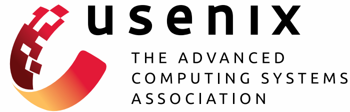

Torturing Databases for Fun and Profit

Mai Zheng, *The Ohio State University;* Joseph Tucek, *HP Labs;* Dachuan Huang and Feng Qin, *The Ohio State University;* Mark Lillibridge, Elizabeth S. Yang, and Bill W. Zhao, *HP Labs;* Shashank Singh, *The Ohio State University*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the
11th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

Torturing Databases for Fun and Profit

Mai Zheng[†] Joseph Tucek[‡] Dachuan Huang[†] Feng Qin [†] Mark Lillibridge[‡]
Elizabeth S. Yang[‡] Bill W. Zhao[‡] Shashank Singh[†]
[†] *The Ohio State University* [‡] *HP Labs*

Abstract

Programmers use databases when they want a high level of reliability. Specifically, they want the sophisticated ACID (atomicity, consistency, isolation, and durability) protection modern databases provide. However, the ACID properties are far from trivial to provide, particularly when high performance must be achieved. This leads to complex and error-prone code—even at a low defect rate of one bug per thousand lines, the millions of lines of code in a commercial OLTP database can harbor thousands of bugs.

Here we propose a method to expose and diagnose violations of the ACID properties. We focus on an ostensibly easy case: power faults. Our framework includes workloads to exercise the ACID guarantees, a record/replay subsystem to allow the controlled injection of simulated power faults, a ranking algorithm to prioritize where to fault based on our experience, and a multi-layer tracer to diagnose root causes. Using our framework, we study 8 widely-used databases, ranging from open-source key-value stores to high-end commercial OLTP servers. Surprisingly, all 8 databases exhibit erroneous behavior. For the open-source databases, we are able to diagnose the root causes using our tracer, and for the proprietary commercial databases we can reproducibly induce data loss.

1 Introduction

Storage system failures are extremely damaging—if your browser crashes you sigh, but when your family photos disappear you cry. Few people use process-pairs or n-versioning, but many use RAID.

Among storage systems, databases provide the strongest reliability guarantees. The atomicity, consistency, isolation, and durability (ACID) properties databases provide make it easy for application developers to create highly reliable applications. However, these properties come at a cost in complexity. Even the relatively simple SQLite database has more than 91 million lines of test code (including the repetition of parameterized tests with different parameters), which is over a thousand times the size of the core library itself [11]. Checking for the ACID properties under failure is notoriously hard since a failure scenario may not be conveniently reproducible.

In this paper, we propose a method to expose and diagnose ACID violations by databases under clean power faults. Unexpected loss of power is a particularly interesting fault, since it happens in daily life [31] and is a threat even for sophisticated data centers [24, 25, 27, 28, 29, 30, 41, 42] and well-prepared important events [18]. Further, unlike the crash model in previous studies (e.g., RIO [16] and EXPLODE [44]) where memory-page corruption can propagate to disk, and unlike the unclean power losses some poorly behaved devices suffer [46], a clean loss of power causes the termination of the I/O operation stream, which is the most idealized failure scenario and is expected to be tolerated by well-written storage software.

We develop four workloads for evaluating databases under this easy power fault model. Each workload has self-checking logic allowing the ACID properties to be checked for in a post-fault database image. Unlike random testing, our specifically designed workloads stress all four properties, and allow the easy identification of incorrect database states.

Given our workloads, we built a framework to efficiently test the behavior of databases under fault. Using a modified iSCSI driver we record a high-fidelity block I/O trace. Then, each time we want to simulate a fault during that run, we take the collected trace and apply the fault model to it, generating a new synthetic trace. We create a new disk image representative of what the disk state may be after a real power fault by replaying the synthetic trace against the original image. After restarting the database on the new image, we run the consistency checker for the

workload and database under test.

This record-and-replay feature allows us to systematically inject faults at every possible point during a workload. However, not all fault points are equally likely to produce failures. User-space applications including databases often have assumptions about what the OS, file system, and block device can do; violations of these assumptions typically induce incorrect behavior. By studying the errors observed in our early experiments, we identify five low-level I/O patterns that are especially vulnerable. Based on these patterns, we create a ranking algorithm that prioritizes the points where injecting power faults is most likely to cause errors; this prioritization can find violations about 20 times faster while achieving the same coverage.

