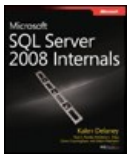

Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 4: Logging and Recovery

Kalen Delaney

In Chapter 3, “Databases and Database Files,” I told you about the data files that are created to hold information in a Microsoft SQL Server database. Every database also has at least one file that stores its transaction log. I referred to SQL Server transaction logs and log files in Chapter 3, but I did not really go into detail about how a log file is different from a data file and exactly how SQL Server uses its log files. In this chapter, I tell you about the structure of SQL Server log files and how they’re managed when transaction information is logged. I explain how SQL Server log files grow and when and how a log file can be reduced in size. Finally, I look at how log files are used during SQL Server backup and restore operations and how they are affected by your database’s recovery model.

Transaction Log Basics

The transaction log records changes made to the database and stores enough information to allow SQL Server to recover the database. The recovery process takes place every time a SQL Server instance is started, and it can take place every time SQL Server restores a database or a log from backup. *Recovery* is the process of reconciling the data files and the log. Any changes to the data that the log indicates have been committed must appear in the data files, and any changes that are not marked as committed must not appear in the data files. The log also stores information needed to roll back an operation if SQL Server receives a request to roll back a transaction from the client (using the *ROLLBACK TRAN* command) or if an error, such as a deadlock, generates an internal *ROLLBACK*.

Physically, the transaction log is one or more files associated with a database at the time the database is created or altered. Operations that perform database modifications write records in the transaction log that describe the changes made (including the page numbers of the data pages modified by the operation), the data values that were added or removed, information about the transaction that the modification was part of, and the date and time of the beginning and end of the transaction. SQL Server also writes log records when certain internal events happen, such as checkpoints. Each log record is labeled with a Log Sequence Number (LSN) that is guaranteed to be unique. All log entries that are part of the same transaction are linked so that all parts of a transaction can be located easily for both undo activities (as with a rollback) and redo activities (during system recovery).

The Buffer Manager guarantees that the transaction log will be written before the changes to the database are written. (This is called *write-ahead logging*.) This guarantee is possible because SQL Server keeps track of its current position in the log by means of the LSN. Every time a page is changed, the LSN corresponding to the log entry for that change is written into the header of the data page. Dirty pages can be written to the disk only when the LSN on the page is less than or equal to the LSN for the last record written to the log. The Buffer Manager also guarantees that log pages are written in a specific order, making it clear which log blocks must be processed after a system failure, regardless of when the failure occurred.

The log records for a transaction are written to disk before the commit acknowledgement is sent to the client process, but the actual changed data might not have been physically written out to the data pages. Although the writes to the log are asynchronous, at commit time the thread must wait for the writes to complete to the point of writing the commit record in the log for the transaction. (SQL Server must wait for the commit record to be written so that it knows the relevant log records are safely on the disk.) Writes to data pages are completely asynchronous. That is, writes to data pages need only be posted to the operating system, and SQL Server can check later to see that they were completed. They don’t have to be completed immediately because the log contains all the information needed to redo the work, even in the event of a power failure or system crash before the write completes. The system would be much slower if it had to wait for every I/O request to complete before proceeding.

Logging involves demarcating the beginning and end of each transaction (and savepoints, if a transaction uses them). Between the beginning and ending demarcations is information about the changes made to the data. This information can take the form of the actual “before and after” data, or it can refer to the operation that was performed so that those values can be derived. The end of a typical transaction is marked with a Commit record, which indicates that the transaction must be reflected in the database’s data files or redone if necessary. A transaction aborted during normal runtime (not system restart) due to an explicit rollback or something like a resource error (for example, an out-of-memory error) actually undoes the operation by applying changes that undo the original data modifications. The records of these changes are written to the log and marked as “compensation log records.”

As mentioned previously, there are two types of recovery, both of which have the goal of making sure the log and the data

agree. A *restart recovery* runs every time SQL Server is started. The process runs on each database because each database has its own transaction log. Your SQL Server error log reports the progress of restart recovery, and for each database, the error log tells you how many transactions were rolled forward and how many were rolled back. This type of recovery is sometimes referred to as *crash recovery* because a crash, or unexpected stopping of the SQL Server service, requires the recovery process to be run when the service is restarted. If the service was shut down cleanly with no open transactions in any database, only minimal recovery is necessary upon system restart. In SQL Server 2008, restart recovery can be run on multiple databases in parallel, each handled by a different thread.

The other type of recovery, *restore recovery* (or *media recovery*), is run by request when a restore operation is executed. This process makes sure that all the committed transactions in the backup of the transaction log are reflected in the data and that any transactions that did not complete do not show up in the data. I'll talk more about restore recovery later in the chapter.

Both types of recovery must deal with two situations: when transactions are recorded as committed in the log but not yet written to the data files, and when changes to the data files don't correspond to committed transactions. These two situations can occur because committed log records are written to the log files on disk every time a transaction commits. Changed data pages are written to the data files on disk completely asynchronously, every time a checkpoint occurs in a database. As I mentioned in Chapter 1, "SQL Server 2008 Architecture and Configuration," data pages can also be written to disk at other times, but the regularly occurring checkpoint operations give SQL Server a point at which *all* changed (or dirty) pages are known to have been written to disk. Checkpoint operations also write log records from transactions in progress to disk because the cached log records are also considered to be dirty.

If the SQL Server service stops after a transaction commits but before the data is written out to the data pages, when SQL Server starts and runs recovery, the transaction must be rolled forward. SQL Server essentially redoes the transaction by reapplying the changes indicated in the transaction log. All the transactions that need to be redone are processed first (even though some of them might need to be undone later during the next phase). This is called the *redo* phase of recovery.

If a checkpoint occurs before a transaction is committed, it writes the uncommitted changes out to disk. If the SQL Server service then stops before the commit occurs, the recovery process finds the changes for the uncommitted transactions in the data files, and it has to roll back the transaction by undoing the changes reflected in the transaction log. Rolling back all the incomplete transactions is called the *undo* phase of recovery.

> **Note** I'll continue to refer to recovery as a system startup function, which is its most common role by far. However, remember that recovery is also run during the final step of restoring a database from backup or attaching a database, and can also be forced manually. In addition, recovery is run when creating a database snapshot, during database mirroring, or when failing over to a database mirror.

Later in this chapter, I'll cover some special issues related to recovery during a database restore. These include the three recovery models that you can set using the *ALTER DATABASE* statement and the ability to place a named marker in the log to indicate a specific point to recover to. The discussion that follows deals with recovery in general, whether it's performed when the SQL Server service restarts or when a database is being restored from a backup.

Phases of Recovery

During recovery, only changes that occurred or were in progress since the last checkpoint are evaluated to determine if they need to be redone or undone. Any transactions that completed prior to the last checkpoint, either by being committed or rolled back, are accurately reflected in the data pages, and no additional work needs to be done for them during recovery.

The recovery algorithm has three phases, which center around the last checkpoint record in the transaction log. The three phases are illustrated in Figure 4-1.

Phase 1: Analysis The first phase is a forward pass starting at the last checkpoint record in the transaction log. This pass determines and constructs a dirty page table (DPT) consisting of pages that might have been dirty at the time SQL Server stopped. An active transaction table is also built that consists of uncommitted transactions at the time SQL Server stops.

