

SQL Server 2008 Administration IN ACTION

SAMPLE CHAPTER

Rod Colledge

FOREWORD BY KEVIN KLINE





SQL Server 2008
Administration in Action
by Rod Colledge

Chapter 10

brief contents

PART 1 PLANNING AND INSTALLATION..... 1

- 1 ■ The SQL Server landscape 3
- 2 ■ Storage system sizing 12
- 3 ■ Physical server design 31
- 4 ■ Installing and upgrading SQL Server 2008 58
- 5 ■ Failover clustering 78

PART 2 CONFIGURATION..... 93

- 6 ■ Security 95
- 7 ■ Configuring SQL Server 128
- 8 ■ Policy-based management 147
- 9 ■ Data management 168

PART 3 OPERATIONS..... 193

- 10 ■ Backup and recovery 195
- 11 ■ High availability with database mirroring 226
- 12 ■ DBCC validation 260
- 13 ■ Index design and maintenance 280
- 14 ■ Monitoring and automation 330
- 15 ■ Data Collector and MDW 360
- 16 ■ Resource Governor 375
- 17 ■ Waits and queues: a performance-tuning methodology 390

Part 3

Operations

In parts 1 and 2, we covered preinstallation planning, installation, and postinstallation configuration. The remainder of the book will be dedicated to day-to-day operational tasks such as backups, index maintenance, and performance tuning. Let's begin with perhaps the most important of these tasks, backups.

10

Backup and recovery

In this chapter, we'll cover

- Backup types
- Recovery models
- Online piecemeal restore
- Database snapshots
- Backup compression

The importance of backups can't be overstated. During normal activity, it's easy to view backing up databases as an administrative chore that complicates the day and offers little benefit. However, when required in an emergency, the presence of valid backups could make all the difference to an organization's ongoing survival. As DBAs, we have a vital role to play in that process.

Successful backup strategies are those that are designed from a restore perspective—that is, they begin with service level agreements covering data loss and restoration times, and work backwards to derive the backup design. Second only to not performing backups, the biggest backup-related mistake a DBA can make is failing to verify backups. There are countless stories of backup tapes being recalled for recovery before finding out that the backups have been failing for the past few

months (or years!). While the backup may *appear* to have succeeded, how can you be sure until you actually restore it?

In this chapter, we begin with an overview of the various types of backups that can be performed with SQL Server before we look at database recovery models. We then move on to cover online piecemeal restores, expanding on the previous chapter's coverage of filegroups. We then explore the benefits of database snapshots, and we conclude with a new backup feature introduced in SQL Server 2008: backup compression.

10.1 Backup types

Unless you're a DBA, you'd probably define a database backup as a complete copy of a database at a given point in time. While that's *one* type of database backup, there are many others. Consider a multi-terabyte database that's used 24/7:

- How long does the backup take, and what impact does it have on users?
- Where are the backups stored, and what is the media cost?
- How much of the database changes each day?
- If the database failed partway through the day, how much data would be lost if the only recovery point was the previous night's backup?

In considering these questions, particularly for large databases with high transaction rates, we soon realize that simplistic backup strategies limited to full nightly backups are insufficient for a number of reasons, not the least of which is the potential for data loss. Let's consider the different types of backups in SQL Server.

Backup methods

There are many tools and techniques for performing database backups, including various third-party products and database maintenance plans (covered in chapter 14). For the purposes of the examples throughout this chapter, we'll use a T-SQL script approach.

10.1.1 Full backup

Full backups are the simplest, most well understood type of database backup. Like standard file backups (documents, spreadsheets, and so forth), a full backup is a complete copy of the database at a given time. But unlike with a normal file backup, you can't back up a database by simply backing up the underlying .mdf and .ldf files.

One of the classic mistakes made by organizations without appropriate DBA knowledge is using a backup program to back up all files on a database server based on the assumption that the inclusion of the underlying database files (.mdf and .ldf) in the backup will be sufficient for a restore scenario. Not only will this backup strategy be unsuccessful, but those who use such an approach usually fail to realize that fact until they try to perform a restore.

For a database backup to be valid, you must use the `BACKUP DATABASE` command or one of its GUI equivalents. Let's look at a simple example in which we'll back up the AdventureWorks database. Check Books Online (BOL) for the full description of the backup command with all of its various options.

```
-- Full Backup to Disk
BACKUP DATABASE [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks.bak'
WITH INIT
```

You can perform backups in SQL Server while the database is in use and is being modified by users. Such backups are known as online backups. In order for the resultant backup to be restored as a transactionally consistent database, SQL Server includes *part* of the transaction log in the full database backup. Before we cover the transaction log in more detail, let's consider an example of a full backup that's executed against a database that's being actively modified.

Figure 10.1 shows a hypothetical example of a transaction that starts and completes during a full backup, and modifies a page *after* the backup process has read it from disk. In order for the backup to be transactionally consistent, how will the backup process ensure this modified page is included in the backup file? In answering this question, let's walk through the backup step by step. The step numbers in the following list correspond to the steps in figure 10.1.

- 1 When the backup commences, a checkpoint is issued that flushes dirty buffer cache pages to disk.
- 2 After the checkpoint completes, the backup process begins reading pages from the database for inclusion in the backup file(s), including page X.
- 3 Transaction A begins.
- 4 Transaction A modifies page X. The backup has already included page X in the backup file, so this page is now out of date in the backup file.
- 5 Transaction B begins, but won't complete until after the backup finishes. At the point of backup completion, this transaction is the oldest active (uncommitted/incomplete) transaction.

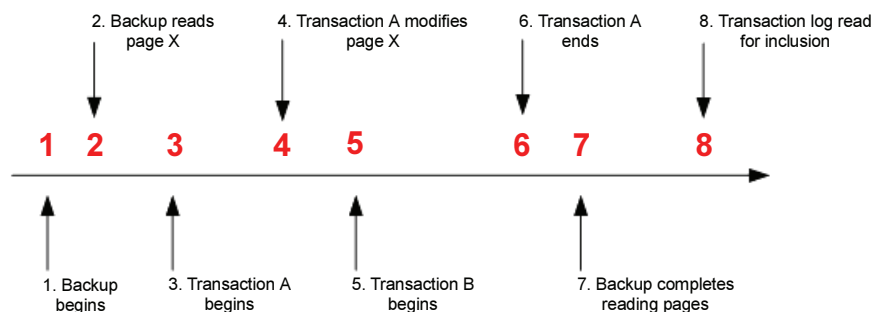


Figure 10.1 Timeline of an online full backup. Based on an example used with permission from Paul S. Randal, managing director of SQLskills.com.

- 6 Transaction A completes successfully.
- 7 The backup completes reading pages from the database.
- 8 As described shortly, the backup process includes *part* of the transaction log in the backup.

If the full backup process didn't include any of the transaction log, the restore would produce a backup that wasn't transactionally consistent. Transaction A's committed changes to page X wouldn't be in the restored database, and because transaction B hasn't completed, its changes would have to be rolled back. By including parts of the transaction log, the restore process is able to roll forward committed changes and roll back uncommitted changes as appropriate.

In our example, once SQL Server completes reading database pages at step 7, it will include all entries in the transaction log since the *oldest* log sequence number (LSN) of one of the following:

- The checkpoint (step 1 in our example)
- The oldest active transaction (step 5)
- The LSN of the last replicated transaction (not applicable in our example)

In our example, transaction log entries since step 1 will be included because that's the oldest of these items. However, consider a case where a transaction starts *before* the backup begins and is still active at the end of the backup. In such a case, the LSN of that transaction will be used as the start point.

This example was based on a blog post from Paul Randal of SQLskills.com. The full post, titled "More on How Much Transaction Log a Full Backup Includes" is available at <http://www.sqlskills.com/BLOGS/PAUL/post/More-on-how-much-transaction-log-a-full-backup-includes.aspx>.

It's important to point out here that even though parts of the transaction log are *included* in a full backup, this doesn't constitute a transaction log *backup*. Another classic mistake made by inexperienced SQL Server DBAs is never performing transaction log backups because they think a full backup will take care of it. A database in full recovery mode (discussed shortly) will maintain entries in the transaction log until it's backed up. If explicit transaction log backups are never performed, the transaction log will continue growing forever (until it fills the disk). It's not unusual to see a 2GB database with a 200GB transaction log!

Finally, when a full backup is restored as shown in our next example, changes since the full backup are lost. In later examples, we'll look at combining a full backup with differential and transaction log backups to restore changes made after the full backup was taken.

```
-- Restore from Disk
RESTORE DATABASE [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks.bak'
WITH REPLACE
```

To reduce the user impact and storage costs of nightly full backups, we can use differential backups.

