PRACTICAL SQL SERVER

SQL Server Database Corruption, Part II: Simulating Corruption

overachiever | Jan 30, 2012

In my last post I provided an overview of what SQL Server database corruption is—and how it's almost always caused by problems at the IO subsystem (or disk) level. However, while it's all fine and well to talk about things in such a theoretical sense, in that post I also mentioned that a great way to get a 'feel' for how corruption works is to simulate it a bit on your own. Accordingly, in this post I'll provide a step-by-step walkthrough of what that looks like by simulating some corruption.

Part I: What Is Corruption?

Setting the Stage

Obviously, when it comes to actively trying to 'corrupt' a database there are a couple of caveats that need to be addressed—above and beyond the OBVIOUS caveat that this is something you'd never want to do with production database. (It IS a great experiment to test against COPIES of your production databases though).

Otherwise, the caveats to simulating corruption are that you'll want to DELETE any potential existing data from msdb's suspect_pages table (which we'll talk about in a future post), and that

you'll obviously want to make sure that you've got a viable backup (even if it's a simple copy/paste of an existing .BAK or .mdf/.ldf files) of whatever database you'll be corrupting.

In my case, I'm corrupting a copy of the AdventureWorks database that I have running in my environmentwhen it comes to actively trying to 'corrupt' a database there are a couple of caveats that need to be addressed—above and beyond the OBVIOUS caveat that this is something you'd never want to do with production database—mostly because I just hate AdventureWorks so much. So, in my case I'm backing it up like so:

Figure1

Simulating Corruption

Then, when it comes to actually simulating corruption, that ends up being a bit hard to do when SQL Server has its 'hooks' into the database in question – so I'll just Detach it using the SSMS GUI, as follows:

Figure2

Once detached, the database is just a collection of 'zeroes and ones' that I can then open up and 'mangle' as needed. At this point I then just need to find the actual .mdf file for this database (which I happen to know is in my D:\SQLData\ drive on my test server), and then I can open it up with an application other than SQL Server.

And, in this case, what I actually want to do is open up the AdventureWorks.mdf in a Hex Editor – something that will painlessly let me look at all of the 'zeroes and ones' that make up the actual data in my file.

For this test I'm using HxD – a great freeware Hex Editor that I like a lot.

Then, once I open up the AdventureWorks.mdf in HxD, I'm greeted by something similar to the following:

Figure3

And, if I scroll down a ways, say to offset 0218D2A0 and friends (or roughly 1/4th of the way into the data file), I see what clearly looks to be a number of sequential IDs – as highlighted below:

Figure4

And, it's at this point that I can 'simulate' some corruption—by simply overwriting those existing values with a bunch of zeroes (or ones, or any other value)—as follows:

Figure5

Granted, this simulation is going to be a bit different than what you might expect with some sort of 'minor disk malfunction' that could result in corruption – but the point of what's going on here should be obvious. I've just arbitrarily chosen a bit of data somewhere in the middle of my file – and 'corrupted' it by clearly setting it to values that SQL Server did not specify.

Corruption Isn't Always Obvious

Once my sample database has been sufficiently corrupted, it's time to bring it back online. Though, just remember that I had to detach it in order to 'hack it' with some corruption. In most 'real-life' corruption scenarios, SQL Server would have asked the OS to write data to disk, and the OS (or the IO subsystem) would then have 'mangled' the data being written – to the point where corruption would have occurred.

Reaching **5.4 million**
IT Professionals

Figure6

But, even in this case, note that if you follow along with the example provided, the AdventureWorks database comes back online without a problem. In fact, you can even query meta-data without a hitch (because meta-data pages weren't corrupted in this example or experiment).

Figure7

So, at this point we've been able to 'inject' some corruption into an existing database – and hopefully this gives you a sense of just how disastrous corruption can be – because there's no indication whatsoever that some of our data has been completely mangled.

And, in this case, we've just 'hacked' some of the data in a non-clustered (but UNIQUE) index for the Sales.Customers table – so this simulation really provides an example of some 'tame' corruption in terms of how much pain has been potentially caused. But, the corruption COULD have been to a larger block of data or to a much more important location – to the point where (instead of losing 'duplicated data' in an index) we could have actually lost actual data in a clustered index and so on.

Or, even worse, if instead of having this corruption 'land' on some of our user data, it could have 'landed' at an inopportune location within the .mdf that that's used by SQL Server to keep track of allocation information – which, in turn, is what SQL Server uses to keep track of WHERE it's storing user data. And, in cases where this kind of data gets destroyed (which is typically fairly rare – yet it DOES happen), then instead of losing your own data, you lose SQL Server's 'references' to where your data is stored – and potentially risk losing much more than just a few snippets of even mission-critical data here and there. In fact, if corruption happens to occur on page 9 – the 'boot page', for example, then you're dead in the water and SQL Server can't actually repair the database in question – at all.

ENGAGE

But, the key thing to note is that, in this example, we've interjected some corruption and there's really no indication at all. In fact, it's not until a query is run against the corrupted index itself that we get any sense that there's a problem. And, once we DO get a sense that there's a problem, it certainly LOOKS ugly based upon the error that SQL Server spits out:

Figure8

Up Next

In following posts we'll take a look at whether or not corruption can be prevented (spoiler alert: it really can't—in most cases) and what that means from a disaster recovery perspective—if, for example, you happen to care about your data and don't want to risk what would happen if corruption were to rear its ugly head.

Part III: Preventing Corruption

TAGS: MICROSOFT SQL SERVER

0 COMMENTS

SECURITY > ENDPOINT SECURITY

Apple Is Ready for the Tech Backlash

The only real guarantee that companies won't go overboard in invading customers' privacy is that they have large revenue streams which make it an unnecessary risk.

Bloomberg | Mar 30, 2018

(Bloomberg View) --It's easy to understand Apple CEO Tim Cook's somewhat self-righteous comments on the Facebook-Cambridge Analytica scandal: His company has never emphasized customer data monetization, and privacy has been one of its persistent selling points. Apple still collects our data — and reserves the right to share it — but the company has a much better track record than newer tech titans.