SQLskills.com
immerse yourself in sql server

Common SQL Server Myths

Back in April 2010 I had the idea to do a DBA-Myth-A-Day series on my blog (www.SQLskills.com/blogs/paul), as a way of helping people debunk myths and misconceptions about SQL Server and how it works. It turned into a bit of a labor-of-love and some of the later blogs posts addressed tens of myths about a single subject.

Many people have asked for these blog posts to be collected together, which I agreed to do but never found the time. One of the blog readers (Peter Maloof—thank you!) very kindly did the work of collating the blog posts into a Word document, and I've worked it up into this PDF, which I present to you now.

Please feel free to re-distribute, reference, or print-out this document, just don't change it. If you like it, let me know and I'll produce some more.

Join our community! Sign up for our monthly newsletter to learn about new events, discount codes for training and consulting, and more exclusive content – click here.

I hope the information here helps you in your work – enjoy!

Paul S. Randal

CEO, SQLskills.com

February 15th 2011

Table of Contents

1/30: In-Flight Transactions Continue After a Failover

Myth #1: *After a failover, any in-flight transactions are continued.*

FALSE

Any time a failover occurs, some form of crash recovery has to occur. If the transaction hadn't committed on the server that the connection was to when the server/database went down, there's no way for SQL Server to automatically recreate all the transaction context and resurrect the in-flight transaction on the failover server - whether the failover used clustering, mirroring, log-shipping, or SAN replication.

For failover clustering, when a failover occurs, the SQL instance is restarted on another cluster node. All the databases go through crash recovery - which will roll-back all uncommitted transactions. (See my TechNet Magazine article from February 2009 for an explanation of crash recovery and how it works: Understanding Logging and Recovery in SQL Server.)

For database mirroring, the transaction log on the mirror database is constantly being used to perform redo operations (of the log records coming from the principal). When the mirror switches to being the principal, the transaction log switches into crash-recovery mode and recovers the database as if a crash had occurred - and then lets connections into the database.

For log shipping, transaction log backups are being replayed on the log shipping secondary database(s) periodically. When a failure on the primary occurs, the DBA brings the log shipping secondary online by completing the restore sequence (either bringing the database immediately online, or replaying all possible transaction log backups to bring the database as up-to-date as possible). The final part of any restore sequence is to run the undo portion of crash recovery - rolling back any uncommitted transactions.

With SAN replication, I/Os to the local SAN are shipped to the remote SAN to be replayed. When a failure occurs, the system connected to the remote SAN comes online and the databases go through crash recovery, in just the same way as for failover clustering (this is an overly simplistic explanation - but you get the idea).

The *only* technology that allows unbroken connections to a database when a failure occurs, is using virtualization with a live migration feature, where the VM comes up and the connections don't know they're talking to a different physical host server of the VM.

But whatever mechanism you're using - if a *connection* is broken, the in-flight transaction is lost. So your application needs to be coded to gracefully cope with broken connections and some form of transaction retry.

1 down - 29 to go!

2/30: DBCC CHECKDB Causes Blocking

Myth #2: *`DBCC CHECKDB` causes blocking because it takes locks by default.*

FALSE

In 7.0 and before, `DBCC CHECKDB` (and the rest of the family of consistency checking commands) was a nasty mess of nested loop C code that took table locks (and the nested loops made the algorithm essentially order-n-squared, for the programmers amongst you). This was not good, and so...

In 2000, a guy called Steve Lindell (who's still on the SQL team) rewrote `DBCC CHECKDB` to get a consistent view of the database using transaction log analysis. Essentially `DBCC CHECKDB` would prevent log truncation and then at the end of reading through the inconsistent (because of concurrent user transactions) database, run crash recovery on the transaction log inside itself. Basically, there was a brand new reproduction of the recovery code, but inside `DBCC CHECKDB`. I helped write a bunch of the log analysis code - tortuous, but fun. No, more tortuous. And there were some little problems with it - like the possibility of false failures... "if it gave errors, run it again and see if you get the same errors". Occasionally it would take table `SCH_S` locks (schema-stability locks) that would only block table scans and table schema modifications. The logging code was overall not good, and so...

In 2005, a guy called Paul Randal rewrote `DBCC CHECKDB` again to use database snapshots to get the consistent view of the database (as a database snapshot automatically provides a transactionally-consistent, point-in-time view of the database). No more nasty transaction log analysis code, not more locks *at all* - as accesses to the source database of a database snapshot never take locks - the buffer pool manages all the possibilities of race conditions.

You can read more on the internals of this stuff (both 2000 and 2005+) in the following posts:

- CHECKDB From Every Angle: Complete description of all CHECKDB stages
- CHECKDB From Every Angle: Why would CHECKDB run out of space?
- Database snapshots - when things go wrong
- Issues around DBCC CHECKDB and the use of hidden database snapshots
- Do transactions rollback when DBCC CHECKDB runs?
- Diskeeper 10 Intelliwrite corruption bug

Now, in all versions, if you use the `WITH TABLOCK` option, `DBCC CHECKDB` will take locks to guarantee a transactionally-consistent view, but I don't recommend doing that. The first thing it will try to do is grab an exclusive database lock, which in the vast majority of cases will fail (it only waits 20 seconds) because of concurrent database connections.

In 2000, the fact that it prevented log truncation could cause some issues - like the log having to grow - and in 2005, there can be issues around the use of database snapshots (see links above).

But by default, `DBCC CHECKDB` has been blocking-free since SQL Server 2000.

3/30: Instant File Initialization Can Be Controlled From Within SQL Server

Myth #3: *Instant file initialization can be a) enabled and b) disabled from within SQL Server.*

a) ***FALSE*** and b) ***TRUE***, respectively.

Instant file initialization is a little-known feature of SQL Server 2005 onwards that allows data files (only, not log files) to skip the usual zero initialization process that takes place. It's a fabulous way to reduce downtime when a disaster occurs and you have to restore a database from scratch - as the new data files that are created don't spend (potentially) hours being zero'd before the actual restore operation can take place.

I've done a blog post about instant file initialization misconceptions before (see Misconceptions around instant initialization) but that didn't cover this aspect of the feature.

You *cannot* enable it from within SQL Server. SQL Server does a one-time check at startup whether the SQL Server service account possesses the appropriate Windows permission (Perform Volume Maintenance Tasks a.k.a. `SE_MANAGE_VOLUME_NAME`) and then instant file initialization is enabled for that instance. Kimberly's excellent blog post Instant Initialization - What, Why, and How has the details on how to enable the feature (and a lot more besides).

You *can* check from within SQL Server to see if it's running. Enable trace flag 3004 (and 3605 to force the output to the errorlog) and then create a database. In the error log you'll see messages indicating that the log file is being zero initialized. If instant file initialization is NOT enabled, you'll also see messages about the data file being zero initialized.

You *can* disable instant file initialization from within SQL Server, albeit only temporarily. Turning on trace flag 1806 will disable instant file initialization while the trace flag is enabled. To turn it off permanently, you'll need to remove the security permission from the SQL Server service account.

These two trace flags were first documented in the SQL Server Premier Field Engineer Blog by MCM Cindy Gross (Twitter) and current-MCM-candidate Denzil Ribeiro - see their post How and Why to Enable Instant File Initialization.

If you're able to - turn this feature on!

4/30: DDL Triggers are INSTEAD OF Triggers

Myth #4: *DDL triggers (introduced in SQL Server 2005) are* `INSTEAD OF` *triggers.*

FALSE

DDL triggers are implemented as `AFTER` triggers, which means the operation occurs and is then caught in the trigger (and optionally rolled-back, if you put a `ROLLBACK` statement in the trigger body).

This means they're not quite as lightweight as you might think. Imagine doing the following:

```
ALTER TABLE MyBigTable ADD MyNewNonNullColumn VARCHAR (20) DEFAULT 'Paul';
```

If there's a DDL trigger defined for `ALTER_TABLE` events, or maybe even something more restrictive like `DDL_TABLE_EVENTS`, every row in the table will be expanded to include the new column (as it has a non-null default), and then the trigger will fire and the operation is rolled back by your trigger body. Not ideal at all. (Try it yourself and look in the log with `fn_dblog` - you'll see the operation rollback.)

What would be better in this case is to specifically `GRANT` or `DENY` the `ALTER` permission, or do something like only permitting DDL operations through stored-procedures that you create.

However, DDL triggers do allow you to effectively stop it happening, but in a relatively expensive way. And they do allow you to perform auditing of who did what, so I'm not saying they're without use - just be careful.

Kimberly has a great post on DDL triggers at "EXECUTE AS" and an important update your DDL Triggers (for auditing or prevention).

5/30: AWE Must Be Enabled on 64-Bit Servers

Myth #5: *AWE must be enabled on 64-bit servers.*

FALSE

(This one was suggested by fellow-MVP Jonathan Kehayias (blog|twitter))

There's a *huge* amount of confusion about AWE settings, locked pages, and what works/doesn't work, and what's required/not required on 32-bit and 64-bit servers, and in which editions.

In a nutshell:

- On 64-bit systems (2005+):
  - AWE is not required (and in fact enabling it does nothing)
  - Turning on the "Locked Pages in Memory" privilege prevents the buffer pool memory (and anything that uses single-page-at-a-time memory allocations) from being paged out
  - When the "Locked Pages in Memory" privilege is set, SQL Server uses the Windows AWE API to do memory allocations as it's a little bit faster
  - "Locked Pages in Memory" is supported by Standard and Enterprise editions (see this blog post for how to enable it in Standard edition)

- On 32-bit systems (2005+):
  - AWE is required to make use of extended virtual address space
  - The "Locked Pages in Memory" privilege must be enabled before AWE can be enabled
  - AWE is supported by Standard and Enterprise editions

No surprise that it's a bit confusing!

My good friend Bob Ward from CSS wrote a very detailed FAQ blog post that explains all of this - see Fun with Locked Pages, AWE, Task Manager, and the Working Set…

6/30: Three Null Bitmap Myths

The null bitmap keeps track of which columns in a record are null or not. It exists as a performance optimization to allow the Storage Engine to avoid having to read all of a record into the CPU when null columns are part of the SELECT list - thus minimizing CPU cache line invalidations (checkout this link for details of how CPU memory caches work and the MESI protocol). There are three pervasive myths to debunk here:

Myth #6a: *The null bitmap isn't always present.*

TRUE

The null bitmap is *always* present in data records (in heaps or the leaf-level of clustered indexes) - even if the table has no nullable columns. The null bitmap is *not* always present in index records (leaf level of nonclustered indexes, and non-leaf levels of clustered and nonclustered indexes).

Here's a simple script to prove it:

```
CREATE TABLE NullTest (c1 INT NOT NULL);
CREATE NONCLUSTERED INDEX NullTest_NC ON NullTest (c1);
GO
INSERT INTO NullTest VALUES (1);
GO

EXEC sp_allocationMetadata 'NullTest';
GO
```

You can get my sp_allocationMetadata script from my post Inside The Storage Engine: sp_AllocationMetadata - putting undocumented system catalog views to work.

