Impact of Disk Corruption on Open-Source DBMS

Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Jeffrey F. Naughton

Department of Computer Science
University of Wisconsin, Madison
{srirams,yupu,vaidyana,haryadi,dusseau,remzi,naughton}@cs.wisc.edu

Abstract— **Despite the best intentions of disk and RAID manufacturers, on-disk data can still become corrupted. In this paper, we examine the effects of corruption on database management systems. Through injecting faults into the MySQL DBMS, we find that in certain cases, corruption can greatly harm the system, leading to untimely crashes, data loss, or even incorrect results. Overall, of 145 injected faults, 110 lead to serious problems. More detailed observations point us to three deficiencies: MySQL does not have the capability to detect some corruptions due to lack of redundant information, does not isolate corrupted data from valid data, and has inconsistent reactions to similar corruption scenarios.**

To detect and repair corruption, a DBMS is typically equipped with an offline checker. Unfortunately, the MySQL offline checker is not comprehensive in the checks it performs, misdiagnosing many corruption scenarios and missing others. Sometimes the checker itself crashes; more ominously, its incorrect checking can lead to incorrect repairs. Overall, we find that the checker does not behave correctly in 18 of 145 injected corruptions, and thus can leave the DBMS vulnerable to the problems described above.

I. Introduction

Disks corrupt data. Although it is well known that entire disks fail [29], [33], recent studies have shown that disks also can corrupt data that they store [4]; in a study of 1.5 million drives over three years, Bairavasundaram *et al.* found that nearly 1% of SATA drives exhibited corruption. More reliable SCSI drives encounter fewer problems, but even within this expensive and carefully-engineered drive class, corruption still takes place.

Fortunately, RAID vendors have employed increasingly sophisticated corruption detection and recovery techniques in order to combat disk corruption [6], [8]. For example, by placing a checksum of data with every block, one can detect whether a block has become corrupted; once detected, the corruption can be repaired by accessing another copy of the block or parity information.

Unfortunately, these schemes are not always enough. Recent work has shown that even with sophisticated protection strategies, the "right" combination of a single fault and certain repair activities (e.g., a parity scrub) can still lead to data loss [19]. Thus, while these schemes reduce the chances of corruption, the possibility still exists; any higher-level client of storage that is serious about managing data reliably must consider the possibility that a disk will return data in a corrupted form.

In this paper, to better understand the possible problems caused by disk corruption, we first observe its impact on a database management system, the MySQL DBMS, through a series of fault injections. By carefully injecting corruptions into a running MySQL server, we can evaluate how MySQL deals with disk corruption. More specifically, we want to answer three questions: Can MySQL detect disk corruption properly? Can MySQL keep running and return valid data despite the presence of corruption? What can we infer about the framework for corruption handling in MySQL?

We find that corruption can be quite damaging, leading to system crashes, data loss, and even incorrect results. Overall, of 145 injected faults, 110 lead to serious problems. More detailed observations point us to three deficiencies in MySQL. First, MySQL ignores some corruptions; some are detectable but ignored and some are undetectable due to lack of redundant information in its data structures. Second, MySQL does not isolate corruption from valid data; a corrupt record can make other valid records inaccessible. Finally, MySQL has widely inconsistent corruption handling, which leads us to the conclusion that MySQL does not employ a proper framework for corruption handling.

Since all corruptions are hard to detect online, a DBMS requires some form of offline corruption detection and repair tools. In the world of file systems, offline tools such as the ubiquitous file system checker (fsck) are used in this capacity today [21]. Originally conceived to help file systems recover from untimely crashes, fsck has remained a useful tool to help the file system recover from unexpected corruption in file system metadata. By carefully combing through the on-disk image, fsck can find and fix many small problems.

One might think that a DBMS does not need such a tool, as concurrency control and recovery has been carefully developed to handle many similar problems [24]. However, while concurrency control and recovery avoid many corruption scenarios and are critical in recovering from others, in general they are not able to catch or repair corrupted metadata resulting from disk malfunctions. The evidence from the marketplace unfortunately confirms this reality; many tools exist to detect and repair these kinds of errors in commercial DBMSs, including SQL Server [22], Oracle [26], [27], [28], and DB2 [16], [17]. To an unfortunate extent, neither the problems these tools address nor the approaches they employ in their solutions have appeared in the research literature. A partial explanation for this might be that such tools require detailed knowledge of proprietary aspects of the DBMSs and how they store and manage their metadata. However, the recent explosion in popularity of open source DBMSs such as PostgreSQL [37] and MySQL [39] has changed the landscape, and it is now possible for the research community at large to explore issues related to metadata corruption, and to do so in the meaningful context of substantial systems with large user communities.

Therefore, in this paper, we also study how effective offline checkers are at detecting corruption in an on-disk image of a database. Specifically, we examine the robustness of *myisamchk*, an offline checker for MySQL. We find that the offline checker is not comprehensive in the checks it performs, misdiagnosing many corruption scenarios and missing others. Sometimes the checker itself crashes. More ominously, its incorrect checking can lead to incorrect repairs. Overall, we find that myisamchk does not behave correctly in 18 of 145 injected corruptions, and thus can leave the DBMS vulnerable to the problems described above, including unexpected crashes, data loss, and incorrect results.

Thus, in this paper, we make two explicit contributions:

- We perform the first study of the effect of corruption on a running database (MySQL), and find that corruption can cause great harm (Section IV).
- We perform the first study of the ability of an existing offline checking tool (myisamchk) to detect corruption in on-disk structures, and find that it misses many significant cases (Section V).

Before describing each of our two main contributions, we first present the background and related work (Section II) and then our fault injection methodology (Section III). After the main body of the paper, we conclude with future directions (Section VI).

II. Background & Related Work

In this section, we provide a brief background on disk failures, with a focus on corruption. We then discuss why RAID is not a complete solution in dealing with corruption. After that, we briefly discuss the state of the art of both online and offline corruption detection techniques within file systems as well as the current approach within DBMS.

A. Disk Corruption: Why It Happens?

We broadly define disk corruption as something that occurs when one reads a block of data from the disk and receives unexpected contents (*e.g.*, the contents are not what were previously written to that location). Thus, the read "succeeds" (*i.e.*, the disk does not return a failure code) but the data within the block is not as expected. For this reason, corruption is sometimes referred to as a *silent* error.

Disk corruption can occur for a multitude of reasons. One cause comes from the magnetic media: the classic problem of "bit rot" which occurs when the magnetism of a single bit or a few bits are flipped. This type of problem can often (but not always) be detected and corrected with low-level ECC embedded in the drive.

