Bulletproof your Database Backup and Recovery Strategy

By Shawn McGehee and Tony Davis
The most critical task for all DBAs is to have a Backup and Recovery strategy that ensures, every day, that in the event of a disaster they can restore and recover any database, within acceptable limits for data loss and downtime.

Even with all the required backups in place, it's easy to miss subtle failings in the overall plan, such as:

- **Inadequate, or missing, backup verification**, failing to spot corruption
- **Poor management of non-scheduled backups**, potentially leading to a failed restore
- **Minimally logged operations** when using BULK_LOGGED recovery model, potentially causing a failed point-in-time restore
- **Poor transaction log management**, slowing down log backup and crash recovery
  - operations

When disaster strikes, and a database restore is required, any such "hole" in the strategy could result in extended downtime, loss of data, and a failure to meet the terms of your Service Level Agreement.
1. Don't rely only on BACKUP...WITH CHECKSUM to catch corruption

Whether there are 10 databases in our environment or 1000, we must ensure that all backups are valid and usable. Without good backups, we will be in a very bad spot when the time comes to bring some data back from the dead.

The default mechanism by which SQL Server detects corruption in data pages is via use of checksums. Whenever it writes a data page to disk it calculates a checksum and stores the value in the page header. Subsequently, when it reads that page into memory, it recalculates the checksum and compares it to the value stored in the header. If the values match, then no corruption occurred while the page was stored in I/O subsystem; if they don't match then SQL Server raises a corruption error, which might look something like this:

```
Msg 824, Level 24, State 2, Line 1
SQL Server detected a logical consistency-based I/O error: incorrect checksum (expected: 0x9a3e399c; actual: 0x9a14b99c).
It occurred during a read of page (1:184) in database ID 23 at offset 0x00000000170000 in file 'D:\SQLDATA\ForcingFailures.mdf'. Additional messages in the SQL Server error log or system event log may provide more detail. This is a severe error condition that threatens database integrity and must be corrected immediately. Complete a full database consistency check (DBCC CHECKDB). This error can be caused by many factors; for more information, see SQL Server Books Online.
```

**BACKUP...WITH CHECKSUM for finding on-disk corruption**

By enabling checksums during backup operations, as shown in Listing 1, we can ensure that every page on disk, to be included in the backup, is free from this form of corruption.

```
BACKUP DATABASE <DatabaseName>
TO DISK = '<Backup_location>.bak'
WITH CHECKSUM;
GO
```

*Listing 1: Backup WITH CHECKSUM syntax*

SQL Server will recalculate checksums as it reads each page to back up. If it finds a page that fails this test, the backup will fail. If the backup succeeds then the backup is valid...or maybe not.
Problems with in-memory corruption

In fact, occasionally, checksum validation can offer a false sense of security and has gotten many DBAs into trouble. It does not guarantee that a database backup is corruption free. The CHECKSUM only verifies that no corruption occurred between writing the data to disk and reading it back out. It does not detect in-memory corruption.

If a data page cached in memory is updated, and then some in-memory corruption occurs (perhaps due to a faulty memory chip) before the modified page is written back to disk, then when the time comes, SQL Server will simply calculate the checksum value on the corrupted page and write it to disk. When it reads that page during a subsequent BACKUP... WITH CHECKSUM operation, it will recalculate the checksum and, assuming no additional problems occurred in the I/O subsystem, the values will match and the corrupted page will be included in the backup.

The most effective way to ensure that your data pages are totally corruption free is to run DBCC CHECKDB on a regular basis. This tool fully validates page structure and content, will check the logical and physical integrity of all the objects in the specified database, and so will catch corruption as early as possible.

A multi-pronged Defense to Data Corruption

In short, the lesson for the DBA here is that he or she requires a multi-pronged defense to the threat of data corruption!

BACKUP...WITH CHECKSUM is a useful tool and I recommend its use. Enabling checksums (and other checks, such as torn page detection) will bring with it only a small CPU overhead and should only have a minimal impact of the speed of your backups.

