source: http://www.MSSQLTips.com/tip.asp?id=2871 -- printed: 2/14/2013 8:01:55 AM

Troubleshooting and Fixing SQL Server Page Level Corruption

Written By: Derek Colley -- 2/7/2013

Problem

Corrupt SQL Server databases are the worst nightmare of any SQL Server professional. In any environment, from small business to enterprise, the compromise of integrity and availability of the data can constitute a business emergency. This is especially the case in those organizations reliant on an OLTP data model, for a high-volume website. SQL Server database corruption and disruption of the transaction processing system can cause business repercussions such as large financial losses, a drop in reputation or customer retention, or contractual SLA problems with the service provider, if not managed in-house. This tip will demonstrate the diagnosis process, discuss one method of correcting page-level corruption without using the REPAIR options with the DBCC CHECKDB command and outline how to get the SQL Server back online.

Solution

Diagnosis - SQL Server Corruption

Normally any business will have contingency plans to deal with SQL Server database corruption scenarios, and any good database professional will have immediately thought of a number of strategies to deal with these. Items such as disaster recovery plans, replication, Database Mirroring\AlwaysOn, Clustering, etc. However, sometimes these options are not appropriate. Imagine you have a 20GB database populated with records of your customers and records of each transaction linked back to your customers. You have two .mdf data files. You are using the full recovery model, full backups are taken daily at 21:00 with transaction log backups taken every 15 minutes. You don't use mirroring, replication or clustering, instead of relying on a robust backup model to protect your data. At 20:00, a message like this occurs:

```
Msg 824, Level 24, State 2, Line 1
SQL Server detected a logical consistency-based I/O error: unable to decrypt page
 due to missing DEK. It occurred during a read of page (3:0) in database ID 10 at
 offset 0000000000000000 in file 'c:\del\corruption_secondary.mdf'.  Additional
 messages in the SQL Server error log or system event log may provide more detail.
 This is a severe error condition that threatens database integrity and must be
 corrected immediately. Complete a full database consistency check (DBCC CHECKDB).
 This error can be caused by many factors; for more information, see SQL Server
 Books Online.
```

It is clear that the SQL Server database integrity has been compromised. As this sample error reports, there's a problem in the 'corruption_secondary.mdf' file (my second data file in this test database) at page 3:0, offset 000..000 (right at the beginning). So what do you do? The first instinct is to think, 'Restore! Restore! Restore!' But in this scenario, this will involve the following steps as a minimum:

- Set database in single-user mode
- Tail log backup of all transactions since last transaction log backup
- Full restore of backup from today - 1
- Restore of 4 transaction logs per hour multiplied by 23 hours = 92 individual logs
- Restore of tail log up to the point of corruption (point in time recovery)
- DBCC check of the database
- Set database online

But wait - there's two things to consider. One, what's your recovery time objective (RTO)? Is it, let's say, four hours? Are you sure you can perform 93 individual backups in four hours, over 20GB of data? What if you're using SATA drives - IOPS constraints will make this a close-run race. What about configuration of the script to do this? What happens if one of those logs is also corrupt? And what happens if the corruption pre-dates the error message (i.e. the error message wasn't a direct response to a very recent event, like disk failure) but instead has festered inside your database? What if (horror of horrors)... the full backups include this corrupted data?

DBCC CHECKDB

Fortunately, there's a few ways to identify corruption in the database. The first, and the most well-known, is DBCC CHECKDB. This utility will perform systematic data integrity checks throughout the datafiles and identify areas of concern. As documented in other excellent articles, this comes with three main options for recovery: REPAIR_ALLOW_DATA_LOSS, REPAIR_FAST and REPAIR_REBUILD. These options are not ideal, especially REPAIR_ALLOW_DATA_LOSS, although it is perfectly possible to restore without data loss using these tools. This article is not going to demonstrate the use of CHECKDB, rather it will demonstrate a different method. Here's the warning about CHECKDB repairs from Books Online:

"Use the REPAIR options only as a last resort. To repair errors, we recommend restoring from a backup. Repair operations do not consider any of the constraints that may exist on or between tables. If the specified table is involved in one or more constraints, we recommend running DBCC CHECKCONSTRAINTS after a repair operation. If you must use REPAIR, run DBCC CHECKDB without a repair option to find the repair level to use. If you use the REPAIR_ALLOW_DATA_LOSS level, we recommend that you back up the database before you run DBCC CHECKDB with this option." -- http://msdn.microsoft.com/en-gb/library/ms176064.aspx

There are other ways of dealing with corruption, including the direct edit of database files in a suitable hex or text editor, and this is the method I'm going to use to demonstrate that it's possible to repair a corrupt data file with direct 'open-heart' surgery. However, you will need access to one copy of the database, whether that's from your QA/Test/dev stack or a restored copy of an old, clean backup. This will be the source of your clean data. You will not need to restore transaction logs or take any other action, providing you're reasonably sure that the corrupted data hasn't been updated since the date of the clean backup.