Phase 2: Redo This phase returns the database to the state it was in at the time the SQL Server service stopped. The starting point for this forward pass is the start of the oldest uncommitted transaction. The minimum LSN in the DPT is the first time SQL Server expects to have to redo an operation on a page, but it needs to redo the logged operations

starting all the way back at the start of the oldest open transaction so that the necessary locks can be acquired. (Prior to SQL Server 2005, it was just allocation locks that needed to be reacquired. In SQL 2005 and later, all locks for those open transactions need to be reacquired.)

Phase 3: Undo This phase uses the list of active transactions (uncommitted at the time SQL Server came down) which were found in Phase 1 (Analysis). It rolls each of these active transactions back individually. SQL Server follows the links between entries in the transaction log for each transaction. Any transaction that was not committed at the time SQL Server stopped is undone so that none of the changes are actually reflected in the database.

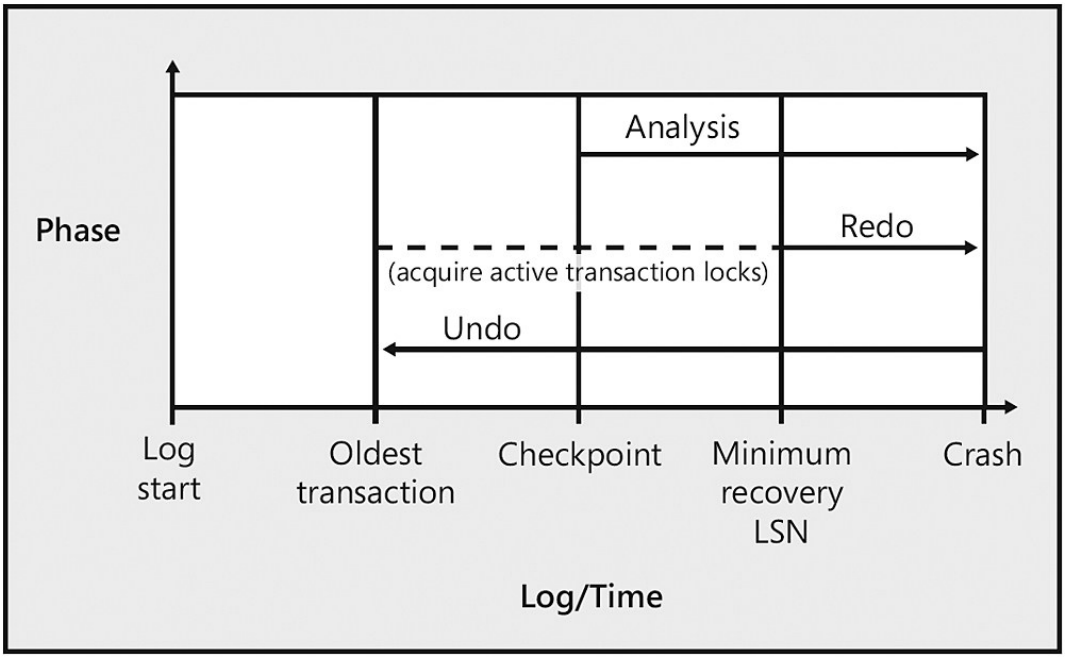

Figure 4-1: The three phases of the SQL Server recovery process

SQL Server uses the log to keep track of the data modifications that were made, as well as any locks that were applied to the objects being modified. This allows SQL Server to support a feature called *fast recovery* when SQL Server is restarted (in the Enterprise and Developer editions only). With fast recovery, the database is available as soon as the redo phase is finished. The same locks that were acquired during the original modification can be reacquired to keep other processes from accessing the data that needs to have its changes undone; all other data in the database remains available. Fast recovery cannot be done during media recovery, but it is used by database mirroring recovery, which uses a hybrid of media recovery and restart recovery.

In addition, SQL Server uses multiple threads to process the recovery operations on the different databases simultaneously, so databases with higher ID numbers don't have to wait for all databases with lower ID numbers to be completely recovered before their own recovery process starts.

Page LSNs and Recovery

Every database page has an LSN in the page header that reflects the location in the transaction log of the last log entry that modified a row on this page. Each log record for changes to a data page has two LSNs associated with it. In addition to the LSN for the actual log record, it also keeps track of the LSN, which was on the data page before the change recorded by this log record. During a redo operation of transactions, the LSNs on each log record are compared to the page LSN of the data page that the log entry modified. If the page LSN is equal to the previous page LSN in the log record, the operation indicated in the log entry is redone. If the LSN on the page is equal to or higher than the actual LSN for this log record, SQL Server skips the *REDO* operation. These two possibilities are illustrated in Figure 4-2. The LSN on the page cannot be between the previous and current values for the log record.

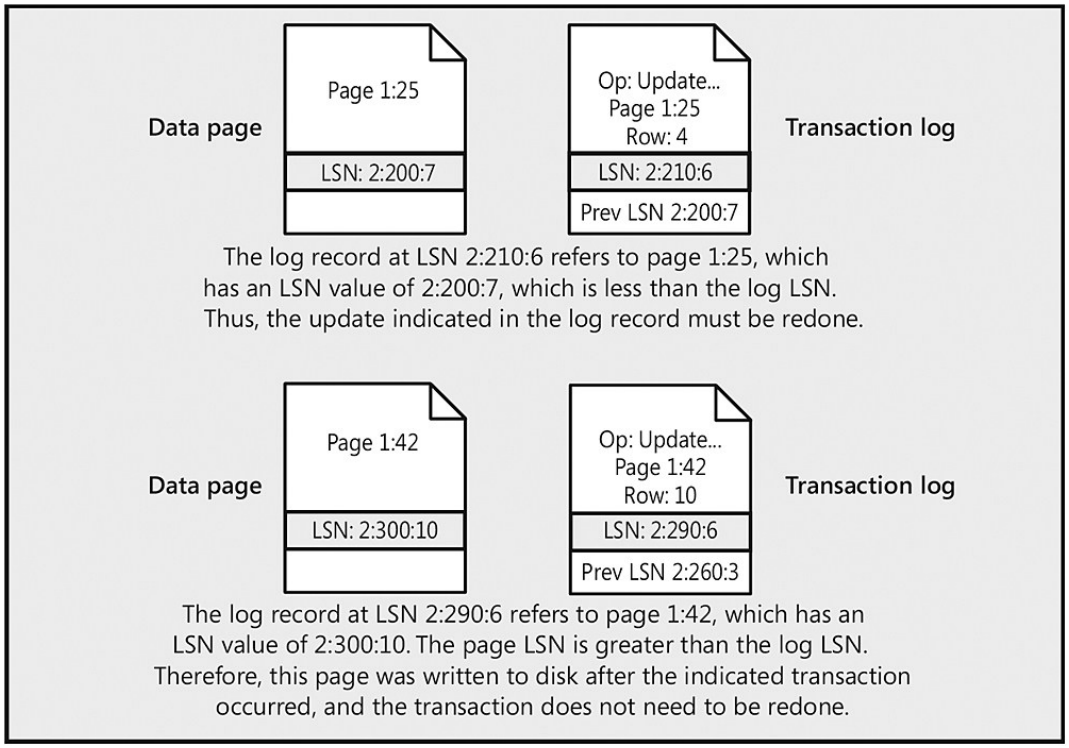

Figure 4-2: Comparing LSNs to decide whether to process the log entry during recovery

Because recovery finds the last checkpoint record in the log (plus transactions that were still active at the time of the checkpoint) and proceeds from there, recovery time is short, and all changes committed before the checkpoint can be purged from the log or archived. Otherwise, recovery could take a long time and transaction logs could become unreasonably large. A transaction log cannot be truncated prior to the point of the earliest transaction that is still open, no matter how many checkpoints have occurred since the transaction started and no matter how many other transactions have started or completed. If a transaction remains open, the log must be preserved because it's still not clear whether the transaction is done or ever will be done. The transaction might ultimately need to be rolled back or rolled forward.