In addition, schedule regular consistency checks for each database, using DBCC CHECKDB. Weekly checks are optimal, but on a server where the spare CPU and disk capacity are not readily available then monthly checks are better no checks. You could also restore databases from a server that cannot sustain weekly integrity checks to a secondary server to perform the maintenance there.
In addition, again, it’s a great idea to perform some test restores. This may seem obvious, but there many DBAs simply assume that their backups are good and let them sit on the shelf. We don’t need to restore every backup in the system to check its health, but doing random spot checks now and again is an easy way to gain peace of mind regarding future restores, and to hone the recovery strategy. Each week, choose a random database, and restore its last full backup. If that database is subject to differential and log backups as well, choose a point-in-time test restore that uses a full, differential and a few log backup files.

A tool such as Red Gate SQL Backup Pro makes it very easy to combine test restores with DBCC CHECKDB integrity checks. We can simply, through the SQL Backup Pro GUI, set up DBCC CHECKDB integrity checks as part of scheduled test restores.

2. Use copy_only for non-scheduled Full Backups

Imagine you are the senior DBA on the team and need to restore to a point in time (say, midday) one of your FULL recovery model databases, which receives nightly full backups (midnight), daily differential backups (midday), and transaction log backups every 30 minutes. All you need here are the most recent nightly full and daily diff backups, so you pull them out and restore them (in this example, over the top of the existing database).

```sql
USE master
GO
RESTORE DATABASE ForcingFailures
FROM DISK = N'D:\SQLBackups\ForcingFailures_Full.bak'
WITH NORECOVERY, REPLACE, STATS = 10;
GO
RESTORE DATABASE ForcingFailures
FROM DISK = N'D:\SQLBackups\ForcingFailures_Diff.bak'
WITH RECOVERY, STATS = 10;
GO
```

*Listing 2: Restoring full and differential backups*
The restore of the full backup proceeds without a hitch, but you're perturbed to see that the restore of the differential backup fails, with the following error message:

```
Msg 3136, Level 16, State 1, Line 1
This differential backup cannot be restored because the database has not been restored to the correct earlier state.
Msg 3013, Level 16, State 1, Line 1
RESTORE DATABASE is terminating abnormally.
```

Quickly, you examine the backup history in the msdb database, assuming you've accidentally pulled the wrong backup files.

```
USE MSDB
GO
SELECT bs.type ,
       bmf.physical_device_name ,
       bs.backup_start_date ,
       bs.user_name
FROM dbo.backupset bs
     INNER JOIN dbo.backupmediafamily bmf
             ON bs.media_set_id = bmf.media_set_id
WHERE bs.database_name = 'ForcingFailures'
ORDER BY bs.backup_start_date ASC ;
```

Listing 3: Checking backup history

This query reveals the type of backup taken (D = full database, I = differential database), the name and location of the backup file, when the backup was started, and who took it.

<table>
<thead>
<tr>
<th>type</th>
<th>physical_device_name</th>
<th>backup_start_date</th>
<th>user_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>D:\SQLBackups\ForcingFailures_Full.bak</td>
<td>2012-09-10 00:00:05.000</td>
<td>JohnB</td>
</tr>
<tr>
<td>D</td>
<td>D:\SQLBackups\ForcingFailures_DEV_Full.bak</td>
<td>2012-09-11 10:41:07.000</td>
<td>JoeD</td>
</tr>
<tr>
<td>I</td>
<td>D:\SQLBackups\ForcingFailures_Diff.bak</td>
<td>2012-09-11 12:00:03.000</td>
<td>JohnB</td>
</tr>
</tbody>
</table>

(3 row(s) affected)
You see your regularly scheduled full and diff backups, but in between the two is a rogue, non-scheduled full backup, taken by the junior DBA at the request of the development team, in order to refresh their environment.

Unless you can find that "rogue" full backup then your differential backup is useless. The reason for this is that there is an inextricable link between a differential database backup and its base full database backup. A differential database backup is a copy of every page that has changed since the last full backup (base full backup). SQL Server keeps track of which data pages changed since the last full backup, and so would need to be included in any differential backup. However, each time we take a full backup, SQL Server clears all markers in this internal tracking mechanism, so each new full backup becomes the base backup for any subsequent differential backups. If we lose a full backup, or it becomes corrupted, any differential backups that rely on that base will be unusable.