Multi-file backups

Backing up a database to multiple files can lead to a significant reduction in backup time, particularly for large databases. When you use the T-SQL `BACKUP DATABASE` command, the `DISK =` clause can be repeated multiple times (separated by commas), once for each backup file, as per this example:

```
BACKUP DATABASE [ADVENTUREWORKS2008]
TO
    DISK = 'G:\SQL BACKUP\ADVENTUREWORKS_1.BAK'
    , DISK = 'G:\SQL BACKUP\ADVENTUREWORKS_2.BAK'
    , DISK = 'G:\SQL BACKUP\ADVENTUREWORKS_3.BAK'
```

10.1.2 Differential backup

While a full backup represents the most complete version of the database, performing full backups on a nightly basis may not be possible (or desirable) for a variety of reasons. Earlier in this chapter we used an example of a multi-terabyte database. If only a small percentage of this database changes on a daily basis, the merits of performing a full nightly backup are questionable, particularly considering the storage costs and the impact on users during the backup.

A differential backup, an example of which is shown here, is one that includes all database changes since the last full backup:

```
-- Differential Backup to Disk
BACKUP DATABASE [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks-Diff.bak'
WITH DIFFERENTIAL, INIT
```

A classic backup design is one in which a full backup is performed weekly, with nightly differential backups. Figure 10.2 illustrates a weekly full/nightly differential backup design.

Figure 10.2 Differential backups grow in size and duration the further they are from their corresponding full backup (base).

Day	Backup Type	Includes ...	Contents
Sunday	Full	Everything	
Monday	Differential	Changes since Sunday	
Tuesday	Differential	Changes since Sunday	
Wednesday	Differential	Changes since Sunday	
Thursday	Differential	Changes since Sunday	
Friday	Differential	Changes since Sunday	
Saturday	Differential	Changes since Sunday	

Compared to nightly full backups, a nightly differential with a weekly full backup offers a number of advantages, primarily the speed and reduced size (and therefore storage cost) of each nightly differential backup. However, there comes a point at which differential backups become counterproductive; the further from the full backup, the larger the differential, and depending on the rate of change, it may be quicker to perform a full backup. It follows that in a differential backup design, the frequency of the full backup needs to be assessed on the basis of the rate of database change.

When restoring a differential backup, the corresponding full backup, known as the *base* backup, needs to be restored with it. In the previous example, if we needed to restore the database on Friday morning, the full backup from Sunday, along with the differential backup from Thursday night, would be restored, as in this example:

```
-- Restore from Disk. Leave in NORECOVERY state for subsequent restores
RESTORE DATABASE [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks.bak'
WITH NORECOVERY, REPLACE
GO

-- Complete the restore process with a Differential Restore
RESTORE DATABASE [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks-Diff.bak'
GO
```

Here, we can see the full backup is restored using the `WITH NORECOVERY` option. This leaves the database in a *recovering* state, and thus able to restore additional backups. We follow the restore of the full backup with the differential restore.

As you'll recall from the restore of the full backup shown earlier, without transaction log backups, changes made to the database since the differential backup will be lost.

10.1.3 *Transaction log backup*

A fundamental component of database management systems like SQL Server is the transaction log. Each database has its own transaction log, which SQL Server uses for several purposes, including the following:

- The log records each database transaction, as well as the individual database modifications made within each transaction.
- If a transaction is canceled before it completes, either at the request of an application or due to a system error, the transaction log is used to undo, or *roll back*, the transaction's modifications.
- A transaction log is used during a database restore to roll forward completed transactions and roll back incomplete ones. This process also takes place for each database when SQL Server starts up.
- The transaction log plays a key role in log shipping and database mirroring, both of which will be covered in the next chapter.

Regular transaction log backups, as shown here, are crucial in retaining the ability to recover a database to a point in time:

```
-- Transaction Log Backup to Disk
BACKUP LOG [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks-Trn.bak'
WITH INIT
```

As you can see in figure 10.3, each transaction log backup forms part of what's called a *log chain*. The head of a log chain is a full database backup, performed after the

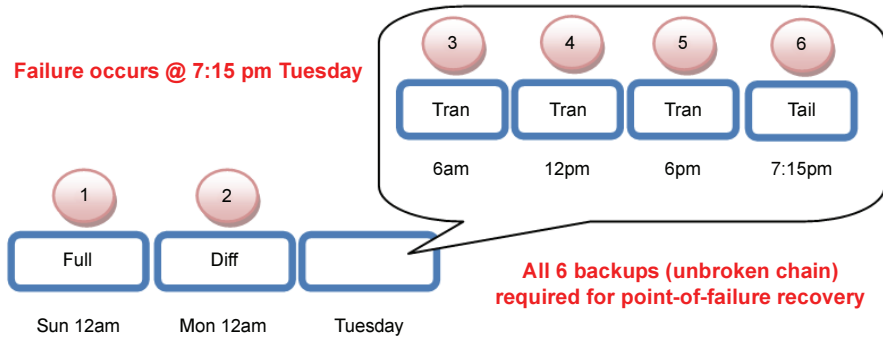


Figure 10.3 An unbroken chain of backups is required to recover to the point of failure.

database is first created, or when the database’s recovery model, discussed shortly, is changed. After this, each transaction log backup forms a part of the chain. To restore a database to a point in time, an unbroken chain of transaction logs is required, from a full backup to the required point of recovery.

Consider figure 10.3. Starting at point 1, we perform a full database backup, after which differential and transaction log backups occur. Each of the backups serves as part of the chain. When restoring to a point in time, an unbroken sequence of log backups is required. For example, if we lost backup 4, we wouldn’t be able to restore past the end of backup 3 at 6 a.m. Tuesday. Attempting to restore the transaction log from log backup 5 would result in an error message similar to that shown in figure 10.4.

In addition to protecting against potential data loss, regular log backups limit the growth of the log file. With each transaction log backup, certain log records, discussed in more detail shortly, are removed, freeing up space for new log entries. As covered earlier, the transaction log in a database in full recovery mode will continue growing indefinitely until a transaction log backup occurs.

The frequency of transaction log backups is an important consideration. The two main determining factors are the rate of database change and the sensitivity to data loss.

TRANSACTION LOG BACKUP FREQUENCY

Frequent transaction log backups reduce the exposure to data loss. If the transaction log disk is completely destroyed, then all changes since the last log backup will be lost.

```

restore log [AdventureWorks2008]
from disk = 'G:\SQL Backup\aw-log-2.bak'
with norecovery

```

Messages

```

Msg 4305, Level 16, State 1, Line 1
The log in this backup set begins at LSN 241000000227300001, which is too recent to apply to the database.
An earlier log backup that includes LSN 241000000223500001 can be restored.
Msg 3013, Level 16, State 1, Line 1
RESTORE LOG is terminating abnormally.

```

Figure 10.4 Attempting to restore an out-of-sequence transaction log

Assuming a transaction log backup was performed 15 minutes before the disk destruction, the maximum data loss would be 15 minutes (assuming the log backup file isn't contained on the backup disk!). In contrast, if transaction log backups are only performed once a day (or longer), the potential for data loss is large, particularly for databases with a high rate of change.

The more frequent the log backups, the more restores will be required in a recovery situation. In order to recover up to a given point, we need to restore each transaction log backup between the last full (or differential) backup and the required recovery point. If transaction log backups were taken every minute, and the last full or differential backup was 24 hours ago, there would be 1,440 transaction log backups to restore! Clearly, we need to get the balance right between potential data loss and the complexity of the restore. Again, the determining factors are the rate of database change and the maximum allowed data loss, usually defined in a service level agreement.

In a moment we'll run through a point-in-time restore, which will illustrate the three backup types working together. Before we do that, we need to cover tail log backups.

TAIL LOG BACKUPS

When restoring a database that's currently attached to a server instance, SQL Server will generate an error¹ unless the *tail* of the transaction log is first backed up. The tail refers to the section of log that hasn't been backed up yet—that is, new transactions since the last log backup.

A tail log backup is performed using the `WITH NORECOVERY` option, which immediately places the database in the restoring mode, guaranteeing that the database won't change after the tail log backup and thus ensuring that all changes are captured in the backup.

WITH NO_TRUNCATE

Backing up the tail of a transaction log using the `WITH NO_TRUNCATE` option should be limited to situations in which the database is damaged and inaccessible. The `COPY_ONLY` option, covered shortly, should be used in its place.