Note Truncating of the transaction log is a logical operation and merely marks parts of the log as no longer needed, so the space can be reused. Truncation is not a physical operation and does not reduce the size of the transaction log files on disk. To reduce the physical size, a shrink operation must be performed.

Some SQL Server administrators have noted that the transaction log seems unable to be truncated, even after the log has been backed up. This problem often results from a user opening a transaction and then forgetting about it. For this reason, from an application development standpoint, you should ensure that transactions are kept short. Another possible reason for an inability to truncate the log relates to a table being replicated using transactional replication when the replication log reader hasn't processed all the relevant log records yet. This situation is less common, however, because typically a latency of only a few seconds occurs while the log reader does its work. You can use *DBCC OPENTRAN* to look for the earliest open transaction or the oldest replicated transaction not yet processed and then take corrective measures (such as killing the offending process or running the *sp_repldone* stored procedure to allow the replicated transactions to be purged). I'll discuss problems with transaction management and some possible solutions in Chapter 10, "Transactions and Concurrency." I'll discuss shrinking of the log in the next section.

Reading the Log

Although the log contains a record of every change made to a database, it is not intended to be used as an auditing tool. The transaction log is used to enable SQL Server to guarantee recoverability in case of statement or system failure and to allow a system administrator to take backups of the changes to a SQL Server database. If you want to keep a readable record of changes to a database, you have to do your own auditing. You can do this by creating a trace of SQL Server activities, using SQL Server Profiler or one of the tracing mechanisms in SQL Server, as discussed in Chapter 2, "Change Tracking, Tracing, and Extended Events."

Note You might be aware that some third-party tools can read the transaction log and show you all the operations that have taken place in a database and can allow you to roll back any of those operations. The developers of these tools spent tens of thousands of hours looking at byte-level dumps of the transaction log files and correlating that

information with the output of an undocumented *DBCC LOG* command. Once they had a product on the market, Microsoft started working with them, which made their lives a bit easier in subsequent releases. However, no such tools are available for SQL Server 2008.

Although you might assume that reading the transaction log directly would be interesting or even useful, it's usually just too much information. If you know in advance that you want to keep track of what your server running SQL Server is doing, you're much better off defining a trace with the appropriate filter to capture just the information that is useful to you.

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 stream. For example, when the *DBCC SHRINKDATABASE* command (discussed in Chapter 3) determines how much the log can be shrunk, it does not consider the log files separately but instead determines the shrinkable size based on the entire log.

Virtual Log Files

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. When a log file is first created, it always has between 2 and 16 VLFs. If the file size is 1 MB or less, SQL Server divides the size of the log file by the minimum VLF size [31 * 8 KB] to determine the number of VLFs. If the log file size is between 1 and 64 MB, SQL Server splits the log into 4 VLFs. If the log file is greater than 64 MB but less than or equal to 1 GB, 8 VLFs are created. If the size is more than 1 GB, there will be 16 VLFs. When the log grows, the same formula is used to determine how many new VLFs to add. A log always grows in units of entire VLFs and can be shrunk only to a VLF boundary. (Figure 4-3 illustrates a physical log file, along with several VLFs.)

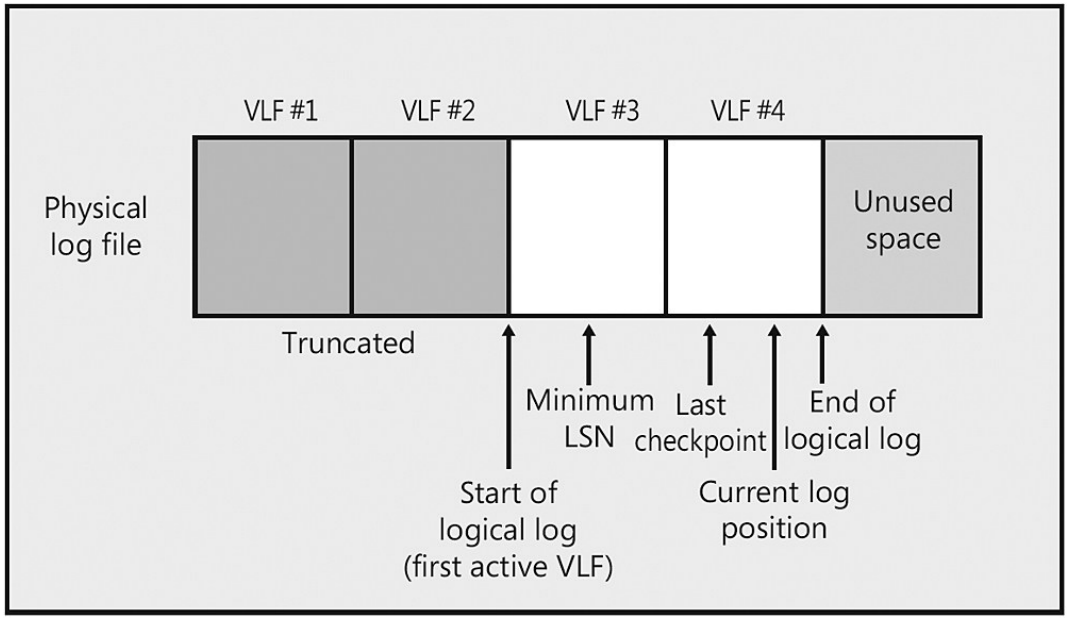

Figure 4-3: Multiple VLFs that make up a physical log file

A VLF can be in one of four states:

Active The active portion of the log begins at the minimum 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.) Figure 4-3 contains two active VLFs.

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 before the oldest active transaction are not needed and can be reused. Truncating or backing up the transaction log changes recoverable VLFs into reusable VLFs. For the purpose of determining which VLFs are reusable, active transactions

include more than just open transactions. The earliest active transaction may be a transaction marked for replication that has not yet been processed, the beginning of a log backup operation, or the beginning of an internal diagnostic scan that SQL Server performs periodically.

Unused One or more VLFs at the physical end of the log files might not have been used yet if not enough logged activity has taken place or if earlier VLFs have been marked as reusable and then reused.

Observing Virtual Log Files

You can observe the same key properties of virtual log files by executing the undocumented command *DBCC LOGINFO.* This command takes no parameters, so it must be run in the database for which you want information. It returns one row for each VLF. When I run this command in my *AdventureWorks2008* database, I get the following eight rows returned (not all columns are shown):

```
FileId   FileSize       StartOffset    FSeqNo      Status      CreateLSN
-------- -------------- -------------- ----------- ----------- -------------------
2        458752         8192           42          2           0
2        458752         466944         41          0           0
2        458752         925696         43          2           0
2        712704         1384448        44          2           0
2        4194304        2097152        47          2           44000000085601161
2        4194304        6291456        46          2           44000000085601161
2        4194304        10485760       40          2           44000000085601161
2        4194304        14680064       0           0           44000000085601161
```

The number of rows tells me how many VLFs are in my database. The *FileID* column indicates which of the log's physical files contains the VLF; for my *AdventureWorks2008* database, there is only one physical log file. *FileSize* and *StartOffset* are indicated in bytes, so you can see that the first VLF starts after 8192 bytes, which is the number of bytes in a page. The first physical page of a log file contains header information, not log records, so the VLF is considered to start on the second page. The *FileSize* column is actually redundant for most rows because the size value can be computed by subtracting the *StartOffset* values for two successive VLFs. The rows are listed in physical order, but that is not always the order in which the VLFs have been used. The use order (logical order) is reflected in the column called *FSeqNo* (which stands for File Sequence Number).