Interesting errors also arise in the disk controllers due to their complexity; modern Seagate drives contain hundreds of thousands of lines of low-level firmware code that manage the operation of the disk [30]. This complexity can lead to a number of bugs which manifest as corruption.

One example of such a bug is a *lost write* (or phantom write), where a disk reports that a write has completed but in fact it was never written to the disk [40]. The next time a client reads such a block, it will receive the old contents, and thus perceive the problem as a corruption. A misconfigured drive can also result in lost writes; for example, if a drive cache is set to write-back mode (instead of write-through), a write will be acknowledged when it is put into the disk cache but before it has been written to disk. If power is lost before the actual write to the media surface, the write is seemingly lost.

A similar problem is known as a misdirected write [44]. In this case, the controller writes the data to disk but to the wrong location. A misdirected write can thus lead to two perceived corruptions; one where the block should have been written, and one where the block was accidentally written. In either case, subsequent reads receive the "wrong" contents.

There are other causes of perceived disk corruption. For example, as data sits in the main memory of a host system, a bad DRAM could corrupt the data [20]; although it is written correctly to disk, the data is corrupt when written and will later be perceived as such. Similarly, buggy operating systems software [10], [12] could accidentally overwrite the in-memory data before writing it to the disk, again leading to a subsequently perceived corruption.

B. Disk Corruption: How Often It Occurs?

Until recently, there was very little data on how often corruption arose in modern storage systems. Although there was much anecdotal information [6], [40], [44], and a host of protection techniques that systems employ to handle such corruption [19], there was little hard data.

Recently, a study by Bairavasundaram *et al.* demonstrates that corruption does indeed occur across a broad range of modern drives [4]. In that study of 1.5 million disk drives deployed in the field, the authors found more than 400,000 blocks have checksum mismatches over three years. They also found that nearline disks develop checksum mismatches an order of magnitude more often than enterprise class disk drives. Furthermore, checksum mismatches within the same disk show high spatial and temporal locality, and checksum mismatches across different disks in the same storage system are not independent. The data shows that corruption takes place, and systems must be prepared to handle it.

C. Doesn't RAID Help?

The end-to-end argument states that failure recovery must be done at the highest level; protection mechanisms at lower levels may improve performance but fundamentally do not solve the desired problem [32]. In the world of storage systems, it would be ideal if RAID storage [29] guaranteed that data was not corrupt. Although we believe that while RAID can indeed improve DBMS reliability, it is not a complete solution for the following reasons.

First, RAID is designed to tolerate the loss of a certain number of disks or blocks (*e.g.*, RAID-5 tolerates one, and RAID-6 tolerates two), but not to to identify corruption. For example, in RAID-5, if a block in a parity set is corrupt, the parity computation will be incorrect, but which block is corrupt cannot be identified with RAID alone.

Second, ironically, commercial RAID systems also corrupt data; a recent paper by Krioukov *et al.* demonstrates how most commercial RAID-5 designs, which should be able to tolerate the loss of any one disk or block, have flaws where a single block loss leads to data loss or silent corruption [19].

Finally, not all systems incorporate more than one disk. For example, consider a typical commodity system running with a single disk drive; in such systems, there is essentially no protection against most forms of corruption described above.

D. Doesn't Checksumming Help?

Checksumming techniques have been used in numerous systems over the years to detect data corruption [6], [11], [36], [38], [40]. For example, Tandem systems have long employed checksums [6]. When a block is read from disk, so too is its stored checksum. A checksum is then computed over the data block and compared to the stored checksum; if the two do not match, the block is declared corrupt and recovered from a mirror copy. Similar to RAID, although checksums can improve corruption detection, it is still not a complete solution for three reasons.

First, memory is not perfect. For example, a bit-flip in memory before a checksum is computed could lead to a corrupt block being written to disk; the disk system will safely store the corrupted block. A recent, large-scale field study by Schroeder *et al.* emphatically show that bit-flips do occur [34].

Second, software is not perfect; large code bases are typically full of bugs [10], [45]. Some of those bugs may indeed corrupt data before it is written to disk, and again may thus survive despite checksum and RAID protections.

Lastly, Krioukov *et al.* also show that checksumming does not protect against complex failures such as torn writes, lost writes, and misdirected writes [19].

E. The File System Approach

Many high-end file systems often claim that they have support for corruption handling. However, their robustness is little known due to the proprietary nature of these systems. With open-source file systems, there is room for evaluation. For example, Prabhakaran *et al.* presented the details of corruption detections in several commodity file systems [30], including Linux ext3 file system [42], ReiserFS [31], IBM JFS [7], and Windows NTFS.

In many cases, they found that these file systems are able to detect metadata corruption in the absence of checksums. Their approach is to store implicit redundant information to cross-check metadata consistency. For example, file systems such as ReiserFS [9] and XFS [41] store page-level information in each internal page of a B-Tree. Thus, a corrupt pointer that does not connect two pages in adjacent levels can be detected checking the page-level information. These file systems show that some redundant information can be useful for online cross-checking without imposing significant overhead.

Although many corruptions are detected, Prabhakaran *et al.* also found that in some cases these file systems fail to check the integrity of their own metadata. They show that the undetected corruptions result in system crashes, the spreading of corruption, and unmountable file systems [30].

To remedy this problem, file system developers created offline tools to scan and repair file system metadata that was inconsistent. The classic repair tool is *fsck* [21]. Despite the presence of RAID and checksumming, fsck remains useful even today; high-end file systems also have their own offline checkers. Some new file systems have tried to make do without an offline checker, *e.g.*, SGI's XFS famously was said to have "no need to fsck, ever" [13], but soon introduced such a tool to handle corruptions that were observed in the field.

Unfortunately, building a robust checker is not straightforward. An analysis of the Linux ext2 checker by Gunawi *et al.* shows that some important repairs are missing, leaving some corruptions unattended, and some repairs are incorrect, making the file system more corrupt and sometimes unusable [15].

F. The DBMS Approach

In the DBMS world, there have been many reports of database corruptions [1], [2]. In many cases, the sources of the corruptions are hard to pinpoint, and hence are not reported. Nevertheless, the fact that error messages such as "Database page corruption on disk" appear in the error logs suggest that database systems read corrupt contents from the disk. But again, the research literature has not extensively addressed how running databases deal with such corruption.

The presence of offline tools to scan and repair database metadata is also less clear. Some tools exist [16], [17], [22], [26], [27], [28]. The existence of the tools certainly indicates that databases are corrupted in practice, despite the presence of RAID and checksums. However, due to the proprietary nature of these database systems and their on-disk formats, there is little published on the details of how these offline check and repair tools work.