When restoring up to the point of failure, the tail log backup represents the very last transaction log backup, with all restores preceding it performed using the `WITH NORECOVERY` option. The tail log is then restored using the `WITH RECOVERY` option to recover the database up to the point of failure, or a time before failure using the `STOPAT` command.

So let's put all this together with an example. In listing 10.1, we first back up the tail of the log before restoring the database to a point in time. We begin with restoring the full and differential backups using the `WITH NORECOVERY` option, and then roll forward the transaction logs to a required point in time.

¹ Unless the `WITH REPLACE` option is used.

Listing 10.1 Recovering a database to a point in time

```
-- Backup the tail of the transaction log
BACKUP LOG [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks-Tail.bak'
WITH INIT, NORECOVERY

-- Restore the full backup
RESTORE DATABASE [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks.bak'
WITH NORECOVERY
GO

-- Restore the differential backup
RESTORE DATABASE [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks-Diff.bak'
WITH NORECOVERY
GO

-- Restore the transaction logs
RESTORE LOG [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks-Trn.bak'
WITH NORECOVERY
GO

-- Restore the final tail backup, stopping at 11.05AM
RESTORE LOG [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks-Tail.bak'
WITH RECOVERY, STOPAT = 'June 24, 2008 11:05 AM'
GO
```

As we covered earlier, the `NO_TRUNCATE` option of a transaction log backup, used to perform a backup without removing log entries, should be limited to situations in which the database is damaged and inaccessible. Otherwise, use the `COPY_ONLY` option.

10.1.4 `COPY_ONLY` backups

Earlier in this chapter we defined a log chain as the sequence of transaction log backups from a given *base*. The base for a transaction log chain, as with differential backups, is a full backup. In other words, before restoring a transaction log or differential backup, we first restore a full backup that preceded the log or differential backup.

Take the example presented earlier in figure 10.3, where we perform a full backup on Sunday night, nightly differential backups, and six hourly transaction log backups. In a similar manner to the code in listing 10.1, to recover to 6 p.m. on Tuesday, we'd recover Sunday's full backup, followed by Tuesday's differential and the three transaction log backups leading up to 6 p.m.

Now let's assume that a developer, on Monday morning, made an additional full backup, and moved the backup file to their workstation. The differential restore from Tuesday would now fail. Why? A differential backup uses a Differential Changed Map (DCM) to track which extents have changed since the last full backup. The DCM in the differential backup from Tuesday now relates to the full backup made by the

developer on Monday morning. In our restore code, we're not using the full backup from Monday—hence the failure.

Now, there are a few ways around this problem. First, we have an unbroken transaction log backup sequence, so we can always restore the full backup, followed by *all* of the log backups since Sunday. Second, we can track down the developer and ask him for the full backup and hope that he hasn't deleted it!

To address the broken chain problem as outlined here, `COPY_ONLY` backups were introduced in SQL Server 2005 and fully supported in 2008.² A `COPY_ONLY` backup, supported for both full and transaction log backups, is used in situations in which the backup sequence shouldn't be affected. In our example, if the developer performed the Monday morning full backup as a `COPY_ONLY` backup, the DCM for the Tuesday differential would still be based on our Sunday full backup. In a similar vein, a `COPY_ONLY` transaction log backup, as in this example, will back up the log without truncation, meaning that the log backup chain will remain intact without needing the additional log backup file:

```
-- Perform a COPY ONLY Transaction Log Backup
BACKUP LOG [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks-Trn_copy.bak'
WITH COPY_ONLY
```

When discussing the different backup types earlier in the chapter, we made several references to the database recovery models. The recovery model of a database is an important setting that determines the usage of the transaction log and the exposure to data loss during a database restore.

10.2 Recovery models and data loss exposure

When a database is created, the recovery model is inherited from the *model* database. You can modify the recovery model in Management Studio or use the `ALTER DATABASE` statement, as shown here:

```
-- Set the Recovery Model to BULK_LOGGED
ALTER DATABASE [ADVENTUREWORKS2008]
SET RECOVERY BULK_LOGGED
```

There are three different recovery models: *simple*, *full*, and *bulk logged*.

10.2.1 Simple recovery model

A database in the simple recovery model will automatically truncate (remove) committed transactions from the log at each checkpoint operation. As a result, no transaction log backups are required in limiting the growth of the log, so maintenance operations are simplified.

² Management Studio in SQL Server 2008 includes enhanced support for `COPY_ONLY` backups with GUI options available for this backup type. Such options were absent in SQL Server 2005, which required a T-SQL script approach.

Despite the reduction in maintenance overhead, the major downside of the simple recovery model is the inability to recover a database to a point in time. As such, the only recovery options are to recover to the previous full or differential backup. This strategy may lead to significant data loss depending on the amount of change since the last full/differential backup.

The simple recovery model is typically used in development and test environments where recovering to the last full or differential backup is acceptable. In such environments, the potential for some data loss is accepted in return for reduced maintenance requirements by avoiding the need to execute and store transaction log backups.

Simple logging vs. no logging

Don't confuse the simple recovery model for the (nonexistent) *no logging* model. Regardless of the recovery model, transactions are logged by SQL Server in order to maintain database integrity in the event of a transaction rollback or sudden server shutdown.

Finally, long-running transactions can still cause significant growth in the transaction log of databases in the simple recovery model. Log records generated by an incomplete transaction can't be removed, nor can any completed transactions that started *after* the oldest open transaction. For example, in figure 10.5, even though transaction D has completed, it can't be removed as it started *after* the incomplete transaction C.

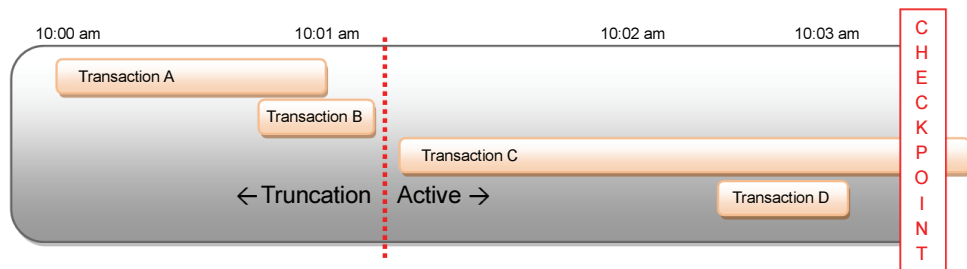


Figure 10.5 Log truncation can't remove log records for active transactions or records from completed transactions that began after the oldest active transaction.

The simple recovery model has further downsides: features such as transaction log shipping, covered in the next chapter, can't be used with this recovery model. For compatibility with SQL Server's full feature set and to minimize data loss, we use the full recovery model.

10.2.2 Full recovery model

A database in the full recovery model will log and retain *all* transactions in the transaction log until the log is backed up, at which point committed transactions will be removed from the log, subject to the same rule that we saw in figure 10.5. Regular

transaction log backups are crucial in limiting the growth of the transaction log in a database in the full recovery model.

As well as recording update, insert, and delete statements, the full recovery model will record index creation and maintenance operations, Bulk Copy Process (BCP) commands, and bulk inserts. As a result, the size of transaction logs (and therefore the backup time) can grow very quickly with the full recovery model, and is therefore an important consideration when using log shipping and/or mirroring. We'll cover this in more detail in later chapters when we address index maintenance techniques.

Disaster recovery plan

A good disaster recovery (DR) plan considers a wide variety of potential disasters, from small events such as corrupted log files and accidentally dropping a table, right through to large environmental disasters such as fires and earthquakes. A crucial component of any DR plan is a well-documented and well-understood backup and restore plan. Perhaps the best way to validate a DR plan is to simulate various disasters on a random/unannounced basis, similar to a fire drill, with each DBA talking it in turns to practice the recovery process. Not only will this ensure documentation is up to date and well understood by all team members, it will liven up the day, and add some competitive spark to the DBA team!

A common technique used when bulk-loading data into a database in the full recovery model is to switch the database to the Bulk_Logged model, discussed next, prior to the load.

10.2.3 *Bulk_Logged recovery model*

When performing large bulk-load operations into a database in the full recovery model, each data and index record modified by the bulk-load process is logged by SQL Server. For very large loads, this can have a significant impact on the load performance.

Under the Bulk_Logged model, SQL Server uses a *Bulk Changed Map* (BCM) to record which *extents*³ the load modified. Unlike the full recovery model, the individual records affected by the bulk load aren't logged. As a result, bulk loads can be significantly quicker than under the full recovery model.