In the output shown previously, you can see that the rows are listed in physical order according to the *StartOffset*, but the logical order does not match. The *FSeqNo* values indicate that the seventh VLF is actually the first one in use (logical) order; the last one in use order is the fifth VLF in physical order. The *Status* column indicates whether the VLF is reusable. A status of 2 means that it is either active or recoverable; a status of 0 indicates that it is reusable or completely unused. (A completely unused VLF has a *FSeqNo* value of 0, as in the eighth row of my output.) As I mentioned previously, truncating or backing up the transaction log changes recoverable VLFs into reusable VLFs, so a status of 2 changes to a status of 0 for all VLFs that don't include active log records. In fact, that's one way to tell which VLFs are active: the VLFs that still have a status of 2 after a log backup or truncation must contain records from active transactions. VLFs with a status of 0 can be reused for new log records, and the log does not need to grow to keep track of the activity in the database. On the other hand, if all the VLFs in the log have a status of 2, SQL Server needs to add new VLFs to the log to record new transaction activity. One last column shown in the *DBCC LOG* output shown previously is called *CreateLSN*. That column contains an LSN value; in fact, it is the current LSN at the time the VLF was added to the transaction log. If the *CreateLSN* value is 0, it means the VLF was part of the original log file created when the database was created. You can also tell how many VLFs were added in any one operation by noticing which VLFs have the same value for *CreateLSN*. In my output, the *CreateLSN* values indicate that my log file only grew once, and four new VLFs were added at the same time.

Multiple Log Files

I mentioned previously that SQL Server treats multiple physical log files as if they were one sequential stream. This means that all the VLFs in one physical file are used before any VLFs in the second file are used. If you have a well-managed log that is regularly backed up or truncated, you might never use any log files other than the first one. If none of the VLFs in multiple physical log files is available for reuse when a new VLF is needed, SQL Server adds new VLFs to each physical log file in a round-robin fashion.

You can actually see the order of usage of different physical files by examining the output of *DBCC LOGINFO*. The first column is the *file_id* of the physical file. If we can capture the output of *DBCC LOGINFO* into a table, we can then sort it in a way that is useful to us. The following code creates a table called *sp_loginfo* that can hold the output of *DBCC LOGINFO*. Because the table is created in the master database and starts with the three characters 'sp_', it can be

accessed and modified in any database:

```
USE master
GO
IF EXISTS  (SELECT 1 FROM sys.tables
                          WHERE name = 'sp_LOGINFO')
        DROP TABLE sp_loginfo;
GO
CREATE TABLE sp_LOGINFO
(FileId tinyint,
 FileSize bigint,
 StartOffset bigint,
 FSeqNo int,
 Status tinyint,
 Parity tinyint,
 CreateLSN numeric(25,0) );
GO
```

The following code creates a new database called *TWO_LOGS* and then copies a large table from the *AdventureWorks2008* sample database into *TWO_LOGS*, causing the log to grow:

```
USE Master
GO
IF EXISTS (SELECT * FROM sys.databases
                          WHERE name = 'TWO_LOGS')
    DROP DATABASE TWO_LOGS;
GO
CREATE DATABASE TWO_LOGS
  ON PRIMARY
  (NAME = Data ,
    FILENAME =
    'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS.mdf'
        , SIZE = 100 MB)
   LOG ON
  (NAME = TWO_LOGS1,
    FILENAME =
    'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS1.ldf'
        , SIZE = 5 MB
        , MAXSIZE = 2 GB),
  (NAME = TWO_LOGS2,
    FILENAME =
    'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\TWO_LOGS2.ldf'
        , SIZE = 5 MB);
GO
```

If you run *DBCC LOGINFO,* you'll notice that it returns VLFs sorted by *FileID,* and initially, the file sequential number values (*FSeqNo*) are also in order:

```
USE TWO_LOGS
GO
DBCC LOGINFO;
GO
```

Now we can insert some rows into the database, by copying from another table:

```
SELECT * INTO Orders
    FROM AdventureWorks2008.Sales.SalesOrderDetail;
GO
```

If you run *DBCC LOGINFO* again, you see that after the *SELECT INTO* operation, even though there are many more rows for each *FileID*, the output is still sorted by *FileID*, and *FSeqNo* values are not related at all. Instead, we can save the output of *DBCC LOGINFO* in the *sp_loginfo* table, and sort by *FSeqNo:*

```
TRUNCATE TABLE sp_LOGINFO;
INSERT INTO sp_LOGINFO
   EXEC ('DBCC LOGINFO');
GO
-- Unused VLFs have a Status of 0, so the CASE forces those to the end
SELECT * FROM sp_LOGINFO
ORDER BY CASE FSeqNo WHEN 0 THEN 9999999 ELSE FSeqNo END;
GO
```

The output of the *SELECT* is shown here:

```
FileId StartOffset          FSeqNo      Status CreateLSN
------ -------------------- ----------- ------ ----------------
2      8192                 43          0      0
2      1253376              44          0      0
2      2498560              45          0      0
2      3743744              46          0      0
3      8192                 47          0      0
3      1253376              48          0      0
3      2498560              49          0      0
3      3743744              50          0      0
2      5242880              51          0      50000000247200092
2      5496832              52          0      50000000247200092
3      5242880              53          0      51000000035600288
3      5496832              54          0      51000000035600288
2      5767168              55          0      53000000037600316
2      6021120              56          0      53000000037600316
3      5767168              57          0      56000000010400488
3      6021120              58          0      56000000010400488
2      6356992              59          0      58000000007200407
2      6610944              60          0      58000000007200407
3      6356992              61          0      60000000025600218
3      6610944              62          0      60000000025600218
2      7012352              63          0      62000000023900246
2      7266304              64          0      62000000023900246
3      7012352              65          0      64000000037100225
3      7266304              66          0      64000000037100225
2      7733248              67          0      66000000037600259
2      7987200              68          0      66000000037600259
2      8241152              69          0      66000000037600259
3      7733248              70          0      68000000035500288
3      7987200              71          0      68000000035500288
3      8241152              72          0      68000000035500288
2      8519680              73          0      71000000037300145
2      8773632              74          0      71000000037300145
2      9027584              75          0      71000000037300145
3      8519680              76          0      75000000018700013
3      8773632              77          2      75000000018700013
3      9027584              0           0      75000000018700013
```

Now you can notice that after the first eight initial VLFs are used (the ones with a *CreateLSN* value of 0), the VLFs alternate between the physical files. Because of the amount of log growth each time, several new VLFs are created, first from *FileID* 2 and then from *FileID* 3. The last VLF added to *FileID* 3 has not been used yet.

So there is really no reason to use multiple physical log files if you have done thorough testing and have determined the optimal size of your database's transaction log. However, if you find that the log needs to grow more than expected and if the volume containing the log does not have sufficient free space to allow the log to grow enough, you might need to create a second log file on another volume.