Preparation

Firstly, you'll need some tools. Here are my favorites:

- Notepad++ (open-source from http://notepad-plus-plus.org/)
- Hex Editor Neo (freeware, from http://www.hhdsoftware.com/free-hex-editor)
- SQL Server Management Studio
- Text comparison tool (e.g. online: http://www.textdiff.com is okay)

And before I get started, here's some assumptions and notes:

- I am using SQL Server 2008 Standard Edition.
- I have full access to the SQL Server data files, which are (in this demonstration) located in c:\del on my file system.
- I have sysadmin access to the SQL Server instance.
- I have access to a full backup which is known to be free of corruption.
- I am using, for this article, a test database populated with two tables, approximately 1.1m data rows, named 'CORRUPTION'. One table is called dbo.Customers and the other is dbo.Transactions. There is an FK relationship between them. The build script to follow along with this tip is available below:

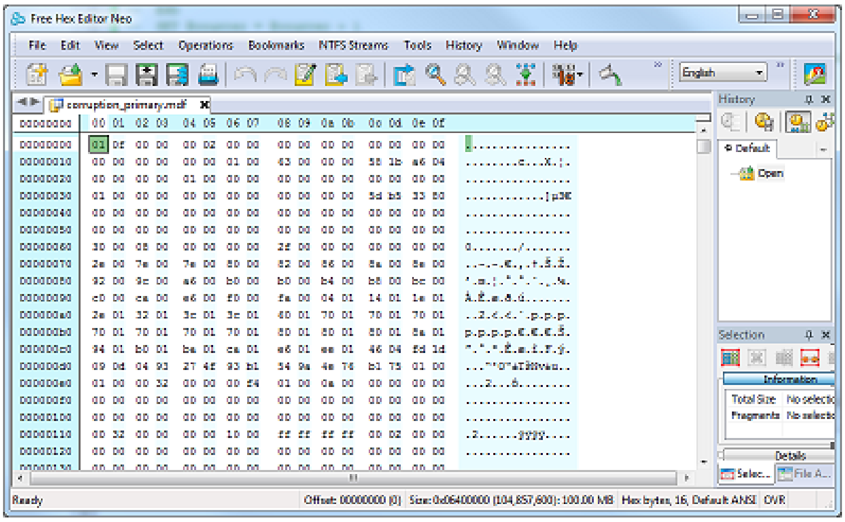
```
 -- PART ONE
-- first, create the test database
SET NOCOUNT ON

CREATE DATABASE CORRUPTION
ON PRIMARY
( NAME = 'corruption_primary', FILENAME = 'c:\del\corruption_primary.mdf', FILEGROWTH=10%, SIZE = 100MB,
( NAME = 'corruption_secondary', FILENAME = 'c:\del\corruption_secondary.mdf', FILEGROWTH=10%, SIZE = 10
LOG ON
( NAME = 'corruption_log', FILENAME = 'c:\del\corruption_log.ldf', SIZE = 50MB )
GO

-- PART TWO
-- create some sample data
USE CORRUPTION
CREATE TABLE dbo.customers
  ( custid INT IDENTITY(1,1) PRIMARY KEY NOT NULL,
   fname VARCHAR(100) NOT NULL,
```

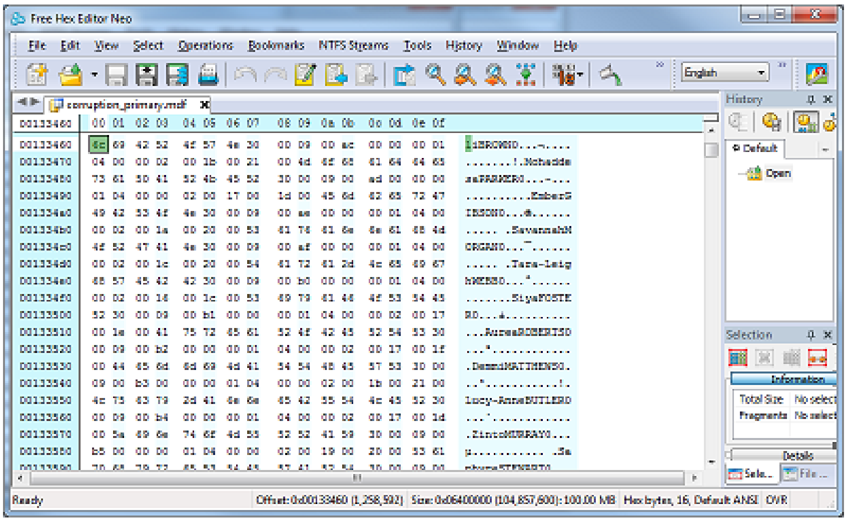
Examining a Corrupt SQL Server Data File

First, let's have a look at a healthy data file in the hex editor. If you're following along, fire up Hex Editor Neo and load one of the datafiles created by my build script (see above). Note you will need to set this database OFFLINE to view the file as SQL Server will maintain a handle on it otherwise. You'll see something like this:



```
00000000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00000000  01 0f 00 00  00 02 00 00  00 00 00 00  00 00 00 00  ................
00000010  00 00 00 00  00 00 01 00  63 00 00 00  58 1b a6 04  ........c...X.¦.
00000020  00 00 00 00  01 00 00 00  00 00 00 00  00 00 00 00  ................
00000030  01 00 00 00  00 00 00 00  00 00 00 00  5d b5 33 80  ............]µ3€
00000040  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000050  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000060  30 00 08 00  00 00 00 00  2f 00 00 00  00 00 00 00  0......./.......
00000070  2e 00 7e 00  7e 00 80 00  82 00 86 00  8a 00 8e 00  ..~.~.€.‚.†.Š.Ž.
00000080  92 00 9e 00  a6 00 b0 00  b0 00 b4 00  b8 00 bc 00  ’.ž.¦.°.°.´.¸.¼.
00000090  c0 00 ca 00  e6 00 f0 00  fa 00 04 01  14 01 1e 01  À.Ê.æ.ð.ú.......
000000a0  2e 01 32 01  3c 01 3c 01  60 01 70 01  70 01 70 01  ..2.<.<.`.p.p.p.
000000b0  70 01 70 01  70 01 70 01  80 01 80 01  80 01 8a 01  p.p.p.p.€.€.€.Š.
000000c0  94 01 b0 01  ba 01 ca 01  e6 01 ee 01  46 04 fd 1d  ”.°.º.Ê.æ.î.F.ý.
000000d0  09 0d 04 93  27 4f 93 b1  54 9a 4e 76  b1 75 01 00  ...“'O“±TšNv±u..
000000e0  01 00 00 32  00 00 00 f4  01 00 0a 00  00 00 00 00  ...2...ô........
000000f0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000100  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000110  00 32 00 00  00 00 10 00  ff ff ff ff  00 02 00 00  .2......ÿÿÿÿ....
00000120  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000130  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
Ready   Offset: 00000000 (0)   Size: 0x06400000 (104,857,600): 100.00 MB   Hex bytes, 16, Default ANSI   OVR
```