Simply triggering errors is not enough. Given the huge code base and the complexity of databases, diagnosing the root cause of an error could be even more challenging. To help in diagnosing and fixing the discovered bugs, we collect detailed traces during the working and recording phases, including function calls, system calls, SCSI commands, accessed files, and accessed blocks. These multi-layer traces, which span everything from the block-level accesses to the workloads' high-level behavior, facilitate much better diagnosis of root causes.

Using our framework, we evaluate 8 common databases, ranging from simple open-source key-value stores such as Tokyo Cabinet and SQLite up to commercial OLTP databases. Because the file system could be a factor in the failure behavior, we test the databases on multiple file systems (ext3, XFS, and NTFS) as applicable. We test each combination with an extensive selection—exhaustively in some cases—of power-fault points. We can do this because our framework does not require the time-consuming process of running the entire workload for every fault, allowing us to emulate thousands of power faults per hour.

To our surprise, **all 8 databases exhibit erroneous behavior**, with 7 of the 8 clearly violating the ACID properties. In some cases, only a few records are corrupted, while in others the entire table is lost. Although advanced recovery techniques (often requiring intimate knowledge of the database and poorly documented options) may reduce the problem, not a single database we test can be trusted to keep all of the data it promises to store. By using the detailed multi-layer traces, we are able to pinpoint the root causes of the errors for those systems we have the source code to; we are confident that the architects of the commercial systems could quickly correct their defects given similar information.

In summary, our contributions are:

- **Carefully designed workloads and checkers to test the ACID properties of databases.** Despite extensive test suites, existing databases still contain bugs. It is a non-trivial task to verify if a database run is correct after fault injection. Our 4 workloads are carefully designed to stress different aspects of a database, and further are designed for easy validation of correctness.

- **A cross-platform method for exposing reliability issues in storage systems under power fault.** By intercepting in the iSCSI layer, our framework can test databases on different operating systems. By recording SCSI commands (which are what disks actually see), we can inject faults with high fidelity. Further, SCSI tracing allows systemic fault injection and ease of repeating any error that is found. Although we focus here on databases, this method is applicable to any software running on top of a block device.

- **A pattern-based ranking algorithm to identify the points most likely to cause problems when power faulted in database operations.** We identify 5 low-level patterns that indicate the most vulnerable points in database operations from a power-fault perspective. Further analysis of the root causes verifies that these patterns are closely related to incorrect assumptions on the part of database implementers. Using these patterns to prioritize, we can accelerate testing by 20 times compared to exhaustive testing while achieving nearly the same coverage.

- **A multi-layer tracing system for diagnosing root causes of ACID violations under fault.** We combine the high-level semantics of databases with the low-level I/O traffic by tracing the function calls, system calls, SCSI commands, files, and I/O blocks. This correlated multi-layer trace allows quick diagnosis of complicated root causes.

- **Experimental results against 8 popular databases.** We apply our framework to a wide range of databases, including 4 open-source and 4 commercial systems. All 8 databases exhibit some sort of erroneous behavior. With the help of the multi-layer tracer, we are able to quickly pinpoint the root causes for the open-source ones.

2 Design Overview

2.1 Fault Model

We study the reliability of databases under a simple fault model: clean power fault. Specifically, we model loss of power as the potential loss of any block-level operations

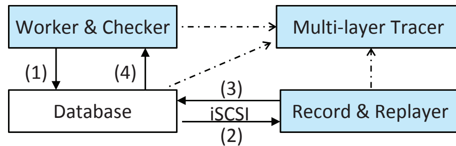

Figure 1: Overall Workflow

that have not been acknowledged as committed to persistent storage, with the proviso that if we drop an operation, we must drop any operations that explicitly depend on it as well. Although our system can induce the more complex faults of other fault models [16, 44, 46], which consider more complex data corruption and inconsistent behavior, we focus here on the simplest case. Such corruption and reordering/dropping of operations seem unreasonable to expect databases to tolerate, given that they so badly violate the API contract of the lower-level storage system.