Use the page IDs in the output from the script in the `First Page` column. Do the following:

```
DBCC TRACEON (3604);
DBCC PAGE (foo, 1, 152, 3); -- page ID from SP output where Index ID = 0
DBCC PAGE (foo, 1, 154, 1); -- page ID from SP output where Index ID = 2
GO
```

From the first `DBCC PAGE` dump of the heap data record:

```
Slot 0 Offset 0x60 Length 11

Record Type = PRIMARY_RECORD         Record Attributes =  NULL_BITMAP
Memory Dump @0x685DC060
```

From the second `DBCC PAGE` dump of the nonclustered index record:

```
Slot 0, Offset 0x60, Length 13, DumpStyle BYTE

Record Type = INDEX_RECORD          Record Attributes =          <<<<<<< No
null bitmap
Memory Dump @0x685DC060
```

Myth #6b: *The null bitmap only contains bits for nullable columns.*

FALSE

The null bitmap, when present, contains bits for all columns in the record, plus 'filler' bits for non-existent columns to make up complete bytes in the null bitmap. I already debunked this one with a long internals blog post last May - see Misconceptions around null bitmap size.

Myth #6c: *Adding another column to the table always results in an immediate size-of-data operation.*

FALSE

The only time that adding a column to a table results in a size-of-data operation (i.e. an operation that modifies every row in a table) is when the new column has a non-null default. In all other cases, the Storage Engine remembers that there are one or more additional columns that may not actually be present in the records themselves. I explained this in a little more depth in the blog post Misconceptions around adding columns to a table.

7/30: Multiple Mirrors and Log Shipping Load Delays

Myth #7: *A database can have multiple mirrors.*

FALSE

This one's pretty cut and dried - database mirroring only allows a single mirror of a principal database. If you want to have extra copies of the principal database, consider using log shipping. You can have as many log shipping secondaries as you want.

One other cool thing about log shipping is that you can have one of the secondaries set to have a load delay of, say, 8 hours. This means the log backups taken on the principal (don't you love it that the various technologies have different nomenclature:

- database mirroring: principal - mirror
- log shipping: primary - secondary
- replication: publisher - subscriber

Ok - this parenthetical clause kind-of got a life of its own...) won't be restored on the log shipping secondary until 8 hours have passed. If someone drops a table in production, it will pretty much immediately get dropped in the mirror (with whatever delay the SEND and WAIT queues have at the time - but you can't *stop it*) but the log shipping secondary with the load delay will still have it intact.

Incidentally, the SQLCAT team wrote a really good article debunking the myth (which stems from Books Online) that you can only mirror 10 databases per instance - see Mirroring a Large Number of Databases in a Single SQL Server Instance. Also take a look at the KB article I wrote for CSS last year which discusses the same thing: KB 2001270 Things to consider when setting up database mirroring in SQL Server.

8/30: Unicorns, Rainbows, and Online Index Operations

Myth #8: *Online index operations do not acquire locks.*

FALSE

Online index operations are not all unicorns and rainbows (for information about unicorns and rainbows - see http://whiteboardunicorns.com/ - safe for work).

Online index operations acquire short-term locks at the beginning and end of the operation, which can cause significant blocking problems.

At the start of the online index operation, a shared (S lock mode) table lock is required. This lock is held while the new, empty index is created; the versioned scan of the old index is started; and the minor version number of the table schema is bumped by 1.

The problem is, this S lock is queued along with all other locks on the table. No locks that are incompatible with an S table lock can be granted while the S lock is queued or granted. This means that updates are blocked until the lock has been granted and the operation started. Similarly, the S lock cannot be granted until all currently executing updates have completed, and their IX or X table locks dropped.

Once all the setup has been done (this is very quick), the lock is dropped, but you can see how blocking of updates can occur. Bumping the minor version number of the schema causes all query plans that update the table to recompile, so they pick up the new query plan operators to maintain the in-build index.

While the long-running portion of the index operation is running, no locks are held (where 'long' is the length of time your index rebuild usually takes).

When then index operation has completed, the new and old indexes are in lock-step as far as updates are concerned. A schema-modification lock (SCH_M lock mode) is required to complete the operation. You can think of this as a super-table-X lock - it's required to bump the major version number of the table - no operations can be running on the table, and no plans can be compiling while the lock is held.

There's a blocking problem here that's similar to when the S lock was acquired at the start of the operation - but this time, no read or write operations can start while the schema-mod lock is queued or granted, and it can't be granted until all currently running read and write activity on the table has finished.

Once the lock is held, the allocation structures for the old index are unhooked and put onto the deferred-drop queue, the allocation structure for the new index are hooked into the metadata for the old index (so the index ID doesn't change), the table's major version number is bumped, and hey presto! You've got a sparkly new index.

As you can see - plenty of potential for blocking at the start and end of the online index operation. So it should really be called 'almost online index operations', but that's not such a good marketing term... You can read more about online index operations in the whitepaper Online Indexing Operations in SQL Server 2005.

9/30: Data File shrink Does Not Affect Performance

Myth #9: *Data file shrink does not affect performance.*

Hahahahahahahahahahahahahaha! *<snort>*

<wipes tears from eyes, attempts to focus on laptop screen, cleans drool from keyboard>

FALSE

The only time a data file shrink won't affect performance is if you use the `WITH TRUNCATEONLY` option and there's free space at the end of file being shrunk.

Shrink affects performance while it's running. It's moving tons of data around, generating I/Os, fully logging everything it does, and burning CPU.

Shrink affects performance after it's run. All that log has to be backed up, log shipped, database mirrored, scanned by transactional replication, and so on. And if the data file grows again, the new space has to be zeroed out again (unless you have instant file initialization enabled).

Worst of all, shrink causes massive index fragmentation - which sucks down performance of range scans.

Unfortunately there was never time for me to rewrite the shrink code (I didn't write it like that in the first place) so we're probably stuck with its massive suckiness (that's a technical term) forever.

Check out this blog post where I go into more details and explain an alternative to using shrink: Why you should not shrink your data files and this one for how to manage your data files correctly: Importance of data file size management. And this one just because it's cool (safe for work): TGIF Time Warp.

And remember kids:

- Data file shrink is evil
- Shrinkdatabase is evil-er
- Auto-shrink is ***evil-est***

Shrink - just say no. With proper education and effective counseling, we can rid the world of this terrible affliction once and for all.

10/30: Database Mirroring Detects Failures Immediately

Myth #10: *Database mirroring detects failures immediately.*

FALSE

The marketing hype around database mirroring is that it provides instant detection of failures and instant failover.

No it doesn't. The speed with which a failure is detected depends on what the failure is, among other things.

The fastest detection of failure occurs when the principal SQL Server instance dies/crashes. When the once-per-second ping comes in from the mirror server, the OS on the principal server will know that there's no process listening on the TCP port the mirror server is pinging, and will let the mirror server know. This takes at most one second.

The next fastest detection of failure is when the OS on the principal server has died. In that case, there's no OS to respond to the ping from the mirror server. The mirror server will continue to ping once-per-second until the mirroring partner timeout expires. By default this is 10 seconds, but you may have increased it (for instance to ensure that a local cluster failover can occur before a mirroring failover to a remote server occurs). In this case, detection takes as long as the mirroring partner timeout is.

The next fastest example is a log drive becoming unavailable. SQL Server will continue to issue write requests to the I/O subsystem, will complain in the errorlog after 20 seconds without an I/O completion, and finally declare the log drive inaccessible after 40 seconds. The database is taken offline and a mirroring failure is declared. SQL Server is very patient, you see - with locks, for example, it will happily wait forever unless it detects a deadlock.

A page corruption might not even trigger a failure at all. If a regular query gets an 823 or 824, mirroring doesn't care (although it will attempt to fix them in 2008 (with a few caveats) - see SQL Server 2008: Automatic Page Repair with Database Mirroring). If the rollback of a query hits the 823 or 824 though, the database goes suspect immediately as it becomes transactionally inconsistent → mirroring failure.

The moral of the story is not to believe everything you read in the brochure :-)

11/30: Database Mirroring Failover is Instantaneous

Following on from yesterday's myth about database mirroring failure detection being instantaneous...

Myth #11: *Database mirroring failover is instantaneous.*

FALSE

A mirroring failover can occur automatically or can be manually initiated.

An automatic failover is performed by the mirror server (yes, the witness does NOT make the decision) if the mirror and witness agree they cannot contact the principal server (this process is called forming quorum) and the mirroring partnership state is `SYNCHRONIZED` (i.e. there were no unsent log records on the principal).

A manual failover is performed by you - either because a witness server wasn't present (and so the mirror cannot ever form the quorum required for an automatic failover) or because the mirroring partnership state was not `SYNCHRONIZED` at the time the principal server died.

Once the failure is initiated, the mirror database will not come online as the principal until the REDO queue has been processed. The REDO queue is the set of log records that have been received on the mirror from the principal, but have not yet been replayed in the mirror database. Even when using synchronous database mirroring, a transaction can commit on the principal once its log records are written to the mirror's log drive - it doesn't have to wait for the log records to be actually replayed in the mirror database. During a failover, the roll-forward of committed transactions must complete before the mirror database comes online, but rolling-back uncommitted transactions happens after the database comes online (using the same mechanism as fast recovery in Enterprise Edition - see my blog post Lock logging and fast recovery).

The roll-forward is single threaded on Standard Edition, and on Enterprise Edition where the server has less than 5 processor cores. On Enterprise Edition where the server has more than 5 processor cores, there's a redo thread for every 4 processor cores. So you can see how the failover time is really entirely dependent on how much log there is to process in the REDO queue, the processing power of the mirror server, and also what other workloads may be running on the mirror server that are competing for resources.

Because of the fact that mirroring is thought of as always performing a fast failover, many people do not monitor the REDO queue on the mirror. It's ***very*** important to do this as the amount of REDO queue correlates to the amount of downtime you'll experience during a mirroring failover.

For a bit more detail on all of this, see the Books Online entry Estimating the Interruption of Service During Role Switching.

12/30: Tempdb Should Always Have One Data File Per Processor Core

This is a myth I hear over and over and over...

Myth #12: *Tempdb should always have one data file per processor core.*

FALSE

Sigh. This is one of the most frustrating myths because there's so much 'official' information from Microsoft, and other blog posts that persists this myth.

There's only one tempdb per instance, and lots of things use it, so it's often a performance bottleneck. You guys know that already. But when does a performance problem merit creating extra tempdb data files?

When you see `PAGELATCH` waits on tempdb, you've got contention for in-memory allocation bitmaps. When you see `PAGEIOLATCH` waits on tempdb, you've got contention at the I/O subsystem level. You can think of a latch as kind of like a traditional lock, but much lighter wait, much more transitory, and used by the Storage Engine internally to control access to internal structures (like in-memory copies of database pages).

Fellow MVP Glenn Berry (twitter|blog) has a blog post with some neat scripts using the `sys.dm_os_wait_stats` DMV - the first one will show you what kind of wait is most prevalent on your server. If you see that it's `PAGELATCH` waits, you can use this script from newly-minted MCM and Microsoft DBA Robert Davis (twitter|blog). It uses the `sys.dm_os_waiting_tasks` DMV to break apart the wait resource and let you know what's being waited on in tempdb.

If you're seeing `PAGELATCH` waits on tempdb, then you can mitigate it using trace flag 1118 (fully documented in KB 328551) and creating extra tempdb data files. I wrote a long blog post debunking some myths around this trace flag and why it's still potentially required in SQL 2005 and 2008 - see Misconceptions around TF 1118.

On SQL Server 2000, the recommendation was one tempdb data file for each processor core. On 2005 and 2008, that recommendation persists, but because of some optimizations (see my blog post) you may not need one-to-one - you may be ok with the number of tempdb data files equal to 1/4 to 1/2 the number of processor cores.

Now this is all one big-ass generalization. I heard just last week of a customer who's tempdb workload was so high that they had to use 64 tempdb data files on a system with 32 processor cores - and that was the only way for them to alleviate contention. Does this mean it's a best practice? Absolutely not!