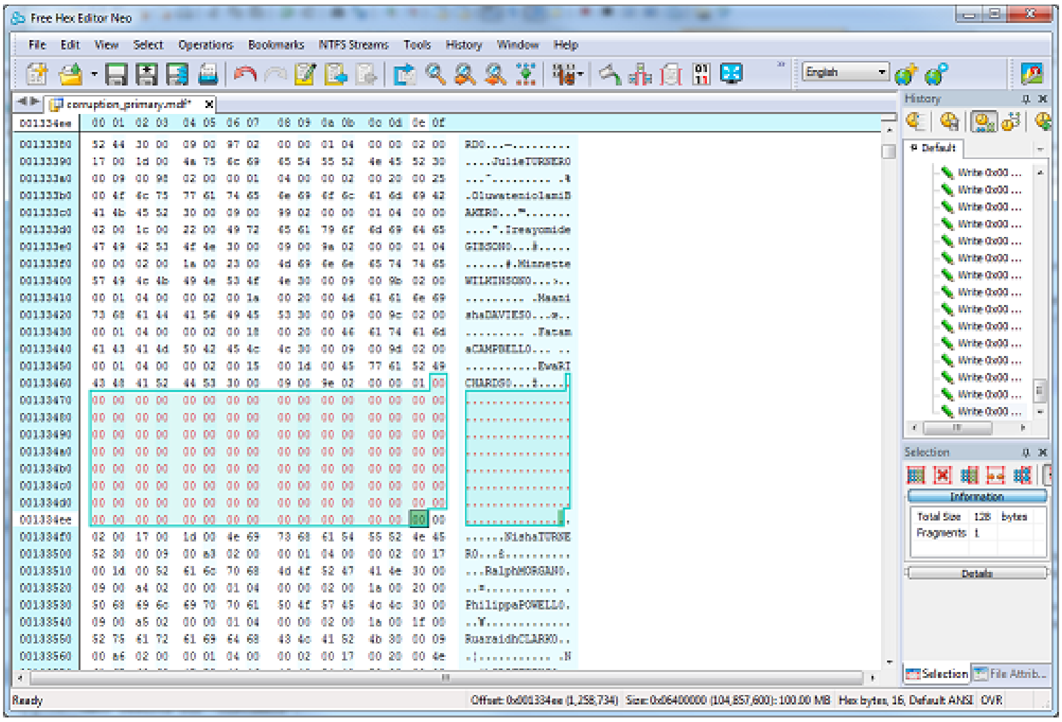
The layout is simple to understand. Each byte is represented in base 16 (hexadecimal) and arranged on the left in sixteen columns. The offset (distance from the start of the file) is on the far left, highlighted in cyan. On the right, the character representation of each byte is shown (ASCII, not Unicode). Scrolling up and down, we can see some of the data in this file. The screenshot below shows some first names and surnames in the dbo.Customers table:



```
00133460  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00133460  6c 69 42 52  4f 57 4e 30  00 09 00 ac  00 00 00 01  liBROWN0...¬....
00133470  04 00 00 02  00 1b 00 21  00 4d 6f 68  61 64 64 65  .......!.Mohadde
00133480  73 61 50 41  52 4b 45 52  30 00 09 00  ad 00 00 00  saPARKER0...-...
00133490  01 04 00 00  02 00 17 00  1d 00 45 6d  62 65 72 47  ..........EmberG
001334a0  49 42 53 4f  4e 30 00 09  00 ae 00 00  00 01 04 00  IBSON0...®......
001334b0  00 02 00 1a  00 20 00 53  61 76 61 6e  6e 61 68 4d  ..... .SavannahM
001334c0  4f 52 47 41  4e 30 00 09  00 af 00 00  00 01 04 00  ORGAN0...¯......
001334d0  00 02 00 1c  00 20 00 54  61 72 61 2d  4c 65 69 67  ..... .Tara-Leig
001334e0  68 57 45 42  42 30 00 09  00 b0 00 00  00 01 04 00  hWEBB0...°......
001334f0  00 02 00 14  00 1c 00 53  69 79 61 46  4f 53 54 45  .......SiyaFOSTE
00133500  52 30 00 09  00 b1 00 00  00 01 04 00  00 02 00 17  R0...±..........
00133510  00 1a 00 41  75 72 65 61  52 4f 42 45  52 54 53 30  ...AureaROBERTS0
00133520  00 09 00 b2  00 00 00 01  04 00 00 02  00 17 00 1f  ...²............
00133530  00 44 65 6d  6d 69 4d 41  54 54 48 45  57 53 30 00  .DemmiMATTHEWS0.
00133540  09 00 b3 00  00 00 01 04  00 00 02 00  1b 00 21 00  ..³...........!.
00133550  4c 75 63 79  2d 41 6e 6e  65 42 55 54  4c 45 52 30  Lucy-AnneBUTLER0
00133560  00 09 00 b4  00 00 00 01  04 00 00 02  00 17 00 1d  ...´............
00133570  00 5a 69 6e  74 6f 4d 55  52 52 41 59  30 00 09 00  .ZintoMURRAY0...
00133580  b5 00 00 00  01 04 00 00  02 00 19 00  20 00 53 61  µ........... .Sa
00133590  70 68 79 72  65 53 54 45  57 41 52 54  30 00 09 00  phyreSTEWART0...
Ready   Offset: 0x00133460 (1,258,592)   Size: 0x06400000 (104,857,600): 100.00 MB   Hex bytes, 16, Default ANSI   OVR
```