Evaluations of open-source file systems have unearthed many weaknesses in the ways modern file systems deal with corruption [15], [30]. However, to the best of our knowledge, there has been no similar published study in the DBMS literature. However, as open-source database systems such as PostgreSQL [37] and MySQL [39] have become both popular and important, we believe a new opportunity has arisen to both evaluate the state of the art of database checking and potentially to improve it. The open nature of these systems make evaluation possible, and in this paper (Sections IV and V), we demonstrate how fault injection can be used to assess the resilience of MySQL (in particular) to various types of corruption.

III. METHODOLOGY

An integral part of ensuring the long-term availability of data is ensuring the reliability and availability of *pointers* and *format information*. Pointers are fundamental to the construction of nearly all data structures, while format information is critical for the correctness of reading the data and metadata. This observation is especially true for database management systems, which rely on pointers to access data correctly and efficiently, and on format information to determine how to parse both metadata and data. Unfortunately, as mentioned in the previous section, information stored on a disk can be corrupt. A robust DBMS should detect and repair such corruption of its metadata.

One difficulty with a pointer-corruption study is the potentially huge exploration space for corruption experiments. To deal with this problem, we utilize a fault injection technique called type-aware pointer corruption (TAC) [5]. TAC reduces the search space by systematically changing the value of only one pointer of each type in the DBMS, then exercising the DBMS and observing its behavior. We further narrow the large search space by corrupting the pointers to refer to each type of data structure, instead of to random values. For example, rather than corrupting a B-Tree pointer to point to a random page, we introduce types to the pages (*e.g.*, grand-child, sibling, parent page), and then change the pointer to point to different types of pages.

TAC simulates field-level corruption. As mentioned in Section II-A, different problems can lead to different types of corruption. For example, a misdirected write can corrupt everything on a page, not just a particular field. This type of page-level corruption can be simulated as well by slightly extending our fault-injection framework. So far, we have only considered field-level corruption as it allows for detailed analysis of the system's responses to different field-corruptions.

To exercise the DBMS as thoroughly as possible, another challenge is to coerce the DBMS down its different code paths to observe how each path handles corruption. This requires that we run workloads exercising all relevant code paths in combination with the induced faults. In this paper, we only focus on read workloads. Specifically, we run three kinds of queries: single selection queries (*e.g.*, WHERE field = X), range selection queries (*e.g.*, WHERE field BETWEEN X AND Y), and full table scans. By running different queries, we can analyze how the injected corruptions affect different workloads.

Section IV presents the results of our online pointer and format corruptions for MySQL. Specifically, we inject corruptions when the server is running and observe if it detects and handles the corruptions. Unfortunately, some corruptions are not detected online. Thus, we then inject the same corruptions

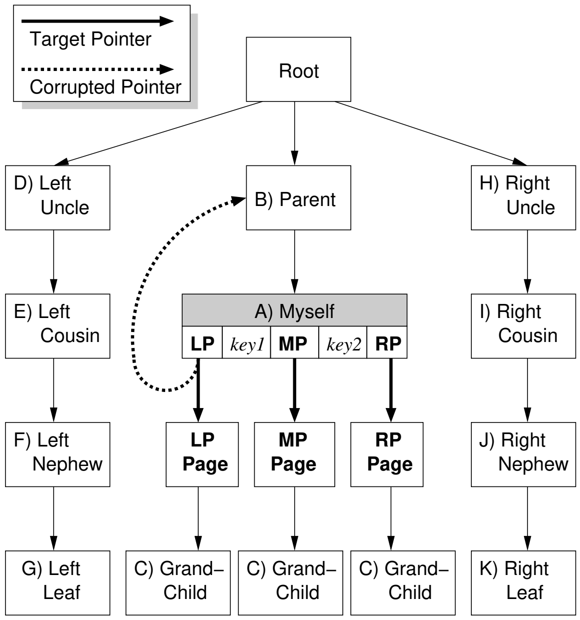

Fig. 1.

B-Tree pointer corruption. The graph above shows key-pages of an index B-Tree. Each box represents a key-page which contains a set of key-pointer pairs. A pointer is a page number. We corrupt three target pointers of a non-leaf page (page A): left-most (LP), middle (MP), and right-most (RP) pointers. For example, the LP pointer can be corrupted to point to the parent page (page B).

and analyze whether the MySQL offline checker is able to detect them (Section V). As we will see, even the offline checker fails to detect some of the corruptions, thus leaving the DBMS vulnerable to on-disk corruption.

All experiments except where specified are performed on the MyISAM Storage Engine of MySQL version 5.0.67 running on the Linux 2.6.12 operating system. We have not tested MySQL with other storage engines. In total we have injected 145 corruption scenarios. Due to the sheer volume of experimental data, it is difficult to present all results for the reader's inspection. We try to present the complete results of our analysis in tables (for those interested in all the data), and then provide qualitative summaries of the results that are presented within the tables.

IV. ONLINE CORRUPTION

Despite the presence of corruption, we expect a running DBMS to be highly reliable and available. More specifically, we expect a reliable DBMS to have a strong mechanism for detecting disk corruption such that corrupt metadata is not wrongly used by the DBMS. Moreover, to be highly available, a DBMS has to keep running and return as much valid data as possible to the users. To see how MySQL stands with respect to these issues, we pose three questions that relate to reliability, availability, and framework for corruption handling:

1) Can MySQL detect disk corruption properly?

	A	B	C	D	E	F	G	H	I	J	K	L	M
MP	×	×	≠	≠	≠	≠	≠	≠	≠	≠	≠	√	√
LP	×	×	≠	≠	≠	≠	≠	≠	≠	≠	≠	√	√
RP	×	×	≠	≠	≠	≠	≠	≠	≠	≠	≠	√	√

(a) Single selection query

	A	B	C	D	E	F	G	H	I	J	K	L	M
MP	×	×	≠	√	√	√	√	≠	≠	≠	≠	√	√
LP	×	×	≠	≠	≠	≠	≠	≠	≠	≠	≠	√	√
RP	×	×	≠	≠	≠	≠	≠	≠	≠	≠	≠	√	√

(b) Range selection query

TABLE I

Online detection of B-Tree pointer corruption. The tables above report the results of our B-Tree pointer corruption. The results depend on the query that is executed. The first and the second tables show the results of a single and a range selection query respectively. The left-most column shows the pointers that we corrupt (*i.e.*, MP, LP, and RP, as illustrated in Figure 1). The row-header represents the new pages (*i.e.*, page A to M) that the corrupted pointer is now pointing to. "√" marks that the corruption is detected; for example, when MP points to an out-of-bound page (M). "×" represents a server crash; for example, when a cycle is introduced when MP points to itself (page A). "≠" implies that the server returns the wrong results to the user; for example, when MP points to its grand-child (page C), records made inaccessible by this corruption are not returned to the user.

