IRON Databases

Adeel Pervez and Joshua Woleben
Department of Computers Sciences, University of Wisconsin-Madison

Abstract: Modern database systems store critical data such as financial transactions, health records, research results, and government data. Continuous access to this data and its integrity are crucial to the business, academic and government environments that rely on them. Database system downtime or data loss can cause millions of dollars in lost business and productivity, and can even result in liability. In this paper we have explored two modern database systems to test their reaction to various types of corruption. We explored a closed source, proprietary system from Intersystems Corporation called Cache, and an open-source database system called PostgreSQL. To accomplish this, fault injection at the pointer level and data structure level was used, then the database tested and its reaction, if any, was recorded. We found that both database systems lack in integrity recovery and could use improvement.

1. Introduction

Our entire economy relies on the availability and integrity of modern database systems, from Oracle to DB2 to MySQL. These databases store everything from health records for patients, government data on criminals and statistics, financial records, accounting, research results, and flight patterns. These databases are required to be secure, reliable, available and perform well. To meet all of these requirements some trade-offs are made. Reliability may be sacrificed for performance, or vice-versa. However, a major question in all database systems is how to guarantee data integrity. What good is high availability if the data that is made available is rendered useless because it is wrong, logically inconsistent or impossible to read?

We decided to test the reaction of various database systems to see how well they reacted to integrity problems. Since the scope of the initial venture was limited, we decided to test two major database systems. The first system was a MUMPS-based RDBMS from Intersystems Corporation called Cache. Cache is a closed-source system that is supported on almost every platform. The other system we tested was PostgreSQL, an open-source, SQL-based system.

To test these systems, we were able to do thorough fault-injection by either corrupting data structures themselves or the pointers within the data structures. Data was either then rewritten to the database or integrity checked over the same points to test the database's reaction. Each case was recorded.

The results were unfortunate but not unexpected. Both database systems reacted poorly to integrity issues, although Cache was able to detect all corruption when its integrity checker was run. PostgreSQL was able to detect some faults through sanity checking and type checking.

2. Intersystems Cache

2.1 Data structure

Cache is a post-relational database system supported on almost all modern platforms. Cache maintains data in persistent data structures called globals. Globals are denoted by the carat symbol followed by the global name, for example ^DATA. Globals are tree structures that resemble arrays of arrays and can have an indeterminate depth. A global can either be assigned a single value, such as a numeric or string, or can be composed of multiple nodes denoted by numeric or string values as subscripts, such as ^DATA(“NAME”)=”WOLEBEN, JOSHUA”.

Globals in Cache are stored on disk in files named CACHE.DAT. Cache won't mount files of any other name, so to have multiple databases, each CACHE.DAT file must be in its own directory. Within CACHE.DAT files are a structure that organize the global data and make it rapid to find.

The basic structure inside a CACHE.DAT file is that of a B+ tree divided into 8 KB blocks. These blocks will have different purposes to help find the data. The basic kinds of blocks in a Cache database file are: global directory block, upper-level pointer block, bottom-level pointer block, global data block, big global data block, and map block. Each block type also has an associated internal number that is used by Cache.

The global directory block contains information for each global in the database. Each entry in the

global data block contains the global name, collation type (a designator internal to Cache that determines how data is stored), protection setting, growth area and the location of the global's top-level pointer block.

Top-level pointer blocks are the uppermost block in the B+ tree for a global. This block is referred to by the global directory block and contains indexes for pointer blocks lower in the tree.

Pointer blocks comprise the middle levels of the B + tree, and contain index values for bottom-level pointers that lead to the data.

Bottom-level pointer blocks are immediately above the data blocks. The bottom-level pointer blocks contain indexes that lead to the actual data being requested.

Data blocks contain the actual data for the global stored on disk. Data is stored in collated order based on the collation type, global name, and subscript name.

Occasionally a global will only need a single pointer block. In this case, there is only one pointer block between the global directory block and the data block itself.

In addition to all of the pointer blocks, each pointer and data block has a right link pointer at the end of the block that points to the next block on its right. This allows for an additional structure of redundancy in the B+ tree structure.

The integrity of this B+ tree structure is critical for continued access to the data. If the structure is corrupted, data can be impossible to access. In most cases only a few pointers are incorrect and can be repaired by the operator. In cases of severe corruption, however, we have to restore the database from a backup and play forward the journal files that Cache keeps to log data transactions to the database. In this paper, we will explore the ways that Cache reacts to a database integrity issue and how they can be repaired, if possible.