The specifics of our fault model are as follows: a fault can occur at any point, the size of data written to media is always an integer block multiple (512 bytes or 4 KB), the device keeps all promises of durable commit of operations (e.g., completed `syncs` are honored as are write barriers), blocks are written (or not) without corruption, and interrupted or dropped operations have no effect. We believe this is the most idealized scenario under power failure, and we expect databases to honor their ACID guarantees under such faults.

2.2 Overall Workflow

Our testing framework injects simulated power faults by intercepting iSCSI [4] commands. iSCSI is a standard allowing one machine (the initiator) to access the block storage of another machine (the target) through the network. To everything above the block driver on the initiator, iSCSI is completely transparent, making it an ideal place to intercept disk activity. Further, the iSCSI target daemon can be implemented in user space [6], which greatly increases the flexibility of fault injection compared to a more complex kernel-level failure emulation layer. Finally, iSCSI interception allows a single fault injection implementation to test databases running on any operating system having an iSCSI initiator (aka all modern OSes).

Figure 1 shows an overview of our framework. There are three main components: the worker and checker, a record and replayer, and a multi-layer tracer. The overall workflow goes as follows: First, the worker applies a workload to stress a database starting from a known disk image. Rather than simply randomly writing to the database, the workloads are carefully designed so that the checker can verify the ACID properties of the post-fault image. Second, the record and replayer monitors the disk activities via the iSCSI layer. All blocks in the data transfer commands are recorded. Third, with the recorded block trace, the replayer simulates a power fault by partially replaying the block operations based on fault injection policies against a copy of the starting image, creating a post-fault disk image. Fourth, the checker opens the database on the post-fault image and verifies the ACID properties of the recovered database. During each of the above steps, the multi-layer tracer traces database function calls, system calls, SCSI commands, accessed files, and accessed blocks to provide unified information to diagnose the root cause of any ACID violations.

3 Worker and Checker

We developed four workloads with different complexities to check if a database provides ACID properties even when under fault. In particular, we check (1) atomicity, which means a transaction is committed "all or nothing"; (2) consistency, which means the database and application invariants always hold between transactions; (3) isolation, which means the intermediate states of a transaction are not visible outside of that transaction; and (4) durability, which means a committed transaction remains so, even after a power failure or system crash.

Each workload stresses one or more aspects of the databases including large transactions, concurrency handling, and multi-row consistency. Each workload has self-checking properties (e.g., known values and orders for writes) that its associated checker uses to check the ACID properties. For simplicity, we present pseudo code for a key-value store; the equivalent SQL code is straightforward. We further log timestamps of critical operations to a separate store to aid in checking.

Workload 1: A single thread performs one transaction that creates `txn_size` (a tunable parameter) rows:

```
Begin Transaction
  for i = 1 to txn_size do
    key   = "k-" + str(i)
    value = "v-" + str(i)
    put(key, value)
  end
  before_commit = get_timestamp()
Commit Transaction
after_commit    = get_timestamp()
```

We use this workload to see whether large transactions (e.g., larger than one block) trigger errors. The two timestamps `before_commit` and `after_commit` record the time boundaries of the commit. The checker for this workload is straightforward: if the fault was after the commit (i.e., later than `after_commit`), check that all

`txn_size` rows are present; otherwise, check that none of the rows are present. If only some rows are present, this is an atomicity violation. Being a single-threaded workload, isolation is not a concern. For consistency checking, we validate that a range scan gives the same result as explicitly requesting each key individually in point queries. Moreover, each retrievable key-value pair should be matching; that is, key N should always be associated with value N.

We report a durability error if the fault was clearly after the commit and the rows are absent. A corner case here is if the fault time lies between `before_commit` and `after_commit`. The ACID guarantees are silent as to what durability properties an in-flight commit has (the other properties are clear) so we do not report durability errors in such cases. We do find that some databases may change back and forth between having and not having a transaction as the point we fault advances during a commit.

Workload 2: This is a multi-threaded version of workload 1, with each thread including its thread ID in the key. Since we do not have concurrent transactions operating on overlapping rows, we do not expect isolation errors. We do, however, expect the concurrency handling of the database to be stressed. For example, one thread may call `msync` and force state from another thread to disk unexpectedly.