2) Can MySQL keep running and return valid data despite the presence of corruption?
3) Based on our results, what can we infer about the framework for corruption handling in MySQL?

To answer these questions, we first present the results of our fault-injection experiments on a running MySQL server (Section IV-A). Then, we answer the questions by presenting our qualitative observations on the results (Section IV-B). Finally, we conclude this section and present some preliminary results for PostgreSQL (Section IV-C).

A. Results

In this section, we present the results of our online pointer and format corruptions. For pointer corruption, we corrupt the B-Tree, record, and overflow pointers. For format corruption, we corrupt the format information stored in the index and data files. For each corruption case, we describe the MySQL data structures that we corrupt, our findings and observations. In all cases, we find that the presence of corruption would lead to server crashes, data loss, or even incorrect results.

1) B-Tree Pointer Corruption: The first class of pointer corruption that we inject is B-Tree pointer corruption. For each database table, MySQL manages three files: an index file (.MYI), a format file (.FRM), and a data file (.MYD). For each index defined on a table, MySQL stores a B-Tree in the index file in the form of key-pages (we also refer to a key-page as a page). A page is usually 1 KB. The index file has a header page (index file header) that has pointers to the root pages of all B-Trees in the index file. A key-page contains a header (page header), describing the key-page, and a set of key-value pairs where the value carries two pointers: a key-pointer (*i.e.*, page number) which points to a child page and a record-pointer which points to the corresponding record stored in the data file. In this experiment, we corrupt the key-pointer by making it point to another page and observe how MySQL handles this class of corruption while it is running.

Figure 1 illustrates a 5-level B-Tree. We corrupt three distinct pointers: the left-most (LP), middle (MP), and right-most (RP) pointers of a non-leaf page (page A). To exercise corruption scenarios, we corrupt these pointers. To reduce the corruption space, we identify eleven categories of pages (pages A to K as shown in Figure 1). For example, we corrupt the left-most pointer (LP) to point to: the parent (page B), left cousin (page E), left nephew (page F), and so on. To be able to detect the corruptions, two keys that wrap the middle pointer (key1 and key2) can be utilized.

Table I summarizes our results. In addition to corrupting the three pointers to point to page A to K, we also force them to point to pages belonging to the index file header (page L) and to out-of-bound pages (page M). To analyze how the corruptions affect different workloads, we also run two types of queries: single and range-selection queries. In total, we have injected 39 B-Tree pointer corruption scenarios. Unfortunately, MySQL does not detect and handle many of these corruptions online; MySQL returns wrong results to users, or the server crashes. Below, we further explain the results.

Detected error (√): Out of the 39 scenarios, MySQL detects only 6 or 10 of them depending if we execute a single or a range-selection query. Most of the corruptions detected are those where pointers point to an out-of-bound page (M) or to a page belonging to an index file header (L). The former is easily detected because reading an out-of-bound page will result in a low-level read error. The latter is detected because MySQL always checks the key-page header, specifically the length of used key-value pairs in the page (which should always be

	00	01	02	03	04	05	06	0D	Data	Out-of-bound
05	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√♠	√♣
06	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√♠	√♣
0D	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√♠	√♣
0B	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√♠	√♣
0C	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√•×	√♠	√♣

TABLE II

Online detection of overflow pointer corruption. The table above reports the results of our overflow pointer corruption. "×" marks that the server hangs and"√" represents that the corruption is detected. When an overflow corruption is detected, depending on the type of the corruption, MySQL reacts differently: sometimes it does not return any valid data and marks the corresponding table as crashed (♠), sometimes returns partial valid data and kills the executed query (♣), and sometimes returns all valid data and skips the corrupt record without notifying the corruption to the user (•).

greater than 4 bytes and less than 1 KB). Pages that belong to the index file header have different structures that always store "0xFE 0xFE" at the same byte offset. Thus, MySQL can detect that they are not valid key-pages.

Wrong results (≠): In many cases, MySQL blindly trusts the corrupt pointers. As a result, incorrect results are returned to users. Specifically, users could get empty records or the wrong number of records (since portions of the tree are silently lost). For example, this can happen when the middle pointer points to its grand-child page (*e.g.*, MP points to page C). All the keys in the grand-child page are valid with respect to key1 and key2. Thus, the corruption is not easily detectable and some portions of the B-Tree (other pages reachable from the MP page) are not reachable anymore.

Server crashes (×): Finally, MySQL does not anticipate a cycle; when the three target pointers are corrupted to point to the page where they are stored (page A) or to the parent of that page (page B), MySQL server does not detect the created loop. The server keeps calling the search routine on the same pages infinitely. This routine only stops when the result is found or not found. Since the MySQL server does not track the previous pages that have been traversed, this loop causes an infinite traversal that eventually causes a stack overflow. The server crashes, and a lost connection error occurs.

2) Record Pointer Corruption: In our next set of experiments, we inject record pointer corruption. Record pointers are stored in key-pages in the index file. A record pointer of a key-value pair points to the actual record that holds the corresponding key. We have injected numerous corruptions. Here, we briefly describe the interesting results.

First, we created a table with fixed-size records with an auto-incremented key and corrupt a record pointer of a key-value in the index file such that it points to another record in the table. Thus, the key stored in a corrupt key-value pair does not match with the key stored in the record that it points to. For example, we take a key-value pair with a key of 500 and corrupt it by making the record pointer points to a record with a key of 600.

We ran a single selection query on the key (SELECT * WHERE key = 500), and the server behaves correctly; the server returns an empty result. We suspect that MySQL verifies that the record pointed by the corrupt key-value pair does not have the same key.

We observed a different behavior when we ran a range selection query (*e.g.*, selecting records with keys between 450 to 550); MySQL only returns a subset of the records, specifically records with keys between 450 to 499. MySQL always trusts the key stored in the record; when the B-Tree traversal hits key 500, it finds that the key in the record is 600, which is larger than the end of the range query (550). Thus, the server stops traversing the B-Tree and only returns a subset of the records. This confirms that when a range selection query is executed, MySQL never checks the fact that the key in the record is different than the key in the key-value pair.

Another interesting result is when we corrupt the record pointer of a dynamic (variable-length) record. With dynamic records, the record pointer is a byte offset, which implies that it can point to any byte in the data file. In this case, MySQL always checks the record information (*e.g.*, record length) in the record header. In the case of a corrupt pointer, the record length is not as desired. MySQL returns an error code to users without giving any result. The error states that the table has been marked as crashed and should be repaired.