2.2 Methodology

The ^REPAIR utility, a Cache routine that can access the pointer structure of the database file, will be used to inject pointer corruption in the database and to also repair it. We will examine various kinds of block corruption at the pointer and data level.

The test Cache instance (the term for an installation of Cache) is hosted on a MacBook Pro with an Intel Core 2 Duo processor at 2.4 GHz and 2 GB of RAM, running Mac OS X 10.5.1. The version of Cache being tested is 5.2.3.

The following kernel parameters had to be set in order to run Cache on Mac OS X: kern.sysv.shmmax to 256,000,000 bytes, and kern.sysv.shmall to 16,777,216 bytes.

First we need a database file to test our code in. Using the ^DATABASE routine, we can create a 5 MB database file in /Applications/cachesys/test/CACHE.DAT.

Next we need a namespace for our code to interact with the database. Using Cache's System Management Portal, we can create a namespace TEST that refers to the database in /Applications/cachesys/test.

Cache is an implementation of the MUMPS programming language and database structure, so our test data will be created using an M routine. Below is our test code, a simple for loop that will assign subscripts of a Cache global a set of values so we have a wide array of data to work with: **f i=1:1:10000000 s ^CS736(i)="Test data"** This code will be run in the TEST namespace at an interactive Cache prompt.

First we should create a backup of our TEST database so we can restore it after a fault injection. To do this, we can shut down the database and copy it to an alternate location. Once we have a backup, we can restart the database and begin probing the global structure and attempting corruption of the pointer blocks.

Once the test data is ready, we can diagram the block structure of the global ^CS736 by proceeding into the %SYS namespace and invoking the ^REPAIR utility.

^REPAIR allows us to actually change nodes of the block we're viewing, so this was used to modify the pointers in various block structures to fault inject. Once this was written to disk, the data was rewritten using a different value and the database's reaction noted. After that, the

database was checked by the integrity check utility in Cache and the error type detected noted.

2.3 Results

2.3.1 Write Results

The write results in Cache are not encouraging. In almost all cases, Cache simply wrote the data to the invalid pointer structure without checking it. Sometimes this would write all correct data, and sometimes it would only write correct data partially through the loop. Cache was inconsistent in how it wrote the data.

For write tests, each type of pointer was changed to a particular value, and the reaction tested. In the table below, a zero (0) denotes no visible reaction. An X denotes a <DATABASE> error, which means Cache detected the integrity fault. An asterisk (*) means that the database encountered a <BLOCKNO> error, which is similar to a <DATABASE> error but specifies Cache detected an invalid block number.

2.3.2 Integrity Test Results

The integrity test results were much more acceptable. Cache detected nearly all of the integrity problems we injected with its independent integrity checker. This shows Cache is capable of detecting the errors in its tree structure at some point. The numbers in the spreadsheet denote the type of error(s) detected by Cache.

2.4 Conclusions

Result Figure 1. Write Test	**Zero**	**Out of Range**	**Expected block type, wrong block**	**Global Directory Block**	**Top Pointer Block**	**Bottom Pointer Block**	**Data Block**	**Linked to itself**
Global Directory Block Pointer	0	0	0	0	0	0	0	0
Top Pointer Block Down Pointer	0	0	0	0	0	0	0	0
Top Pointer Block Right Link Pointer	0	X	X	X	X	X	X	X
Bottom Pointer Block Down Pointer	0	0	0	0	0	0	0	0
Bottom Pointer Block Right Link Pointer	0	*	0	0	0	0	0	0
Data Block Right Link Pointer	0	0	0	0	0	0	0	0

Results Figure 2. Integrity	Zero	Out of Range	Expected block type, wrong block	Global Directory Block	Top Pointer Block	Bottom Pointer Block	Data Block	Linked to itself
Global Directory Block Pointer	0	0	0	0	0	0	0	0
Top Pointer Block Down Pointer	27	27	26	9	25	26	25	25
Top Pointer Block Right Link Pointer	0	28	4 & 5	8	4 & 5	4 & 5	8	4
Bottom Pointer Block Down Pointer	27	27	5 & 26	5 & 26	5 & 26	5 & 26	5 & 26	5 & 26
Bottom Pointer Block Right Link Pointer	26 & 29	26	26 & 4	26 & 8	26 & 4	26 & 4	26 & 8	26 & 4
Data Block Right Link Pointer	11	13	13	None	26	26	13	26