Workload 3: This workload tests single-threaded multi-row consistency by simulating concurrent, non-overlapping banking transactions. It performs `txn_num` transactions sequentially, each of which moves money among a different set of `txn_size` accounts. Each account starts with some money and half of the money in each even numbered account is moved to the next higher numbered account (i.e., half of k-t-$2i$'s money is moved to k-t-$(2i+1)$ where t is the transaction ID):

```
for t = 1 to txn_num do
  key_prefix = "k-" + str(t) + "-"
  Begin Transaction
    for i in 1 to txn_size/2 do
      k1 = key_prefix + str(2*i)
      k2 = key_prefix + str(2*i+1)
      tmp1 = get(k1)
      put(k1, tmp1 - tmp1/2)
      tmp2 = get(k2)
      put(k2, tmp2 + tmp1/2)
    end
    before_commit[t] = get_timestamp()
  Commit Transaction
  after_commit[t]    = get_timestamp()
end
```

As with workload 1, we check that each transaction is all-or-nothing, that transactions committed before the fault are present (and those after are not), and that the results for range-scans and point-queries match. Further, since none of the transactions change the total amount of money, the checker tests every pair of rows with keys of the form k-t-$2i$, k-t-$(2i+1)$ to see whether their amounts sum to the same value as in the initial state.

	key	value
meta rows	THR-1-TXN-1	k-2-k-5-TS-00:01
	THR-1-TXN-2	k-6-k-8-TS-00:13
	THR-2-TXN-1	k-7-k-6-TS-00:03
	THR-2-TXN-2	k-3-k-7-TS-00:14
work rows	k-1	v-init-1
	k-2	v-THR-1-TXN-1
	k-3	v-THR-2-TXN-2
	k-4	v-init-4
	k-5	v-THR-1-TXN-1
	k-6	v-THR-1-TXN-2
	k-7	v-THR-2-TXN-2
	k-8	v-THR-1-TXN-2

Figure 2: An example workload 4 output table.

Workload 4: This is the most stressful (and time-consuming) workload. It has multiple threads, each performing multiple transactions, and each transaction writes to multiple keys. Moreover, transactions from the same or different threads may update the same key, fully exercising database concurrency control.

Figure 2 shows an example output table generated by workload 4. In this example, there are two threads (`THR-1` and `THR-2`), each thread contains two transactions (`TXN-1` and `TXN-2`), and each transaction updates two work-row keys (e.g., `k-2` and `k-5` for `THR-1-TXN-1`). The table has two parts: the first 4 rows (meta rows) keep the metadata about each transaction, including the non-meta keys written in that transaction and a timestamp taken immediately before the commit. The next 8 rows (work rows) are the working region shared by the transactions. Each transaction randomly selects two keys within the working region, and updates their values with its thread ID and transaction ID. For example, the value `v-THR-1-TXN-1` of the key `k-2` means the transaction 1 in thread 1 updated the key `k-2`. All rows start with initial values (e.g., `v-init-1`) to give a known starting state.

The following pseudo-code shows the working logic of the first transaction of thread one; the runtime values in the comments are the ones used to generate Figure 2:

```
me = "THR-1-TXN-1"
Begin Transaction
  // update two work rows:
  key1 = get_random_key()          //k-2
```

```
  put(key1, "v-" + me)
  key2 = get_random_key()        //k-5
  put(key2, "v-" + me)

  // update my meta row:
  before_commit = get_timestamp() //TS-00:01
  v = key1 + "-" + key2 + "-"
    + str(before_commit)  //k-2-k-5-TS-00:01
  put(me, v)
Commit Transaction
after_commit[me] = get_timestamp()
```

Each transaction involves multiple rows (one meta row and multiple work rows) and stores a large amount of self-description (i.e., the meta row records the work rows updated and the work rows specify which meta row is associated with their last updates). The `after_commit` timestamps allow greater precision in identifying durability, but are not strictly necessary.

As with workloads 1–3, we validate that range and point queries match. In addition, since the table contains initial values (e.g., `k-1 = v-init-1`) before the workload starts, we expect the table should at least maintain the original values in the absence of updates—if any of the initial rows are missing, we report a durability error. A further quick check verifies that the format of each work row is either in the initial state (e.g., `k-1 = v-init-1`), or was written by a valid transaction (e.g., `k-2 = v-THR-1-TXN-1`). Any violation of the formatting rules is a consistency error.