3) Overflow Pointer Corruption: Next, we inject pointer corruptions into the data file. With fixed-size records, the data file does not store any pointers because a record can be fetched given its record number. With the variable-length record format, a record cannot always span contiguous bytes. Thus, a record can be put in one or more frames. When a record is deleted, all the frames that it occupies are marked deleted. When a record is inserted, it can reuse unused frames. If the new record does not fit in a frame, multiple frames are allocated for the record. Thus, in each frame, MySQL stores a pointer (the overflow pointer) to the next frame and a signature header that describes the frame. Only frames with hexadecimal signatures 05, 06, 0B, 0C, and 0D have an overflow pointer. This overflow pointer cannot point to all types of frames; a valid pointer can only point to a frame with a signature between 07 and 0C; more details can be found elsewhere [25].

Table II shows the result of our overflow pointer corruption. We inject corruptions that make an overflow pointer invalid. For example, a starting frame of a small record (05) should

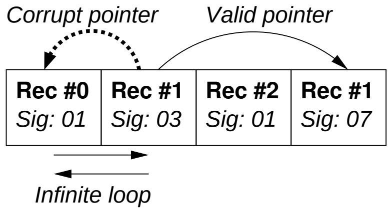

Fig. 2.
Server hangs. The figure illustrates a corruption scenario that causes MySQL server hangs.

not point to a deleted framed (00). Furthermore, because an overflow pointer is a byte offset (*i.e.*, it can point to any byte in the data file), we also force an overflow pointer points to data and to an out-of-bound offset.

We found that MySQL detects all overflow pointer errors ($\surd$). However, depending on the corruption, different results are returned and different error messages are thrown. For example, if an overflow pointer accidentally points to data, MySQL is very conservative by not returning any valid data (even though it has fetched some), but rather emits an error message stating that the table has been marked as crashed and should be repaired (♠). However, if an overflow pointer points to an out-of-bound offset, the server kills the executed query by returning only valid records that have been fetched so far (♣). Finally, if an overflow pointer points to an invalid frame, the server detects the error, skips this corrupt record, and continues scanning the next record ($\bullet$). The users then would get all valid records, even those that are located after the corrupted record. In this case, the server does not propagate the error message to users.

Moreover, a certain scenario of overflow pointer corruption makes the server enter an infinite loop ($\times$). Specifically, this happens on a full-scan query when an overflow pointer points to an invalid frame that is located *before* the frame that holds the overflow pointer. Figure 2 illustrates the bug. MySQL scans the variable-length frames one-by-one, looking for any starting frame. When there is an invalid overflow pointer (*e.g.*, the starting frame of record #1 points to the starting frame of record #0), the corruption is detected from the given signatures. But, rather than moving to the next valid frame (*i.e.*, record #2), MySQL scans the wrong next frame, (*i.e.*, record #1, which is the frame next to the invalid frame). In this case, the server gets stuck in an infinite loop.

Beyond the corruption scenarios shown in the matrix in Table II, we also performed a more specific fault injection: an overflow pointer is corrupted to point to a "valid" frame that actually belongs to another record. But, in MySQL, a frame does not hold information about its owner. Thus, it is not straightforward for MySQL to detect this corruption online. As a result, the corrupt record is presented to users like a valid record, except part of the data belongs to another record.

4) Index Format Corruption: We now corrupt important format information that is stored in the index file header, shown in the left column of Table III. This format information is crucial for parsing both metadata (*e.g.*, keys, key-pointers, etc.) and data (*e.g.*, columns). Due to space constraints, we do not provide the descriptions of the fields; their descriptions can be found elsewhere [25]. For each field, we corrupt the value to zero (0), a value less than the actual one ($<$), a value larger than the actual one ($>$), and the maximum possible value (Max). Format information is used differently depending on the query workload. Thus, we ran three types of query: full-scan, single selection, and range selection.

Table III depicts how various types of format corruption are handled in an inconsistent manner; some corruptions are detected ($\surd$), some are not. When a corruption is not detected, MySQL sometimes returns incorrect results to the user ($\neq$), sometimes returns valid results (.), leaving the corruption unnoticeable, and sometimes crashes ($\times$) in some unanticipated scenarios.

5) Record Format Corruption: In our final online experiment, we corrupt dynamic-record length information stored in the data file. MySQL is able to detect the discrepancy between the length of a record and the total length of its frames. MySQL tracks the cumulative length of the frames that have been fetched with respect to a record. If the cumulative length is larger than the record length, MySQL stops the query and returns only valid records that have been fetched so far. However, if the cumulative length is less than the record length, the server emits a hard error message saying that the table has been marked as crashed and should be repaired.

B. Observations

We now answer the questions we posed earlier in the paper. In short, our results have shown that MySQL does not detect all kinds of corruption that can arise, the MySQL server is not highly available in the midst of corruptions, and finally MySQL does not have a consistent framework for corruption handling. Below, we describe these observations in more detail.

1) Incomplete Detection: We find that MySQL ignores many corruptions, which leads to incorrect results being returned, crashes, and data loss. After further analysis, we find two reasons for these problems: in some cases MySQL ignores detectable corruptions and in some other cases MySQL does not have the ability to detect certain corruptions.

Ignored detectable corruptions: There are cases where corruption can be detected from implicit redundant information stored in MySQL data structures. Thus, with some additional work, some corruptions are actually detectable. However, detectability does not always lead to detection as we see in these three examples:

First, in B-Tree pointer corruption (Section IV-A.1), when a pointer is corrupt such that it points to a page not reachable from the parent page (*e.g.*, MP points to page D through K in Figure 1), MySQL could detect this by checking the keys with respect to key1 and key2. However, since MySQL does