(incidentally, this is an excellent way of hacking SQL Server for non-Windows usernames/passwords, for white-hat purposes of course).

The next screenshot shows me deliberately corrupting some data, by zeroing-out a portion of the data. This is quite close to what would happen during corruption by a third party (such as disk failure) if, for example, marked by the O/S as unusable. Let's corrupt the dbo.Customers table starting at offset 0013346f and continuing until offset 001334ef (about 128 bytes) by replacing with hex 00 (binary 0x00):



```
001334ee  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00133380  52 44 30 00  09 00 97 02  00 00 01 04  00 00 02 00  RD0...—.........
00133390  17 00 1d 00  4a 75 6c 69  65 54 55 52  4e 45 52 30  ....JulieTURNER0
001333a0  00 09 00 98  02 00 00 01  04 00 00 02  00 20 00 25  ...˜........ .%
001333b0  00 4f 6c 75  77 61 74 65  6e 69 6f 6c  61 6d 69 42  .OluwateniolamiB
001333c0  41 4b 45 52  30 00 09 00  99 02 00 00  01 04 00 00  AKER0...™.......
001333d0  02 00 1c 00  22 00 49 72  65 61 79 6f  6d 69 64 65  ....".Ireayomide
001333e0  47 49 42 53  4f 4e 30 00  09 00 9a 02  00 00 01 04  GIBSON0...š.....
001333f0  00 00 02 00  1a 00 23 00  4d 69 6e 6e  65 74 74 65  ......#.Minnette
00133400  57 49 4c 4b  49 4e 53 4f  4e 30 00 09  00 9b 02 00  WILKINSON0...›..
00133410  00 01 04 00  00 02 00 1a  00 20 00 4d  61 61 6e 69  .......... .Maani
00133420  73 68 61 44  41 56 49 45  53 30 00 09  00 9c 02 00  shaDAVIES0...œ..
00133430  00 01 04 00  00 02 00 18  00 20 00 46  61 74 61 6d  .......... .Fatam
00133440  61 43 41 4d  50 42 45 4c  4c 30 00 09  00 9d 02 00  aCAMPBELL0... ..
00133450  00 01 04 00  00 02 00 15  00 1d 00 45  77 61 52 49  ...........EwaRI
00133460  43 48 41 52  44 53 30 00  09 00 9e 02  00 00 01 00  CHARDS0...ž.....
00133470  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00133480  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00133490  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334a0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334b0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334c0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334d0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334ee  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
001334f0  02 00 17 00  1d 00 4e 69  73 68 61 54  55 52 4e 45  ......NishaTURNE
00133500  52 30 00 09  00 a3 02 00  00 01 04 00  00 02 00 17  R0...£..........
00133510  00 1d 00 52  61 6c 70 68  4d 4f 52 47  41 4e 30 00  ...RalphMORGAN0.
00133520  09 00 a4 02  00 00 01 04  00 00 02 00  1a 00 20 00  ..¤......... .
00133530  50 68 69 6c  69 70 70 61  50 4f 57 45  4c 4c 30 00  PhilippaPOWELL0.
00133540  09 00 a5 02  00 00 01 04  00 00 02 00  1a 00 1f 00  ..¥.............
00133550  52 75 61 72  61 69 64 68  43 4c 41 52  4b 30 00 09  RuaraidhCLARK0..
00133560  00 a6 02 00  00 01 04 00  00 02 00 17  00 20 00 4e  .¦.......... .N
```

Offset: 0x001334ee (1,258,734) Size: 0x06400000 (104,857,600): 100.00 MB Hex bytes, 16, Default ANSI OVR

Now let's try and get this database online:

```
ALTER DATABASE [CORRUPTION] SET ONLINE;
Result:
Command(s) completed successfully.
```

That's good. But the corruption in this case only extends for 128 bytes and evidently only hits data pages (see http://msdn.microsoft.com/en-gb/library/ms190969(v=sql.105).aspx for a full rundown of page types in SQL Server). If I had corrupted GAM or SGAM pages, or IAM (for indexes), or indeed BCM/DCM pages, I would likely have received a message similar to the type shown at the beginning of this article. So this shows there are two key types of corruption which I will term detected corruption, and undetected corruption. The deliberate corruption just witnessed is undetected, and is arguably even more dangerous than detected corruption, since the database can continue to jog along fine until this data is accessed. This allows corruption to fester within the database and make recovery situations even more complex.