The trade-offs of the Bulk_Logged model are significant: when a transaction log backup occurs after a bulk-load operation, SQL Server includes the entire contents of each extent touched by the bulk-load process, even if only a small portion of the extent was actually modified. As a result, the transaction log backup size can be massive, potentially almost as big as the entire database, depending on the amount of modified extents.

³ An *extent* is a collection of eight 8K pages.

The other downside to this recovery model is the inability to restore a transaction log containing bulk-load operations to a point in time. Given these limitations, it's generally recommended that the Bulk_Logged model be used as a temporary setting for the period of the bulk load before switching back to the full recovery model. Making a transaction log backup before and after entering and leaving the Bulk_Logged model will ensure maximum data protection for point-in-time restores while also benefiting from increased performance during the bulk-load operation(s).

Before looking at the backup recovery process in more detail, let's consider some additional backup options at our disposal.

10.3 Backup options

SQL Server 2008 includes a rich array of options that can be employed as parts of a customized backup and recovery strategy. In this section, we'll consider three such options: *checksums*, *backup mirroring*, and *transaction log marks*. But before we cover these options, let's have a look at an important part of any backup strategy: the backup location and retention policy.

10.3.1 Backup location and retention policy

A key component of a well-designed backup strategy is the location of the backups: disk or tape (or both). Let's consider each of these in turn before looking at a commonly used backup retention policy.

TAPE

Historically, organizations have chosen tape media as a backup destination in order to reduce the cost of online storage while retaining backups for long periods of time. However, a *tape-only* approach to backups presents a number of challenges:

- Tape systems typically have a higher failure rate when compared to disk.
- Typically, tapes are rotated offsite after a predefined period, sometimes as soon as the morning after the backup. Should the backup be required for restore, there may be a time delay involved in retrieving the tape for restore.
- Depending on the tape system, it may be difficult/cumbersome to restore a tape backup to a different server for restoration verification, or to use it as a source for DBCC checks or other purposes.

In addressing these concerns, disk backups are frequently used, although they too have some challenges to overcome.

DISK

Due to some of the limitations with the tape-only approach, backup verification, whereby backups are restored on a regular basis to ensure their validity, are often skipped. As a result, problems are often discovered for the first time when a *real* restore is required.

In contrast to tape, disk-based backups offer the following advantages:

- When required for a restore, they are immediately available.
- Disk media is typically more reliable than tape, particularly when RAID protected.

- Disk-based backups can be easily copied to other servers when required, making the verification process much simpler compared with a typical tape-based system.

Despite its advantages, a disk-based backup approach has some drawbacks. The main one is the extra disk space (and associated cost) required for the backup files. Further, the cost advantage of tape is fully realized when considering the need to store a history of backups—for example, daily backups for the last 30 days, monthly backups for the past 12 months, and yearly backups for the last 7 years. Storing all of these backups on disk is usually more expensive compared to a tape-based system, not to mention the risk of losing all of the disk backups in an environmental disaster.

With the introduction of third-party backup compression tools and the inclusion of backup compression as a standard feature of SQL Server 2008 (Enterprise edition), the cost of disk storage for backups is significantly reduced, but the overall cost is still typically higher than a tape-based system.

In addressing the negative aspects of both tape and disk, a common approach is to combine both methods in what's known as a *disk then tape* approach.

DISK THEN TAPE

As shown in figure 10.6, the ideal backup solution is to combine both disk and tape backups in the following manner:

- 1 Database backups are performed to disk.
- 2 Later in the day/night, the disk backup files are archived to tape in the same manner as other files would be backed up (documents, images and so forth).
- 3 Typical restore scenarios use the most recent backup files on disk. After a number of days, the oldest disk-based backup files are removed in order to maintain a sliding window; for example, the past 5 days of backups are stored on disk.
- 4 If older backups are required, they can be sourced from tape.

The advantages of such a system are numerous:

- Backups are stored in two locations (disk and tape), thus providing an additional safety net against media failure.

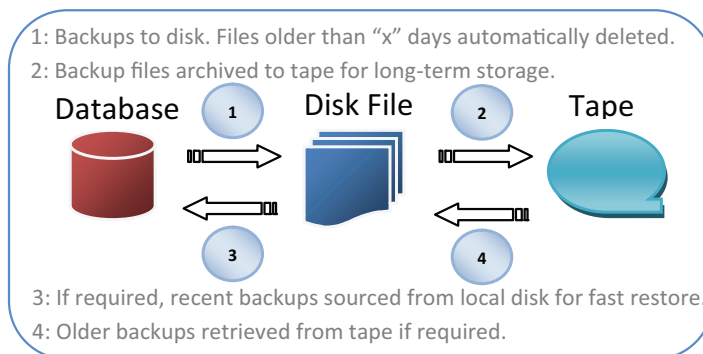


Figure 10.6 The disk then tape backup methodology provides fast restore, dual backup protection, and long-term archive at a moderate cost.

- The most common restore scenario, typically that of the previous night's backup, is available on disk for fast restore without requiring tapes to be requested from offsite.
- A full history of backups is available for restore from tape.
- The cost of the overall system is reduced, while still providing all of the advantages of disk-based backups for the most common restore scenarios.

A variation on this approach is using SAN-based backup solutions. In chapter 3 we covered the benefits that SANs provide in disaster recovery scenarios. Most of the enterprise-class SANs available today provide several methods of *snapping*, or *cloning*, LUNs in order to provide near instant backup/restore solutions. Once snapped, the cloned LUN can be archived to tape, thereby providing long-term storage like the disk then tape approach. If using these backup types, take care to ensure the backup method used is SQL Server compatible and enables transaction log roll forwards for point-in-time recovery.

Regardless of the backup destination, an important consideration is how long to retain the backups before deleting or overwriting them.

BACKUP RETENTION

Assuming the disk then tape backup method is used, the retention period for each location needs to be considered separately. For the disk backups, the retention period is dependent on the backup model. For example, if a weekly full, nightly differential system is in place, then the weekly backup would need to be retained on disk for the whole week for use with the previous night's differential backup. If disk space allows, then additional backups can be retained on disk as appropriate.

In considering the tape rotation policy (how long to keep a tape before overwriting it), the classic rotation policy typically used is the *grandfather-father-son* (GFS) system, whereby 22 tapes are used per year.

The GFS tape rotation policy, as shown in table 10.1, uses 6 sons, 3 fathers, and 13 grandfathers (52 weeks per year divided by 4-week periods) for a total of 22 tapes per year. Optionally, one of the grandfather tapes can be retained as a yearly backup tape for a period of years.

Table 10.1 Grandfather-father-son tape rotation policy

Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	Son1	Son2	Son3	Son4	Son5	Son6	Father1
2	Son1	Son2	Son3	Son4	Son5	Son6	Father2
3	Son1	Son2	Son3	Son4	Son5	Son6	Father3
4	Son1	Son2	Son3	Son4	Son5	Son6	Grandfather-x

Regardless of the disk location and retention period, ensuring backups are valid is an important consideration. Short of actually restoring each backup, one of the options available for detecting media failure is to use *backup checksums*.

10.3.2 Backup checksums

One of the features introduced in SQL Server 2005 was the ability for backups to verify the validity of pages as part of the backup process, and for the backup itself to include a checksum.

When using the optional⁴ `WITH CHECKSUM` option of the `BACKUP` command as shown here, the backup process verifies the checksum of each page as the backup is performed, assuming the `PAGE_VERIFY` database option, covered in more detail in chapter 12, is set to `CHECKSUM` (which is the default).

```
-- Verify Page Validity during backup with CHECKSUM
BACKUP DATABASE [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks.bak'
WITH CHECKSUM
```

The `PAGE_VERIFY` option calculates and stores a checksum value for each database page written to disk. When read from disk, the checksum is verified against the page read and used to alert the presence of suspect pages.

The `WITH CHECKSUM` option of the `BACKUP` command calculates and verifies the checksum value of each page as it's read by the backup process. If a checksum error occurs, the backup fails, unless the `CONTINUE_AFTER_ERROR` option is used. In that case, the backup is flagged as containing errors and the suspect page(s) are marked in the `suspect_pages` table in the `msdb` database.

In addition to verifying the checksum of each page, the `WITH CHECKSUM` option calculates a checksum for the entire backup process. When a database is restored from a backup created with the checksum option, the restore process verifies the checksums as part of the restore process, unless the `NO_CHECKSUM` option is used. If a checksum error is found as part of the restore, the restore will fail, unless the `CONTINUE_AFTER_ERROR` option is used.