Our conclusion is that Cache is more focused on I/O errors than integrity errors. Cache's reaction to any I/O error in any of its databases is to immediately freeze all updates and require manual intervention, which is designed to maintain current data integrity. More subtle database errors are undetected by the write daemons but detected by integrity checks. The current methodology to recover data is to restore from backup and apply journal files, which is effective but requires extensive downtime. We propose that Cache check the integrity of the blocks it is going to write to disk before doing the write, and that Cache keep journal files of the pointer structure it changes to allow possible live repair of the database rather than requiring restore from a backup. These features could be optional in environments where performance trumps reliability.

3. PostgreSQL

3.1 Methodology

Our methodology is to apply type-aware corruption to on-disk structures as explained in [1] and [2], run some workload and analyze the results to determine PostgreSQL's failure policy. The advantage of applying type-aware corruption is that a lot more into the system's working is obtained than by corrupting structures in a type-oblivious manner. Another advantage, as

Item	Description
PageHeaderData	20 bytes long. Contains general information about the page, including free space pointers.
ItemPointerData	Array of (offset,length) pairs pointing to the actual items. 4 bytes per item.
Free space	The unallocated space. New item pointers are allocated from the start of this area, new items from the end.
Items	The actual items (rows in tables) themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.
	Table 1. Overall Page Layout

Item	Description
pd_lsn	LSN: next byte after last byte of xlog record for last change to this page
pd_tli	TimelineID of last change
pd_lower	Offset to start of free space
pd_upper	Offset to end of free space
pd_special	Offset to start of special space
pd_pagesize_version	Page size and layout version number information
	Table 2. PageHeaderData Layout

Item	Description
t_xmin	Insert transaction ID stamp
t_xmax	Delete transaction ID stamp
t_cmin	Insert command ID stamp
t_cmax	Delete command ID stamp
t_xvac	Transaction ID for vacuum
t_ctid	Current TID or newer row version
t_natts	Number of attributes
t_infomask	Flags
t_hoff	Offset to user data
	Table 3. HeapTupleHeaderData

System Catalog	Description
pg_aggregate	Stores information regarding aggregate functions
pg_attrdef	Stores default attribute values
pg_attribute	Stores information about individual columns in tables
pg_class	Catalogs tables, indexes, sequences etc.
pg_constraint	Stores constraint information about tables
pg_database	Stores information about available databases
pg_depend	Records dependency information among tables
pg_index	Contains partial information about indexes
pg_operator	Stores information about operators
pg_type	Stores data type information
	Table 4. Tested System Catalog Tables

mentioned in [2], is that the exploration space of the tests to be performed is drastically reduced. Corruption is done in user tables, system catalog tables and various indexes on these.

Our workloads consist of SQL statements (select, insert, alter etc.). Our goal is to exercise the systems as thoroughly as possible so many variations of these are tried for each possible data corruption.

3.2 PostgreSQL Data Structures

PostgreSQL stores its data as files on the file system. Each table has a separate file. Each file is divided into fixed size pages. Table 1, 2 summarize the data structures present in a single page. Every Item (row in a table) is has a header, the HeapTupleHeaderData. This is summarized in Table 3. The system catalog tables that were tested are described in Table 4.

3.3 Results

Our result are summarized Figures 1,2,3,4.. Each cell in a figure shows the behavior when a particular workload is applied to a particular kind of corruption. We make use of the IRON taxonomy as presented in [1] for classifying the corruption detection and recovery mechanisms. The taxonomy for our case is as follows:

3.3.1 Levels of Detection:

None: No detection is performed
Type: Corruption was detected using type checking. Type checking means checking some predefined magic numbers to verify validity of data blocks.
Sanity: Corruption was detected using sanity checking. Sanity checking entails checking data values against well-known values to detect corruption.

