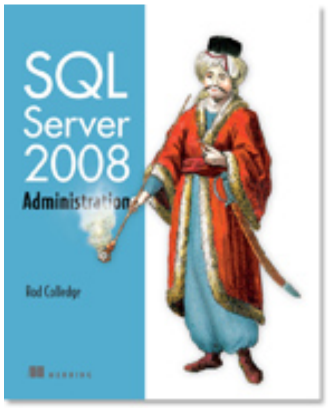
MANNING PUBLICATIONS

Backup types

Excerpted from



SQL Server 2008 Administration in Action

Rod Colledge
MEAP Release: May 2008
Softbound print: August 2009 | 464 pages
ISBN: 193398872X

Unless you're a DBA, you probably consider a database backup as being a complete copy of a database at a given point in time. Whilst that's *one* type of database backup, there are many others. Consider a multi terabyte database that's used 24 x 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 part way 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 a simplistic full nightly backup is insufficient on a number of fronts, in particular 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 3rd party products and database maintenance plans. For the purposes of the examples throughout this article, we'll use a T-SQL script approach

Full Backup

Full backups are the simplest, most well understood type of backup. Like backing up a file (document, spreadsheet, and so forth), a full backup is exactly that; a complete copy of the database at a given time, but unlike a file backup, backing up a database cannot be performed 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 backup all files on a database server on the assumption that the inclusion of the underlying database files (.mdf/.ldf) in the backup will be sufficient for a restore scenario. Not only will this backup "strategy" fail, those that use such an approach usually fail to realize it until they try and perform a restore.

In order for a backup to be valid, we use the BACKUP DATABASE command, or one of its GUI equivalents. A simple example for backing up the AdventureWorks database follows. The full description of the backup command with all of its various options can be found in Books Online.

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

Backups in SQL Server can be performed whilst the database is in use and being modified by users. 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 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 presented below correspond to the steps in figure 1.

1. When the backup commences, a checkpoint is issued which flushes dirty buffer cache pages to disk,
2. After the checkpoint completes, the backup process commences 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 will not complete until after the backup finishes. At the point of backup completion, this transaction is the oldest active (uncommitted / incomplete) transaction,
6. Transaction A completes successfully,
7. The backup completes reading pages from the databases

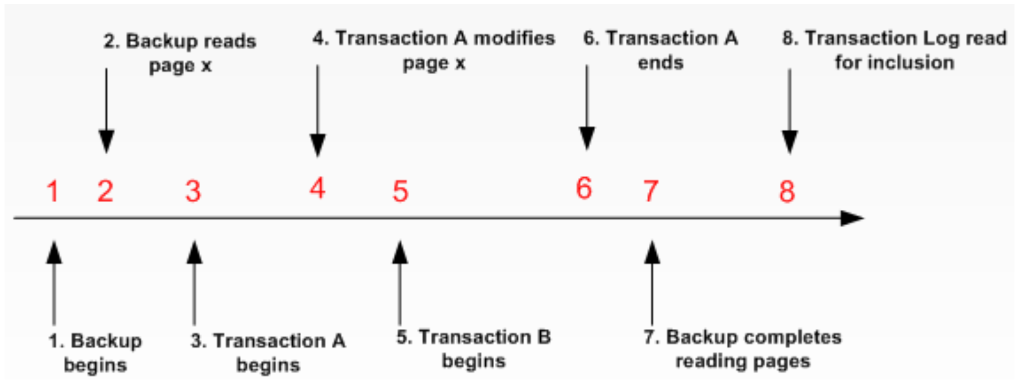


Figure 1 Timeline of an online full backup. Based on an example used with permission from Paul S. Randal, Managing Director of sqlskills.com

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* would not be in the restored database, and transaction B has not completed, so it's changes need 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 the above example, once SQL Server completes reading databases pages at point 7, it will include all entries in the transaction log since the *oldest* log sequence number (LSN) of either;

- The Checkpoint (step 1 in the above example), or
- The oldest active transaction (step 5 in the above example), or
- The LSN of the last replicated transaction (out of scope for this example)

In our above example, transaction log entries since step 1 will be included as that is the oldest of the above items, however, consider a case where a transaction starts *before* the backup commences, 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 Randall of SQLSkills.com. The link to the full post titled *More on How Much Transaction Log a Full Backup Includes*, is available at http://www.sqlCrunch.com/backup.

It's important to point out here that even though parts of the transaction log are *included* in a full backup, this does not constitute a transaction log *backup*. Another classic mistake made by inexperienced SQL Server DBAs is never performing transaction log backups on the assumption that 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 per this 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. Using the T-SQL BACKUP DATABASE command, the "disk =" clause can be repeated multiple times (comma separated), once for each backup file

Differential Backup

Whilst 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 article we covered 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 below, 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 2 illustrates a weekly full / nightly differential backup design.