	Full scan				Single selection				Range selection			
Format info	0	$<$	$>$	Max	0	$<$	$>$	Max	0	$<$	$>$	Max
State header												
header length	$\surd$	$\surd$	.	$\surd$	$\surd$	$\surd$	.	$\surd$	$\surd$	$\surd$	.	$\surd$
keys	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
number of records	$\neq$	.	.	.	$\neq$	.	.	.	$\neq$	.	.	.
data file length	$\neq$	$\neq$	$\surd$	$\surd$	$\neq$	$\neq$	.	.	$\neq$	$\neq$	$\surd$	.
Base header												
record length	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\times$	$\times$
pack rec. length	$\surd$	$\surd$	$\times$	$\times$	$\neq$	$\neq$	$\neq$	$\times$	$\neq$	$\neq$	$\neq$	$\times$
rec ref. length	$\surd$	.	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$
key ref. length	.	.	.	.	$\neq$	$\surd$	$\neq$	$\surd$	$\times$	$\surd$	$\times$	$\surd$
max key blk len	.	.	.	.	$\times$	.	.	.	$\times$	.	.	.
fields	.	.	$\surd$	$\surd$	.	.	$\surd$	$\surd$	.	.	$\surd$	$\surd$
Key def.												
key segments	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
block length	.	.	.	.	$\times$	.	$\surd$	$\surd$	$\times$	.	$\surd$	$\surd$
Key segment												
length	.	.	.	.	$\neq$	$\surd$	.	$\times$	$\surd$	$\surd$	$\surd$	$\times$
Record info												
length	.	.	$\surd$	$\surd$	.	.	$\surd$	$\surd$	.	.	$\surd$	$\surd$

TABLE III

Online detection of format corruption. The table above reports MySQL corruption handling of different format corruptions. We corrupt a format value to zero (0), a value less than its actual value ($<$), a value larger than its actual value ($>$), and the maximum possible value (Max). This format information is stored in the index file header. "$\times$" represents a server crash, "$\neq$" implies that the server returns wrong results to the users, "." marks that the corruption is silently ignored, and "$\surd$" marks that the corruption is detected.

not perform such a check, incorrect results are returned ("$\neq$" in Table I).

Second, a record pointer corruption (Section IV-A.2) should be easily detectable; the MySQL server could compare the key stored in the index with the one stored in the record. But, rather than utilizing this redundant information, MySQL always trusts the keys stored in the records. As a result, incorrect results are returned.

Third, in the index format corruption (Section IV-A.4), when the data file length specified in the state header is corrupted to zero, MySQL returns no result to the user without any error message, blindly believing that the data file is empty although the number of records stored in the state header can give the correct information. A similar situation occurs when the data file length is corrupted to half of the actual value; MySQL only scans half of the data file. Another example is when the server crashes because the record length stored in the base header is corrupted to a maximum value. These corruptions actually can be caught simply by verifying the same information stored in the format file.

Undetectable corruptions: We find that several corruptions are hard to detect because MySQL does not store enough implicit redundant information in its data structures. We find many instances of this issue:

First, in the B-Tree pointer corruption (Section IV-A.1), it is hard to verify that a pointer properly connects two pages in adjacent levels because a page does not store its page level. For example, if a pointer is corrupt such that it points to one of its grand-children (*e.g.*, MP points to its C in Figure 1), MySQL cannot detect this easily.

Second, it is hard to detect an invalid root pointer because the index file header does not store the height of the B-Tree and the root page does not store its page level. Thus, a root pointer that points to a non-root page is considered valid, leading to a silent data loss (*i.e.*, some pages connected from the original root page are not reachable anymore). If the index file header stores the height of the B-Tree and each key-page has page-level information, their values can be cross-checked.

Third, it is difficult to catch a page in a B-Tree that points to another page belonging to another B-Tree because a page does not store information about to which B-Tree it belongs to. A table can have more than one index thus more than one B-Tree can be saved in the same index file. A page in a B-Tree should not be allowed to point to a page belonging to another B-Tree. However, since the page does not specify owner information, such a corruption scenario is not detected. As a result, users get incorrect results or the server kills the executed query with an error message.

Fourth, in the overflow pointer corruption (Section IV-A.3), it is also hard to catch a frame in a record that points to another frame belonging to another record because a frame does not hold information about its owner. Thus, when a corrupt overflow pointer points to a "valid" frame that actually belongs to another record, MySQL cannot easily detect this corruption online. As a result, the corrupt record is presented to users like a valid record, except part of the data belongs to another record.

Fifth, in the index format corruption (Section IV-A.4), it is challenging to verify true leaf and non-leaf pages. The page header has a one bit field that specifies whether the page is a leaf page (bit = 0) or a non-leaf page (bit = 1). When we corrupt the bit, thus making a non-leaf page a leaf page and vice-versa, the server sometimes hits an infinite loop, sometimes returns an empty result to users, and sometimes detects incorrect keys due to incorrect parsing. Detecting this corruption is challenging if not impossible. If only redundant information such as page level were stored in the page header, such detection would be straightforward.

In summary, MySQL should peruse available information in its data structures to cross-check its metadata consistency to the greatest extent possible. Furthermore, our findings also show that adding extra information might be useful for corruption detection or even recovery. The file system story in Section II shows that adding implicit redundancy can be done efficiently.

2) Reduced Availability: A system crash reduces availability. Thus, failure should be avoided in most systems. Unfortunately, in our experiments, we have shown that MySQL crashes in many cases of corruption.

Reduced availability also happens when MySQL fails to return valid data to users. When a minimal corruption occurs we might wish MySQL give us as many valid records as possible. For example, if there is only one corrupt record (*e.g.*, due to a corrupt overflow pointer), we might wish valid records were still accessible. However, that is not always the case in MySQL. In the overflow pointer corruption (Section IV-A.3), when an overflow pointer accidentally points to data, MySQL does not return any valid records ("♠" in Table II). When an overflow pointer points to an out-of-bound offset, the server only returns valid records that have been fetched so far ("♣" in Table II). Hence, due to this inconsistent handling, a small corruption in MySQL can make a large number of records inaccessible.

To improve availability, corruption should be detected and isolated. Detection is crucial; our findings have shown that corrupt metadata can lead to crashes. Worse, it might lead to the propagation of the corruption. This result emphasizes that catching corrupt metadata is a crucial factor in increasing availability. Furthermore, after corrupt metadata is detected, the corruption and also the operation on the metadata should be isolated; more specifically, the operation should be able to continue processing other valid metadata.

3) No Framework for Corruption Handling: Finally, we believe that MySQL might not have a framework for corruption handling. This conclusion is suggested by its *inconsistent* reactions in handling corruption. We define inconsistent handling as the case where similar failure scenarios are handled differently. From our results, we find five cases of inconsistent handling in each class of corruption we injected:

First, in the B-Tree pointer corruption (Section IV-A.1), when we corrupt the middle pointer to point to any page reachable from the left-uncle, MySQL detects the corruptions ("$\surd$" in Table I-b when MP points to D, E, F, or G). However, when the middle pointer is corrupted to point to any page reachable from the right-uncle, MySQL does not detect the corruptions and delivers the wrong results to the users ("$\neq$" in Table I-b when MP points to H, I, J, or K). These two cases are similar but handled differently. It turns out that, for the first case, MySQL "coincidentally" detects the corruption; the error message actually comes from the detection of an out-of-bound key-pointer due to the abnormal behavior of the search routine after it follows the corrupted middle pointer.