In this case, the rogue _DEV_Full backup is now the base backup for the regular diff backup. If that rogue backup is still available, you can restore that instead, followed by the diff backup. If the junior DBA deleted the rogue backup after completing the restore to development, then the problem is more serious, but still not unfixable. You can still restore the midnight full backup followed by all the transaction log backups up to midday. If the rogue backup is gone, and this had happened to a SIMPLE recovery model database, with no log backups, then the problem would be more serious still, since all you can really do is restore the midnight full backup.

The way to avoid this problem in the first place is to always, when adopting a backup strategy that uses differential backups, use the COPY_ONLY option when capturing any full backups that fall outside the regular, scheduled backup jobs.

```
BACKUP DATABASE ForcingFailures
TO DISK = N'D:\SQLBackups\ForcingFailures_DEV_Full.bak'
WITH COPY_ONLY ;
GO
```

*Listing 4: Using COPY_ONLY with full backups*
In terms of the data it contains and the manner in which we restore it, a copy-only full backup is just like a normal full backup. However, the big difference is that, unlike a regular full backup, with a copy-only full backup, SQL Server’s internal mechanism for tracking data pages changed since the last full backup is left untouched and so the core backup strategy is unaffected.

The lessons here are that DBAs need to make sure that:

- Only those people who need to take backups have the permissions granted to do so
- The DBA team, and certain administrative users, know how and when to use a copyonly full backup.

### 3. Check your SLA before using BULK_LOGGED recovery

It is not possible to restore a database to a point in time within a log file that contains minimally logged operations, recorded while the database was operating in the BULK_LOGGED recovery model. In order to visualize this, Figure 1 depicts an identical backup timeline for two databases, each of which we wish to restore to the same point in time (represented by the arrow). The green bar represents a full database backup and the yellow bars represent a series of log backups. The only difference between the two databases is that the first is operating in FULL recovery model, and the second in BULK LOGGED.

![Database backup timeline](image)

*Figure 1: Database backup timeline.*
During the time span of the fifth log backup of the day, there was a BULK INSERT command against each database, to load a set of data. This bulk data load completed without a hitch but in an unrelated incident, within the time span of the fifth log backup, a user ran a "rogue" transaction and crucial data was lost. The project manager informs the DBA team, and requests that they restore the database to a point just where the transaction that resulted in data loss started.

In the FULL recovery model database, this is not an issue. The bulk data load was fully logged and we can restore the database to any point in time within that log file. We simply restore the last full database backup, without recovery, and apply the log files to the point in time right before the unfortunate data loss incident occurred.

In the BULK LOGGED database, we have a problem. We can restore to any point in time within the first four log backups, but not to any point in time within the fifth log backup, which contains the minimally logged operations. For that log file, we are in an "all or nothing" situation; we must apply none of the operations in this log file, so stopping the restore at the end of the fourth file, or we must apply all of them, proceeding to restore to any point in time within the sixth log backup.

In other words, we can restore the full database backup, again without recovery, and apply the first four log backups to the database. Unfortunately, we will not have the option to restore to any point in time within the fifth log. If we apply the whole of the fifth log file backup, this would defeat the purpose of the recovery, since the errant process committed its changes somewhere inside of that log backup file, so we’d simply be removing the data we were trying to get back! We have little choice but to restore up to the end of the fourth log, enter database recovery, and report the loss of any data changes that were made after this time.

Hopefully, this will never happen to you and, unless your SLA adopts a completely "zero tolerance" attitude towards any risk of data loss, it is not a reason to avoid BULK_LOGGED recovery model altogether. There are valid reasons using this recovery model in order to reduce the load on the transaction log, and if we follow best practices, we should not find ourselves in this type of situation.
The lessons here are that:

- We should always check the SLA before switching a database, even temporarily, to BULK_LOGGED recovery model
- If use of BULK_LOGGED model is permissible, then we must take steps to minimize any potential impact on point-in-time restore operations