Although backup checksums provide additional confidence in the validity of the backups, they do introduce additional CPU overhead during the backup process. Before enabling this option, ensure the overhead is measured on a test system, particularly in cases where the additional overhead may extend the backup window beyond the desired time frame. That being said, the additional confidence this option provides is well worth the typically small CPU overhead.

Another technique commonly used for backup assurance is the mirroring option.

10.3.3 Backup mirroring

There is no such thing as too many backups. One of the optional backup clauses is `MIRROR TO`. Here's an example:

```
-- Mirror the backup to a separate backup server using a UNC path
BACKUP DATABASE [AdventureWorks2008]
TO DISK = N'G:\SQL Backup\AdventureWorks-20080701.bak'
MIRROR TO DISK = '\\BACKUP-SERVER\SQL-Backups\AdventureWorks-20080701.bak'
WITH FORMAT
```

⁴ Enabled by default on compressed backups.

The `MIRROR TO` clause allows a backup to be streamed to multiple destinations. The typical use of this option is for making a duplicate backup on a file server using a Universal Naming Convention (UNC) path to a file share (in the previous example, `\\BACKUP-SERVER\SQL-Backups`). This option provides multiple advantages:

- Additional backups for protection against media failure.
- Different retention periods for different locations; for example, the file server backups can be retained for a longer period on disk when compared to the backup file on the database server.
- The tape archive process can archive from the file share rather than the database server. Not only does this reduce the additional load the tape archive process places on the database server, it also avoids the need for tape drivers and associated software to be installed on the database server.

In concluding this section, let's take a look at the challenge of coordinating backups across multiple databases.

10.3.4 Transaction log marks

A common backup requirement is for coordinated backups across multiple databases. This is usually a requirement for the restore process rather than the backup—when a database is restored, all associated databases must be restored to exactly the same point.

Synchronized restores are enabled using *transaction log marks*. Before we take a look at using them in a restore scenario, let's see how they're used in recovering from an unintended action. Consider the following statement, which increases product prices by 2 percent:

```
-- Update all prices by 2%
BEGIN TRANSACTION updatePrices WITH MARK 'Updating Prices Now';
    UPDATE Products
        SET Price = Price * 1.02
COMMIT TRANSACTION updatePrices
```

Let's imagine we only intended to update *some* products, not *all* of them, as shown in the previous statement. Short of running additional commands to roll back the price increase (and other flow-on effects), we'd be looking at a database restore, but if we can't remember the time of the update, a transaction log recovery using the `STOPAT` option won't help.

One of the optional clauses we used in the update price transaction was `WITH MARK`, and we can use that in a restore command. After performing a restore of a full backup in `NORECOVERY` mode, we can then restore a transaction log backup made after the transaction to the point immediately before the mark, using the `STOPBEFOREMARK` option:

```
-- After restoring the full backup, roll forward the transaction log
-- Use the STOPBEFOREMARK option to stop before the marked transaction
RESTORE LOG [AdventureWorks2008]
FROM DISK = N'G:\SQL Backup\AdventureWorks-log.bak'
WITH RECOVERY, STOPBEFOREMARK = 'updatePrices'
GO
```

Now that's all well and good (and very handy), but how does that help us with coordinating backups and restores across multiple databases? Well, by encapsulating statements that update multiple databases within a single marked transaction, we can achieve the desired result (see listing 10.2).

Listing 10.2 Marking multiple transaction logs for coordinated restores

```
-- Use a dummy transaction to mark multiple databases
-- If required, each database can be restored to the same point in time
BEGIN TRANSACTION backupMark WITH MARK
    UPDATE db1.dbo.dummytable set coll = 1
    UPDATE db2.dbo.dummytable set coll = 1
    -- other databases here ...
COMMIT TRANSACTION backupMark
```

By executing a simple update statement in multiple databases *within one transaction*, we're marking the transaction log of each database at the same time. Such an update statement could be executed immediately before transaction log backups are performed, thus enabling the backups to be restored to the same point in time using the `STOPBEFOREMARK` that we saw earlier. Bear in mind, however, that data entered in the databases *after* this transaction will be lost, and this is an important consideration in a coordinated restore scenario.

Using transaction marks to enable synchronized restores across multiple databases is one example of using backup/restore features beyond the basics. While a basic backup/restore approach may suffice for small databases, it's insufficient for very large databases (VLDBs). In the previous chapter, we covered the use of filegroups as a mechanism for enabling enhanced administration options. We also explored a best practice whereby user objects are placed on secondary filegroups so that the only objects in the primary filegroup are system objects. Let's take a look at that process in more detail, and see how it can be used to minimize the user impact of a restoration process.

10.4 Online piecemeal restores

Consider a very large, multi-terabyte database in use 24/7. One of the challenges with databases of this size is the length of time taken for various administration tasks such as backups and the effect such operations have on database users.

One of the advantages of using multiple filegroups is that we're able to back up individual filegroups instead of (or as well as) the entire database. Such an approach not only minimizes the user impact of the backup operation, but it also enables *online piecemeal restores*, whereby parts of the database can be brought online and available for user access while other parts are still being restored.⁵ In contrast, a traditional restore process would require users to wait for the entire database to restore before being able to access it, which for a VLDB could be quite a long time.

In this section we'll walk through the process of an online piecemeal restore using filegroups. Online restores can also be performed at the individual page level, and we'll take a look at that in chapter 12 when we cover the `DBCC` tool.

⁵ Online restores are available in the Enterprise version of SQL Server 2005 and 2008 only.

The database used for our examples is structured as shown in listing 10.3. This code creates a database with three filegroups.

Listing 10.3 Creating a database with multiple filegroups for online restore

```
-- Create "Sales" database with 3 secondary filegroups
-- Each filegroup has 2 files and 1 table

CREATE DATABASE [Sales] ON PRIMARY (
    NAME = N'Sales'
    , FILENAME = N'E:\SQL Data\Sales.mdf'
    , SIZE = 51200KB
    , FILEGROWTH = 1024KB
)

, FILEGROUP [FG1] (
    NAME = N'Sales1'
    , FILENAME = N'E:\SQL Data\Sales1.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
), (
    NAME = N'Sales2'
    , FILENAME = N'E:\SQL Data\Sales2.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
)

, FILEGROUP [FG2] (
    NAME = N'Sales3'
    , FILENAME = N'E:\SQL Data\Sales3.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
), (
    NAME = N'Sales4'
    , FILENAME = N'E:\SQL Data\Sales4.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
)

, FILEGROUP [FG3] (
    NAME = N'Sales5'
    , FILENAME = N'E:\SQL Data\Sales5.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
), (
    NAME = N'Sales6'
    , FILENAME = N'E:\SQL Data\Sales6.ndf'
    , SIZE = 51200KB
    , FILEGROWTH = 10240KB
)

LOG ON (
    NAME = N'Sales_log'
    , FILENAME = N'F:\SQL Log\Sales_log.ldf'
    , SIZE = 10240KB
    , FILEGROWTH = 10%
)

GO
```

```

-- Set FG1 to be the default filegroup
ALTER DATABASE [Sales]
MODIFY FILEGROUP [FG1] DEFAULT
GO

USE [SALES]
GO

-- Create a table on each filegroup
CREATE TABLE dbo.Table_1 (
    Col1 nchar(10) NULL
) ON FG1
GO

CREATE TABLE dbo.Table_2 (
    Col1 nchar(10) NULL
) ON FG2
GO

CREATE TABLE dbo.Table_3 (
    Col1 nchar(10) NULL
) ON FG3
GO

```

As you can see in listing 10.3, we've created a database with three filegroups and one table on each. We've also ensured that user objects won't be created in the primary filegroup by marking the secondary filegroup, FG1, as the default. Listing 10.4 sets up the basis for our restore by seeding the tables and making a filegroup backup of the primary and secondary filegroups. For this example, all of the filegroup backups occur in sequence, but in a real example, we'd perform the filegroup backups over a number of nights to reduce the nightly backup impact. Once the filegroup backups are complete, we'll modify some data for a transaction log backup in a later step.