Second, in the record pointer corruption (Section IV-A.2), MySQL reacts to a corrupt record pointer differently depending on the executed query. In the case of a single selection query, users get correct (empty) result; in the case of a range selection query, users get wrong (partial) results without any errors thrown; in the case of a dynamic length record, a hard error is thrown and no result is returned (even the valid ones). This shows that MySQL corruption handling is sometimes soft and sometimes hard.

Third, in the overflow pointer corruption (Section IV-A.3), depending on the corrupt value, MySQL gives widely different reactions ranging from marking the table as crashed (♠ in Table II) to killing the executed query (♣), and sometimes silently returning without any error-code (•).

Fourth, in the index format corruption (Section IV-A.4), Table III clearly depicts how format corruptions are handled in an inconsistent manner, depending on the workload and on the corrupt value. For example, when the key reference length in the base header is corrupted, sometimes the corruption is detected ($\surd$), but sometimes it is not. When the corruption is not detected, MySQL sometimes returns incorrect results to the user ($\neq$) and at times crashes ($\times$).

Fifth, in the record format corruption (Section IV-A.5), when a query hits a corrupt dynamic-record length, depending on the corrupt value, MySQL sometimes stops the query and returns only valid records that have been fetched so far, but sometimes emits a hard error message saying that the table has been marked as crashed.

In summary, we believe that MySQL does not have a proper framework for corruption handling. When inconsistent handling is observed, usually it implies that the corruption handling code is diffused throughout the code base [14], [30]. Such diffusion usually results in unpredictable and often undesirable fault-handling strategies, which might turn into frustration for human debugging [30].

C. Summary

We have found that MySQL does not detect and handle corruptions well. We believe that the observations we have made are not specific to MySQL; in addition to MySQL, we have applied our fault injection method to PostgreSQL version 8.3, another open source DBMS. Our initial experiment shows that PostgreSQL has similar problems as MySQL. For example, in PostgreSQL, pages in the index file store left and right sibling pointers. When the right sibling pointer of a page is corrupted so that it points to one of its left sibling pages, the SELECT query on the table based on index scan makes the server to hang as it hits an infinite loop. Beyond the scenario described above, we have also injected 24 more corruptions to PostgreSQL and found that 12 of them highlight the problems observed in this section.

V. Offline Corruption

Online detection of hundreds of possible corruption scenarios is often not feasible. One primary reason is because full cross-checks must be performed to detect all scenarios. Thus, a DBMS offline checker should be the last tool that catches all corruptions in the database. When a corruption has been detected by an offline checker, a repair utility can be run, thus restoring the tables to a consistent condition. However, if the offline checker misses some corruption scenarios, one would not run the repair utility and corrupt data can leak into the running system, which may cause more corruptions.

In this section, we analyze the robustness of the MySQL offline checker, myisamchk, in dealing with the same corruption scenarios we have injected in the online case. This checker runs in two modes: check and repair. In this first mode, myisamchk attempts to find all corruptions in the database, while in the second, it tries to rebuild the tables and index files. Thus, we pose two questions:

1) Can myisamchk find all corruptions in the database?
2) Can myisamchk correctly repair the database?

To answer these questions, we first present the results of our fault-injection experiments on myisamchk (Section V-A) and then summarize our observations (Section V-B).

A. *Results*

1) Check Mode: We have injected the same B-Tree and overflow pointer corruptions described in Sections IV-A.1 and IV-A.3. All cases except one are detected by myisamchk; myisamchk crashes when a left-most key points to the same page where the key is stored. More detailed observation shows that in many cases of detected corruptions, the error messages thrown do not precisely describe the injected corruptions. This suggests that the checks performed do not capture the actual corruptions. Hence, perhaps it is not surprising to discover a corner-case bug.

The most interesting findings of our offline experiments arose when we inject format corruptions (as in Section IV-A.4). As depicted in Table IV, the offline checker blindly trusts some format information. As a result, the checker crashes ($\times$) when such information is not as expected. This system crash is unacceptable because a checker should not trust any value it retrieves from the disk; its basic purpose is to find corrupt metadata. Other than this, Table IV also shows that many corruption scenarios are left undetected.

Format info	0	$<$	$>$	Max
State header				
header length	$\surd$	$\surd$	.	$\surd$
keys	$\surd$	$\surd$	$\surd$	$\surd$
number of records	$\surd$	$\surd$	$\surd$	$\surd$
data file length	$\surd$	$\surd$	$\surd$	$\surd$
Base header				
reclength	$\surd$	$\surd$	.	.
pack reclength	.	.	$\surd$	$\times$
rec reflength	$\surd$	$\surd$	$\surd$	$\surd$
key reflength	$\surd$	$\surd$	$\surd$	$\surd$
max key blk len	$\times$	.	.	.
fields	.	.	$\surd$	$\surd$
Key def				
keysegs	$\surd$	$\surd$	$\surd$	$\surd$
block length	$\surd$	.	$\surd$	$\times$
Key segment				
length	$\surd$	$\surd$	$\surd$	$\times$
Record info				
length	.	.	$\surd$	$\surd$

TABLE IV

Offline detection of format corruption. The table reports myisamchk corruption handling of different format corruptions. "$\times$", ".", and "$\surd$" represent server crash, ignored corruption, and detection respectively.

2) Repair Mode: When we inject format corruptions, we also find that the repair performed by myisamchk could be problematic. For example, when the record length ("reclength") specified in the base header of the index file is corrupted, myisamchk throws an error message saying that it found wrong records in the data file and suggests a repair. When the repair is finished, however, all records in the table are discarded and the record length still remains corrupted.

After studying the code, we determined the reason. In MySQL the record length is essential to parsing records from the data file. However, myisamchk assumes that this field is always correct. Thus, once the field gets corrupted, it will never locate the corruption. Then during the repair, myisamchk will not be able to read any record from the data file by using the wrong record length, thus leaving no record after the repair.

In fact, this erroneous repair could be avoided by a simple fix, which makes use of the redundant information inside the data file itself and from the format file.

B. *Observations*

In summary, our results show that the offline checker myisamchk is far from robust; it does not catch all corruptions and it does not always repair the database correctly. Our observations point to the same issues faced by the running MySQL (Section IV-B). Mainly, some detectable corruptions are ignored and some corruptions are not detectable due to the lack of redundant information. As a result, the checker itself can crash and even worse an erroneous repair could happen.