If we need to perform maintenance operations that can be minimally logged and we wish to switch to BULK_LOGGED model, we should take a log backup immediately before switching to BULK_LOGGED, and immediately after switching the database back to FULL recovery, as demonstrated in Listing 5. This will isolate the minimally logged transactions in a single log backup file.

```sql
USE master;
GO
BACKUP LOG SampleDB TO DISK = '\path\example\filename.trn';
GO
ALTER DATABASE SampleDB SET RECOVERY BULK_LOGGED WITH NO_WAIT;
GO
-- Perform minimally logged transactions here
-- Stop minimally logged transactions here
ALTER DATABASE SampleDB SET RECOVERY FULL WITH NO_WAIT;
GO
BACKUP LOG SampleDB TO DISK = '\path\example\filename.trn';
GO
```

*Listing 5: A template for temporarily switching a database to BULK_LOGGED recovery model*
4. Use Good Log Management to avoid slow log backups and database recovery

Consider a situation where, as a DBA, a database falls under your care. Having implemented some monitoring, you find that the log is almost full and that there isn't the capacity on its disk drive to accommodate an urgent database maintenance operation. You try a log backup but for reasons you need to investigate further, SQL Server will not truncate the log. In order to buy some time, you add a secondary log file, on a separate disk, and the operation proceeds as planned.

Factors that may delay log truncation
For a FULL or BULK_LOGGED recovery model database, only a log backup will result in truncation of the log (i.e. enable reuse of space in the log). However, other factors that may delay log truncation. For example, a long-running uncommitted transaction may keep large areas of the log "active", or another database process, such as database mirroring or transactional replication, may still be require the log records. See http://www.sqlservercentral.com/articles/Transaction+Logs/72488/ for more details.

You investigate why the log ballooned in size and it turns out to be an application leaving "orphaned transactions" in the database. The issue is fixed, and the next log backup truncates the log, creating plenty of reusable space.

So far so good: you've allowed the maintenance operation to proceed, and you've fixed the problem that caused the explosive log growth. However, you still have two outstanding problems:

- A database with multiple log files
- A database with a principal log file that is bloated and likely highly fragmented

Firstly, you need to get rid of that secondary log file as soon as possible. SQL Server does not write log records in parallel to multiple log files, even when created on separate drives, so there is no performance advantage to having multiple log files. As soon as an "emergency" secondary log file is no longer required, we need to remove it, as all it will do is slow down any restore or recovery operations, since SQL Server has to zero-initialize the log during full and differential restore operations.
Instant file initialization and log files

In the code download available with this whitepaper, the script `McGeheeDavis_MyMessages_2logs.sql` creates a MyMessages database with a small initial log file (2 MB), which is set to grow in small increments (2 MB). It sets the database to FULL recovery model and performs a full database backup. In Listing 6, we create a simple `MessageTable` in the MyMessages database, and insert into it a million rows.

```sql
USE MyMessages;
GO
CREATE TABLE dbo.MessageTable
(
    MessageData nvarchar(200) NOT NULL,
    MessageDate datetime2 NOT NULL
);
GO
INSERT INTO MessageTable
(MessageData, MessageDate)
SELECT TOP 1000000
    REPLICATE('a', 200),
    GETDATE()
FROM msdb.sys.columns a
    CROSS JOIN msdb.sys.columns b;
GO
```

Listing 6: Create `MessageTable` and insert a million rows

Let's look at the log size and space usage. Our initial size for the log was 2 MB but SQL Server has had to auto-grow it to 745 MB to accommodate our insert.
DBCC SQLPERF(LOGSPACE);
-- MyMessages Log Size: 745 MB, Log Space Used: 72%

Listing 7: Check log stats

A transaction log that auto-grows frequently, in small increments, will have a very large number of small Virtual Log Files (VLFs). This phenomenon is log fragmentation. If a database process, such as the crash recovery process, has to read the log file, it starts by reading in all the VLFs. If there are many of them, this will be a longer operation, and may affect the overall time taken to recover the database. A similar argument applies to other operations that read the log, such as log backups.

Essentially, the initial size and relatively small growth increments we've chosen for this database are inappropriate for this sort of data load and lead to the creation of a large number of VLFs. We can interrogate the VLF architecture using a command called DBCC LogInfo, as shown in Listing 8.