Listing 10.4 Filegroup backups

```

-- Seed tables
INSERT table_1
VALUES ('one')
GO

INSERT table_2
VALUES ('two')
GO

INSERT table_3
VALUES ('three')
GO

-- Take FileGroup Backups
BACKUP DATABASE [Sales]
FILEGROUP = N'PRIMARY'
TO DISK = N'G:\SQL Backup\Sales_Primary_FG.bak'
WITH INIT
GO

```

```

BACKUP DATABASE [Sales]
FILEGROUP = N'FG1'
TO DISK = N'G:\SQL Backup\Sales_FG1_FG.bak'
WITH INIT
GO

BACKUP DATABASE [Sales]
FILEGROUP = N'FG2'
TO DISK = N'G:\SQL Backup\Sales_FG2_FG.bak'
WITH INIT
GO

BACKUP DATABASE [Sales]
FILEGROUP = N'FG3'
TO DISK = N'G:\SQL Backup\Sales_FG3_FG.bak'
WITH INIT
GO

-- Modify data on FG2
INSERT table_2
VALUES ('two - two')
GO

```

At this point, let's imagine that the disk(s) containing the filegroups is completely destroyed, but the transaction log disk is okay. Restoring a multi-terabyte database as a single unit would take a fair amount of time, during which the entire database would be unavailable. With filegroup restores, what we can do is prioritize the restores in order to make the most important data available as soon as possible. So for this example, let's imagine that filegroup 2 was the most important filegroup from a user perspective. Let's get filegroup 2 back up and running first (see listing 10.5).

Listing 10.5 Online piecemeal restore: restoring the most important filegroup first

```

-- Disaster at this point. Prioritize Restore of Filegroup 2
USE MASTER
GO

-- Start by performing a tail backup
BACKUP LOG [Sales]
TO DISK = N'G:\SQL Backup\Sales_log_tail.bak'
WITH NORECOVERY, NO_TRUNCATE
GO

-- recover Primary and FG2
RESTORE DATABASE [Sales]
FILEGROUP='Primary'
FROM DISK = N'G:\SQL Backup\Sales_Primary_FG.bak'
WITH PARTIAL, NORECOVERY

RESTORE DATABASE [Sales]
FILEGROUP='FG2'
FROM DISK = N'G:\SQL Backup\Sales_FG2_FG.bak'
WITH NORECOVERY

RESTORE LOG [Sales]

```

```

FROM DISK = N'G:\SQL Backup\Sales_log_tail.bak'
WITH RECOVERY
GO

-- At this point the database is up and running for Filegroup 2 only
-- Other filegroups can now be restored in the order required

```

As shown in listing 10.5, the first step in performing a piecemeal restore is to back up the tail of the transaction log. This will enable us to restore up to the point of failure. Once this backup is completed, we can then start the restore process by restoring the primary filegroup. According to our best practice, this is very small as it contains system objects only. As a result, the primary filegroup restore is quick, and as soon as it completes, the database is online and available for us to prioritize the remainder of the filegroup restores.

In line with our priorities, we proceed with a restore of the FG2 filegroup. The last statement restores the transaction log tail backup, which rolls forward transactions for FG2. At this point, FG2 is online and available for users to query. Attempting to query tables 1 and 3 at this moment will fail as these filegroups are offline pending restore. An error message will appear when you attempt to access these tables:

```

Msg 8653, Level 16, State 1, Line 1
The query processor is unable to produce a plan for the table or view
'table_3' because the table resides in a filegroup which is not online.

```

Let's recover the remaining filegroups now, as shown in listing 10.6.

Listing 10.6 Online piecemeal restore for remaining filegroups

```

-- restore FG1
RESTORE DATABASE [Sales]
FILEGROUP='FG1'
FROM DISK = N'G:\SQL Backup\Sales_FG1_FG.bak'
WITH NORECOVERY

RESTORE LOG [Sales]
FROM DISK = N'G:\SQL Backup\Sales_log_tail.bak'
WITH RECOVERY
GO

-- restore FG3
RESTORE DATABASE [Sales]
FILEGROUP='FG3'
FROM DISK = N'G:\SQL Backup\Sales_FG3_FG.bak'
WITH NORECOVERY

RESTORE LOG [Sales]
FROM DISK = N'G:\SQL Backup\Sales_log_tail.bak'
WITH RECOVERY
GO

```

Listing 10.6 assumed *all* the filegroups were damaged and needed to be restored. Should some of the filegroups be undamaged, then a restore for those filegroups is unnecessary. Let's imagine that filegroups 1 and 3 were undamaged. After FG2 is

restored, we can bring the remaining filegroups online with a simple recovery statement such as this:

```
RESTORE DATABASE [Sales] FILEGROUP='FG1', FILEGROUP='FG3' WITH RECOVERY
```

While normal full backup/restores on single filegroup databases may be acceptable for small to medium databases, very large databases require more thought to reduce the backup impact and minimize user downtime during recovery scenarios. By placing user objects on secondary filegroups, filegroup backups and the online piecemeal restore process enable both of these goals to be met.

As we covered earlier, online restores are available only in the Enterprise edition of SQL Server 2005 and 2008. Another Enterprise-only feature is the *database snapshot*, which we explore next.

10.5 Database snapshots

A common step in deploying changes to a database is to take a backup of the database prior to the change. The backup can then be used as a rollback point if the change/release is deemed a failure. On small and medium databases, such an approach is acceptable; however, consider a multi-terabyte database: how long would the backup and restore take either side of the change? Rolling back a simple change on such a large database would take the database out of action for a considerable period of time.

Database snapshots, not to be confused with snapshot backups,⁶ can be used to address this type of problem, as well as provide additional functionality for reporting purposes.

First introduced in SQL Server 2005, and only available in the Enterprise editions of SQL Server, snapshots use a combination of Windows *sparse files* and a process known as *copy on write* to provide a point-in-time copy of a database. After the snapshot has been created, a process typically taking only a few seconds, modifications to pages in the database are delayed to allow a copy of the affected page to be posted to the snapshot. After that, the modification can proceed. Subsequent modifications to the same page proceed without delay. Initially empty, the snapshot grows with each database modification.

Sparse files

Database snapshots are created on the NTFS file system, which provides the necessary *sparse file* support. Unlike traditional files, sparse files only occupy space on disk when data is actually written to them, with the size of the file growing as more data is added. As a result, very large files can be created quickly, even on file systems with limited free space.

⁶ Snapshot backups are specialized backup solutions commonly used in SANs to create near-instant backups using split-mirror (or similar) technology.

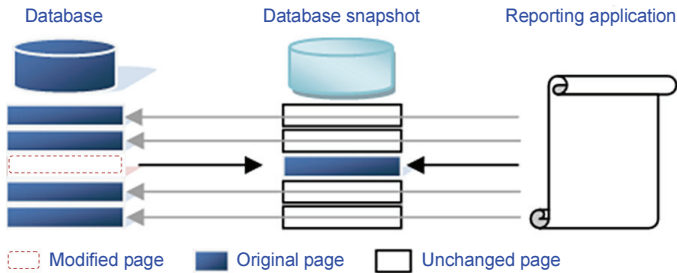


Figure 10.7 Pages are copied to a database snapshot before modification; unchanged page requests are fulfilled from the source database.

As figure 10.7 shows, when a page in a database snapshot is read, if the page hasn't been modified since the snapshot was taken, the read is redirected to the source database. Conversely, modified pages will be read from the snapshot, thus allowing consistent, point-in-time results to be returned.

Let's take a look now at the process of creating a snapshot.

10.5.1 Creating and restoring snapshots

A database snapshot can be created using T-SQL, as shown here:

```
-- Create a snapshot of the AdventureWorks database
CREATE DATABASE AdventureWorks2008_Snapshot_20080624 ON (
    NAME = AdventureWorks2008_Data
    , FILENAME = 'E:\SQL Data\AdventureWorks_Data.ss'
)
AS SNAPSHOT OF [AdventureWorks2008];
GO
```

As you can see in figure 10.8, a snapshot is visible after creation in SQL Server Management Studio under the Database Snapshots folder. You can select it for querying as you would any other database.

Given its read-only nature, a snapshot has no transaction log file, and when created, each of the data files in the source database must be specified in the snapshot creation statement along with a corresponding filename and directory. The only exceptions are files used for FileStream data, which aren't supported in snapshots.

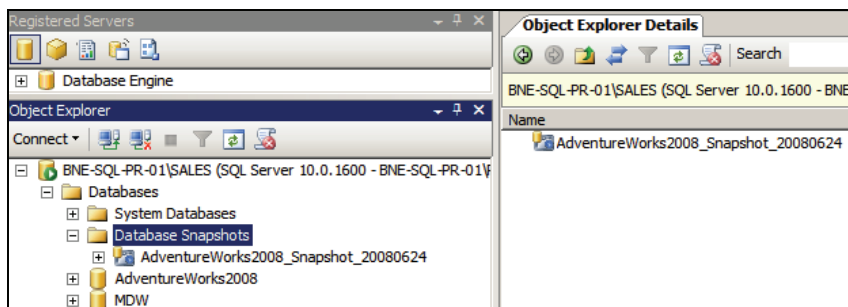


Figure 10.8 A database snapshot is visible in SQL Server Management Studio under the Database Snapshots folder.