Automatic Truncation of Virtual Log Files

SQL Server assumes you're not maintaining a sequence of log backups if any of the following is true:

- You have configured the database to truncate the log on a regular basis by setting the recovery model to SIMPLE.
- You have never taken a full database backup.

Under any of these circumstances, SQL Server truncates the database's transaction log every time it gets "full enough." (I'll explain this in a moment.) The database is considered to be in autotruncate mode.

Remember that truncation means that all log records prior to the oldest active transaction are invalidated and all VLFs not containing any part of the active log are marked as reusable. It does not 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—the *recovery interval*. You can change the recovery interval manually by using the *sp_configure* stored procedure or by using SQL Server Management Studio, as discussed in Chapter 1. However, it is best to let SQL Server autotune this value. In most cases, this recovery interval value is set to one minute. By default, *sp_configure* shows zero minutes, meaning SQL Server autotunes the value. SQL Server bases its recovery interval on the estimate that 10 MB worth of transactions can be recovered in one minute.

The actual log truncation is invoked by the checkpoint process, which is usually sleeping and is awakened only 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 of the log.

In addition, if the log ever gets to 70 percent full while the database is in autotruncate mode, the log manager wakes 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 manager is never needed, the checkpoint process won't be invoked and the truncation never happens. If you have a database in autotruncate mode, for which the transaction log has VLFs with a status of 2, you do not see the status change to 0 until some logging activity is required in the database.

If the log is regularly truncated, SQL Server can reuse space in the physical file by cycling back to an earlier VLF when it reaches the end of the physical log file. In effect, SQL Server recycles the space in the log file that is no longer needed for recovery or backup purposes. My *AdventureWorks2008* database is in this state because I have never made a full database backup.

Maintaining a Recoverable Log

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. The VLF status stays at 2 until the log backup occurs. After the log backup, the status changes to 0 and SQL Server can cycle back to the beginning of the file. Figure 4-4 depicts this cycle in a simplified fashion. As you can see from the *FSeqNo* values in the earlier output from the *AdventureWorks2008* database, SQL Server does not always reuse the log files in their physical sequence.

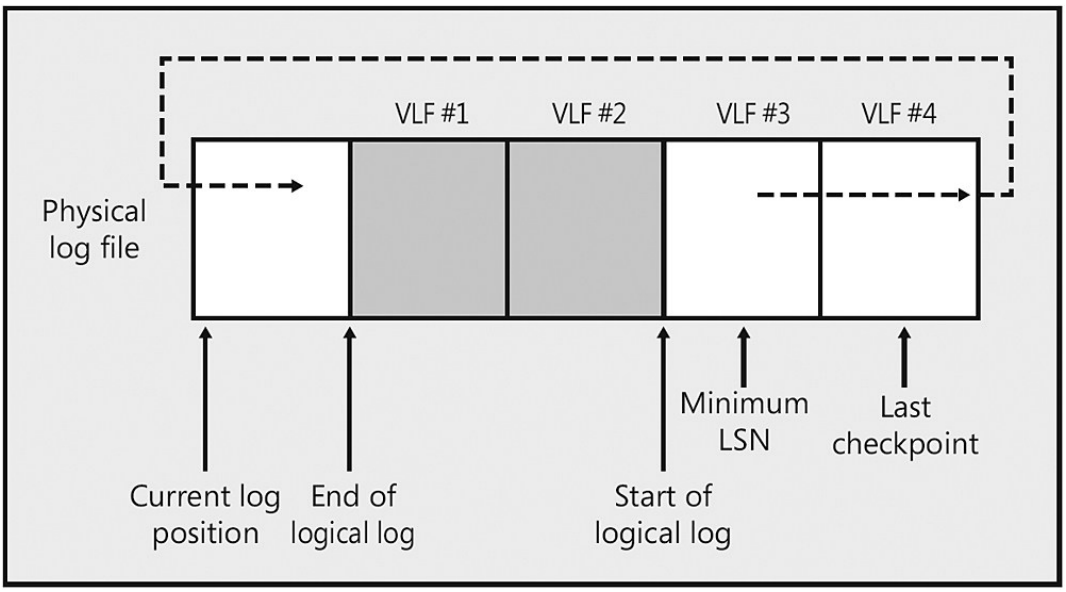

Figure 4-4: The active portion of the log cycling back to the beginning of the physical log file

Note If a database is not in autotruncate mode and you are not performing regular log backups, your transaction log is never truncated. If you are doing only full database backups, you must truncate the log manually to keep it at a manageable size.

The easiest way to tell whether a database is in autotruncate mode is by using the catalog view called *sys.database_recovery_status* and looking in the column called *last_log_backup_lsn*. If that column value is null, the

database is in autotruncate mode.

You can actually observe the difference between a database that is in autotruncate mode and a database that isn't by running a simple script in the *pubs* database, which is shown at the end of this paragraph. This script works so long as you have never made a full backup of the *pubs* database. If you have never made any modifications to *pubs*, and you installed it using the *Instpubs.sql* script, the size of its transaction log file is just about 0.75 MB, which is the size at creation. The following script creates a new table in the *pubs* database, inserts three records, and then updates those records 1,000 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, and the number of VLFs does not increase even after 3,000 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* again, which you can download from the companion Web site, *http://www.SQLServerInternals.com/companion*.) However, even though the number of VLFs does not change, you see that the *FSeqNo* values change. Log records are being generated, and as each VLF is reused, it gets a new *FSeqNo* value:

```
USE pubs;
-- First look at the VLFs for the pubs database
DBCC LOGINFO;
-- Now verify that pubs is in auto truncate mode
SELECT last_log_backup_lsn
FROM master.sys.database_recovery_status
WHERE database_id = db_id('pubs');
GO
CREATE TABLE newtable (a int);
GO
INSERT INTO newtable VALUES (10);
INSERT INTO newtable VALUES (20);
INSERT INTO newtable VALUES (30);
GO
SET NOCOUNT ON
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 model. I'll discuss recovery models later in this chapter, but for now, you can just make sure that *pubs* is in the appropriate recovery model 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, or the path to any backup location:

```
BACKUP DATABASE pubs to disk =
    'c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\backup\pubs.bak';
```

As soon as you make the full backup, you can verify that the database is not in autotruncate mode, again by looking at the *database_recovery_status* view:

```
SELECT last_log_backup_lsn
FROM master.sys.database_recovery_status
WHERE database_id = db_id('pubs');
```

This time, you should get a non-null value for *last_log_backup_lsn* to indicate that log backups are expected. Next, 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 and that there are more VLFs. The initial space in the log could not be reused because SQL Server assumed that you were saving that information for transaction log backups.

Now you can try to shrink the log back down again. The first thing that you need to do is truncate the log, which you can do by setting the recovery model to SIMPLE as follows:

```
ALTER DATABASE pubs SET RECOVERY SIMPLE;
```

If you then issue the following command, or if you issue the *DBCC SHRINKDATABASE* command for the *pubs* database, SQL Server shrinks the log file:

```
DBCC SHRINKFILE (2);
```

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

In SQL Server 7.0, where this log architecture was first introduced, running the preceding commands exactly as specified did not always shrink the physical log file. When the log file did not shrink, it was because 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 versions later than SQL Server 7.0, this process is unnecessary. If a shrink command has already been issued, truncating the log internally generates a series of NO-OP (or dummy) 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.

Automatic Shrinking of the Log

Remember that truncating is not shrinking. A database should be truncated so that it is most shrinkable, and if the log is in autotruncate mode and the autoshrink option is set, the log is physically shrunk at regular intervals.

If a database has the autoshrink option on, an autoshrink process kicks in every 30 minutes (as discussed in Chapter 3) 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 autoshrink processes. The autoshrink process marks the shrinkpoint of the log as 125 percent of the maximum log space actually 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 it gets truncated or backed up. It's possible to have autoshrink without having the database in autotruncate mode, although you cannot guarantee that the log actually shrinks. For example, if the log is never backed up, none of the VLFs are marked as reusable, so no shrinking can take place.

As a final note, you need to be aware that just because a database is in autotruncate mode, you cannot guarantee that the log won't grow. (It is the converse that you can be sure of— that if a database is not in autotruncate mode, the log *will* grow.) *Autotruncate* means only that VLFs that are considered recoverable are marked as reusable at regular intervals. But VLFs in an active state are not affected. If you have a long-running transaction (which might be a transaction that someone forgot to commit), all the VLFs that contain any log records since that long-running transaction started are considered active and can never be reused. One uncommitted transaction can mean the difference between a very manageable transaction log size and a log that uses more disk space than the database itself and continues to grow.

Log File Size

You can see the current size of the log file for all databases, as well as the percentage of the log file space that has been used, by running the command *DBCC SQLPERF('logspace').* However, because it is a DBCC command, it's hard to filter the rows to get just the rows for a single database. Instead, you can use the dynamic management view *sys.dm_os_performance_counters* and retrieve the percentage full for each database's log:

```
SELECT instance_name as [Database],
       cntr_value as "LogFullPct"
FROM sys.dm_os_performance_counters
WHERE counter_name LIKE 'Percent Log Used%'
    AND instance_name not in ('_Total', 'mssqlsystemresource')
    AND cntr_value > 0;
```

The final condition is needed to filter out databases that have no log file size reported. This includes any database that is unavailable because it has not been recovered or is in a suspect state, as well as any database snapshots, which have no transaction log.

Backing Up and Restoring a Database

As you probably know by now, this book is not intended to be a how-to book for database administrators. The bibliography in the companion content lists several excellent books that can teach you the mechanics of making database backups and restoring and can offer best practices for setting up a backup-and-restore plan for your organization. Nevertheless, some important issues relating to backup and restore processes can help you understand why one backup plan might be better

suited to your needs than another. Most of these issues involve the role the transaction log plays in backup and restore operations, so I'll discuss the main ones in this section.

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 opt for the fastest possible speed for data file access at the cost of fault tolerance, backups provide insurance in case your data files are damaged. In addition, backups are also the preferred way to manage copying of databases to other machines or other instances.

If you're using a backup to restore lost data, the amount of data that is potentially recoverable depends on the type of backup. SQL Server 2008 has four main types of backups (and a couple of 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, or a local tape drive.

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 tell quickly which extents need to be backed up by examining the bits on the Differential Changed Map (DCM) pages for each data file in the database. DCM pages are big bitmaps, with one bit representing an extent in a file, just like the Global Allocation Map (GAM) and Shared Global Allocation Map (SGAM) pages that I discussed in Chapter 3. Each time a full backup is made, all the bits in the DCM 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.

File and filegroup backup File and filegroup backups are intended to increase flexibility in scheduling and media handling compared to full backups, in particular for very large databases. File and filegroup backups are also useful for large databases that contain data with varying update characteristics, meaning some filegroups allow both read and write operations and some are read-only.

More Info For full details on the mechanics of defining backup devices, making backups, and scheduling backups to occur at regular intervals, consult *SQL Server Books Online* or one of the SQL Server administration books listed in the bibliography in the online companion content.

A full backup can be made while your database 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 full, differential, or file backups, SQL Server records the current log sequence number (LSN) at the time the backup starts and again at the time the backup ends. This allows the backup to capture the relevant parts of the log as well. The relevant part starts with the oldest active transaction at the time of the first recorded LSN and ends with the second recorded LSN.

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

Recovery Models

As I said in Chapter 3 when I discussed database options, the RECOVERY option has three possible values: FULL, BULK_LOGGED, or SIMPLE. The value that you choose determines the size of your transaction log, the speed and size of your transaction log backups (or whether you can make log backups at all), as well as the degree to which you are at risk of losing 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. This 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 2008 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.

In FULL recovery mode, SQL Server also fully logs *CREATE INDEX* operations. 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. Prior to SQL Server 2000, 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 are trade-offs. The biggest trade-off is that the size of your transaction log files can be enormous, so it can take much longer to make log backups than with releases prior to SQL Server 2000.

BULK_LOGGED Recovery Model

The BULK_LOGGED recovery model allows you to restore a database completely in case of media failure and also gives you the best performance and least log space usage for certain bulk operations. In FULL recovery mode, these operations are fully logged, but in BULK_LOGGED recovery mode, they are logged only minimally. This can be much more efficient than normal logging because in general, when you write data to a user database, you must write it to disk twice: once to the log and once to the database itself. This is because the database system uses an undo/redo log so it can roll back or redo transactions when needed. Minimal logging consists of logging only the information that is required to roll back the transaction without supporting point-in-time recovery. These bulk operations include:

- *SELECT INTO*
  - This command always creates a new table in the default filegroup.
- Bulk import operations, including the following:
  - The *BULK INSERT* command
  - The *bcp* executable
- The *INSERT INTO … SELECT* command, when data is selected using the *OPENROWSET(BULK…)* function.
- The *INSERT INTO … SELECT* command, when more than an extent's worth of data is being inserted into a table without nonclustered indexes and the TABLOCK hint is used. If the destination table is empty, it can have a clustered index. If the destination table is already populated, it cannot. (This option can be useful to create a new table in a nondefault filegroup with minimal logging. The *SELECT INTO* command does not allow specifying a filegroup.)
- Partial updates to columns having a large value data type (which will be discussed in Chapter 7, "Special Storage").
- Using the .WRITE clause in the *UPDATE* statement when inserting or appending new data.
- *WRITETEXT* and *UPDATETEXT* statements when inserting or appending new data into LOB data columns (text, ntext, or image).
  - Minimal logging is not used in these cases when existing data is updated.
- Index operations
  - *CREATE INDEX*, including indexes on views
  - *ALTER INDEX REBUILD* or *DBCC DBREINDEX*
  - *DROP INDEX.* The creation of the new heap is minimally logged, but the page deallocation is always fully logged.

When you execute one of these bulk operations, SQL Server logs only the fact that the operation occurred and information

about space allocations. Every data file in a SQL Server 2008 database has at least one special page called a Bulk Changed Map (BCM) page, or also called a Minimally Logged Map (ML Map) page, which is managed much like the GAM and SGAM pages that I discussed in Chapter 3 and the DCM pages that I mentioned previously. 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 transaction log backup. A BCM page is located on the eighth 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 log backup occurs.

Because of the ability to minimally log bulk operations, the operations themselves can potentially be carried out much faster than in FULL recovery mode. However, the speed improvement is not guaranteed. The only guarantee with minimally logged operations is that the log itself is smaller. Minimal logging might actually be slower than fully logged operations in certain cases. Although there are not as many log records to write, with minimal logging SQL Server forces the data pages to be flushed to disk before the transaction commits. This forced flushing of the data pages can be very expensive, especially when the I/O for these pages is random. You can contrast this to full logging, which is always sequential I/O. If you don't have a fast I/O subsystem, it can become very noticeable that minimal logging is slower than full logging.

In general, minimal logging does not mean no logging, and it doesn't minimize logging for all operations. It is a feature that minimizes the amount of logging for the operations described previously, and if you have a high-performance I/O subsystem, performance likely improves as well. But on lower-end machines, minimally logged operations are slower than fully logged operations.

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 contains a full sequential record of all changes to your database.

The trade-off to having a smaller log 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 whenever a checkpoint occurs, which happens at regular, frequent intervals. Therefore, the only types of backups that can be made are those that don't require log backups. These types of backups are full database backups, differential backups, partial full and differential backups, and filegroup backups for read-only filegroups. 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 that it contains are committed or rolled back, and the transactions are no longer needed for recovery from server or transaction failure. In fact, as soon as you change your database to SIMPLE recovery model, the log is truncated.

Keep in mind that SIMPLE logging does not mean no logging. What's "simple" is your backup strategy, because you never need to worry about log backups. However, all operations are logged in SIMPLE mode, even though some operations may have fewer log records in that mode than in FULL mode. A log for a database in SIMPLE mode might not grow as much as a database in FULL mode because the bulk operations discussed earlier in this chapter also are minimally logged in SIMPLE mode. This does not mean you don't have to worry about the size of the log in SIMPLE mode. As in any recovery mode, log records for active transactions cannot be truncated, and neither can log records for any transaction that started after the oldest open transaction. So if you have large or long-running transactions, you still might need lots of log space.

Compatibility with Database Options

Microsoft introduced these recovery models in SQL Server 2000 and intended them to replace the *select into/bulkcopy* and *trunc. log on chkpt.* database options. SQL Server 7.0 and earlier versions required that the *select into/bulkcopy* option be set for you 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 by using the *sp_dboption* procedure, but not by using the *ALTER DATABASE* command. However, with versions later than SQL Server 7.0, changing either of these options using *sp_dboption* also changes your recovery model, and changing your recovery model

changes the value of one or both of these options, as you'll see here. 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 inspect the *sys.databases* view. For example, this query returns the recovery mode and the state of the *AdventureWorks2008* database:

```
SELECT name, database_id, suser_sname(owner_sid) as owner ,
        state_desc, recovery_model_desc
FROM sys.databases
WHERE name = 'AdventureWorks2008'
```

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 changes to BULK_LOGGED. Conversely, if you force the database back into FULL mode by using *ALTER DATABASE,* the value of the *select into/ bulkcopy* option changes. If you're using SQL Server 2008 Standard or Enterprise edition, the *model* database starts in FULL recovery mode, so all your new databases will also be in FULL mode. You can change the mode of the *model* database or any other user database by using the *ALTER DATABASE* command.

To make best use of your transaction log, you can switch between FULL and BULK_LOGGED mode without worrying about your backup scripts failing. Prior to SQL Server 2000, once you performed a *SELECT INTO* command 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, they 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. You might want to 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 simply takes longer and is larger.

You can't easily switch to and from SIMPLE mode if you're trying to maintain a sequence of log backups. Switching into SIMPLE mode is no problem, but when you switch back to FULL or BULK_LOGGED, you need to plan your backup strategy and be aware that there are no log backups up to that point. So when you use the *ALTER DATABASE* command to change from SIMPLE to FULL or BULK_LOGGED, you should first make a complete database backup in order for the change in behavior to be complete. Remember that in SIMPLE recovery mode, your transaction log is truncated at regular intervals. This recovery mode 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 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 need to choose not only a recovery model but also the 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 restore of a database, you must make a full database backup occasionally, to use as a starting point for other types of backups. In addition, you may choose among a differential backup, a log backup, or a combination of both. Here are the characteristics of these last two types, which can help you decide between them.

A differential backup

- Is faster if your environment includes a lot of changes to the same data. It backs up only the most recent change, whereas a log backup captures 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 contains 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 the database media fails, so long as the log is available. This allows 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.

Remember that backups can be created as compressed backups, as briefly discussed in Chapter 1. This can greatly reduce the amount of time and space required to actually create the backup (full, differential, or log) on the backup device. The algorithm for compressing backups is very different than the algorithms used for row or page data compression. I elaborate on the differences in Chapter 7.

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 4-5, which shows a database fully backed up on Sundays. The log is backed up daily, and a differential backup is made on Tuesdays and Thursdays. A drive failure occurs on a 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.

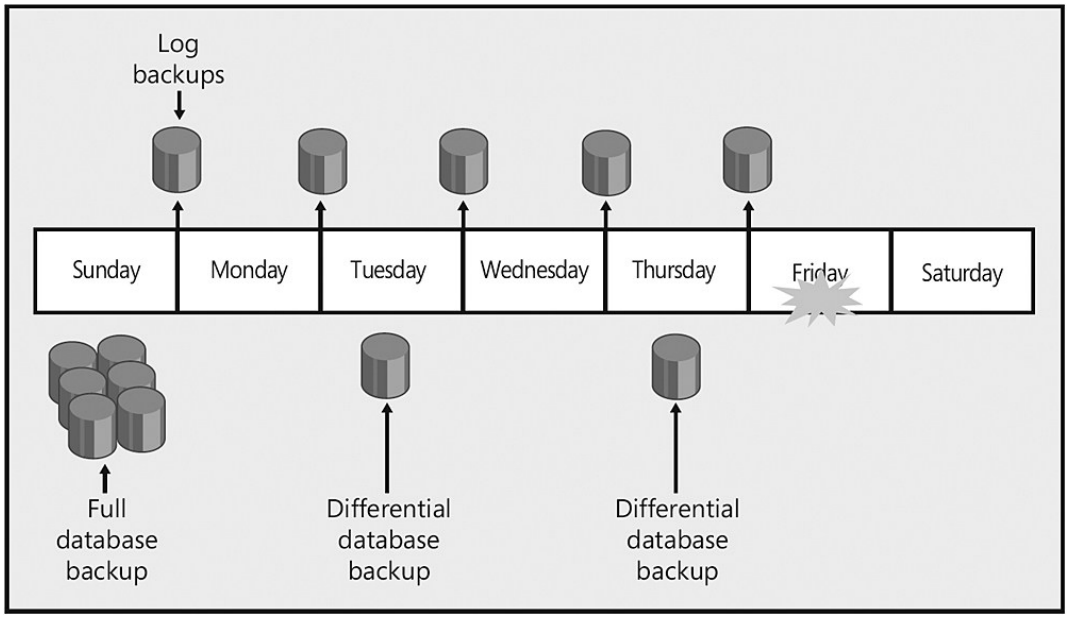

Figure 4-5: The combined use of log and differential backups, which reduces total restore time

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 also need to have available any filegroups containing data inserted by a minimally logged operation.

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 and index extents, as well as all the log blocks, 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 rolls back any incomplete transactions and opens 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 is left in an inconsistent state and is not usable.

If you choose WITH NORECOVERY, you can then apply the next backup. In the scenario depicted in Figure 4-5, 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 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 differential backup is restored, you must restore, in sequence, all the log backups made after the 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 Restore recovery (media recovery) is similar to restart recovery, which I described previously in this chapter, but it is a *REDO-only* operation. It includes an analysis pass to determine how much work might need to be done, and then a roll-forward pass to redo completed transactions and return the database to the state it was in when the backup was complete. Unlike restart 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. Once a *RESTORE WITH RECOVERY* is specified, after the redo pass, the database is restarted and SQL Server runs a restart recovery to undo incomplete transactions. In addition, SQL Server might need to make some adjustments to metadata after the recovery is complete, so no access to the database is allowed until all phases of recovery are finished. In other words, you don't have the option to use "fast" recovery as part of a *RESTORE*.

Backing Up and Restoring Files and Filegroups

SQL Server 2008 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. This also can be useful when you have an isolated media failure on a single drive and you think that restoring the entire database would take too long.

Here are some details to keep in mind about backing up and restoring files and filegroups:

- Individual files and filegroups with the *read-write* property 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. Read-only filegroups and the files in them can be backed up in SIMPLE mode.
- 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 that you back up filegroup *FG1* at 10 A.M. on Monday. The database is still in use, and changes happen to data in *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 that make 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. The tail of the log is backed up using the special WITH NO_TRUNCATE option, but you can also use the NORECOVERY option. When backing up the tail of the log WITH NORECOVERY, the database is put into the RESTORING state and can prevent an accidental background change from interfering with the restore sequence.

You can then restore *FG1* using the *RESTORE DATABASE* command, specifying just filegroup *FG1*. Your database is not in a consistent state because the restored *FG1* has changes only through 10 A.M., and the rest of the database has 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 do not reach that point until you apply the 6 P.M. log backup.

Partial Backups

A partial backup can be based either on a full or a differential backup, but a partial backup does not contain all the filegroups. Partial backups contain all the data in the primary filegroup and all the read-write filegroups. In addition, you can specify that any read-only files also be backed up. If the entire database is marked as read-only, a partial backup contains only the primary filegroup. Partial backups are particularly useful for very large databases (VLDBs) using the SIMPLE recovery model because they allow you to back up only specific filegroups, even without having log backups.

Page Restore