A Deeper Look at the SQL Server Corruption

Let's run DBCC CHECKDB to see if we can uncover the corruption using SQL Server Management Studio:

```
...
DBCC results for 'customers'.
Msg 8928, Level 16, State 1, Line 1
Object ID 2105058535, index ID 1, partition ID 72057594038779904, alloc unit ID 72057594039697408 (type
Msg 8939, Level 16, State 98, Line 1
Table error: Object ID 2105058535, index ID 1, partition ID 72057594038779904, alloc unit ID 72057594039
...
repair_allow_data_loss is the minimum repair level for the errors found by DBCC CHECKDB (CORRUPTION).
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

Argh! Note the key phrase here: "repair_allow_data_loss is the minimum repair level for the errors found by DBCC CHECKDB (CORRUPTION)"

However unhelpful this statement is, the detailed results produced by DBCC CHECKDB do give us some information on where the corruption lies. We can also find this information using DBCC PAGE and DBCC IND (Background information on these commands here - http://www.sqlskills.com/blogs/paul/inside-the-storageengine-using-dbcc-page-and-dbcc-ind-to-find-out-if-page-splits-ever-roll-back/, here - http://blogs.msdn.com/b/sqlserverstorageengine/archive/2006/12/13/more-undocumented-fun_3a00_-dbcc-ind_2c00_-dbcc-page_2c00_-and-off_2d00_row-columns.aspx) and here - http://www.mssqltips.com/sqlservertip/1578/using-dbcc-page-to-examine-sql-server-table-and-index-data/.

DBCC CHECKDB gave us the page affected, so let's use DBCC PAGE to examine the page's hex contents (1:153). You will need to switch on trace flag 3604 first.

```
DBCC TRACEON(3604)
DBCC PAGE(CORRUPTION,1,153,2)
```

Here's some of the output (too long to show in full):

```
PAGE: (1:153)
BUFFER:
BUF @0x000000010EFF5400
bpage = 0x000000010EE68000           bhash = 0x0000000000000000          bpageno = (1:153)
bdbid = 10                           breferences = 0                     bUse1 = 64
bstat = 0x2c00809                    blog = 0x79ca2159                   bnext = 0x0000000000000000
PAGE HEADER:
Page @0x000000010EE68000
m_pageId = (1:153)                   m_headerVersion = 1                 m_type = 1
m_typeFlagBits = 0x4                 m_level = 0                         m_flagBits = 0x200
m_objId (AllocUnitId.idObj) = 27     m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594039697408
Metadata: PartitionId = 72057594038779904                                Metadata: IndexId = 1
Metadata: ObjectId = 2105058535      m_prevPage = (0:0)                  m_nextPage = (1:156)
pminlen = 9                          m_slotCnt = 252                     m_freeCnt = 1
m_freeData = 7687                    m_reservedCnt = 0                   m_lsn = (19:464:16)
m_xactReserved = 0                   m_xdesId = (0:0)                    m_ghostRecCnt = 0
m_tornBits = 1132858382
```

You can see that this is where I deliberately corrupted the file. But you'll notice the page location differs from our original absolute (file) offset figure of 0x013346f (1258607) - instead, it's 0x10EE68000. This gives us a problem - it's difficult to locate the corrupted data.

Locating SQL Server Database Corruption

Happily, there's a simple way to both confirm the page number and find the physical offset of the I/O corruption - force a logical consistency error. We know from this line in the DBCC CHECKDB output the name of the table or index affected:

```
"DBCC results for 'customers'..."
```

So let's force a logical consistency error by attempting to read this table. This will provide us with a physical offset address from the start of the affected file:

```
SELECT * FROM dbo.Customers
```

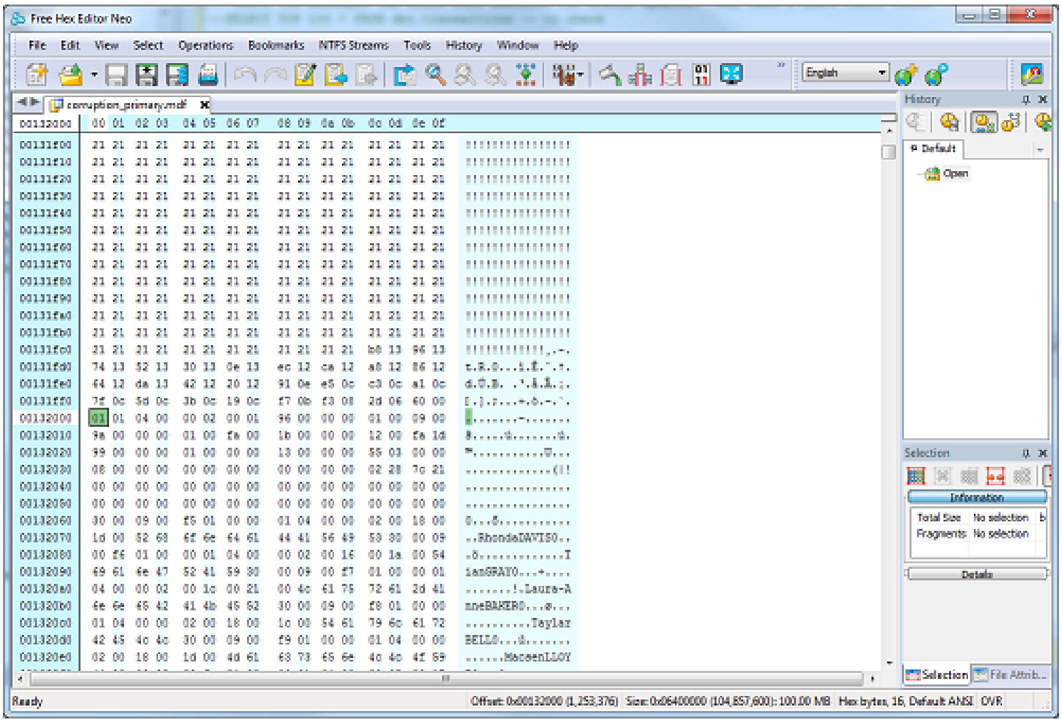
We then receive the following error:

```
Msg 824, Level 24, State 2, Line 2
SQL Server detected a logical consistency-based I/O error: incorrect checksum
(expected: 0x43860c0e; actual: 0x958afc9c). It occurred during a read of page (1:153)
 in database ID 10 at offset 0x00000000132000 in file 'c:\del\corruption_primary.mdf'.
  Additional messages in the SQL Server error log or system event log may provide more
 detail. This is a severe error condition that threatens database integrity and must
 be corrected immediately. Complete a full database consistency check (DBCC CHECKDB).
 This error can be caused by many factors; for more information, see SQL Server Books
 Online.
```

We can confirm this address is reasonably accurate by confirming the page number is 153 (as per the DBCC PAGE output and DBCC CHECKDB output), and therefore confirm the physical location of the corruption as follows:

- Offset(h) / 2000(h) = Page Id (h) (divide the offset given by 8192 in hexadecimal (2000))
- 0x00132000 / 2000 = 99(h)
- 99(h) = 153 (convert back to decimal to get the page number)

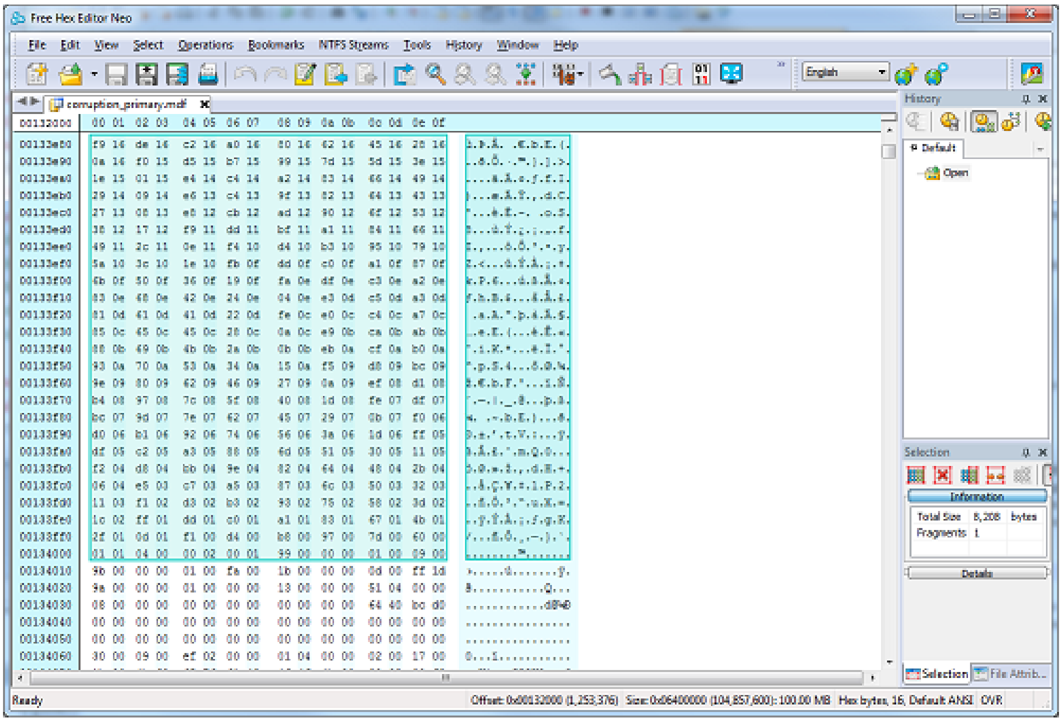
Now, 132000(h) differs from 13346f(h) which is where we KNOW we started to corrupt the data. Let's take a look at the hex editor at physical offset 132000 to try and explain the difference:



```
00132000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00131f00  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f10  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f20  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f30  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f40  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f50  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f60  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f70  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f80  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131f90  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131fa0  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131fb0  21 21 21 21  21 21 21 21  21 21 21 21  21 21 21 21  !!!!!!!!!!!!!!!!
00131fc0  21 21 21 21  21 21 21 21  21 21 21 21  b8 13 96 13  !!!!!!!!!!!!,.-.
00131fd0  74 13 52 13  30 13 0e 13  ec 12 ca 12  a8 12 86 12  t.R.0...ì.Ê.¨.†.
00131fe0  64 12 da 13  42 12 20 12  91 0e e5 0c  c3 0c a1 0c  d.Ú.B. .'.å.Ã.¡.
00131ff0  7f 0c 5d 0c  3b 0c 19 0c  f7 0b f3 08  2d 06 60 00  ..].;...÷.ó.-.`.
00132000  01 01 04 00  00 02 00 01  96 00 00 00  01 00 09 00  ........–.......
00132010  9a 00 00 00  01 00 fa 00  1b 00 00 00  12 00 fa 1d  š.....ú.......ú.
00132020  99 00 00 00  01 00 00 00  13 00 00 00  55 03 00 00  ™...........U...
00132030  08 00 00 00  00 00 00 00  00 00 00 00  02 28 7c 21  .............(|!
00132040  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00132050  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00132060  30 00 09 00  f5 01 00 00  01 04 00 00  02 00 18 00  0...õ...........
00132070  1d 00 52 68  6f 6e 64 61  44 41 56 49  53 30 00 09  ..RhondaDAVIS0..
00132080  00 f6 01 00  00 01 04 00  00 02 00 16  00 1a 00 54  .ö.............T
00132090  69 61 6e 47  52 41 59 30  00 09 00 f7  01 00 00 01  ianGRAY0...÷....
001320a0  04 00 00 02  00 1c 00 21  00 4c 61 75  72 61 2d 41  .......!.Laura-A
001320b0  6e 6e 65 42  41 4b 45 52  30 00 09 00  f8 01 00 00  nneBAKER0...ø...
001320c0  01 04 00 00  02 00 18 00  1c 00 54 61  79 6c 61 72  ..........Taylar
001320d0  42 45 4c 4c  30 00 09 00  f9 01 00 00  01 04 00 00  BELL0...ù.......
001320e0  02 00 18 00  1d 00 4d 61  63 73 65 6e  4c 4c 4f 59  ......MacsenLLOY
```