So, why is one-to-one not always a good idea? Too many tempdb data files can cause performance problems for another reason. If you have a workload that uses query plan operators that require lots of memory (e.g. sorts), the odds are that there won't be enough memory on the server to accommodate the operation, and it will spill out to tempdb. If there are too many tempdb data files, then the writing out of the temporarily-spilled data can be really slowed down while the allocation system does round-robin allocation.

Why would round-robin allocation cause things to slow down for memory-spills to tempdb with a large number of files? Round-robin allocation is per filegroup, and you can only have one filegroup in tempdb. With 16, 32, or more files in tempdb, and very large allocations happening from just a few threads, the extra synchronization and work necessary to do the round-robin allocation (looking at the allocation weightings for each file and deciding whether to allocate or decrement the weighting, plus quite frequently recalculating the weightings for all files - every 8192 allocations) starts to add up and become noticeable. It's very different from lots of threads doing lots of small allocations. It's also very different from allocating from a single-file filegroup - which is optimized (obviously) to not do round-robin. I really need to do a benchmarking blog post to show what I mean - but in the mean time, I've heard this from multiple customers who've created large numbers of tempdb files, and I know this from how the code works (my dev team owned the allocation code).

So you're damned if you do and damned if you don't, right? Potentially - yes, this creates a bit of a conundrum for you - how many tempdb data files should you have? Well, I can't answer that for you - except to give you these guidelines based on talking to many clients and conference/class attendees. Be careful to only create multiple tempdb data files to alleviate contention that you're experiencing - and not to push it to too many data files unless you really need to - and to be aware of the potential downfalls if you have to. You may have to make a careful balance to avoid helping one workload and hindering another.

Hope this helps.

PS: To address a comment that came in - no, the extra files don't *have* to be on separate storage. If all you're seeing it PAGELATCH contention, separate storage makes no difference as the contention is on in-memory pages. For PAGEIOLATCH waits, you most likely will need to use separate storage, but not necessarily - it may be that you need to move tempdb itself to different storage from other databases rather than just adding more tempdb data files. Analysis of what's stored where will be necessary to pick the correct path to take.

13/30: You Cannot Run DMVs When in the 80 Compat Mode

Myth #13: *You cannot run DMVs when in the 80 compat mode*.

FALSE

To start with, there's a lot of confusion about what compat mode means. Does it mean that the database can be restored/attached to a SQL Server 2000 server? No. It means that some T-SQL parsing, query plan behavior, hints and a few other things behave as they did in SQL Server 2000 (or 2005, if you're setting it to 90 on a 2008 instance).

In SQL Server 2008 you can use `ALTER DATABASE SET COMPATIBILITY_LEVEL` to change the compatibility level; in prior versions you use `sp_dbcmptlevel`. To see what the compability level controls, see:

- For SQL Server 2008, the Books Online entry ALTER DATABASE Compatibility Level.
- For SQL Server 2005, the Books Online entry sp_dbcmptlevel (Transact-SQL).

Compatibility level has no effect on the database physical version - which is what gets bumped up when you upgrade, and prevents a database being restored/attached to a previous version - as they have a maximum physical version number they can understand. See my blog post Search Engine Q&A #13: Difference between database version and database compatibility level for more details, and Msg 602, Level 21, State 50, Line 1 for details on the error messages you get when trying to attach/restore a database to a previous version.

But I digress, as usual :-)

One of the things that looks like it doesn't work is using DMVs when in the 80 compat mode. Here's a simple script to show you, using SQL Server 2005:

```
CREATE DATABASE DMVTest;
GO
USE DMVTest;
GO
CREATE TABLE t1 (c1 INT);
CREATE CLUSTERED INDEX t1c1 on t1 (c1);
INSERT INTO t1 VALUES (1);
GO

EXEC sp_dbcmptlevel DMVTest, 80;
GO

SELECT * FROM sys.dm_db_index_physical_stats (
     DB_ID ('DMVTest'), -- database ID
     OBJECT_ID ('t1'),  -- object ID       <<<<<< Note I'm using 1-part
naming
     NULL,              -- index ID
     NULL,              -- partition ID
     'DETAILED');       -- scan mode
GO
```

And the really useful error I get back is:

```
Msg 102, Level 15, State 1, Line 2
Incorrect syntax near '('.
```

How incredibly useful is that? It pinpoints the problem exactly - not.

Edit: After writing this I realized I'd fallen victim to my own myth too! DMVs *are* supported in the 80 compat-mode completely. What's *not* supported is calling a function (e.g. `OBJECT_ID`) as one of the DMV parameters. Thanks to Aaron Bertrand for pointing this out! (Apparently he pointed that out in the recent Boston class we taught, but I missed it.)

Here's the trick to using the functions as parameters. You change context to a database in the 90 or higher compatibility level - and then you can point the DMV at the database in the 80 compatibility level.

Very cool. Check it out:

```
USE master;
GO

SELECT * FROM sys.dm_db_index_physical_stats (
 DB_ID ('DMVTest'),          -- database ID
 OBJECT_ID ('DMVTest..t1'), -- object ID   <<<<<< Note I'm using 3-part
naming here now
 NULL,                       -- index ID
 NULL,                       -- partition ID
 'DETAILED');                -- scan mode
GO
```

And it works, even though the database `DMVTest` is in the 80 compatibility level.

One thing to be *very* careful of - you need to make sure you're using the correct object ID. If I'd just left the second parameter as `OBJECT_ID ('t1')`, it would have tried to find the object ID of the `t1` table in the `master` database. If it didn't find it, it will use the value NULL, which will cause the DMV to run against all tables in the `DMVTest` database. If by chance there's a t1 table in `master`, it's likely got a different object ID from the `t1` table in `DMVTest`, and so the DMV will puke (that's a technical term folks :-).

And `sys.dm_db_index_physical_stats` isn't a true DMV - Dynamic Management View - it's a Dynamic Management Function which does a *ton* of work potentially to return results - so you want to make sure you limit it to only the tables you're interested in. See my recent blog post Inside sys.dm_db_index_physical_stats for details of how it works and how expensive it can be.

So, you'll need to use the new 3-part naming option of `OBJECT_ID` in SQL Server 2005 onwards to make sure you're grabbing the correct object ID when going across database contexts.

Another way to do it is to use variables and pre-assign the values to them, which you can do from within the 80 compat-mode database:

```
DECLARE @databaseID INT;
DECLARE @objectID   INT;

SELECT @databaseID = DB_ID ('DMVTest');
SELECT @objectID   = OBJECT_ID ('t1');

SELECT * FROM sys.dm_db_index_physical_stats (
     @dbid,        -- database ID
     @objid,       -- object ID
     NULL,         -- index ID
     NULL,         -- partition ID
     'DETAILED');  -- scan mode
GO
```

Bottom line: another myth bites the dust!

14/30: Clearing the Log Zeroes Out Log Records

Myth #14: *Clearing the log zeroes out log records*.

FALSE

The transaction log is *always* zero initialized when first created, manually grown, or auto-grown. Do not confuse this with the process of clearing the log during regular operations. That simply means that one or more VLFs (Virtual Log Files) are marked as inactive and able to be overwritten. When log clearing occurs, nothing is cleared or overwritten. 'Clearing the log' is a very confusing misnomer. It means the exact same as 'truncating the log', which is another unfortunate misnomer, because the size of the log doesn't change at all.

You can read more about zero initialization of the log in my blog post Search Engine Q&A #24: Why can't the transaction log use instant initialization? And about how log clearing works in my TechNet Magazine article from February 2009 Understanding Logging and Recovery in SQL Server.

You can prove this to yourself using trace flag 3004. Turning it on will let you see when SQL Server is doing file zeroing operations (as I described in A SQL Server DBA myth a day: (3/30) instant file initialization can be controlled from within SQL Server). Turn it on and watch for messages coming during the day - you shouldn't see anything unless the log grows.

Here's a script to show you what I mean:

```
DBCC TRACEON (3004, 3605);
GO

-- Create database and put in SIMPLE recovery model so the log will clear on
checkpoint
CREATE DATABASE LogClearTest ON PRIMARY (
    NAME = 'LogClearTest_data',
    FILENAME = N'D:\SQLskills\LogClearTest_data.mdf')
LOG ON (
    NAME = 'LogClearTest_log',
    FILENAME = N'D:\SQLskills\LogClearTest_log.ldf',
    SIZE = 20MB);
GO

-- Error log mark 1
ALTER DATABASE LogClearTest SET RECOVERY SIMPLE;
GO

USE LogClearTest;
GO

-- Create table and fill with 10MB - so 10MB in the log
CREATE TABLE t1 (c1 INT IDENTITY, c2 CHAR (8000) DEFAULT 'a');
GO
INSERT INTO t1 DEFAULT VALUES;
GO 1280
```

```
-- Clear the log
CHECKPOINT;
GO

-- Error log mark 2
ALTER DATABASE LogClearTest SET RECOVERY SIMPLE;
GO
```

And in the error log we see:

```
2010-04-13 13:20:27.55 spid53      DBCC TRACEON 3004, server process ID (SPID) 53.
This is an informational message only; no user action is required.
2010-04-13 13:20:27.55 spid53      DBCC TRACEON 3605, server process ID (SPID) 53.
This is an informational message only; no user action is required.
2010-04-13 13:20:27.63 spid53      Zeroing D:\SQLskills\LogClearTest_log.ldf from page 0
to 2560 (0x0 to 0x1400000)
2010-04-13 13:20:28.01 spid53      Zeroing completed on D:\SQLskills\LogClearTest_log.ldf
2010-04-13 13:20:28.11 spid53      Starting up database 'LogClearTest'.
2010-04-13 13:20:28.12 spid53      FixupLogTail() zeroing
D:\SQLskills\LogClearTest_log.ldf from 0x5000 to 0x6000.
2010-04-13 13:20:28.12 spid53      Zeroing D:\SQLskills\LogClearTest_log.ldf from page 3
to 63 (0x6000 to 0x7e000)
2010-04-13 13:20:28.14 spid53      Zeroing completed on D:\SQLskills\LogClearTest_log.ldf
2010-04-13 13:20:28.16 spid53      Setting database option RECOVERY to SIMPLE for
database LogClearTest.
2010-04-13 13:20:29.49 spid53      Setting database option RECOVERY to SIMPLE for
database LogClearTest.
```

The two `ALTER DATABASE` commands serve as markers in the error log. There's clearly no zeroing occuring from the `CHECKPOINT` between the two `ALTER DATABASE` commands. To further prove to yourself, you can add in calls to `DBCC SQLPERF (LOGSPACE)` before and after the `CHECKPOINT`, to show that the log is clearing when the `CHECKPOINT` occurs (watch the value in the `Log Space Used (%)` column decrease).

15/30: Checkpoint Only Writes Pages From Committed Transactions

I'm half way through my debunking month!

Myth #15: *Checkpoint only writes pages from committed transactions*.

FALSE

This myth has persisted for *ages* and is linked to a misunderstanding of how the overall logging and recovery system works. A checkpoint always writes out all pages that have changed (known as being *marked dirty*) since the last checkpoint, or since the page was read in from disk. It doesn't matter whether the transaction that changed a page has committed or not - the page is written to disk regardless. The only exception is for tempdb, where data pages are not written to disk as part of a checkpoint. Here are some blog post links with deeper information:

- TechNet Magazine article on Understanding Logging and Recovery in SQL Server
- Blog post: How do checkpoints work and what gets logged
- Blog post: What does checkpoint do for tempdb?

You can watch what a checkpoint operation is doing using a few traceflags:

- 3502: writes to the error log when a checkpoint starts and finishes
- 3504: writes to the error log information about what is written to disk

To use these traceflags, you must enable them for all threads using DBCC TRACEON (3502, 3504, -1) otherwise you won't see any output.