Interrogating VLFs using DBCC LogInfo

DBCC LogInfo is an undocumented and unsupported command. We'll use it in this whitepaper to peek at the VLFs, but we won't go into detail about the information it returns. Kalen Delany has a good blog post that explains its use, and output: http://sqlblog.com/blogs/kalen_delaney/archive/2009/12/21/exploring-the-transaction-logstructure.aspx.

USE MyMessages
DBCC Loginfo;
-- returns 1488 rows

Listing 8: A fragmented log!

All we're interested in at this stage is that DBCC LogInfo returns 1488 rows, meaning 1488 VLFs. Let's assume, in our simplified example, this is the point at which the DBA needs to add a second 3 GB log file, to accommodate the database maintenance operations.
USE master;
GO
ALTER DATABASE MyMessages
ADD LOG FILE ( NAME = N'MyMessages_log2',
    FILENAME = N'D:\SQLData\MyMessages2.ldf',
    SIZE = 512000KB , FILEGROWTH = 512000KB );
GO
USE master
GO
ALTER DATABASE MyMessages MODIFY FILE
    ( NAME = N'MyMessages_log2', SIZE = 3146752KB );
GO

Listing 9: Adding a secondary log file

If we re-interrogate DBCC LOGINFO, it returns 1512 rows. We've created a 3 GB secondary log file but this time set a reasonable initial size (500 MB), followed by a manual growth, and the secondary log file comprises only 24 new VLFs, a very reasonable number of virtual files.

The database maintenance operation can now proceed. Sometime later, we've also fixed the problem with the orphaned transaction, and a log backup successfully truncates the log. At this stage, we no longer need our secondary log file but it still exists. Let's restore the MyMessages database.

BACKUP LOG MyMessages
TO DISK = 'D:\SQLBackups\MyMessages.trn';
USE master;
go
RESTORE DATABASE MyMessages
FROM DISK = 'D:\SQLBackups\MyMessages_full.bak'
WITH REPLACE, NORECOVERY;
RESTORE DATABASE MyMessages
FROM DISK='D:\SQLBackups\MyMessages.trn'
WITH RECOVERY;
/*<output truncated>…
Processed 0 pages for database 'MyMessages', file 'MyMessages' on file 1.
Processed 65704 pages for database 'MyMessages', file 'MyMessages_log' on file 1.
Processed 0 pages for database 'MyMessages', file 'MyMessages_log2' on file 1.
RESTORE LOG successfully processed 65704 pages in 53.476 seconds (9.598 MB/sec).*/

Listing 10: Restoring MyMessages (with secondary log file)

The restore took over 50 s. If we repeat the exact same steps, but without adding the secondary log file, the comparative restore, in our tests, took about 6 seconds. This is a substantial impact, even for a relatively modestly size secondary log. In cases where the log file is much larger, the effect on backup times can be dramatic.

In order to remove the secondary log file, we need to wait until it contains no part of the active log. Since our goal is to remove it, it's permissible to shrink this secondary log file (demonstrated shortly), and turn off auto-growth for this file. Shrinking the secondary log to zero will return it to its original size (500 MB) and so "encourage" the active log to move swiftly back into the primary log file. It's important to note that this will not move any log records in the secondary log over to the primary log (some people expect this behavior because if we specify the EMPTYFILE parameter, when shrinking a data file, SQL Server will move the data to another data file in the same filegroup).

As soon as a log backup means that the secondary log file contains no part of the active log, we can simply remove it.

USE MyMessages;
GO
ALTER DATABASE MyMessages REMOVE FILE MyMessages_Log2;
GO

Listing 11: Removing the secondary log file
This is one problem solved, but we may still have a bloated and fragmented primary log. As discussed earlier, having a log this fragmented may slow down operations that need to read the log file, such as log backups and the crash recovery process.

The way to reduce the primary log file to a reasonable size, and remove the fragmentation, is to shrink it, and then manually resize it. Note that we should never shrink the log as part of our standard maintenance operations, as it will simply need to grow again we add more data, and modify our existing data, and these log growth events are expensive, since SQL Server has to zero-initialize the new log space.

