulk operations include BULK INSERT, *bcp*, CREATE INDEX, SELECT INTO, WRITETEXT, and UPDATETEXT. In FULL recovery mode, these operations are fully logged, but in BULK_LOGGED recovery mode, they are only minimally logged.

When you execute one of these bulk operations, SQL Server logs only the fact that the operation occurred. However, the operation is fully recoverable because SQL Server keeps track of what extents were actually modified by the bulk operation. Every data file in a SQL Server 2000 database now has an additional allocation page called a BCM page, which is managed much like the GAM and SGAM pages that I discussed earlier in the chapter. Each bit on a BCM page represents an extent, and if the bit is 1 it means that this extent has been changed by a minimally logged bulk operation since the last full database backup. A BCM page is located at the 8th page of every data file, and every 511,230 pages thereafter. All the bits on a BCM page are reset to 0 every time a full database backup or a log backup occurs.

Because of the ability to minimally log bulk operations, the operations themselves can be carried out much faster than in FULL recovery mode. There is a little overhead to setting the bits in the appropriate BCM page, but compared to the cost of logging each individual change to a data or index row, the cost of flipping bits is almost negligible.

If your database is in BULK_LOGGED mode and you have not actually performed any bulk operations, you can restore your database to any point in time or to a named log mark because the log will contain a full sequential record of all changes to your database.

The tradeoff comes during the backing up of the log. In addition to copying the contents of the transaction log to the backup media, SQL Server scans the BCM pages and backs up all the modified extents along with the transaction log itself. The log file itself stays small, but the backup of the log can be many times larger. So the log backup takes more time and might take up a lot more space than in FULL recovery mode. The time it takes to restore a log backup made in BULK_LOGGED recovery mode is similar to the time it takes to restore a log backup made in FULL recovery mode. The operations don't have to be redone; all the information necessary to recover all data and index structures is available in the log backup.

SIMPLE Recovery Model

The SIMPLE recovery model offers the simplest backup-and-restore strategy. Your transaction log is truncated at regular, frequent intervals. Therefore, only full database backups and differential backups are allowed. You get an error if you try to back up the log while in SIMPLE recovery mode. Because the log is not needed for backup purposes, sections of it can be reused as soon as all the transactions it contains are committed or rolled back, and the transactions are no longer needed for recovery from server or transaction failure.

Converting from SQL Server 7

Microsoft intended these recovery models to replace the *select into/bulkcopy* and *trunc. log on chkpt.* database options. Earlier versions of SQL Server required that the *select into/bulkcopy* option be set in order to perform a SELECT INTO or bulk copy operation. The *trunc. log on chkpt.* option forced your transaction log to be truncated every time a checkpoint occurred in the database. This option was recommended only for test or development systems, not for production servers. You can still set these options using the *sp_dboption* procedure, but not using the ALTER DATABASE command. However, in SQL Server 2000, changing either of these options using *sp_dboption* also changes your recovery mode, and changing your recovery mode changes the value of one or both of these options, as you'll see below. The recommended method for changing your database recovery mode is to use the ALTER DATABASE command:

```
ALTER DATABASE <database_name>
    SET RECOVERY [FULL | BULK_LOGGED | SIMPLE]
```

To see what mode your database is in, you can use the DATABASEPROPERTYEX() property function:

```
SELECT DATABASEPROPERTYEX('<database_name>', 'recovery')
```

As I just mentioned, you can change the recovery mode by changing the database options. For example, if your database is in FULL recovery mode and you change the *select into/bulkcopy* option to *true*, your database recovery mode automatically changes to BULK_LOGGED. Conversely, if you force the database back into FULL mode using ALTER DATABASE, the value of the *select into/bulkcopy* option changes automatically. In fact, *sysdatabases* doesn't record any special information for the recovery mode. The recovery mode is determined by the status bits for these two database options. If bit 3 in *sysdatabases.status* is set, the database has *select into/bulkcopy* enabled, and if bit 4 is set, the database has *trunc. log on chkpt.* enabled. Table 5-4 shows the relationship between the database options and the new recovery modes.