Here's a quick script that proves that dirty pages from uncommitted transactions are written out during a checkpoint. First the setup:

```
CREATE DATABASE CheckpointTest;
GO
USE CheckpointTest;
GO
CREATE TABLE t1 (c1 INT IDENTITY, c2 CHAR (8000) DEFAULT 'a');
CREATE CLUSTERED INDEX t1c1 on t1 (c1);
GO

SET NOCOUNT ON;
GO

CHECKPOINT;
GO

DBCC TRACEON (3502, 3504, -1);
GO
```

And now an uncommitted transaction that causes 10MB of pages to be dirtied, followed by a checkpoint:

```
BEGIN TRAN;
GO
INSERT INTO t1 DEFAULT VALUES;
GO 1280

CHECKPOINT;
GO
```

And in the error log we see:

```
2010-04-15 13:31:25.09 spid52      DBCC TRACEON 3502, server process ID
(SPID) 52. This is an informational message only; no user action is required.
2010-04-15 13:31:25.09 spid52      DBCC TRACEON 3504, server process ID
(SPID) 52. This is an informational message only; no user action is required.
2010-04-15 13:31:25.09 spid52      Ckpt dbid 8 started (0)
2010-04-15 13:31:25.09 spid52      About to log Checkpoint begin.
2010-04-15 13:31:25.09 spid52      Ckpt dbid 8 phase 1 ended (0)
2010-04-15 13:31:25.71 spid52      FlushCache: cleaned up 1297 bufs with 50
writes in 625 ms (avoided 0 new dirty bufs)
2010-04-15 13:31:25.71 spid52                  average throughput:  16.21
MB/sec, I/O saturation: 70
2010-04-15 13:31:25.71 spid52                  last target outstanding: 2
2010-04-15 13:31:25.71 spid52      About to log Checkpoint end.
2010-04-15 13:31:25.71 spid52      Ckpt dbid 8 complete
```

Clearly all the pages were written out, even though the transaction had not committed.

16/30: Corruptions and Repairs

Myth #16: *Variety of myths around corruptions and repairs...*

All of them are FALSE

There are a bunch of things I hear over and over around what repair can and cannot do, what can cause corruptions, and whether corruptions can disappear. A bunch of these I've already written up in blog posts over the last few years so rather than regurgitate the same stuff, this mythbuster post is some interesting links to keep you happy.

Firstly, around what repair can and cannot do. I wrote a blog post Misconceptions around database repair that covers 13 separate myths and misconceptions - from whether you can run repair separately from `DBCC CHECKDB` (no!) to whether `REPAIR_ALLOW_DATA_LOSS` will cause data loss (I'm confused as to why the name is confusing :-).

Secondly, I've heard many people complaining the `DBCC CHECKDB` shows corruptions which then 'disappear' when they run `DBCC CHECKDB` again. There's a very good reason for this - the database pages that were exhibiting corruptions are no longer part of the allocated set of pages in the database by the time `DBCC CHECKDB` is run a second time - so they don't show as corruptions. I explain this in great detail in the blog post Misconceptions around corruptions: can they disappear?.

Lastly, there's a pervasive myth that interrupting a long-running operation (like shrink, index rebuild, bulk load) can cause corruption. No. Unless there's a corruption bug in SQL Server (which happens sometimes, but rarely), nothing you can do from T-SQL can cause corruption. I wrote a detailed blog post on this a couple of years ago - see Search Engine Q&A #26: Myths around causing corruption.

17/30: Page Checksums

A few people have suggested some of the myths around page checksums so today is another multi-mythbusting extravaganza! Well, I get excited at least :-)

I described page checksums in depth in the blog post How to tell if the IO subsystem is causing corruptions?

Myth #17: *variety of myths around page checksums*.

All of them are FALSE

17a) *page checksums are enabled automatically when you upgrade from SQL Server 2000 or 70*

No. You must explicitly enable page checksums on upgraded databases using `ALTER DATABASE blah SET PAGE_VERIFY CHECKSUM`. Databases that are created on SQL Server 2005 and 2008 will have page checksums enabled automatically unless you change the setting in the model database - which you shouldn't.

17b) *page checksums are error correcting*

No. Page checksums can detect errors in a page but are not like CRC-based checksums in network protocols that can correct single-bit errors.

17c) *enabling page checksums kicks off a background task to put a page checksum on each database page*

No. There is no process, background or otherwise, that can put a page checksum on each page. This is a major bummer (technical term :-) as it means you must perform index rebuilds or other size-of-data operations to actually put a page checksum on the pages. This myth goes hand-in-hand with 17d below...

17d) *simply reading the pages is enough to put a page checksum on them (e.g. with a backup or DBCC CHECKDB)*

No. A page checksum is only put on a page when it is read into memory, changed, and then written back out to disk.

17e) *when a database is changed from torn-page detection to page checksums, all torn-page detection is lost*

No. Pages know whether they are protected through torn-page detection, a page checksum, or nothing at all. As mentioned above, pages aren't changed to a page checksum until they're physically altered. I went into this in great detail with an example script in the blog post Inside The Storage Engine: Does turning on page checksums discard any torn-page protection?

17f) *page checksums detect corruption immediately*

This myth was suggested for debunking by fellow MVP Gail Shaw (twitter|blog) and is of course untrue. A damaged page cannot be detected until it is read into memory and the buffer pool checks the validity of the page checksum.

18/30: FILESTREAM Storage, Garbage Collection, and More

Myth #18: *Various FILESTREAM data myths*.

All of them are FALSE

18a) *FILESTREAM data can be stored remotely*

No. A FILESTREAM data container (the invented name for the NTFS directory structure that stores the FILESTREAM data) must adhere to the same locality rules as regular database data and log files - i.e. it must be placed on storage 'local' to the Windows server running SQL Server. FILESTREAM data can be *accessed* using a UNC path, as long as the client has contacted the local SQL Server and obtained the necessary transaction context to use when opening the FILESTREAM file.

18b) *FILESTREAM data containers can be nested*

No. Two FILESTREAM data containers for the same database may share a root directory, but data containers cannot be nested, and data containers from different databases cannot share a directory. I blogged an example script that shows this at Misconceptions around FILESTREAM storage.

18c) *partial updates to FILESTREAM files are supported*

No. Any update to a FILESTREAM file creates an entirely new FILESTREAM file, which will be picked up by log backups. This is why FILESTREAM cannot be used with database mirroring - the amount of data to be pushed to the mirror would be prohibitive. Hopefully a future version of SQL Server will implement a differencing mechanism that will allow partial updates and hence database mirroring compatibility.

18d) *FILESTREAM garbage collection occurs instantaneously*

No. Garbage collection occurs once a FILESTREAM file is no longer required (usually meaning it's been backed up by a log backup) AND a further checkpoint occurs. This is very non-intuitive and leads many people to think that FILESTREAM garbage collection isn't working for them. I explained this in detail in my post FILESTREAM garbage collection.

18e) *FILESTREAM directory and filenames cannot be determined*

No. There is method to the seeming madness of the GUIDs and weird filenames. The actual FILESTREAM filenames are the character representation of the LSN of the log record that described the creation of the file. The table and column directory names are GUIDs that you can get from system tables. (To be entirely accurate, the table directories are actually 'rowset' directories as far as the Storage Engine is concerned - with a rowset being equal to the portion of a table in a single partition of a partitioned tabled).

I have two blog posts which crack open the various names and system tables and show you how to figure out what's what:

- FILESTREAM directory structure explains how to figure out a FILESTREAM filename from a table row
- FILESTREAM directory structure - where do the GUIDs come from? has a kind of self-explanatory post title :-)

19/30: TRUNCATE TABLE is Non-Logged

Today's myth is very persistent, so it's high time it was debunked with a nice script to prove it too!

Myth #19: *A TRUNCATE TABLE operation is non-logged*.

FALSE

There is no such thing as a non-logged operation in a user database. The only non-logged operations that SQL Server performs are those on the version store in tempdb.

A `TRUNCATE TABLE` operation does a wholesale delete of all data in the table. The individual records are not deleted one-by-one, instead the data pages comprising the table are simply deallocated. The allocations are unhooked from the table and put onto a queue to be deallocated by a background task called the deferred-drop task. The deferred-drop task does the deallocations instead of them being done as part of the regular transaction so that no locks need to be acquired while deallocating entire extents. Before SQL Server 2000 SP3 (when this process was put into SQL Server), it was possible to run out of memory while acquiring locks during a `TRUNCATE TABLE` operation.

Here's an example script:

```
CREATE DATABASE TruncateTest;
GO
USE TruncateTest;
GO
ALTER DATABASE TruncateTest SET RECOVERY SIMPLE;
GO
CREATE TABLE t1 (c1 INT IDENTITY, c2 CHAR (8000) DEFAULT 'a');
CREATE CLUSTERED INDEX t1c1 on t1 (c1);
GO

SET NOCOUNT ON;
GO

INSERT INTO t1 DEFAULT VALUES;
GO 1280

CHECKPOINT;
GO
```

The database is in the `SIMPLE` recovery mode so the log clears out on each checkpoint (for simplicity - ha ha :-)

Wait for a minute or so (there may be some ghost record cleanup that occurs) and check how many rows are in the log:

```
SELECT COUNT (*) FROM fn_dblog (NULL, NULL);
GO
```

If you don't get a result of 2, do another checkpoint and check the log record count again until it comes back at 2. Now the database is completely quiescent and any new log records are from stuff we're doing. Now we'll do the truncate:

```
TRUNCATE TABLE t1;
GO

SELECT COUNT (*) FROM fn_dblog (NULL, NULL);
GO
```