3.3.2 Levels of Recovery:

None: No recovery is performed
Propagate: An error is propagated to the user. The system does not handle the error itself
Redundancy: Data redundancy is used to deal with the error

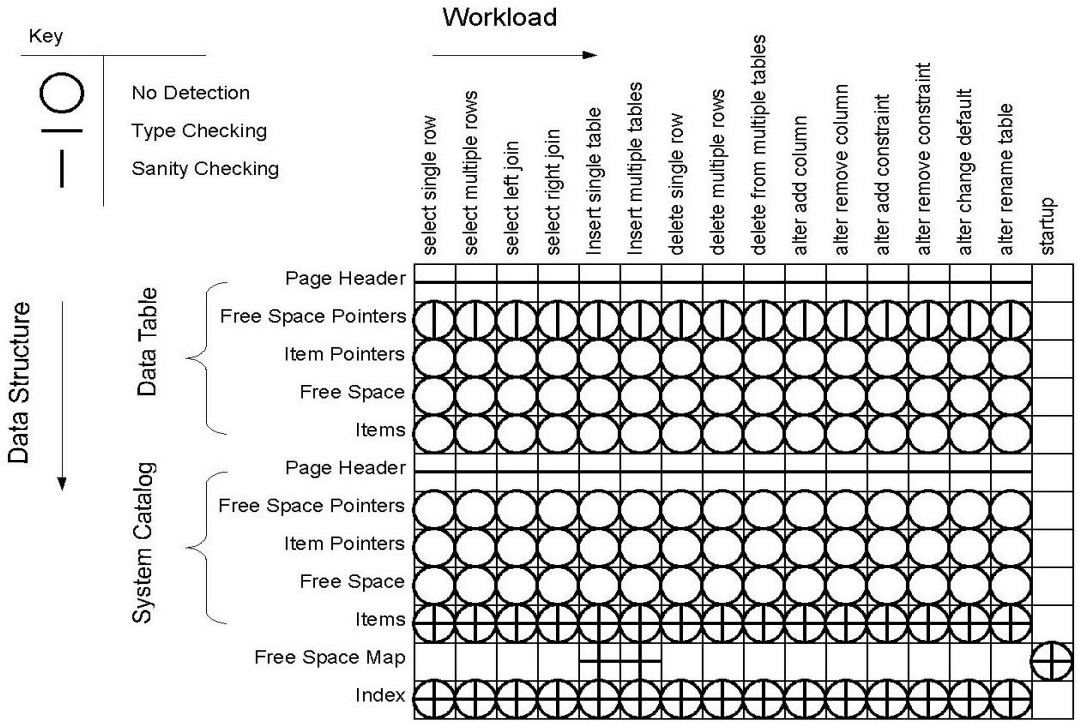
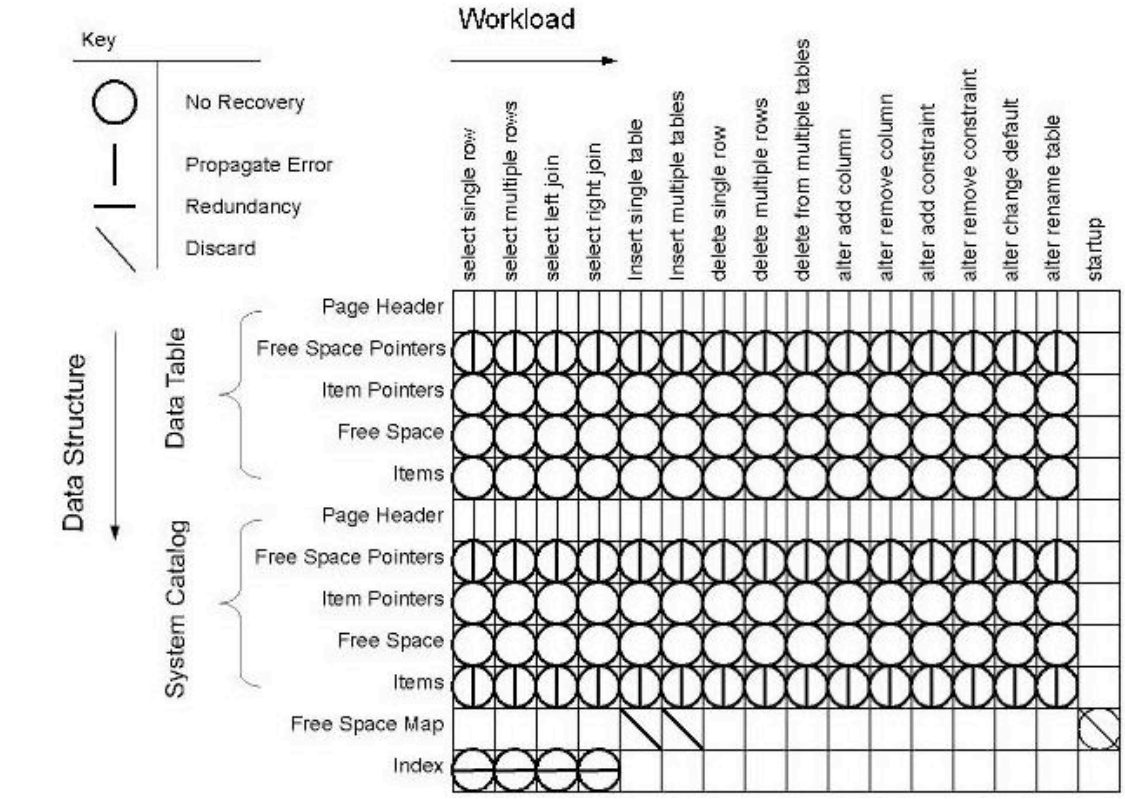
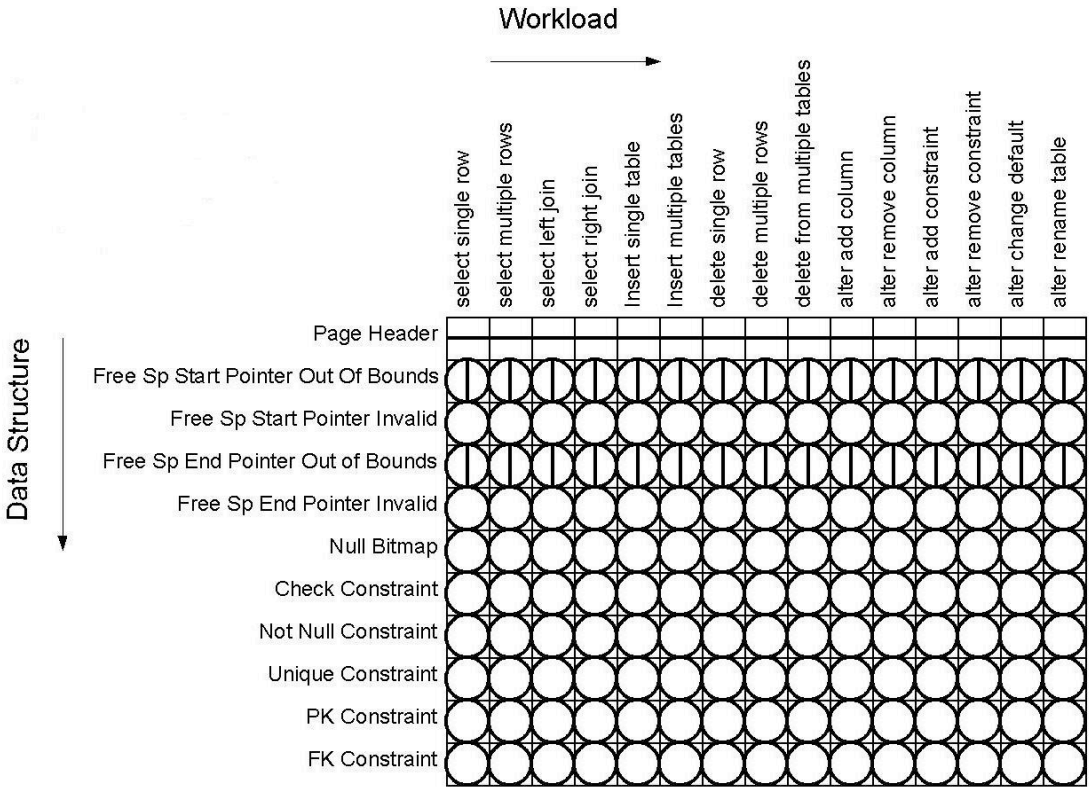
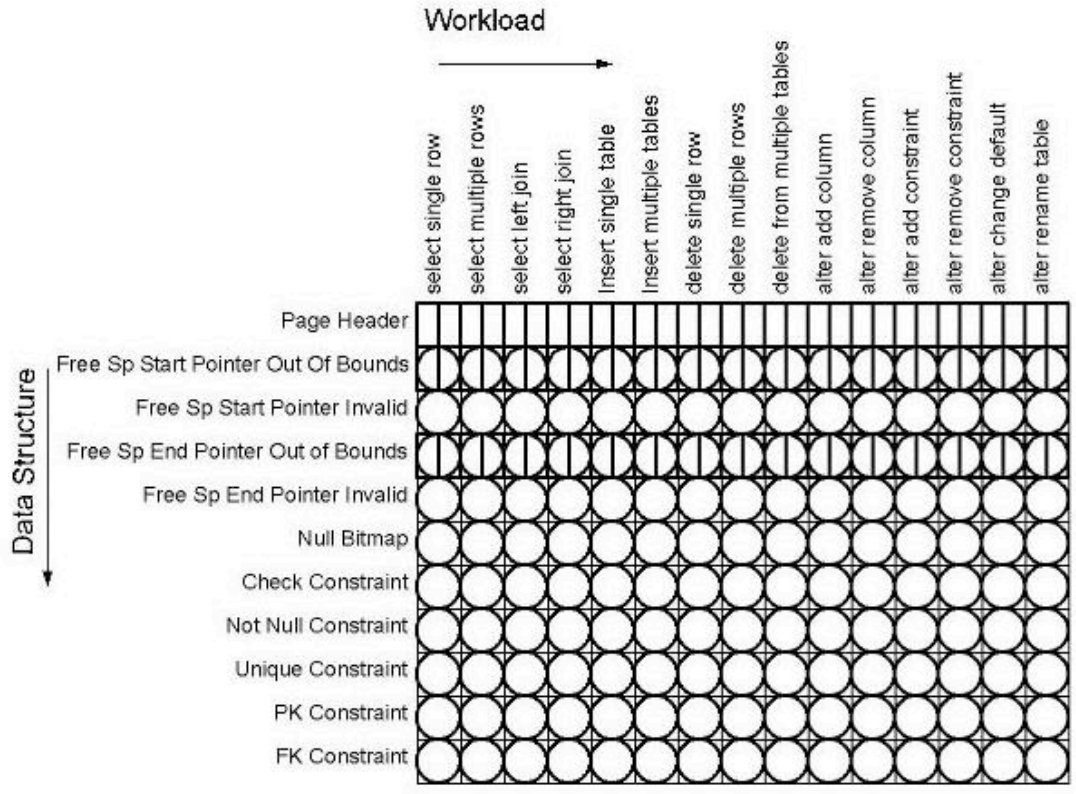