If trunc. log on chkpt. is:	*And select into/bulkcopy is:*	*The recovery mode is:*
FALSE	FALSE	FULL
FALSE	TRUE	BULK_LOGGED
TRUE	FALSE	SIMPLE
TRUE	TRUE	SIMPLE

Table 5-4. *The relationship between SQL Server 7 database options and recovery modes.*

If you're using SQL Server 2000 Standard or Enterprise Edition, the *model* database starts in FULL recovery mode, so all your new databases will also be in FULL mode. If you're using SQL Server Personal Edition or the Microsoft SQL Server Desktop Engine, the *model* database starts in SIMPLE recovery mode. You can change the mode of the *model* database or any other user database by using the ALTER DATABASE command.

The new recovery model offers you two major benefits over previous versions. First, you can always perform a SELECT INTO operation without having to worry about what options you've set. Prior to SQL Server 2000, you could only run the SELECT INTO or minimally logged bulk copy operation if you had set the specific database option to true, and only a database owner could change that option. That sometimes meant a restriction on what non-DBO developers could accomplish.

Second, you can freely switch between the FULL and BULK_LOGGED modes without worrying about your backup scripts failing. Prior to SQL Server 2000, once you performed a SELECT INTO or a bulk copy, you could no longer back up your transaction log. So if you had automatic log backup scripts scheduled to run at regular intervals, these would break and generate an error. This can no longer happen. You can run SELECT INTO or bulk copy in any recovery mode, and you can back up the log in either FULL or BULK_LOGGED mode.

In addition, you can easily switch between FULL and BULK_LOGGED modes if you usually operate in FULL mode but occasionally need to perform a bulk operation quickly. You can change to BULK_LOGGED and pay the price later when you back up the log; the backup will simply take longer and be larger.

You can't easily switch to and from SIMPLE mode. When you use the ALTER DATABASE command to change from SIMPLE to FULL or BULK_LOGGED, you must first make a complete database backup in order for the change in behavior to be complete. Remember that SIMPLE recovery mode is comparable to the database option that truncates the log at regular intervals. The truncation option isn't recommended for production databases, where you need maximum transaction recoverability. The only time that SIMPLE mode is really useful is in test and development situations or for small databases that are primarily read-only. I suggest that you use FULL or BULK_LOGGED for your production databases and that you switch between those modes whenever you need to.

Choosing a Backup Type

If you're responsible for creating the backup plan for your data, you'll not only need to choose a recovery mode but also decide what kind of backup to make. I mentioned the three main types: full, differential, and log. In fact, you can use all three types together. To accomplish any type of full restoration of a database, you must occasionally

make a full database backup. In addition, you can also make differential or log backups. Here are some facts to help you decide between these last two:

A differential backup:

- Is faster if your environment includes a lot of changes to the same data. It will back up only the most recent change, whereas a log backup will capture every individual update.
- Captures the entire B-tree structures for new indexes, whereas a log backup captures each individual step in building the index.
- Is cumulative. When you recover from a media failure, only the most recent differential backup needs to be restored because it will contain all the changes since the last full database backup.

A log backup:

- Allows you to restore to any point in time because it is a sequential record of all changes.
- Can be made after a failure of the database media, as long as the log is available. This will allow you to recover right up to the point of the failure. The last log backup (called the tail of the log) must specify the WITH NO_TRUNCATE option in the BACKUP LOG command if the database itself is unavailable.
- Is sequential and discrete. Each log backup contains completely different log records. When you use a log backup to restore a database after a media failure, all log backups must be applied in the order that they were made.

Restoring a Database

How often you make each type of backup determines two things: how fast you can restore a database and how much control you have over which transactions are restored. Consider the schedule in Figure 5-5, which shows a database fully backed up on Sunday. The log is backed up daily, and a differential backup is made on Tuesday and Thursday. A drive failure occurs on Friday. If the failure does not include the log files or if you have mirrored them using RAID 1, you should back up the tail of the log with the NO_TRUNCATE option.

> **WARNING** If you are operating in BULK_LOGGED recovery mode, backing up the log also backs up any data that was changed with a BULK_LOGGED operation, so you might need to have more than just the log file available to back up the tail of the log. You'll also need to have available any filegroups containing data inserted with a bulk copy or SELECT INTO command.