I get back a result of 541 log records - clearly the operation is not non-logged, but it's clearly also not deleting each record - as I inserted 1280 records. If we look in the log we'll see:

```
SELECT
 [Current LSN], [Operation], [Context],
 [Transaction ID], [AllocUnitName], [Transaction Name]
FROM fn_dblog (NULL, NULL);
GO

Current LSN             Operation           Context             Transaction
ID  AllocUnitName               Transaction Name
----------------------  ------------------  ------------------  -------------
-  -------------------------  ----------------
00000081:000001a6:0016  LOP_BEGIN_CKPT      LCX_NULL
0000:00000000   NULL                        NULL
00000081:000001a9:0001  LOP_END_CKPT        LCX_NULL
0000:00000000   NULL                        NULL
00000081:000001aa:0001  LOP_BEGIN_XACT      LCX_NULL
0000:00001072   NULL                        TRUNCATE TABLE
00000081:000001aa:0002  LOP_LOCK_XACT       LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:0003  LOP_LOCK_XACT       LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:0004  LOP_LOCK_XACT       LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:0005  LOP_COUNT_DELTA     LCX_CLUSTERED
0000:00000000   sys.sysallocunits.clust     NULL
00000081:000001aa:0006  LOP_COUNT_DELTA     LCX_CLUSTERED
0000:00000000   sys.sysrowsets.clust        NULL
00000081:000001aa:0007  LOP_COUNT_DELTA     LCX_CLUSTERED
0000:00000000   sys.sysrowsetcolumns.clust  NULL
00000081:000001aa:0008  LOP_COUNT_DELTA     LCX_CLUSTERED
0000:00000000   sys.sysrowsetcolumns.clust  NULL
00000081:000001aa:0009  LOP_COUNT_DELTA     LCX_CLUSTERED
0000:00000000   sys.sysrowsetcolumns.clust  NULL
00000081:000001aa:000a  LOP_HOBT_DDL        LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:000b  LOP_MODIFY_ROW      LCX_CLUSTERED
0000:00001072   sys.sysallocunits.clust     NULL
00000081:000001aa:000c  LOP_MODIFY_COLUMNS  LCX_CLUSTERED
0000:00001072   sys.sysallocunits.clust     NULL
00000081:000001aa:000d  LOP_DELETE_ROWS     LCX_MARK_AS_GHOST
0000:00001072   sys.sysserefs.clust         NULL
00000081:000001aa:000e  LOP_MODIFY_HEADER   LCX_PFS
```

```
0000:00000000   Unknown Alloc Unit          NULL
00000081:000001aa:000f  LOP_SET_BITS        LCX_PFS
0000:00000000   sys.sysserefs.clust         NULL
00000081:000001aa:0010  LOP_INSERT_ROWS     LCX_CLUSTERED
0000:00001072   sys.sysserefs.clust         NULL
00000081:000001aa:0011  LOP_MODIFY_ROW      LCX_SCHEMA_VERSION
0000:00000000   sys.sysobjvalues.clst       NULL
00000081:000001aa:0012  LOP_INSERT_ROWS     LCX_CLUSTERED
0000:00001072   sys.sysallocunits.clust     NULL
00000081:000001aa:0013  LOP_INSERT_ROWS     LCX_CLUSTERED
0000:00001072   sys.sysserefs.clust         NULL
00000081:000001aa:0014  LOP_HOBT_DDL        LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:0015  LOP_MODIFY_ROW      LCX_CLUSTERED
0000:00001072   sys.sysrowsets.clust        NULL
00000081:000001aa:0016  LOP_IDENT_SENTVAL   LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001aa:0017  LOP_MODIFY_ROW      LCX_CLUSTERED
0000:00001072   sys.syscolpars.clst         NULL
00000081:000001aa:0018  LOP_COMMIT_XACT     LCX_NULL
0000:00001072   NULL                        NULL
00000081:000001b0:0001  LOP_BEGIN_XACT      LCX_NULL
0000:00001073   NULL                        DeferredAllocUnitDrop::Process
00000081:000001b0:0002  LOP_LOCK_XACT       LCX_NULL
0000:00001073   NULL                        NULL
00000081:000001b0:0003  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0004  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0005  LOP_SET_BITS        LCX_SGAM
0000:00000000   Unknown Alloc Unit          NULL
00000081:000001b0:0006  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0007  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0008  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0009  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000a  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000b  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000c  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000d  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000e  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:000f  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0010  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0011  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0012  LOP_MODIFY_ROW      LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
```

```
00000081:000001b0:0013  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0014  LOP_SET_BITS        LCX_SGAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0015  LOP_SET_BITS        LCX_GAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0016  LOP_SET_BITS        LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0017  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0018  LOP_SET_BITS        LCX_GAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:0019  LOP_SET_BITS        LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:001a  LOP_MODIFY_ROW      LCX_PFS
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:001b  LOP_SET_BITS        LCX_GAM
0000:00001073   Unknown Alloc Unit          NULL
00000081:000001b0:001c  LOP_SET_BITS        LCX_IAM
0000:00001073   Unknown Alloc Unit          NULL
etc
```

The transaction with ID `0000:00001072` is the implicit transaction of my `TRUNCATE TABLE` statement (as you can see from the transaction name). It commits at LSN `00000081:000001aa:0018` and then straight afterwards is the start of the deferred-drop transaction. As you can see from the log records, it's just deallocating the pages and extents.

Well, you can't really see that unless you know what all the log records are doing, so let's have a quick look at the descriptions:

```
SELECT
 [Current LSN], [Operation], [Lock Information], [Description]
FROM fn_dblog (NULL, NULL);
GO
```

and you'll be able to see the locks that are logged to allow fast recovery to work (see my blog post [Lock logging and fast recovery](#) for an in-depth explanation) and also the description of the operations being performed. Here's a small selection from the start of the deferred-drop transaction:

```
Operation       Lock
Information                                              Description
--------------  ---------------------------------------------------------
-------------------------------------------------------
LOP_BEGIN_XACT  NULL
DeferredAllocUnitDrop::Process
LOP_LOCK_XACT   HoBt 0:ACQUIRE_LOCK_IX ALLOCATION_UNIT: 8:72057594042384384
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:0
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:152
Deallocated 0001:00000098
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:1
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:156
Deallocated 0001:0000009c
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:2
```

```
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:157
Deallocated 0001:0000009d
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:3
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:158
Deallocated 0001:0000009e
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:4
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:159
Deallocated 0001:0000009f
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:5
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:160
Deallocated 0001:000000a0
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:6
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:161
Deallocated 0001:000000a1
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X RID: 8:1:153:7
LOP_MODIFY_ROW  HoBt 72057594042384384:ACQUIRE_LOCK_X PAGE: 8:1:162
Deallocated 0001:000000a2
LOP_SET_BITS    NULL
ClearBit 0001:000000a0
LOP_SET_BITS    NULL
Deallocated 1 extent(s) starting at page 0001:000000a0
LOP_SET_BITS    NULL
LOP_MODIFY_ROW
Deallocated 0001:000000a8;Deallocated 0001:000000a9;Deallocated
0001:000000aa;Deallocated 0001:000000ab;Deallocated 0001:000000ac;Deallocated
0001:000000ad;Deallocated 0001:000000ae;Deallocated 0001:000000af
LOP_SET_BITS    NULL
Deallocated 1 extent(s) starting at page 0001:000000a8
LOP_SET_BITS    NULL
LOP_MODIFY_ROW
Deallocated 0001:000000b0;Deallocated 0001:000000b1;Deallocated
0001:000000b2;Deallocated 0001:000000b3;Deallocated 0001:000000b4;Deallocated
0001:000000b5;Deallocated 0001:000000b6;Deallocated 0001:000000b7
```

The first 8 operations are deallocating the 8 pages that are allocated from mixed extents when the table was first populated and after that it switches to deallocating an entire extent at a time. Have a poke around - this stuff's really fascinating. Note also the `LOP_LOCK_XACT` log record, which just describes the acquisition of a lock - not a change to the database. You'll notice that the extent deallocations don't have any locks protecting them - that's what the allocation unit IX lock is doing.

By the way, if you have nonclustered indexes on the table too, they are also dealt with the same way and there will be a single deferred-drop transaction which deallocates all the pages from both the table and all nonclustered indexes, one allocation unit at a time. Try it and you'll see what I mean.

Myth debunked!

PS There's another myth that a `TRUNCATE TABLE` can't be rolled back - I debunk that in this old blog post: Search Engine Q&A #10: When are pages from a truncated table reused?

20/30: Restarting a Log Backup Chain Requires a Full Database Backup

This myth is one of the most common and I've come across very few people who know the truth.

Myth #20: *after breaking the log backup chain, a full database backup is required to restart it*.

FALSE

A normal transaction log backup contains all the transaction log generated since the previous log backup (or since the first ever full backup if it's the first ever log backup for the database). There are various operations that will break the log backup chain - i.e. prevent SQL Server from being able to take another log backup until the chain is restarted. The list of such operations includes:

- Switching from the `FULL` or `BULK_LOGGED` recovery models into the `SIMPLE` recovery model
- Reverting from a database snapshot
- Performing a `BACKUP LOG` using the `WITH NO_LOG` or `WITH TRUNCATE_ONLY` (which you can't do any more in SQL Server 2008 - yay!)
  - See the blog post BACKUP LOG WITH NO_LOG - use, abuse, and undocumented trace flags to stop it

Here's an example script that shows you what I mean:

```
CREATE DATABASE LogChainTest;
GO
ALTER DATABASE LogChainTest SET RECOVERY FULL;
GO
BACKUP DATABASE LogChainTest TO DISK = 'C:\SQLskills\LogChainTest.bck' WITH
INIT;
GO
BACKUP LOG LogChainTest TO DISK = 'C:\SQLskills\LogChainTest_log1.bck' WITH
INIT;
GO
ALTER DATABASE LogChainTest SET RECOVERY SIMPLE;
GO
ALTER DATABASE LogChainTest SET RECOVERY FULL;
GO

Processed 152 pages for database 'LogChainTest', file 'LogChainTest' on file
1.
Processed 1 pages for database 'LogChainTest', file 'LogChainTest_log' on
file 1.
BACKUP DATABASE successfully processed 153 pages in 0.088 seconds (14.242
MB/sec).
Processed 2 pages for database 'LogChainTest', file 'LogChainTest_log' on
file 1.
BACKUP LOG successfully processed 2 pages in 0.033 seconds (0.341 MB/sec).
```

I created a database, put it into the `FULL` recovery model, started the log backup chain, and then momentarily bounced it into the `SIMPLE` recovery model and back to `FULL`.

Now if I try to take a log backup:

```
BACKUP LOG LogChainTest TO DISK = 'C:\SQLskills\LogChainTest_log2.bck' WITH
INIT;
GO

Msg 4214, Level 16, State 1, Line 1
BACKUP LOG cannot be performed because there is no current database backup.
Msg 3013, Level 16, State 1, Line 1
BACKUP LOG is terminating abnormally.
```

SQL Server knows that I performed an operation which means the next log backup will NOT contain all the log generated since the previous log backup, so it doesn't let me do it.

The myth says that a full database backup is required to restart the log backup chain. In reality, all I need is a data backup that bridges the LSN gap. A differential backup will do:

```
BACKUP DATABASE LogChainTest TO DISK = 'C:\SQLskills\LogChainTest_Diff1.bck'
WITH INIT, DIFFERENTIAL;
GO
BACKUP LOG LogChainTest TO DISK = 'C:\SQLskills\LogChainTest_log2.bck' WITH
INIT;
GO

Processed 40 pages for database 'LogChainTest', file 'LogChainTest' on file
1.
Processed 1 pages for database 'LogChainTest', file 'LogChainTest_log' on
file 1.
BACKUP DATABASE WITH DIFFERENTIAL successfully processed 41 pages in 0.083
seconds (4.040 MB/sec).
Processed 1 pages for database 'LogChainTest', file 'LogChainTest_log' on
file 1.
BACKUP LOG successfully processed 1 pages in 0.010 seconds (0.768 MB/sec).
```

This is really cool because you don't need to take a (potentially very large) full database backup to be able to continue with regular log backups.

If you have a backup strategy that involves file or filegroup backups as well as database backups, you can even restart the log backup chain after a single file differential backup! Take note, however, that to be able to restore that database, you'd need to have a data backup of each portion of it that bridges the LSN gap (i.e. a file or filegroup full or differential backup) but that's more complicated than I want to go into in this post.

Another myth bites the dust!

21/30: Corruption Can Be Fixed By Restarting SQL Server

This myth (and derivatives) are very common among non-DBAs as so many Windows problems can be fixed by rebooting the computer (yes, I still see this on servers, Windows 7 etc. - try changing the terminal services port number without a reboot).

Myth #21: *Database corruption can be fixed by restarting SQL Server or rebooting the Windows server or detaching/attaching the database*.

FALSE

None of these operations will cause SQL Server to fix a corruption. A corrupt page needs to be restored or repaired in some way - neither of which occur when SQL Server, Windows, or the physical machine is restarted, nor when a corrupt database is detached.

In fact, detaching and then trying to re-attach a corrupt database might be one of the worst things you can do if the corruption is such that the database cannot have crash recovery run on it (i.e. it is in the SUSPECT or RECOVERY_PENDING state) - as part of the process of attaching a database that needs crash recovery is... to run crash recovery - and if it can't be done, the attach fails. This is when the hack-the-database-back-in trick becomes necessary (see my blog post TechEd Demo: Creating, detaching, re-attaching, and fixing a suspect database). Don't ever detach a corrupt database.