The fact that myisamchk does not perform a complete set of checks is not surprising given the minimal implementation

#	Checks Performed
4	**Checking data file:** Check validity of deleted block links, deleted frames, overflow pointers, size of deleted blocks
9	**Checking keys:** Check delete links (range-check and alignment), compare key-value pairs (range-check and alignment), check record-pointer, page length, auto-increment key.
2	**Checking file sizes:** check length of index and data file
15	**Total**

TABLE V

Checks performed by myisamchk. The table summarizes the 15 checks performed by the MySQL offline checker.

of the checker (under 2000 lines of code). A more detailed study shows that the checker only performs 15 checks, shown in Table V. Many important checks are either omitted or overlooked. Redundant information in the format file (*e.g.*, column and key definitions, file size, record count, etc.) is not used to verify the consistency of the index file. B-Tree checks are also not comprehensive. For example, key-value pairs comparison is done only on per-page level; key-value ordering across siblings and parent/child is not checked. Thus, there is room for improvement in building a more robust MySQL checker.

VI. Conclusion

In the world of storage systems, it would be ideal if RAID storage guaranteed that data was not corrupt. Unfortunately, no such guarantee is possible (though techniques can make the odds of perceived corruption lower). Thus, a DBMS must, at the highest level, be responsible for the correctness of its data. This notion is particularly true of the DBMS metadata, which no client of the DBMS can even access; if the DBMS does not safeguard its own metadata, no other components can.

In this paper, we have begun the exploration of the data corruption problem on database management systems. We have shown that the MySQL and PostgreSQL DBMSs do not tolerate such faults particularly well, and that MySQL offline checker catches some but not all corruptions, thus leaving the system susceptible to corruption if it arises.

However, we believe our work is only the first step towards the "hardening" of database management systems to the problems of corruption. Many problems remain, including:

Online checking: A running DBMS should likely perform internal integrity checks while it runs to protect against other forms of corruption, including those from bad memory [23] as well as from disk.

There is a large body of work regarding techniques for detecting and recovering from data corruption [35], including the use of in-memory redundancy with checksums and replicas, or the use of fault-tolerant data structures [3], where a single pointer fault cannot lose a large amount of data, unlike what we have seen in Section IV-A.1. Although all these techniques are not new, it would be interesting to find out why they are not deployed in practice. One reason might be a lack of study in quantifying how much performance overhead is imposed and how much reliability is gained when a certain redundancy or protection is added. This would be an important issue to look into further.

Aside from existing techniques, we believe a proper framework is needed in deploying the techniques. One possible solution is having a centralized framework that focuses on corruption handling [14]. Without a centralized framework, handling hundreds of corruption scenarios is proven to be difficult, diffused, and inconsistent.

Robust offline checkers: The existence of repair tools again indicates that we need them in practice. However, as we have observed, the repair process of checkers (both for DBMSs and file systems) is typically *ad hoc*. Thus, the quest in building more robust checkers has begun recently. For example, Gunawi *et al.* utilize a declarative approach to write hundreds of checks and repairs in a clear and compact manner [15]. Others have used more formal frameworks as the foundation for corruption repair. For example, Khurshid *et al.* suggest the use of symbolic execution [18] and Wang *et al.* define the problem of corruption-repair as a global optimization problem by using structural Hamming and edit distance [43].

Thus, further work is clearly required. Only through a combined offline and online approach will a high-performance, robust, and truly corruption-robust DBMS be realized.

References

[1] `http://bugs.mysql.com/search.php?search_for=corruption&status=All&cmd=display`.

[2] `http://search.postgresql.org/search?m=1&q=corruption`.

[3] Yonatan Aumann and Michael A. Bender. Fault Tolerant Data Structures. In *The 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, Burlington, Vermont, October 1996.

[4] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[5] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Systematically Benchmarking the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[6] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[7] Steve Best. JFS Overview. `www.ibm.com/developerworks/library/l-jfs.html`, 2000.

[8] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.

[9] Florian Buchholz. The structure of the Reiser file system. `http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php`, January 2006.

[10] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[11] Michael H. Darden. Data Integrity: The Dell|EMC Distinction. `http://www.dell.com`, May 2002.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[13] Rob Funk. fsck / xfs. `http://lwn.net/Articles/226851/`.

[14] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.

[15] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[16] IBM. `http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.pd.doc/pd/c0020760.htm`.

[17] IBM. `http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/core/c0009137.htm`.

[18] Sarfraz Khurshid, Ivan Garca, and Yuk Lai Suen. Repairing Structurally Complex Data. In *12th International SPIN Workshop on Model Checking of Software (SPIN '05)*, San Francisco, CA, August 2005.

[19] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[20] Xin Li, Michael C. Huang, , and Kai Shen. An Empirical Study of Memory Hardware Errors in A Server Farm. In *The 3rd Workshop on Hot Topics in System Dependability (HotDep '07)*, Edinburgh, UK, June 2007.

[21] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[22] Microsoft. `http://technet.microsoft.com/en-us/library/ms176064.aspx`.

[23] Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, and Alberto Munoz. Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models. In *9th ACM SIGOPS European Workshop 'Beyond the PC: New Challenges for the Operating System'*, Kolding, Denmark, September 2000.

[24] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[25] MySQL Team. MySQL Internals: MyISAM. `http://forge.mysql.com/wiki/MySQL_Internals_MyISAM`.

[26] Oracle. `http://www.ordba.net/Tutorials/OracleUtilities~DBVERIFY.htm`.

[27] Oracle. `http://www.oracleutilities.com/Packages/dbms_repair.html`.

[28] Oracle. `http://www.oracle-base.com/articles/8i/DetectAndCorrectCorruption.php`.

[29] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[30] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[31] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[32] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[33] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, pages 1–16, San Jose, California, February 2007.

[34] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[35] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, FairFax County, Virginia, November 2005.

[36] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[37] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1986.

[38] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[39] Sun Microsystems. MySQL White Papers, 2008.

[40] Rajesh Sundaram. The Private Lives of Disk Drives. `http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html`, February 2006.

[41] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[42] Stephen C. Tweedie. EXT3, Journaling File System. `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, July 2000.

[43] Hongyi Wang, Bingsheng He, Vijayan Prabhakaran, and Lidong Zhou. Crystal: The Power of Structure Against Corruptions. In *The 5th Workshop on Hot Topics in System Dependability (HotDep '09)*, Lisbon, Portugal, June 2009.

[44] Glenn Weinberg. The Solaris Dynamic File System. `http://members.visi.net/~thedave/sun/DynFS.pdf`, 2004.

[45] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank Swaminathan Sundararaman and Abhishek Rajimwale for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from NetApp, Inc and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.