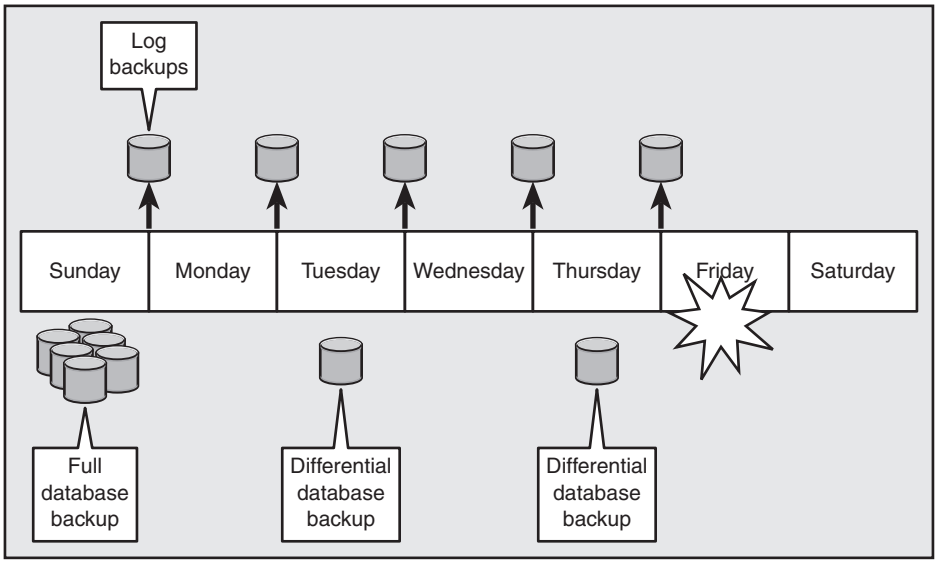

Figure 5-5. *Combined usage of log and differential backups reduces total restore time.*

To restore this database after a failure, you must start by restoring the full backup made on Sunday. This does two things: it copies all the data, log, and index pages from the backup media to the database files, and it applies all the transactions in the log. You must determine whether incomplete transactions are rolled back. You can opt to recover the database by using the WITH RECOVERY option of the RESTORE command. This will roll back any incomplete transactions and open the database for use. No further restoring can be done. If you choose not to roll back incomplete transactions by specifying the WITH NORECOVERY option, the database will be left in an inconsistent state and will not be usable.

If you choose WITH NORECOVERY, you can then apply the next backup. In the scenario depicted in Figure 5-5, you would restore the differential backup made on Thursday, which would copy all the changed extents back into the data files. The differential backup also contains the log records spanning the time the differential backup was being made, so again you have to decide whether to recover the database. Complete transactions are always rolled forward, but you determine whether incomplete transactions are rolled back.

After the last differential backup is restored, you must restore, in sequence, all the log backups made after the last differential backup was made. This includes the tail of the log backed up after the failure if you were able to make this last backup.

> **NOTE** The recovery done during a restore operation works almost exactly the same way as restart recovery, which I described in Chapter 3. There is an analysis pass to determine how much work might need to be done, a roll-forward pass to redo completed transactions and return the database to the state it was in when the backup was complete, and a rollback pass to undo incomplete transactions. The big difference between restore recovery and restart recovery is that with restore recovery you have control over when the rollback pass is done. It should not be done until all the rolling forward from all the backups has been applied. Only then should you roll back any transactions that are still not complete.

Backing Up and Restoring Files and Filegroups

SQL Server 2000 allows you to back up individual files or filegroups. This can be useful in environments with extremely large databases. You can choose to back up just one file or filegroup each day, so the entire database does not have to be backed up as often. It also can be useful when you have an isolated media failure on just a single drive and you think that restoring the entire database would take too long.

Here are a few details you should keep in mind when backing up and restoring files and filegroups:

- Individual files and filegroups can be backed up only when your database is in FULL or BULK_LOGGED recovery mode because you must apply log backups after you restore a file or filegroup and you can't make log backups in SIMPLE mode.
- Unlike differential and full database backups, a backup of a file or filegroup does not back up any portion of the transaction log.
- You can restore individual file or filegroup backups from a full database backup.
- Immediately before restoring an individual file or filegroup, you must back up the transaction log. You must have an unbroken chain of log backups from the time the file or filegroup backup was made.
- After restoring a file or filegroup backup, you must restore all the transaction logs made between the time you backed up the file or filegroup and the time you restored it. This guarantees that the restored files are in sync with the rest of the database.