Now - here are some interesting behaviors that could look like rebooting fixes the corruption:

- If the corruption was just a corrupt page image in memory, but the on-disk image of the page is not corrupt, the corruption will seem to have been fixed by rebooting.
- If the corruption was real, but you did something else as part of the reboot that caused that page to no longer be allocated, the corruption will seem to have been fixed by rebooting. This is the same as the myth I debunked a while ago in the blog post Misconceptions around corruptions: can they disappear?
- If the I/O subsystem is rebooted too, and an I/O was 'stuck' somehow in the I/O subsystem (e.g. a persistent stale read issue) then the corruption will seem to have been fixed by rebooting. This isn't really fixing the corruption; this is allowing a broken I/O subsystem to recover. The I/O subsystem is still broken. I've seen this case maybe three or four times in my life.

Bottom line - to recover from corruption, you need some combination of backups and/or a redundant system to failover to. Rebooting is not the answer and will almost invariably just waste time.

22/30: Resource Governor allows I/O Governing

Resource governor is a great feature in SQL Server 2008, but there are some misconceptions about what it can do...

Myth #22: *Resource governor allows governing of I/O activity*.

FALSE

Resource governor does not govern I/O activity in any way - hopefully that will be something that's added in the next major release of SQL Server. It'll be a lot more useful once you can use to prevent run-away queries doing huge table scans, or spills into tempdb.

Some other things that resource governor does *not* do in the first version:

- Allow governing of buffer pool memory. The memory governing it performs is for query execution memory grants only - not for how much space is used in the buffer pool by pages being processed by a query.
- Allow two instances of SQL Server to cooperatively govern CPU and memory resources. Multi-instance governing has to be done with Windows Server Resource Manager, and then with resource governor for each instance.
- Allow a connection to be notified that it has been governed in some way.

Don't get me wrong - it's great, but it will be a lot better with these additions too.

Our friend and fellow-MVP Aaron Bertrand (twitter|blog) and SQL PM Boris Baryshnikov wrote a comprehensive whitepaper that you should read for more details: Using the Resource Governor.

23/30: Lock Escalation

Another really commonly-held belief...

Myth #23: *lock escalation goes row-to-page and then page-to-table*.

FALSE

Nope, never. Lock escalation in SQL Server 2005 and before goes directly to a table lock always.

In SQL Server 2005 (and 2008) you can change the behavior of lock escalation (if you really know what you're doing) using these trace flags:

- 1211 - disables lock escalation totally and will allow lock memory to grow to 60% of dynamically allocated memory (non-AWE memory for 32-bit and regular memory for 64-bit) and will then further locking will fail with an out-of-memory error
- 1224 - disables lock escalation until 40% of memory is used and then re-enables escalation

1211 takes precedence over 1224 if they're both set - so be doubly careful. You can find more info on these trace flags in Books Online.

In SQL Server 2008, you can change the behavior of lock escalation per table using the `ALTER TABLE blah SET (LOCK_ESCALATION = XXX)` where `XXX` is one of:

- `TABLE`: always escalate directly to a table lock.
- `AUTO`: if the table is partitioned, escalate to a partition-level lock, but then don't escalate any further.
- `DISABLE`: disable lock escalation. This doesn't disable table locks - as the Books Online entry says, a table lock may be required under some circumstances, like a table scan of a heap under the `SERIALIZABLE` isolation level.

Back in January 2008 I blogged an example of setting up a partitioned table and showing partition-level lock escalation in action - see SQL Server 2008: Partition-level lock escalation details and examples.

You may ask why the `AUTO` option isn't the default in SQL Server 2008? It's because some early-adopters found that their applications started to deadlock using that option. So, just as with the lock escalation trace flags, be careful about turning on the `AUTO` option in SQL Server 2008.

24/30: Twenty-Six Restore Myths

One area I haven't touched on yet in the series is `RESTORE` - and there are a *ton* of misconceptions here (so many, in fact, that I can't cover them all in a single post!). Last Saturday's post busted 6 page checksum myths, and last Sunday's busted 5 FILESTREAM myths so I need to beat those today.

In fact, I'm going to do one myth for each letter of the alphabet as everyone else is still asleep here - it's another multi-mythbusting extravaganza!

Myth #24: *Twenty-six myths around restore operations...*

All of them are FALSE!

24a) *it is possible to do a point-in-time restore using `WITH STOPAT` on a full or differential backup*

No. The syntax looks like it allows it, but it's just a syntactical nicety to allow you to do the best practice of using `WITH STOPAT` on every restore operation in the point-in-time restore sequence so you don't accidentally go past it. I go into more details in the old blog post Debunking a couple of myths around full database backups.

24b) *it is possible to continue with a restore sequence after having to use `WITH CONTINUE_AFTER_ERROR`*

No. If a backup is corrupt such that you must use `WITH CONTINUE_AFTER_ERROR` to restore it, that's restore terminates your restore sequence. If you're restoring a bunch of transaction log backups and one is corrupt, you may want to think carefully on whether you want to force it to restore or not. Forcing a corrupt log backup to restore could mean you've got inconsistent data in the database, or worst case, structural corruption. I'd most likely recommend not restoring it.

24c) *it is possible to restore different parts of a database to different points-in-time*

No. A portion of the database cannot be brought online unless it is at the same point in time as the primary filegroup. The exception, of course, is a read-only filegroup.

24d) *it is possible to restore filegroups from different databases together in a new database*

No. All the files in a database have a GUID in the fileheader page. Unless the GUID matches that of data file ID 1 in the database, it cannot be restored as part of the same database.

24e) *restore removes index fragmentation (or updates statistics, etc.)*

No. What you backup is what you get when you restore. I explain this a bit more in a blog post over on our SQL Server Magazine Q&A blog.

24f) *it is possible to shrink a database during a restore*

No. This is an often-requested feature in SQL Server - be able to restore a very large, but mostly empty, database on a dev or QA server and have it only be the size of the data in the original database. But you can't.

24g) *you can restore a database to any downlevel version of SQL Server*

No. This is one of the most pervasive myths. SQL Server cannot understand databases from more recent versions (e.g. SQL Server 2005 cannot understand a SQL Server 2008 database). I already explained about this a bunch in A DBA myth a day: (13/30) you cannot run DMVs when in the 80 compat mode (T-SQL Tuesday #005).

24h) *you can always restore a database to any edition of SQL Server*

No. In SQL Server 2005, if there's a table/index partitioning in the database, it can only be restored on Enterprise (or Enterprise Eval or Developer) Edition. On SQL Server 2008 the list is partitioning, transparent data encryption, change data capture, and data compression. I blogged about this issue, the new DMV you can use in SQL Server 2008, and an example script in the blog post SQL Server 2008: Does my database contain Enterprise-only features?

24i) *using* `WITH STANDBY` *breaks the restore sequence*

No. The `WITH STANDBY` option allows you to get a read-only transactionally-consistent look at the database in the middle of the restore sequence. As far as the restore sequence is concerned, it's as if you used `WITH NORECOVERY`. You can stop as many times as you like using `WITH STANDBY`. This is what log shipping uses when you ask it to allow access to a log-shipping secondary between log backup restores. Beware though, that using `WITH STANDBY` might cause some seemingly weird behavior - see Why could restoring a log-shipping log backup be slow?

24j) *instant file initialization during a restore doesn't work if the database wasn't backed up on a server with instant file initialization enabled*

No. Whether instant file initialization is used is entirely dependent on whether the SQL Server instance performing the restore has it enabled. There is nothing in the backup itself that controls this. You can read a bunch about instant file initialization starting in the blog post A SQL Server DBA myth a day: (3/30) instant file initialization can be controlled from within SQL Server.

24k) *restore is the best way to recover from corruption*

No, not necessarily. Depending on what backups you have, restore may be the best way to recover with zero or minimal data loss, but it may be waaaay slower than running a repair and accepting some data loss, or pulling damaged/lost data back from a log shipping secondary. The best way to recover from corruption is the one that allows you to best meet your downtime and data-loss service level agreements.

24l) *you can take a tail-of-the-log backup after starting a restore sequence*

No. As soon as you start to restore over a database you lose the ability to back up the tail-of-the-log. The very first thing in a disaster recovery plan should always be to check whether a tail-of-the-log backup needs to be taken, just in case.

24m) *you can always do a point-in-time restore to a time covered by a log backup*

No. If the log backup contains a minimally-logged operation then you cannot stop at a point in time covered by that log backup. You can only restore it in its entirety. This is because a log backup following a minimally-logged operation must include the data extents that were changed by the operation, but there's nothing in the backup that says *when* the extents were changed (that would be the transaction log - that wasn't generated because the operation was minimally logged!). You can figure out how much data will be included in such a log backup using the script in New script: how much data will the next log backup include?

24n) *as long as the backup completes successfully, the restore will work too*

No, no, no, no. A backup file is just like a data file - it sits on an I/O subsystem. And what causes most corruptions? I/O subsystems. You must periodically check that your backups are still valid otherwise you could be in for a nasty surprise when disaster strikes. See Importance of validating backups. The other thing to consider is that an out-of-band full or log backup could have been taken that breaks your restore sequence if it's not available. See `BACKUP WITH COPY_ONLY` - how to avoid breaking the backup chain.

24o) *all SQL Server page types can be single-page restored*

No. Various allocation bitmaps and critical metadata pages cannot be single-page restored (or fixed using automatic page repair with database mirroring in SQL Server 2008). My blog post Search Engine Q&A #22: Can all page types be single-page restored? explains more.

24p) *using* `RESTORE ... WITH VERIFYONLY` *validates the entire backup*

No. Using `VERIFYONLY` only validates the backup header looks like a backup header. It's only when you take the backup using `WITH CHECKSUM` and do `RESTORE ... WITH VERIFYONLY` *and* using `WITH CHECKSUM` that the restore does more extensive checks, including the checksum over the entire backup.

24q) *it is possible to restore a backup of an encrypted database without first having restored the server certificate*

No. That's the whole point of transparent data encryption. Lose the server certificate, lose the database.

24r) *a restore operation performs all REDO and UNDO operations when the restore sequence is completed*

No. The REDO portion of recovery is performed for each restore operation in the restore sequence. The UNDO portion is not done until the restore sequence is completed.

24s) *a compressed backup can only be restored using Enterprise Edition in SQL Server 2008*

No. All editions can restore a compressed backup. New in SQL Server 2008 R2, Standard Edition can create a compressed backup as well as Enterprise Edition.

24t) *the restore of a database from an earlier version of SQL Server can be made to skip the upgrade process*

No. It is not possible to skip any necessary upgrade or recovery during a database restore or attach.

24u) *a backup taken on a 32-bit instance cannot be restored on a 64-bit instance, and vice-versa*

No. There is not difference in the database format on different CPU architectures.

24v) *restoring the database is everything the application needs to continue*

No. Just like with a high-availability failover to a database mirror or log shipping secondary, everything in (what I call) the application ecosystem must be there for the application to work. That may include ancillary databases, logins, jobs, stored procedures etc.

24w) *to restore a damaged file from a multi-file filegroup you must restore the entire filegroup*

No. This used to be the case before SQL Server 2000, but not any more.

24x) *you can restore a backup to any uplevel version of SQL Server*

No. You can only restore a database from two versions back (i.e. you cannot directly restore a SQL Server 7.0 database to SQL Server 2008).