You can create multiple snapshots of the same database. The only limitations are the performance overhead and the potential for the snapshots to fill the available disk space. The disk space used by a snapshot is directly determined by the amount of change in the source database. After the snapshot is first created, its footprint, or used space, is effectively zero, owing to the sparse file technology. With each change, the snapshot grows. It follows that if half of the database is modified since the snapshot was created, the snapshot would be roughly half the size of the database it was created from.

Once created, a database can be reverted to its snapshot through the `RESTORE DATABASE T-SQL` command using the `FROM DATABASE_SNAPSHOT` clause as shown here (this example will fail if the AdventureWorks database contains FileStream data). During the restore process, both the source and snapshot databases are unavailable and marked *In Restore*.

```
-- Restore the AdventureWorks database from the snapshot
USE master
GO
RESTORE DATABASE AdventureWorks2008
FROM DATABASE_SNAPSHOT = 'AdventureWorks2008_Snapshot_20080624';
GO
```

There are a number of restrictions with reverting to snapshots, all of which are covered in Books Online. The major ones are as follows:

- A database can't revert to a snapshot if more than one snapshot exists. In such a case, all snapshots should be removed except the one to revert to.
- Despite the obvious advantages of snapshots, they're no substitute for a good backup strategy. Unlike a database restore with point-in-time roll-forward capabilities, a database reverted to a snapshot loses all data modifications made after the snapshot was taken.
- Restoring a snapshot breaks the transaction log backup chain; therefore, after the restore, a full backup of the database should be taken.
- Databases with FileStream data can't be reverted.

Given the copy-on-write nature of snapshots, there's a performance overhead in using them, and their unique nature means update and delete modifications aren't permitted against them—that is, they're effectively read-only databases for the duration of their existence. To reduce the performance overhead, older snapshots that are no longer required should be dropped using a `DROP DATABASE` command such as this one:

```
-- Drop the snapshot
DROP DATABASE AdventureWorks2008_Snapshot_20080624
```

To fully understand the power of database snapshots, let's cover some of the many different ways they can be used.

10.5.2 Snapshot usage scenarios

Database snapshots are useful in a variety of situations. Let's cover the most common uses, beginning with reporting.

REPORTING

Given a snapshot is a read-only view of a database at a given moment, it's ideal for reporting solutions that require data accurate as at a particular moment, such as at the end of a financial period.

The major consideration for using snapshots in this manner is the potential performance impact on the source database. In addition to the copy-on-write impact, the read impact needs to be taken into account: in the absence of a snapshot, would you run reports against the source database? If the requested data for reporting hasn't changed since the snapshot was taken, data requested from the snapshot will be read from the source database.

A common snapshot scenario for reporting solutions is to take scheduled snapshots, for example, once a day. Given each snapshot is exposed as a new database with its own name, reporting applications should ideally be configured so that they are aware of the name change and be capable of dynamically reconnecting to the new snapshot. To assist in this process, name new snapshots consistently to enable a programmatic reconnection solution. Alternatively, *synonyms* (not covered in this book) can be created and updated to point to the appropriate snapshot objects.

READING A DATABASE MIRROR

We'll cover database mirroring in the next chapter, but one of the restrictions with the mirror copy of a database is that it can't be read.

When you take a snapshot of the database mirror, you can use it for reporting purposes, but the performance impact of a snapshot may lead to an unacceptable transaction response time in a synchronous mirroring solution, a topic we'll cover in the next chapter.

ROLLING BACK DATABASE CHANGES

A common use for snapshots is protecting against database changes that don't go according to plan, such as a schema change as part of an application deployment that causes unexpected errors. Taking a snapshot before the change allows a quick roll-back without requiring a full database backup and restore.

The major issue with rolling back to a snapshot in this manner is that all data entered after the snapshot was created is lost. If there's a delay after the change and the decision to roll back, there may be an unacceptable level of data changes that can't be lost.

For changes made during database downtime, when change can be verified while users aren't connected to the database, snapshots can provide an excellent means of reducing the time to deploy the change while also providing a safe rollback point.

TESTING

Consider a database used in a testing environment where a given set of tests needs to be performed multiple times against the same data set. Traditionally, a database backup is restored between each test to provide a repeatable baseline. If the database is very large, the restore delay may be unacceptably long. Snapshots provide an excellent solution to this type of problem.

DBCC SOURCE

Finally, as you'll see in chapter 12, a DBCC check can be performed against a database snapshot, providing more control over disk space usage during the check.

In closing the chapter, let's focus on a very welcome addition to SQL Server 2008: backup compression.

10.6 Backup compression

New in 2008

To reduce the time and space required for backups, some organizations choose to purchase third-party backup tools capable of compressing SQL Server backups. While such products are widely used and proven, other organizations are reluctant to use them for a variety of reasons, such as the following:

- *Cost*—Despite the decreased disk usage (and therefore cost) enabled by such products, some organizations are reluctant to provide the up-front expenditure for new software licenses.
- *Portability*—Depending on the product, compressed backups performed on one licensed server may not be able to be restored on an unlicensed server.
- *Non-Microsoft software*—Some organizations feel uncomfortable with using non-Microsoft software to control such a critical operational process.

In avoiding backup compression products for these reasons, many organizations choose suboptimal backup designs, such as tape only, in order to reduce storage costs. Such designs are often in conflict with their service level agreements for restoration times and acceptable data loss, and often the limitations of such designs are realized for the first time after an actual data loss event.

Introduced in the Enterprise edition of SQL Server 2008, backup compression allows native SQL Server backups to be compressed, which for many organizations will introduce a whole range of benefits and cost savings. No doubt some companies will upgrade to SQL Server 2008 for this reason alone. Although compressed backups can only be *created* using the Enterprise edition of SQL Server 2008, they can be *restored* to any edition of SQL Server 2008.

As with data compression, covered in the previous chapter, there is some CPU overhead involved in backup compression (about 5 percent is typical). To control whether a backup is compressed, you have a number of options, beginning with a server-level default setting called Backup Compression Default, which you can set using `sp_configure` or SQL Server Management Studio, as shown in figure 10.9.

For individual backups, you can override the default compression setting using options in SQL Server Management Studio, or by using the `WITH COMPRESSION/NO_COMPRESSION` T-SQL options as shown here:

```
-- Backup the AdventureWorks database using compression
BACKUP DATABASE AdventureWorks2008
TO DISK = 'G:\SQL Backup\AdventureWorks-Compressed.bak'
WITH INIT, COMPRESSION
```

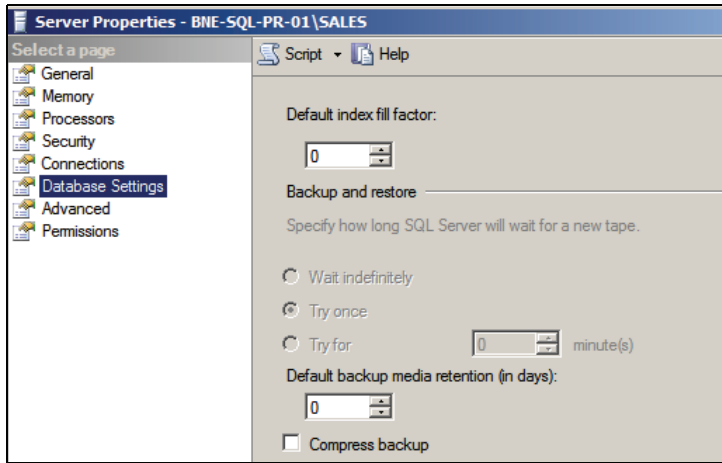


Figure 10.9 The Compress Backup option enables the default backup compression to be set. Individual backups can explicitly override this setting.

As with data compression, the actual compression rates achieved depend largely on the makeup of the data within the database. Similar to data compression, the goal of backup compression is *not* to achieve the maximum possible compression, but to strike a balance between CPU usage and compression rates.

Given that, the observed compression rates are quite impressive considering the moderate CPU overhead. For example, as you can see in figure 10.10, the observed size of a native AdventureWorks2008 database backup was 188MB compared with the compressed backup size of 45MB. Further, the time taken to back up the database in uncompressed form was 10 seconds compared to 7 seconds for a compressed backup.

Although the actual results will differ depending on the scenario, extrapolating out the compression and duration figures to a very large database scenario promises significant savings in disk space (and therefore money) as well as time.