SQL Server 2008 also allows you to restore individual pages. When SQL Server detects a damaged page, it marks it as suspect and stores information about the page in the *suspect_pages* table in the *msdb* database.

Damaged pages can be detected when activities such as the following take place:

- A query needs to read a page.
- *DBCC CHECKDB* or *DBCC CHECKTABLE* is being run.
- *BACKUP* or *RESTORE* is being run.
- You are trying to repair a database with *DBCC DBREPAIR*.

Several types of errors can require a page to be marked as suspect and entered into the *suspect_pages* table. These can include checksum and torn page errors, as well as internal consistency problems, such as a bad page ID in the page header. The column *event_type* in the *suspect_pages* table indicates the reason for the status of the page, which usually reflects the reason the page has been entered into the *suspect_pages* table. *SQL Server Books Online* lists the following possible values for the *event_type* column:

***event_type* value**	**Description**
1	823 error caused by an operating system CDC error or 824 errors other than a bad checksum or a torn page (for example, a bad page ID).
2	Bad checksum.
3	Torn page.
4	Restored. (The page was restored after it was marked as bad.)
5	Repaired. (DBCC repaired the page.)
7	Deallocated by DBCC.

Some of the errors recorded in the *suspect_pages* table might be transient errors, such as an I/O error that occurs because a cable has been disconnected. Rows can be deleted from the *suspect_pages* table by someone with the appropriate permissions, such as someone in the *sysadmin* server role. In addition, not all errors that cause a page to be inserted in the *suspect_pages* table require that the page be restored. A problem that occurs in cached data, such as in a nonclustered index, might be resolved by rebuilding the index. If a *sysadmin* drops a nonclustered index and rebuilds it, the corrupt data, although fixed, is not indicated as fixed in the *suspect_pages* table.

Page restore is specifically intended to replace pages that have been marked as suspect because of an invalid checksum or a torn write. Although multiple database pages can be restored at once, you aren't expected to be replacing a large number of individual pages. If you do have many damaged pages, you should probably consider a full file or database restore. In addition, you should probably try to determine the cause of the errors; if you discover a pending device failure, you should do your full file or database restore to a new location. Log restores must be done after the page restores to bring the new pages up to date with the rest of the database. Just as with file restore, the log backups are applied to the database files containing a page that is being recovered.

In an online page restore, the database is online for the duration of the restore, and only the data being restored is offline. Note that not all damaged pages can be restored with the database online.

Note Online page restore is allowed only in SQL Server 2008 Enterprise Edition.

SQL Server Books Online lists the following basic steps for a page restore:

1. Obtain the page IDs of the damaged pages to be restored. A checksum or torn write error returns the page ID, which is the information needed for specifying the pages. You can also get page IDs from the *suspect_pages* table.
2. Start a page restore with a full, file, or filegroup backup that contains the page or pages to be restored. In the *RESTORE DATABASE* statement, use the PAGE clause to list the page IDs of all the pages to be restored. The maximum number of pages that can be restored in a single file is 1,000.
3. Apply any available differentials required for the pages being restored.
4. Apply the subsequent log backups.

5. Create a new log backup of the database that includes the final LSN of the restored pages—that is, the point at which the last restored page was taken offline. The final LSN, which is set as part of the first restore in the sequence, is the redo target LSN. Online roll-forward of the file containing the page can stop at the redo target LSN. To learn the current redo target LSN of a file, see the *redo_target_lsn* column of *sys.master_files*.
6. Restore the new log backup. Once this new log backup is applied, the page restore is complete and the pages are usable. All the pages that were bad are affected by the log restore. All other pages have a more recent LSN in their page header, and there is nothing to redo. In addition, no UNDO phase is needed for page-level restore.

Partial Restore

SQL Server 2008 lets you do a partial restore of a database in emergency situations. Although the description and the syntax look similar to file and filegroup backup and restore, there is a big difference. With file and filegroup restore, 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 that you don't restore are treated as offline when you attempt to refer to 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 at all. 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 2008, 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 during the update and then after it, the first log backup has the beginning of the transaction and some of the updates, and the second log backup has the remainder of the updates and the commit. Suppose that you then need to restore these log backups after restoring the full database. If you run recovery after restoring the first log backup, the first part of the transaction is rolled back. If you then try to restore the second log backup, it starts in the middle of a transaction, and SQL Server won't have information about what the beginning of the transaction was. You certainly can't recover transactions from this point because their operations might depend on this update that you've partially lost. SQL Server, therefore, does not allow any more restoring to be done. The alternative is not to run recovery to roll back the first part of the transaction and instead to leave the transaction incomplete. SQL Server takes into account that the database is inconsistent and does 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 rolls back incomplete transactions but keeps track of the rolled-back work in the specified file, which is known as a *standby file*. The next restore operation reads the contents of the standby file and redoes the operations that were rolled back, and then it restores the next log. If that restore also specifies WITH STANDBY, incomplete transactions again are rolled back, but a record of those rolled-back transactions is saved. Keep in mind that you can't modify any data if you've restored WITH STANDBY (SQL Server generates 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 is kept) to make the database fully usable.

Summary

In addition to one or more data files, every database in a SQL Server instance has one or more log files that keep track of changes to that database. (Remember that database snapshots do not have log files because no changes are ever made directly to a snapshot.) SQL Server uses the transaction log to guarantee consistency of your data, at both a logical and a physical level. In addition, an administrator can make backups of the transaction log to make restoring a database more efficient. An administrator or database owner can also set a database's recovery mode to determine the level of detail stored in the transaction log.