Tech Tip #17: File Corruption Detection

Overview: Finding database corruption from status codes, PVSW.LOG messages or KeyCheck

Even with the best of intentions, sometimes things just go wrong. Such is the case with everything in life, and your PSQL database environment is no exception. While PSQL has safeguards built into the environment to prevent data loss, like shadow paging, transaction logging, and more, in the end the database is still stored on a computer hard disk, and failures do occur. Further, the individual bits in today's hard disks are VERY small – about 13nm, or about a dozen atoms. It's amazing they work at all!

When something "bad" happens to a hard disk, be it an operating system crash, hardware failure, power outage, or just about anything else, there is a chance that it will impact your database files. This is why we have discussed the issues of backup and disaster recovery in this Tech Tip series. When such a failure doe simpact your database, the engine will be the first to know about it because it is the only thing that is actually touching the database files.

When the engine runs into a corrupted data file, it usually is able to detect that fact and report the error back to the application. This is most commonly reported as a **Status 2** (I/O Error) or a **Status 54** (Variable Page Error). When these errors get back to the application, however, the application is then responsible for notifying the user so that some action steps can be taken. In many cases, applications will error out and display this message immediately – but in other cases, the corruption may go hidden for some time, or it may hide behind other cryptic messages.

Luckily, this is not the only notification option. The database engine also logs unusual events (such as corrupted files) to the **PVSW.LOG** on the server. In order to detect corruption more quickly after a server crash or other outage, you should periodically check the PVSW.LOG for such messages, especially in the first few days following a failure. In most cases, the messages will be flaged as *System Error* messages, as indicated in this example:

```
NTDBSMGR64.EXE  Nostromo        E     System Error: 116.6.0 File: R:\REPAIR\APDISTH
NTDBSMGR64.EXE  Nostromo        E     System Error: 116.6.0 File: R:\REPAIR\APHDRH
NTDBSMGR64.EXE  Nostromo        E     System Error: 116.12.0 File: R:\REPAIR\APLINH
NTDBSMGR64.EXE  Nostromo        E     System Error: 116.6.0 File: R:\REPAIR\APSERIAL
```

A *System Error* will aways be reported with three numbers. The first number indicates the location (within the engine code) where the error was detected. The second number indicates the database operation that was being attempted. The third number indicates the OS return code, if any. If the last code is non-zero, check the OS documentation for that status code for more information. If you see a 0, as in this example, then you likely have a damaged file that needs to be fixed.

Note that the engine has to access a bad page or bad pointer in order to report the error – this is why corruption can remain hidden for days or weeks after a crash. It is possible to force the database engine to access every page of every file (and thus detect corruption when it has occurred) using a carefully crafted process that reads each record of a file by Key 0, Key 1, and so on. This can be done with the built in BUTIL tool via a simply batch script like this:

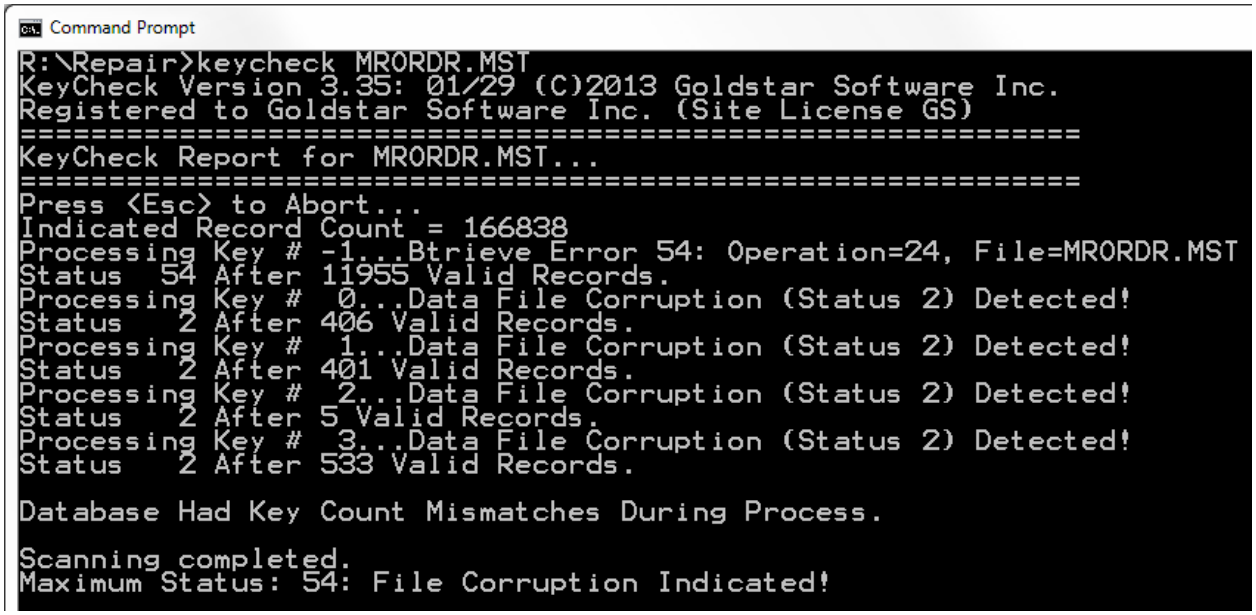
```
BUTIL –RECOVER datafile NUL
BUTIL –SAVE datafile NUL N 0
BUTIL –SAVE datafile NUL N 1
BUTIL –SAVE datafile NUL N 2
```

If you look carefully, you'll see that this process reads through the file in every possible way. However, you still have to watch the process and determine whether it succeeds or fails.

The Goldstar Software utility called **KeyCheck** does all of this, and more:

http://www.goldstarsoftware.com/keycheck.asp

Internally, **KeyCheck** is very simple – it reads every record from a database table in physical order as well as by each key. Where it is superior to the **BUTIL** process, though, is in both performance and reporting. First, **KeyCheck** can be considerably faster, especially since the registered version provides a multi-threaded scanning process that can process all of the keys concurrently. Second, KeyCheck can immediately report an error and stop processing, or it can continue to process all of the available data records on other keys, to see how bad the file actually is. Here is one example:



In this example, **KeyCheck** has detected corruption on every key path, and it has reported both Status 2 and Status 54 errors. This file is indeed in some trouble. Another related feature of KeyCheck is the ability to locate missing records on a given key. In other words, when a file is mangled, try to extract enough information about the missing record so that it can be recovered somehow.

Of course, once corruption is detected, it then needs to be fixed. The first line of defense is the system backup, as we've discussed in a previous Tech Tip. With this option, simply restoring the damaged file may be the entire solution. In other cases, linkages between the various tables (common in a relational database, where data in one table is related to other tables) may force you to restore the entire database. This is where the database expert cannot help – you really need to contact the application developer to figure out the best way to recover from any given corruption.

What if you don't have a good backup? Tsk, tsk. Have we taught you nothing? Luckily, there *are* other options, which we will explore in our next Tech Tip!