However, shrinking the log is permissible in situations such as this, in the knowledge that we have investigated and resolved the cause of the excessive log growth, and will then correctly size the log such that shrinking the log should be a “one off” event.

The recommended approach is use DBCC SHRINKFILE (see http://msdn.microsoft.com/enus/library/ms189493.aspx) to reclaim the space, and remove fragmentation. If we specify 0 (zero) as the target size, or don’t specify a size, SQL Server will attempt to shrink the log back to its initial size. Alternatively, we can specify a target size to which to shrink the file (such as “1”, if we wish SQL Server to order shrink the log to its smallest possible size). Here, we use zero in order to shrink the log back to its initial size (2 MB).

```sql
USE MyMessages;
GO
DBCC SHRINKFILE (N'MyMessages_log', target_size=0);
GO
```

<table>
<thead>
<tr>
<th>Disk</th>
<th>Field</th>
<th>Current Size</th>
<th>Minimum Size</th>
<th>Used Pages</th>
<th>Estimated Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>56316</td>
<td>256</td>
<td>66816</td>
<td>256</td>
</tr>
</tbody>
</table>

Listing 12: Shrinking the primary log file (partial success)
In the output from this command, we see the current database size (66816*8-KB pages) and minimum possible size after shrinking (256*8-KB pages). This is actually an indication that the shrink did not work fully. SQL Server shrunk the log to the point where the last VLF in the file contained part of the active log and then stopped. Check the messages tab for confirmation.

```sql
/*
* Cannot shrink log file 2 (MyMessages_log) because the logical log file located at the end of the file is in use.
(1 row(s) affected)
DBCC execution completed. If DBCC printed error messages, contact your system administrator.*/
```

Perform a log backup and try again.

```sql
USE master;
go
BACKUP DATABASE MyMessages
TO DISK = 'D:\SQLBackups\MyMessages_full.bak'
WITH INIT;
go
BACKUP LOG MyMessages
TO DISK = 'D:\SQLBackups\MyMessages.trn'
WITH INIT;
USE MyMessages;
go
DBCC SHRINKFILE (N'MyMessages_log', 0);
go
```

<table>
<thead>
<tr>
<th>Dbid</th>
<th>FileId</th>
<th>CurrentSize</th>
<th>MinimumSize</th>
<th>UsedPages</th>
<th>EstimatedPages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

Listing 13 Shrinking the primary log file after log backup
Having done this, we can now manually grow the log to the required size, in a similar way to that demonstrated in Listing 13.

**DBA lesson, Part 1:** if a log file grows excessively, and you need to add a temporary secondary log file, remove it as soon as it is no longer required. If you leave it hanging around it may slow down considerably database restore and recovery operations.

**DBA lesson, Part 2:** it is a bad idea to undersize the transaction log, and then let SQL Server auto-grow it in an uncontrolled fashion, in small increments. This can lead to log fragmentation, which may slow down log backup and database recovery operations. The way to avoid issues relating to expensive log growth events, and log fragmentation, is simply to set the correct initial size for the log (and data) file, allowing for current requirements, and predicted growth over a set period.

Ideally, having done this, the log would never auto-grow, which isn't to say that we should disable the auto-growth facility. It must be there as a safety mechanism, and we should set a reasonable auto-growth increment in order to avoid growth events fragmenting the log. However, having sized the log appropriately, we are not relying on auto-growth being the mechanism that controls log growth.

**Summary**

Every DBA should leave their organization's Backup and Recovery strategy in better shape than they found it, for quick disaster recovery with data, credibility, and customers intact; we hope this whitepaper helps you "bulletproof" yours.

If you'd like to learn more about how to build a “bullet-proof” backup and restore strategy for your databases, and optimize your transaction log management, check out the free eBooks [SQL Server Backup and Restore](#), by Shawn McGehee and [SQL Server Transaction Log Management](#), by Tony Davis and Gail Shaw.
Compress, verify, and encrypt SQL Server backups

Backup, restore, and verify – the three main features of any good backup strategy – are simple with SQL Backup Pro. It takes fast, small, secure SQL Server database backups, based on whatever schedule you set.

Learn more and download a free trial of SQL Back-up