Figure 1. Data Table Detection Results


Workload
Key
No Recovery
Propagate Error
Redundancy
Discard
select single row
select multiple rows
select left join
select right join
Insert single table
Insert multiple tables
delete single row
delete multiple rows
delete from multiple tables
alter add column
alter remove column
alter add constraint
alter remove constraint
alter change default
alter rename table
startup
Data Structure
Data Table
Page Header
Free Space Pointers
Item Pointers
Free Space
Items
System Catalog
Page Header
Free Space Pointers
Item Pointers
Free Space
Items
Free Space Map
Index


Workload
select single row
select multiple rows
select left join
select right join
Insert single table
Insert multiple tables
delete single row
delete multiple rows
delete from multiple tables
alter add column
alter remove column
alter add constraint
alter remove constraint
alter change default
alter rename table
Data Structure
Page Header
Free Sp Start Pointer Out Of Bounds
Free Sp Start Pointer Invalid
Free Sp End Pointer Out of Bounds
Free Sp End Pointer Invalid
Null Bitmap
Check Constraint
Not Null Constraint
Unique Constraint
PK Constraint
FK Constraint



Discard: Corrupt data is discarded.

For some workloads we might see multiple mechanisms employed either for recovery or detection. These have also been summarized in the results.

Below we summarize some of our observations regarding PostgreSQL's failure policy.

3.3.4 Detection Observations

Observation 1 Corruption detection in PostgreSQL is done primarily using type and sanity checking.

PostgreSQL checks magic numbers associated with data structures (the PageHeader, for example) for verification when such information is available (see observation 2). Sanity checking is done when applicable; for instance Free Space Pointers are checked in this way. An out-of-bounds Free Space Pointer is detected and an error is reported.

Observation 2 Type information is not available for most important data structures.

Figure 2. Overall Recovery Results

Figure 3. Data Table Detection Results

Figure 4. Data Table Recovery Results

No type information is associated with important data structures such as an Item in a relation or an index. An item in a relation represents a row in a table and a tree node in an index. A corrupt pointer to an Item is not detected due to the lack of such type information.