24y) *a restore operation will always take the same time as the backup operation*

No. There are a ton of things that can affect restore time - like whether there's a long-running transaction that need to be rolled back, or whether the database files need to be created and zero-initialized. There's no guarantee.

24z) *you should always drop a database before restoring*

No. If you drop the database then the database files need to be created and zero initialized (or at least the log file does if you have instant file initialization enabled). Also, you should *always* have a copy of the damaged database just in case the restore fails for some reason.

25/30: Fill Factor

Myth #25: *Various myths around fill factor*.

All are FALSE

25a) *fill factor is adhered to at all times*

No. From Books Online:

Important:
The fill-factor setting applies only when the index is created, or rebuilt. The SQL Server Database Engine does not dynamically keep the specified percentage of empty space in the pages. Trying to maintain the extra space on the data pages would defeat the purpose of fill factor because the Database Engine would have to perform page splits to maintain the percentage of free space specified by the fill factor on each page as data is entered.

25b) *fill factor of 0 is different from fill factor of 100*

No: From Books Online:

Note:
Fill-factor values 0 and 100 are the same in all respects.

25c) *fill factor of 0 leaves some space in the upper levels of the index*

No. This one isn't in Books Online and I don't know where this myth came from, but it's completely untrue. You can easily convince yourself of this using a script like the one below:

```
CREATE DATABASE foo;
GO
USE foo;
GO
CREATE TABLE t1 (c1 INT IDENTITY, c2 CHAR (1000) DEFAULT 'a');
CREATE CLUSTERED INDEX t1c1 ON t1 (c1);
GO
SET NOCOUNT ON;
GO
INSERT INTO t1 DEFAULT VALUES;
GO 10000
```

Now check the fill factor is 0 and perform an index rebuild.

```
SELECT [fill_factor] FROM sys.indexes
WHERE NAME = 't1c1' AND [object_id] = OBJECT_ID ('t1');
GO
ALTER INDEX t1c1 ON t1 REBUILD WITH (FILLFACTOR = 100);
GO
```

Then figure out the index pages above the leaf level and look at the `m_freeCnt` value in the page header, the amount of free space on the page:

```
EXEC sp_allocationMetadata 't1';
GO
DBCC TRACEON (3604);
DBCC PAGE (foo, 1, 164, 3);     -- the root page, from the SP output
GO
DBCC PAGE (foo, 1, 162, 1);     -- the page ID in the DBCC PAGE output above
GO
```

I see a value of 10 bytes - clearly no space was left on the page. It's a myth. Btw - you can get the sp_allocationMetadata script from this blog post.

26/30: Nested Transactions Are Real

Nested transactions are an evil invention designed to allow developers to make DBAs' lives miserable. In SQL Server, they are even more evil...

Myth #26: *Nested transactions are real in SQL Server.*

FALSE!!!

Nested transactions do not actually behave the way the syntax would have you believe. I have no idea why they were coded this way in SQL Server - all I can think of is someone from the dim and distant past is continually thumbing their nose at the SQL Server community and going "ha - fooled you!!".

Let me explain. SQL Server allows you to start transactions inside other transactions - called nested transactions. It allows you to commit them and to roll them back.

The commit of a nested transaction has absolutely no effect - as the only transaction that really exists as far as SQL Server is concerned is the outer one. Can you say 'uncontrolled transaction log growth'? Nested transactions are a common cause of transaction log growth problems because the developer thinks that all the work is being done in the inner transactions so there's no problem.

The rollback of a nested transaction rolls back the entire set of transactions - as there is no such thing as a nested transaction.

Your developers should not use nested transactions. ***They are evil***.

If you don't believe me, here's some code to show you what I mean. First off - create a database with a table that each insert will cause 8k in the log.

```
CREATE DATABASE NestedXactsAreNotReal;
GO
USE NestedXactsAreNotReal;
GO
ALTER DATABASE NestedXactsAreNotReal SET RECOVERY SIMPLE;
GO
CREATE TABLE t1 (c1 INT IDENTITY, c2 CHAR (8000) DEFAULT 'a');
CREATE CLUSTERED INDEX t1c1 ON t1 (c1);
GO
SET NOCOUNT ON;
GO
```

Test #1: Does rolling back a nested transaction only roll back that nested transaction?

```
BEGIN TRAN OuterTran;
GO

INSERT INTO t1 DEFAULT Values;
GO 1000

BEGIN TRAN InnerTran;
GO

INSERT INTO t1 DEFAULT Values;
GO 1000

SELECT @@TRANCOUNT, COUNT (*) FROM t1;
GO
```

I get back the results 2 and 2000. Now I'll roll back the nested transaction and it should only roll back the 1000 rows inserted by the inner transaction...

```
ROLLBACK TRAN InnerTran;
GO

Msg 6401, Level 16, State 1, Line 1
Cannot roll back InnerTran. No transaction or savepoint of that name was
found.
```

Hmm... from Books Online, I can only use the name of the outer transaction or no name. I'll try no name:

```
ROLLBACK TRAN;
GO

SELECT @@TRANCOUNT, COUNT (*) FROM t1;
GO
```

And I get the results 0 and 0. As Books Online explains, `ROLLBACK TRAN` rolls back to the start of the outer transaction and sets `@@TRANCOUNT` to 0. All changes are rolled back. The only way to do what I want is to use `SAVE TRAN` and `ROLLBACK TRAN` to the savepoint name.

Test #2: Does committing a nested transaction really commit the changes made?

```
BEGIN TRAN OuterTran;
GO

BEGIN TRAN InnerTran;
GO

INSERT INTO t1 DEFAULT Values;
GO 1000

COMMIT TRAN InnerTran;
GO

SELECT COUNT (*) FROM t1;
GO
```

I get the result 1000, as expected. Now I'll roll back the outer transaction and all the work done by the inner transaction should be preserved...

```
ROLLBACK TRAN OuterTran;
GO

SELECT COUNT (*) FROM t1;
GO
```

And I get back the result 0. Oops - committing the nested transaction did not make its changes durable.

Test #3: Does committing a nested transaction at least let me clear the log?

I recreated the database again before running this so the log was minimally sized to begin with, and the output from `DBCC SQLPERF` below has been edited to only include the `NestedXactsAreNotReal` database.

```
BEGIN TRAN OuterTran;
GO

BEGIN TRAN InnerTran;
GO

INSERT INTO t1 DEFAULT Values;
GO 1000

DBCC SQLPERF ('LOGSPACE');
GO

Database Name         Log Size (MB) Log Space Used (%) Status
--------------------- ------------- ------------------ -----------
NestedXactsAreNotReal 12.05469      95.81983           0
```

Now I'll commit the nested transaction, run a checkpoint (which will clear all possible transaction log in the SIMPLE recovery model), and check the log space again:

```
COMMIT TRAN InnerTran;
GO

CHECKPOINT;
GO

DBCC SQLPERF ('LOGSPACE');
GO
```

```
Database Name         Log Size (MB) Log Space Used (%) Status
--------------------- ------------- ------------------ -----------
NestedXactsAreNotReal 12.05469      96.25324           0
```

Hmm - no change - in fact the Log Space Used (%) has increased slightly from writing out the checkpoint log records (see How do checkpoints work and what gets logged). Committing the nested transaction did not allow the log to clear. And of course not, because a rollback can be issued at any time which will roll back all the way to the start of the outer transaction - so all log records are required until the outer transaction commits or rolls back.

And to prove it, I'll commit the outer transaction and run a checkpoint:

```
COMMIT TRAN OuterTran;
GO

CHECKPOINT;
GO

DBCC SQLPERF ('LOGSPACE');
GO
```

```
Database Name         Log Size (MB) Log Space Used (%) Status
--------------------- ------------- ------------------ -----------
NestedXactsAreNotReal 12.05469      26.4339            0
```

And it drops right down.

Nested transactions - just say no! (a public service announcement from the nice folks at SQLskills.com :-)

27/30: Use BACKUP WITH CHECKSUM to Replace DBCC CHECKDB

Myth #27: *you can avoid having to run `DBCC CHECKDB` (or equivalent) by using `BACKUP ... WITH CHECKSUM`*

FALSE

On first glance, this seems like it should be true as the `WITH CHECKSUM` option will cause all existing page checksums on allocated pages to be checked. But it's not and here's why:

Firstly, on a database that's been upgraded from SQL Server 2000 or earlier, page checksums must be enabled. After they're enabled, not all pages in the database will have page checksums on them - and an I/O subsystem does not distinguish between pages with and without page checksums when causing corruption. So if all you do is use `BACKUP ... WITH CHECKSUM`, there may be corruption that you won't find until it's too late... (I'll just leave the 'until it's too late' hanging there... you can imagine scary consequences for yourselves :-)

Secondly, you might only take a full database backup once per month, which isn't a frequent enough consistency check for my liking (I recommend at least once per week). Even with a weekly differential backup, that's not going to check all allocated pages in the database - only those in extents that it backs up (those extents that changed since the last full backup).

Lastly, and most insidiously, relying solely on `BACKUP ... WITH CHECKSUM` leaves you susceptible to in-memory corruptions. If a bad memory chip, badly-written XP, or other rogue Windows process corrupts a SQL Server data file page in memory, and then it gets written to disk, you've got a corrupt page with a valid checksum - and nothing will catch that except `DBCC CHECKDB`.

Bottom line - you can't avoid running consistency checks. If you're having trouble, take a look at my old blog post CHECKDB From Every Angle: Consistency Checking Options for a VLDB.

And while we're on the subject, check out this post: Search Engine Q&A #26: Myths around causing corruption.

28/30: BULK_LOGGED Recovery Model

The BULK_LOGGED recovery model continues to confuse people...

Myth #28: *various myths around the BULK_LOGGED recovery model*.

28a) *regular DML operations can be minimally-logged*

No.

Only a small set of bulk operations can be minimally-logged when in the BULK_LOGGED (or SIMPLE) recovery model. The list is in the Books Online topic Operations That Can Be Minimally Logged has the list. This is the 2008 link - make sure to check the link for the version you're running on.

28b) *using the BULK_LOGGED recovery model does not affect disaster recovery*

No.

Firstly, if a minimally-logged operation has been performed since the last log backup, and one or more data files were damaged and offline because of the disaster, a tail-of-the-log backup cannot be performed and so all user transactions performed since the last log backup will be lost.

Secondly, if a log backup contains a minimally-logged operation, a point-in-time restore cannot be performed to any time covered by the log backup. The log backup can either not be restored, or be restored in its entirety (plus additional log backups if required) - i.e. you can restore to a point:

- Before the beginning of that log backup
- At the end of that log backup
- After the end of that log backup

But you can't restore to a point during that log backup.

28c) *using the BULK_LOGGED recovery model also reduces the size of log backups*

No.

A log backup that includes a minimally-logged operation must backup the minimal amount of transaction log *and* all the data file extents changed by that operation - otherwise the restore of the log backup would not fully reconstitute the minimally-logged operation. This means that log backups are roughly the same size whether in the FULL or BULK_LOGGED recovery model.

29/30: Fixing Heap Fragmentation

Myth #29: *Fix heap fragmentation by creating and dropping a clustered index.*

Noooooooooooo!!!

This is just about one of the worst things you could do outside of shrinking a database.

If you run sys.dm_db_index_physical_stats (or my old DBCC SHOWCONTIG) on a heap (a table without a clustered index) and it shows some fragmentation, don't *EVER* create and drop a clustered index to build a nice, contiguous heap. Do yourself a favor and just create the well-chosen clustered index and leave it there - there's a ton of info out there on choosing a good clustering key - narrow+static+unique+ever-increasing is what you need. Kimberly has a blog post from 2005(!) that sums things up: Ever-increasing clustering key - the Clustered Index Debate.....again! and I've got An example of a nasty cluster key.