For those organizations with a tape-only backup approach, backup compression presents an excellent argument to move to a disk then tape approach. For those

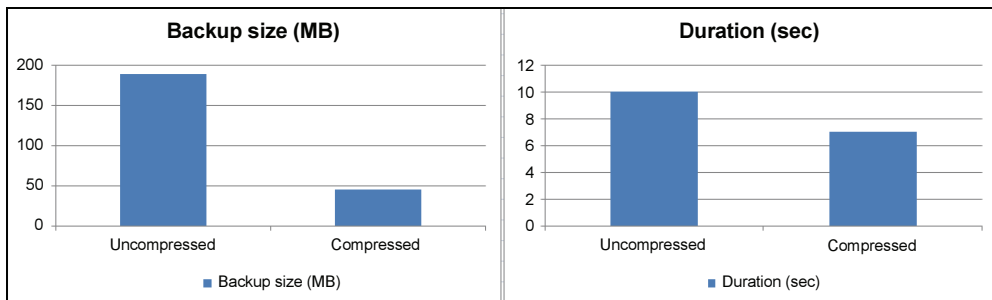


Figure 10.10 Backing up the AdventureWorks2008 database with and without compression. For a moderate CPU overhead, compressed backups yield significant space and duration savings.

already using disk-based backups, the opportunities for storage savings and greater backup availability are compelling reasons for an upgrade to SQL Server 2008.

Finally, as you saw in chapter 6, a backup produced from a database that's protected with Transparent Data Encryption (TDE) will also be encrypted and can't be restored to another server unless that server has the appropriate certificate restored. From a compression perspective, the space savings of a compressed backup on a TDE-encrypted database will be minimal. As such, I don't recommend compressing backups of TDE-encrypted databases.

10.7 **Best practice considerations: backup and recovery**

Developing a reliable backup strategy is arguably the most fundamental and important of all DBA tasks. Fortunately, there are a number of well-established best practices to assist in this process.

- Design a backup strategy for the speed and ease of restoration, not the convenience of the backup. The design should be centered around the service level agreements for restoration time and acceptable data loss.
- Thoroughly document the backup and restore process and include actual code for various restore scenarios. Anyone with moderate DBA skills should be able to follow the documentation to ensure the correct restore process is executed in the shortest possible time.
- When developing disaster recovery plans, consider smaller events as potential disasters in addition to complete site failure. "Small" disasters such as the accidental deletion of a production table can have just as much impact as big ones.
- Simulate and practice recovering from disasters on a regular basis to ensure that documentation is up to date and that all appropriate support staff are comfortable with, and trained in, the recovery process. Consider implementing random "fire drills" to more accurately simulate disaster.
- To minimize the performance impact, schedule full backups for periods of low-usage times.
- Ensure system databases (with the exception of tempdb) are backed up on a regular basis, and immediately after the installation of any service packs, hotfixes, or cumulative updates. System databases store important instance-level data such as login information, maintenance plans, SQL Agent job definitions, and execution history. Restoring a master database backup that was taken when an earlier service pack version was installed is not an experience I recommend!
- Use `COPY_ONLY` backups to avoid breaking backup chains when additional backups are required.
- Backing up the tail of a transaction log using the `WITH NO_TRUNCATE` option should be limited to situations in which the database is damaged and inaccessible; otherwise, the `COPY_ONLY` option should be used in its place.
- After first creating a database or changing the recovery model, take a full backup to initialize the log chain.

- To provide point-in-time restore capabilities and manage transaction log growth, production databases should be in the full recovery mode with regular transaction log backups.
- Development and test databases that don't require point-in-time restoration capabilities should be placed in the simple recovery mode to limit administration overhead and disk space usage.
- Use the Bulk_Logged model on a temporary basis only during bulk-load operations. Take transaction log backups immediately before and after using the bulk logged model for maximum point-in-time protection.
- Consider the disk then tape backup technique whereby backups are written to disk before being archived to tape and removed from disk after a number of days. As well as enabling two copies of recent backups for resilience against media failure, the local disk copies provide fast restoration if needed, and you maintain offsite tape copies for long-term archival purposes.
- Assuming the CPU overhead is measured and within the available headroom, consider backup checksums (along with page checksums) as a means of enabling constant and ongoing I/O verification.
- Consider the MIRROR TO DISK option when performing disk backups to create an *off-server* disk backup for tape archive. With this approach, you avoid the need for tape backup software and drivers on the SQL Server, and you create an additional disk backup with independent retention periods.
- If using the MIRROR TO DISK option to back up to a secondary backup file over the network, consider a private LAN connection to the backup server to maximize network performance and minimize the effect on the public LAN.
- Streaming a backup to multiple backup files can produce a significant performance increase compared to single file backups, particularly for very large databases.
- For small databases, full nightly backups with regular transaction log backups through the day are ideal. For larger databases, consider a weekly full, daily differential, and hourly transaction log model. For very large databases running on the Enterprise edition of SQL Server, consider a filegroup backup/restore design centered around online piecemeal restores.
- Keep in mind the diminishing returns of differential backups. The frequency of the full backup needs to be assessed on the basis of the rate of database change.
- Restore backups on purpose-built backup verification servers or as part of an infrastructure solution, such as a reporting server with automated restores. Log shipping (covered in the next chapter) is an excellent way of verifying transaction log backup validity as well as providing a mechanism to enable reporting databases to be refreshed with current data.
- An alternate means of verification is the RESTORE WITH VERIFYONLY operation, which will read the contents of the backup file to ensure its validity without actually restoring it. In the absence of an automated restore process, this is a good method for verifying that backups are valid.

- Consider the use of backup devices (not covered in this book) for more flexibility when scripting backup jobs. Rather than creating script jobs containing hard-coded directory paths and filenames, using backup devices enables portability of backup scripts; each environment's backup devices can be configured for the appropriate drive letters, directory paths, and filenames.
- If using database snapshots for reporting purposes, ensure they're consistently named to assist with programmatically redirecting access to new snapshots, and make sure old snapshots are removed to reduce disk space requirements and the performance overhead of copy-on-write.
- If using the Enterprise edition of SQL Server, consider backup compression as a means of reducing backup disk cost. Alternatively, consider keeping more backups on disk for longer periods (or use both strategies).
- Compressing backups of databases that use Transparent Data Encryption isn't recommended because the compression rate is likely to be low while still incurring CPU overhead.

Additional information on the best practices covered in this chapter can be found online at <http://www.sqlCrunch.com/backup>.

As we've covered in this chapter, transaction logs form a fundamental part of a backup and recovery plan. In the next chapter, we'll take a look at log shipping, an excellent mechanism for ongoing verification of the validity of the transaction log backups.

SQL Server 2008 Administration in ACTION

Rod Colledge • Foreword by Kevin Kline

Ensuring databases are secure, available, reliable, and recoverable are core DBA responsibilities. This is a uniquely practical book that drills down into techniques, procedures, and practices that will keep your databases running like clockwork.

Open **SQL Server 2008 Administration in Action** and find sharply focused and practical coverage of

- Selecting and configuring server components
 - Configuring RAID arrays
 - Working with SANs and NUMA hardware
- New features in SQL Server 2008
 - Policy-based management
 - Resource Governor
 - Management Data Warehouse
- And much more!
 - Performance tuning techniques
 - Index design and maintenance
 - SQL Server clustering and database mirroring
 - Backup and restore

The techniques and practices covered in this book are drawn from years of experience in some of the world's most demanding SQL Server environments. It covers new features of SQL Server 2008 in depth. Its best practices apply to all SQL Server versions.



Rod Colledge is an SQL Server consultant based in Brisbane, Australia, and founder of sqlCrunch.com, a site specializing in SQL Server best practices. He's a frequent speaker at SQL Server user groups and conferences.

For online access to the author, code samples, and a free ebook for owners of this book, go to:

www.manning.com/SQLServer2008AdministrationinAction



“Simply loaded with excellent and immediately useful information.”

—From the Foreword by Kevin Kline, Technical Strategy Manager, Quest Software

“I thought I knew SQL Server until I read this book.”

—Tariq Ahmed, coauthor of *Flex 4 in Action*

“A refreshing database administration book.”

—Michael Redman, Principal Consultant (SQL Server), Microsoft

“Required for any MS DBA.”

—Andrew Siemer, Architect, OTX Research

“It delivered way beyond my expectations... Packed with useful enterprise-level knowledge.”

—Darren Neimke, Author of *ASP.NET 2.0 Web Parts in Action*

ISBN 13: 978-1-933988-72-6
ISBN 10: 1-933988-72-X



9 781933 988726



MANNING

\$44.99 / Can \$56.99 [INCLUDING eBOOK]