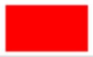
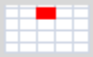
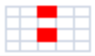
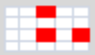
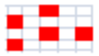
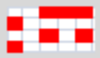
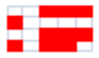
Day	Backup Type	Includes ...	Contents
Sunday	Full	Everything	
Monday	Differential	Changes since Sunday	
Tuesday	Differential	Changes since Sunday	
Wednesday	Differential	Changes since Sunday	
Thursday	Differential	Changes since Sunday	
Friday	Differential	Changes since Sunday	
Saturday	Differential	Changes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

Figure 2 Differential backups grow in size and duration the further from their corresponding full backup (base)

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 counter-productive; 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 above 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 per 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
```

In the above example, we can see the full backup is restored *with norecovery*. This leaves the database in a *recovering* state, able to restore additional backups. We follow the restore of the full backup with the differential restore.

As per the restore of the full backup presented earlier, without transaction log backups, discussed next, changes made to the database since the differential backup will be lost.

Transaction Log Backup

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

- Records each database transaction, and the individual database modifications made within each transaction,
- Should a transaction be cancelled prior to completion, either at the request of an application or due to a system error, the transaction log is used to undo, or *rollback* the transaction's modifications,
- A transaction log is used during a database restore to roll forward completed transactions, and roll back incomplete transactions. This process is also followed for each database when SQL Server starts up,
- The Transaction Log plays a key role in log shipping and database mirroring.

Regular transaction log backups, an example of which is shown below, 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 shown in figure 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 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. In order to restore a database to a point in time, an unbroken chain of transaction logs is required, from a full backup up to the required point of recovery.

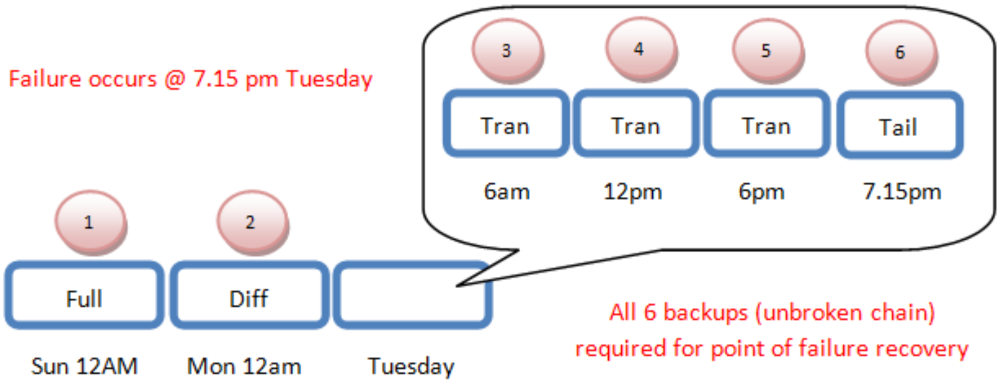
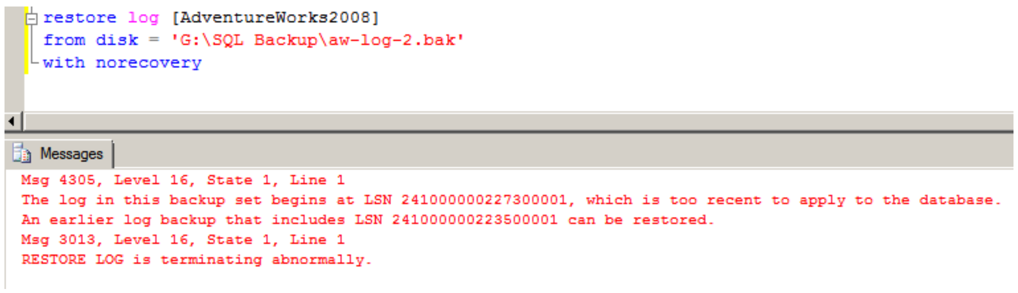

Failure occurs @ 7.15 pm Tuesday
1
Full
Sun 12AM
2
Diff
Mon 12am
Tuesday
3
Tran
6am
4
Tran
12pm
5
Tran
6pm
6
Tail
7.15pm
All 6 backups (unbroken chain) required for point of failure recovery

Figure 3 An unbroken chain of backups are required to recover to the point of failure

Consider figure 3. Starting at point 1, we perform a full database backup, after which differential and transaction log backups occur. Each of the backups act 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 would not be able to restore past the end of backup 3 at 6am Tuesday morning. Attempting to restore the transaction log from log backup 5 would result in an error message similar to that shown in figure 4;



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

```
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 4 Attempting to restore an out of sequence transaction log

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 continuing growing indefinitely until a transaction log backup occurs.

The frequency of transaction log backups is an important consideration, with the two main determining factors being 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. 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 is not 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 here 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 is currently attached to a server instance, SQL Server will generate an error[1] unless the *tail* of the transaction log is first backed up. The tail refers to the section of log that has not yet been backed up, 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 will not change after the tail log backup and therefore 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 proceeding it performed with the *NORECOVERY* option. The tail log is then restored with the *RECOVERY* option to recover the database up to the point of failure, or a time before failure using the *STOP* AT command.

So let's put all this together with an example. In listing 1, we first backup the tail of the log before restoring the database to a point in time. We begin with restoring the full and

[1] Unless the WITH REPLACE option is used

differential backups with *norecovery*, and then roll forward the transaction logs to a required point in time;

Listing 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, the *COPY_ONLY* option should be used.

COPY_ONLY Backups

Earlier in this article we discussed a log chain as being 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 3 where we perform a full backup on Sunday night, nightly differential backups and 6 hourly transaction log backups. In a similar manner to the code presented in listing 1, to recover to 6pm on Tuesday night, we would recover Sunday's full backup, followed by Tuesday's differential and the 3 transaction log backups leading up to 6 pm.

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* or 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 above, we're not using the full backup from Monday, hence the failure.

Now there are a few ways around this problem. Firstly, 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. Secondly, we can track down the developer and ask him for the full backup and hope that he hasn't deleted it!

In order to address the broken chain problem as outlined above, 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, are used in situations in which the backup sequence should not be affected. In our above 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 per this example, will backup the log without truncation, meaning the log backup chain remains 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 article, several references were made 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.

[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 requiring a T-SQL script approach