Yes, you can use ALTER TABLE ... REBUILD in SQL Server 2008 to remove heap fragmentation, but that is almost as bad as creating and dropping a clustered index!

Why am I having a conniption fit about this? Well, every record in a nonclustered index has to link back to the matching row in the table (either a heap or clustered index - you can't have both - see Kimberly's recent SQL Server Magazine blog post What Happens if I Drop a Clustered Index? for an explanation). The link takes the form of:

- if the table is a heap, the actual physical location of the table record (data file:page number:record number)
- if the table has a clustered index, the clustering key(s)

The blog post link at the bottom of this post explains in a lot more detail.

If you create a clustered index, all the linkages to the heap records are no longer valid and so all the nonclustered indexes must be rebuilt automatically to pick up the new clustering key links. If you drop the clustered index again, all the clustering key links are now invalid so all the nonclustered indexes must be rebuilt automatically to pick up the new heap physical location links.

In other words, if you create and then drop a clustered index, all the nonclustered indexes are rebuilt twice. Nasty.

If you think you can use ALTER TABLE ... REBUILD in SQL Server 2008 to fix heap fragmentation, you can, but it causes all the nonclustered indexes to be rebuilt as the heap record locations obviously change.

Now, what about if you *rebuild* a clustered index? Well, that depends on what version you're on and whether you're doing a simple rebuild or changing the definition of the index. One major point of misconception is that moving a clustered index or partitioning it changes the cluster keys - it doesn't. For a full list of when the nonclustered indexes need to be rebuilt, see Indexes From Every Angle: What happens to non-clustered indexes when the table structure is changed?

30/30: Backup Myths

The month is finally over so time for the grand finale!

Although it's been fun debunking all these myths, it's been a tad stressful making sure I come up with an interesting and useful myth to debunk every day. I'd like to give kudos to fellow-MVP Glenn Berry (blog|twitter) who's been running an excellent DMV-a-Day series through April too!

To round out the month, I present to you 30 myths around backups - one for each day of the month of April. Last night I sat down to write this post and was a few myths short so reached out to the fabulous SQL community on Twitter (follow me!) for help - too many people to list (you know who you are) - I thank you!

A few folks have asked if I'll pull the month's posts into a PDF e-book - let me know if you'd like that.

I *really* hope you've enjoyed the series over the last month and have had a bunch of myths and misconceptions debunked once and for all - I know quite a few of you are going to use these explanations as ammunition against 3rd-party vendors, developers, and other DBAs who insist on incorrect practices.

Ok - here we go with the last one...

Myth #30: *Various myths around backups...*

All are FALSE!!

For a good primer on understanding backups and how they work see my TechNet Magazine article Understanding SQL Server Backups.

30-01) *backup operations cause blocking*

No. Backup operations do not take locks on user objects. Backups do cause a really heavy read load on the I/O subsystem so it might *look* like the workload is being blocked, but it isn't really. It's just being slowed down. There's a special case where a backup that has to pick up bulk-logged extents will take a file lock which could block a checkpoint operation - but DML is never blocked.

30-02) *switching from the* `FULL` *recovery model to the* `BULK_LOGGED` *recovery model and back again breaks the log backup chain*

No. It just doesn't. Switching from either `FULL` or `BULK_LOGGED` to `SIMPLE` *does* break the log backup chain however.

30-03) *breaking the log backup chain requires a full backup to restart it*

No. You can restart the log backup chain with either a full or differential backup - anything that bridges the LSN gap from the point at which the log backup chain was broken. See my

blog post A SQL Server DBA myth a day: (20/30) restarting a log backup chain requires a full database backup for more details.

30-04) *concurrent log backups are not possible while a full or differential backup is in progress*

No, this changed in SQL Server 2005. See my blog post Search Engine Q&A #16: Concurrent log and full backups.

30-05) *a full or differential backup clears the log*

No. A log backup includes all the log since the last log backup - nothing can change that - no matter whether that log was also backed up by a full or differential backup. I had a famous argument on Twitter last year and wrote this blog post as proof: Misconceptions around the log and log backups: how to convince yourself. In the `FULL` or `BULK_LOGGED` recovery models, the *only* thing that clears the log is a log backup.

30-06) *using the `BULK_LOGGED` recovery model for minimally-logged operations reduces the size of the next transaction log backup*

No. A minimally-logged operation is so-named because only the page allocations are logged. A log backup needs all the information necessary to resconstitute the transaction, so a log backup following a minimally-logged operation must backup the log plus all extents changed by the minimally-logged operation. This will result in the log backup being roughly the same size as if the operation was fully logged.

30-07) *full and differential backups only contain the log generated while the backup* was running

No. A full or differential backup contains enough log to be able to recover the database to a transactionally-consistent view of the database at the time the data-reading portion of the backup finished (or as far back as the oldest log record that transactional replication has not yet processed - to ensure that replication works properly after a restore). Check out these two blog posts for details:

- Debunking a couple of myths around full database backups
- More on how much transaction log a full backup includes

30-08) *backups always test existing page checksums*

No. It only does it when you use the `WITH CHECKSUM` option - which you should.

30-09) *backups read data through the buffer pool*

No. The backup subsystem opens its own channels to the database files to avoid the performance hit of having to read everything into SQL Server's memory and back out to the backup device (and also effectively flushing the buffer pool in the process). If you ask the for page-checksum checking, it uses its own small portion of memory.

30-10) *backups perform consistency checks (a la DBCC CHECKDB)*

No. Nothing else to say.

30-11) *if the backup works, the restore will too*

No. Please don't fall into this trap. You must regularly validate your backups to give yourself a high level of confidence that they will work if a disaster occurs. See Importance of validating backups for more details.

30-12) *a mirrored backup will succeed if the mirror location becomes unavailable*

No. If any one of the paths to a mirrored backup fails, the entire mirrored backup operation fails. I'd really like it to work the other way around - where the local backup succeeds and the remote backups fail, but it doesn't unfortunately.

30-13) *a tail-of-the-log backup is always possible*

No. A tail-of-the-log backup is one that backs up all the log generated since the last log backup, in an exceptional situation. If the data files are damaged, you can still do a tail-of-the-log backup EXCEPT if the un-backed-up log contains a minimally-logged operation. That would require reading data extents - which cannot be done if the data files are damaged. For this reason, the `BULK_LOGGED` recovery model should not be used on databases that have 24x7 user transactions.

30-14) *you can use backups instead of* `DBCC CHECKDB`

No. See A SQL Server DBA myth a day: (27/30) use `BACKUP WITH CHECKSUM` to replace `DBCC CHECKDB`.

30-15) *you can back up a database snapshot*

No. It's not implemented, but would be great if you could.

30-16) *you can use database snapshots instead of log backups*

No. A database snapshot is only usable while the database on which it is based is usable and online. If the source database is corrupted, the database snapshot most likely is too. If the source database goes suspect, so does the database snapshot.

Also, having multiple database snapshots (equating to multiple log backups) incurs an increasing performance drain - as every page that changes in the source database may need to be synchronously written to all existing snapshots before it can be written to the source database data files, and all existing database snapshots will grow as more pages are pushed into them.

30-17) *log backups will be the size of the log*

No. The log has to accommodate the space necessary to roll back active transactions, the amount of space returned by `DBCC SQLPERF (LOGSPACE)` on a busy system doesn't

accurately reflect the amount of log records in the log. This blog spot explains: Search Engine Q&A #25: Why isn't my log backup the same size as my log? And apart from that, a log backup is just all the log generated since the last log backup - not the whole log file usually - and if it happens to be, the first part of the explanation comes into play.

30-18) *you cannot backup a corrupt database*

No. In most cases you can use the `WITH CONTINUE_AFTER_ERROR` option to back up the corrupt database. If that fails (maybe because of a damaged boot page or file-header page), there are no other options apart from OS-level file backups.

30-19) *you cannot stop someone doing a `BACKUP LOG .. WITH NO_LOG or TRUNCATE_ONLY` operation*

No. In SQL Server 2008 it's not possible any more (yay!) and in 2005 and before, use trace flag 3231 which turns the operation into a no-op.

30-20) *log backups always clear the log*

No.

If there's no concurrent data backup running, a log backup will always *try* to clear the log, and only succeed in clearing the inactive portion of the log - the log that's only considered 'required' by SQL Server because it hasn't yet been backed up. If anything else is holding the log 'required', it cannot be cleared, even though it has been backed up. Subsequent log backups will check again and again until the time comes when that portion of the log can be cleared. The TechNet Magazine article Understanding Logging and Recovery in SQL Server I wrote last year explains a lot more about how the log works.

Also, if there is a concurrent data backup running, the log clearing will be delayed until the data backup finishes. See the blog post in myth 30-04 for more details.

30-21) *differential backups are incremental*

No. Differential backups are all the data extents that have changed since the last full backup - so they are cumulative. Log backups are incremental - all log generated since the last log backup. Many people call differential backups 'incrementals', when they're not really.

30-22) *once a backup completes, you can safely delete the previous one*

No. No. No.

If you go to restore, and you find your full backup is corrupt, what do you do? Well, if you don't have an older full backup, you most likely start updating your resume. You need to keep a rolling-window of backups around in case a disaster occurs and you need to restore from an older set of backups.

30-23) *you can back up a mirror database*

No. A mirror database is not accessible except through a database snapshot. And you can't back up that either.

30-24) *you can back up a single table*

No. You can effectively back up single table if it happens to be wholely contained on a single filegroup, but there's no way to say `BACKUP TABLE`.

30-25) *SQL Server has to be shut down to take a backup*

No. No idea how this myth started... [Edit: apparently this myth started with Oracle - and we all know how good Oracle is compared to SQL Server... :-)]

30-26) *my transaction is guaranteed to be contained in the backup if it committed before the backup operation completed*

No. The commit log record for the transaction has to have been written out before the data-reading portion of the backup finished. See my blog post Search Engine Q&A #6: Using fn_dblog to tell if a transaction is contained in a backup for more details.

30-27) *you should shrink the database before a backup to reduce the backup size*

No. Shrink just moves pages around so won't make any difference. See my old blog post Conference Questions Pot-Pourri #10: Shrinking the database before taking a backup. And of course, shrink is evil. See A SQL Server DBA myth a day: (9/30) data file shrink does not affect performance. And what's even worse as someone reminded me, is if you do the shrink *after* the full backup, the next differential backup may be huge, for no actual data changes!

30-28) *backups are always the best way to recover from a disaster*

No. Backups are usually the best way to recover with zero data-loss (as long as you have log backups up to the point of the disaster), but not necessarily the best way to recover with minimal downtime. It may be way faster to fail over, or to run repair and accept some data loss if the business requirements allow it.

30-29) *you don't need to back up master, msdb, model...*

No. You should always back up the system databases. Master contains all the security info, what databases exist - msdb contains all the SSIS packages, Agent jobs, backup history - model contains the configuration for new databases. Don't fall into the trap of only backing up user databases otherwise you'll be in a world of hurt if you have to do a bare-metal install.

30-30) *you should always plan a good backup strategy*

No. Now you're thinking 'Huh?'...

You should plan a restore strategy. Use the business requirements and technical limitations to figure out what you need to be able to restore in what time, and then use that to figure out what backups you need to take to allow those restores to happen. See the blog posts:

- Importance of having the right backups
- Planning a backup strategy - where to start?

The vast majority of the time people plan a backup strategy without testing or thinking about restores - and come a disaster, they can't restore within their SLAs. Don't let that be you.