Observation 3 Error detection mechanisms such as parity and checksums are not employed.

Observation 4 Most data corruptions are not detected.

In the tests we ran it was found that almost all actual table data corruptions passed through undetected. This we feel is a direct consequence of the detection policy employed as explained next.

3.4 Lessons

Lesson 1 Type and sanity checking do not/cannot detect everything.

The sanity checking for Free Space Pointers does not detect invalid pointers that are not out-of-bounds. Without additional information sanity checks alone cannot detect such errors. For some structures such information can be made available while for others it is not possible to do so. For instance, for actual table data, sanity checking does not help detect corruption.

Another example of a structure where sanity checking does not help is the Null Bitmap. The Null Bitmap for a data row tells which columns have null values. In a page in a table file, the Null bitmap appears just before the actual table row. In one of the tests we performed, a single bit corruption in the bitmap caused the data in a row to shift one column to the right. So data belonging to one column appeared in the next column! No amount of sanity checking (without additional information) can detect such a corruption.

There are other cases in which sanity checking might be able to detect corruption but is not feasible to do. Examples are the constraints on a specific column. Suppose a corruption causes a to have multiple primary keys. With sanity checking the only way to detect this is to compare every value in that column with every other value. This is clearly infeasible to do.

Lesson 2 Sanity checking can be made more comprehensive by providing additional information.

PostgreSQL stores Free Space start and end pointers that point to the start and end of free space in a data page. As we have mentioned before, in general, invalid pointers are not detected. This can be fixed by storing the actual amount of free space in the page and comparing that with the amount of free space pointed to. This way almost all Free Space Pointer errors can be detected.

Lesson 3 Include type information for more data structures.

Most pointer corruptions can be detected by including type information with the data structure pointed to. Storing type information with an Item would allow us to detect corrupt Item pointers. Type information can also be a substitute for the fix mentioned in Lesson 2. A magic number could be added to the beginning of the array of Items in a page. A Free Space End pointer corruption could then be detected by checking the value pointed to by the pointer.

Lesson 4 Use checksums

Data structures for which it is infeasible to do sanity/type checking checksums can be used. This should be done, at least, for important structures if not for everything.

3.5 Recovery Observations

According to our findings PostgreSQL does not have much machinery in the way of recovering from data corruption. Error recovery in PostgreSQL is tailored toward absolute system crashes and not partial corruption. Below we summarize our observations of the way PostgreSQL responds to data corruption.

Observation 5 Errors encountered are mostly propagated to the user.

PostgreSQL reports errors when they are detected. In most cases the table causing the error becomes inaccessible. However, there are some instances in which doing further operations on the table is still possible. There are still other instances in which the entire database becomes inaccessible.

Observation 6 Errors in performance-enhancing data structures usually cause complete failure.

Examples are indexes. An error in an index, if detected, will usually cause it to be propagated to the user and the operation aborted. Even if the actual data which is present in the table is available, it is not checked and returned.

An example where the behavior is different is the Free Space Map. The Free Space Map is meant for speeding up searches for new free space when required and is not essential for correctness. If an error is detected in the Free Space Map the action is to discard the map and start anew.

Observation 7 Replication is not used for recovery.

PostgreSQL does not maintain replicas of data with the intention of using them for recovery when needed. PostgreSQL does do some things that have the side effect of a full or partial replica being created. Indexes for instances are an example of this.

Another example are the system catalog tables. When creating a new database the system catalog tables are copied from a preexisting template of tables. Suppose during operation of a database a system catalog table is rendered inaccessible. The template replica is not used for recovery even if the table remained unchanged after being copied.

4. Related Work

Related work is being done in the filesystem field at the University of Wisconsin-Madison, for IRON file systems [1]. Work has also been done in distributed systems fault injection [5]. The field of database integrity needs further exploration and research in order to solidify an understanding of what can make database systems more reliable. However, those findings must also be applied in the constraints of performance requirements.

References

1. IRON File Systems.
2. L. N. Bairavasundaram et al., The Effects of Disk Corruption: Case Study of a Commercial File System.
3. PostgreSQL documentation: http://www.postgresql.org/docs/
4. PostgresSQL source code
5. Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment http://www.loria.fr/~festor/DSOM2001/proceedings/S5-2.pdf