```
Ready    Offset: 0x00132000 (1,253,376)  Size: 0x06400000 (104,857,600): 100.00 MB  Hex bytes, 16, Default ANSI  OVR
```

You will note that offset 132000(h) corresponds to the START of the corrupted page. We will not be able to determine exactly WHERE in the page the corruption occurs, simply because corruption is detected using checksums. The checksum will calculate whether or not the page is valid - not pinpoint where exactly the corruption begins. We know a page is 8,192 bytes in SQL Server - that's the default page size, and we now know that this converts to 2000(h). So simply slice a page (offset start 132000(h) to 134000(h)) from the corrupted database file into your text comparison tool. I'm going to take the hex representation of the data, for simplicity:



```
00132000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00133e80  f9 16 de 16  c2 16 a0 16  80 16 62 16  45 16 28 16
00133e90  0a 16 f0 15  d5 15 b7 15  99 15 7d 15  5d 15 3e 15
00133ea0  1e 15 01 15  e4 14 c4 14  a2 14 83 14  66 14 49 14
00133eb0  29 14 09 14  e6 13 c4 13  9f 13 82 13  64 13 43 13
00133ec0  27 13 08 13  e8 12 cb 12  ad 12 90 12  6f 12 53 12
00133ed0  38 12 17 12  f9 11 dd 11  bf 11 a1 11  84 11 66 11
00133ee0  49 11 2c 11  0e 11 f4 10  d4 10 b3 10  95 10 79 10
00133ef0  5a 10 3c 10  1e 10 fb 0f  dd 0f c0 0f  a1 0f 87 0f
00133f00  6b 0f 50 0f  36 0f 19 0f  fa 0e df 0e  c3 0e a2 0e
00133f10  83 0e 68 0e  42 0e 24 0e  04 0e e3 0d  c5 0d a3 0d
00133f20  81 0d 61 0d  41 0d 22 0d  fe 0c e0 0c  c4 0c a7 0c
00133f30  85 0c 65 0c  45 0c 28 0c  0a 0c e9 0b  ca 0b ab 0b
00133f40  88 0b 69 0b  4b 0b 2a 0b  0b 0b eb 0a  cf 0a b0 0a
00133f50  93 0a 70 0a  53 0a 34 0a  15 0a f5 09  d8 09 bc 09
00133f60  9e 09 80 09  62 09 46 09  27 09 0a 09  ef 08 d1 08
00133f70  b4 08 97 08  7c 08 5f 08  40 08 1d 08  fe 07 df 07
00133f80  bc 07 9d 07  7e 07 62 07  45 07 29 07  0b 07 f0 06
00133f90  d0 06 b1 06  92 06 74 06  56 06 3a 06  1d 06 ff 05
00133fa0  df 05 c2 05  a3 05 88 05  6d 05 51 05  30 05 11 05
00133fb0  f2 04 d8 04  bb 04 9e 04  82 04 64 04  48 04 2b 04
00133fc0  06 04 e5 03  c7 03 a5 03  87 03 6c 03  50 03 32 03
00133fd0  11 03 f1 02  d3 02 b3 02  93 02 75 02  58 02 3d 02
00133fe0  1c 02 ff 01  dd 01 c0 01  a1 01 83 01  67 01 4b 01
00133ff0  2f 01 0d 01  f1 00 d4 00  b8 00 97 00  7d 00 60 00
00134000  01 01 04 00  00 02 00 01  99 00 00 00  01 00 09 00
00134010  9b 00 00 00  01 00 fa 00  1b 00 00 00  0d 00 ff 1d
00134020  9a 00 00 00  01 00 00 00  13 00 00 00  51 04 00 00
00134030  08 00 00 00  00 00 00 00  00 00 00 00  64 40 bc d0
00134040  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00134050  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
00134060  30 00 09 00  ef 02 00 00  01 04 00 00  02 00 17 00
```

```
Ready    Offset: 0x00132000 (1,253,376)  Size: 0x06400000 (104,857,600): 100.00 MB  Hex bytes, 16, Default ANSI  OVR
```

However, what we have done is narrowed the range where the corruption lies.

Fixing the SQL Server Corruption Problem

We can now use a data comparison tool (pick one that suits you) to compare the page from a valid copy of the database to the corrupted page. You will then get a side-by-side comparison of exactly what has changed, and what bytes within that range need changing. I'm going to use an online service, http://www.textdiff.com, to do this. The text highlighted in red, below, shows the changed values:

```
00 01 04 00 00 02 00 15 00 1d 00 45 77 61 52 49
43 48 41 52 44 53 30 00 09 00 9e 02 00 00 01 0400
00 00 02 00 18 00 1d 00 44 65 65 63 61 6e 43 4c
41 52 4b 30 00 09 00 9f 02 00 00 01 04 00 00 0200 00 00 00 00
00 19 00 1f 00 54 61 73 6e 69 65 6d 46 4f 53 54
45 52 30 00 09 00 a0 02 00 00 01 04 00 00 02 00
1b 00 22 00 50 68 69 6c 69 70 70 6f 73 52 4f 42
45 52 54 53 30 00 09 00 a1 02 00 00 01 04
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00
02 00 18 00 20 00 48 6f 6e 6f 75 72 54 48 4f 4d
50 53 4f 4e 30 00 09 00 a2 02 00 00 01 04 00 00
02 00 17 00 1d 00 4e 69 73 68 61 54 55 52 4e 45
```

The easiest way of getting a valid version of the affected page is to restore an older copy of the database as a new, separate, database, then set it offline and open it in your hex editor. Providing you have a *reasonable expectation* that the data in that page is unlikely to have changed (naturally) in the interval between the date of the backup and the date of the corruption (you can calculate the probability for this depending on your circumstances) then you will be able to copy and paste the entire data page (don't copy and paste just the block that's changed, since the 'old' checksum will invalidate it) from the hex editor window with the valid copy to the hex editor window with the corrupted copy.

Once you have done this, save and close the corrupted file, then SET ONLINE in SQL Server Management Studio. Now run DBCC CHECKDB. You should find no anomalies with the data.

Miscellany

CAVEATS:

- If corruption is widespread, i.e. in more than one location or over a large area, other types of pages may

be affected which will cause SQL Server to be unable to open the file. You may wish to restore instead, or do a larger copy/paste operation by directly comparing an old backup with the corrupted copy and replacing larger swathes of the data file.
- Direct editing of the file is a method that is intolerant of faults. Checksums are calculated on a page-by-page basis and should you fail to copy/paste correctly, I/O errors will still occur or you may be unable to open the database.
- This is not a 'quick-fix' solution - you will need to take time beforehand practicing this kind of operation for it to be successful.

Another interesting tidbit of information relating to this is a method of calculating the allocation unit ID. The allocation unit ID simply refers to the ID of the allocation unit, which is a logical container of data than spans multiple pages. There are three types - you can read more about them here - http://sqlserverdownanddirty.blogspot.co.uk/2011/04/what-are-allocation-units.html. Allocation unit views such as sys.allocation_units are helpful for finding out, for example, how many pages in a particular allocation unit are used (giving you an idea of, in this context, how much data is affected). By checking data_space_id in the aforementioned view, you can see which filegroup the corrupted data is located in. However what's not commonly known is that you can get the allocation unit ID from the object ID by using some simple mathematics. Note this doesn't apply if the object is a LOB (m_objid will be 255):

```
m_objid * 65536 + (2^56) = Allocation Unit ID
```

Using the DBCC PAGE information from earlier:

- m_objId = 27
- 27 * 65536 = 1769472
- 1769472 + (2^56) = 72057594039697408
- Metadata: AllocUnitId = 72057594039697408 (confirmed)

Also, bear in mind that if you're attempting to find out whether the corruption affects an index (in this case, the clustered index of dbo.customers) you may find this difficult to establish, especially if you have had to start the database in EMERGENCY mode. Fortunately, we can find out whether or not the corruption affects an index or not by using some system views.

So we know the allocation unit ID using the above method - let's find the hobt_id (which is identical to the container_id if type = IN_ROW_DATA) and we can then query sys.partitions to give us the index number, if not on the clustered index for the affected object.

```
SELECT * FROM sys.allocation_units
WHERE allocation_unit_id = 72057594039697408
```

Results:

```
allocation_unit_id type  type_desc container_id  data_space_id total_pages used_pages data_pages
72057594039697408 1  IN_ROW_DATA 72057594038779904 1    25   21   19
```

OK, so we know that if type = 1 or 3 ('in-row data') then container_id = sys.partitions.hobt_id (otherwise sys.partitions.partition_id for LOB data). Using this information:

```
SELECT * FROM sys.partitions WHERE hobt_id = 72057594038779904
```

Results:

```
partition_id  object_id index_id partition_number hobt_id    rows filestream_filegroup_id data_compressi
72057594038779904 1131151075 1  1   72057594038779904  2506 0   0   NONE
```

This is useful information - it tells us the number of potential affected rows, plus the index_id on the object, which we can find out more information about by querying sys.objects, sys.tables and sys.indexes, JOINing on this ID where necessary.

Next Steps

- I hope this article has given you an insight into recovery of SQL Server databases using non-standard tooling. Feel free to comment and I'll respond as soon as possible. If this topic interests you, please follow the links scattered throughout this piece for further information, particularly for the DBCC commands PAGE, IND and CHECKDB. Many thanks for reading.
- Further MSSQLTips.com reading:
    - Table-level Recovery - http://www.mssqltips.com/sqlservertip/2814/table-level-recovery-for-selected-sql-server-tables/
    - Using the Emergency State for a Corrupt SQL Server Database - http://www.mssqltips.com/sqlservertip/2333/using-the-emergency-state-for-a-corrupt-sql-server-database/
    - SQL Server Disaster Recovery Planning and Testing - http://www.mssqltips.com/sqlservertip/1258/sql-server-disaster-recovery-planning-and-testing/