  For example, suppose you backed up Filegroup FG1 at 10 A.M. Monday. The database is still in use, and changes happen to data on FG1 and

transactions are processed that change data in both FG1 and other filegroups. You back up the log at 4 P.M. More transactions are processed that change data in both FG1 and other filegroups. At 6 P.M., a media failure occurs and you lose one or more of the files making up FG1.

To restore, you must first back up the tail of the log containing all changes that occurred between 4 P.M. and 6 P.M. You can then restore FG1 using the RESTORE DATABASE command, specifying just filegroup FG1. Your database will not be in a consistent state because the restored FG1 will have changes only through 10 A.M. but the rest of the database will have changes through 6 P.M. However, SQL Server knows when the last change was made to the database because each page in a database stores the LSN of the last log record that changed that page. When restoring a filegroup, SQL Server makes a note of the maximum LSN in the database. You must restore log backups until the log reaches at least the maximum LSN in the database, and you will not reach that point until you apply the 6 P.M. log backup.

Partial Restore

SQL Server 2000 lets you do a partial restore of a database in emergency situations. Although the description and the syntax seem similar to file and filegroup backups, there is a big difference. With file and filegroup backups, you start with a complete database and replace one or more files or filegroups with previously backed up versions. With a partial database restore, you don't start with a full database. You restore individual filegroups, which must include the primary filegroup containing all the system tables, to a new location. Any filegroups you don't restore no longer exist and are treated as OFFLINE when you attempt to reference data stored on them. You can then restore log backups or differential backups to bring the data in those filegroups to a later point in time. This allows you the option of recovering the data from a subset of tables after an accidental deletion or modification of table data. You can use the partially restored database to extract the data from the lost tables and copy it back into your original database.

Restoring with Standby

In normal recovery operations, you have the choice of either running recovery to roll back incomplete transactions or not running recovery. If you run recovery, no further log backups can be restored and the database is fully usable. If you don't run recovery, the database is inconsistent and SQL Server won't let you use it at all. You have to choose one or the other because of the way log backups are made.

For example, in SQL Server 2000, log backups do not overlap—each log backup starts where the previous one ended. Consider a transaction that makes hundreds of updates to a single table. If you back up the log in the middle of the updating and again after the updating is finished, the first log backup will have the beginning of

the transaction and some of the updates and the second log backup will have the remainder of the updates and the commit. Suppose you then need to restore these log backups after restoring the full database. If, after restoring the first log backup, you run recovery, the first part of the transaction is rolled back. If you then try to restore the second log backup, it will start in the middle of a transaction and SQL Server won't know what the beginning of the transaction did. You certainly can't recover transactions from this point because their operations might depend on this update that you've lost part of. So, SQL Server will not allow any more restoring to be done. The alternative is to not run recovery to roll back the first part of the transaction, but instead to leave the transaction incomplete. SQL Server will know that the database is inconsistent and will not allow any users into the database until you finally run recovery on it.

What if you want to combine the two approaches? It would be nice to be able to restore one log backup and look at the data before restoring more log backups, particularly if you're trying to do a point-in-time recovery, but you won't know what the right point is. SQL Server provides an option called STANDBY that allows you to recover the database and still restore more log backups. If you restore a log backup and specify WITH STANDBY = '<*some filename*>', SQL Server will roll back incomplete transactions but keep track of the rolled-back work in the specified file, which is known as a standby file. The next restore operation will first read the contents of the standby file and redo the operations that were rolled back, and then it will restore the next log. If that restore also specifies WITH STANDBY, incomplete transactions will again be rolled back but a record of those rolled back transactions will be saved. Keep in mind that you can't modify any data if you've restored WITH STANDBY (SQL Server will generate an error message if you try), but you can read the data and continue to restore more logs. The final log must be restored WITH RECOVERY (and no standby file will be kept) to make the database fully usable.

SUMMARY

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 *pubs*, *Northwind*, and *msdb*.) 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 the 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.

Now that you understand database basics, it's time to move on to tables, the fundamental data structures you'll work with.