Another check involves multiple rows within each transaction. Specifically, the work rows and meta rows we observe need to match. When we observe at least one row (either a work or a meta row) updated by a transaction T_a and there is a missing row update from T_a, we classify the potential errors based on the missing row update: If that row is corrupted (unreadable), then we report a durability error. If furthermore the transaction T_a definitely committed after the fault point (i.e., T_a's `before_commit` is after the fault injection time), we also report an isolation error because the transaction's uncommitted data became visible.

Alternatively, if the row missing the update (from transaction T_a) contains either the initial value or the value from a transaction T_b known to occur earlier (i.e., transaction T_b's `after_commit` is before transaction T_a's `before_commit`), then we report an atomicity error since partial updates from transaction T_a are observed. If furthermore transaction T_a definitely committed before the fault point, we also report a durability error, and if it definitely committed after the fault point we report an isolation error.

Note that because each transaction saves timestamps, we can determine if a work row might have been legitimately overwritten by another transaction. As shown in Figure 2, the first transaction of thread 2 (`THR-2-TXN-1`) writes to `k-7` and `k-6`, but the two rows are overwritten by `THR-2-TXN-2` and `THR-1-TXN-2`, respectively. Based on the timestamp bounds of the commits, we can determine if these overwritten records are legitimate. One last check is for transactions that definitely committed but do not leave any update; we report this as a durability error.

4 Record and Replay

The record-and-replay component records the I/O traffic generated by the Worker under the workload, and replays the I/O trace with injected power faults. As mentioned in Section 2.2, the component is built on the iSCSI layer. This design choice gives fine-grained and high-fidelity control over the I/O blocks, and allows us to transparently test databases across different OSes.

4.1 Record

Figure 3(a) shows the workflow for the record phase. The Worker exercises the database, which generates filesystem operations, which in turn generate iSCSI requests that reach the backing store. By monitoring the iSCSI target daemon, we collect detailed block I/O operations at the SCSI command level. More specifically, for every SCSI command issued to the backing store, the SCSI Parser examines its command descriptor block (CDB) and determines the command type. If the command causes data transfer from the initiator to the target device (e.g., `WRITE` and its variants), the parser records the timestamp of the command and further extracts the logical block address (LBA) and the transfer length from the CDB, then invokes the Block Tracer. The Block Tracer fetches the blocks to be transferred from the daemon's buffer and records it in an internal data log. The command timestamp, the LBA, the transfer length, and the offset of the blocks within the data log are further recorded in an index log for easy retrieval. In this way, we obtain a sequence of block updates (i.e., the Worker's block trace) that can be used to generate a disk image representative of the state after a power fault.

The block trace collected from SCSI commands is enough to generate simulated power faults. However, given the huge number of low-level block accesses in a trace, how to inject power faults efficiently is challenging. Moreover, the block I/Os themselves are too low-level to infer the high-level operations of the database under testing, which are essential for understanding why an ACID violation happens and how to fix it. To address these challenges, we design a multi-layer tracer, which correlates the low-level block accesses with various high-level semantics.

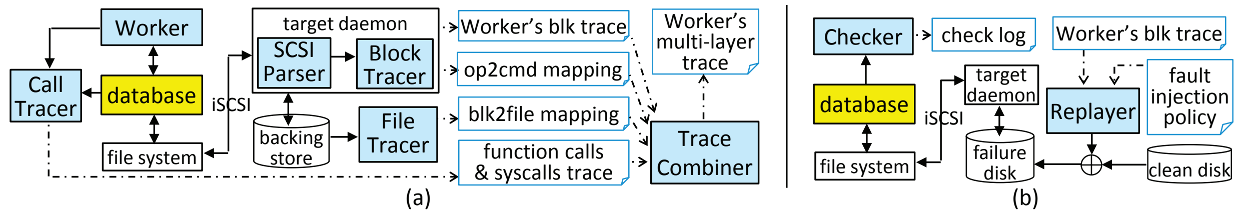

Figure 3: (a) Record phase (b) Replay for testing

In particular, we collect three more types of traces in the record phase. First, our op2cmd mapper (inside the Block Tracer, not shown) maps each block operation (512 B or 4 KB) to the specific SCSI command it is part of (a single SCSI command can cover over a megabyte); the resulting mapping (i.e., op2cmd mapping) lets us infer which operations the file system (or database) treated as one logical unit and thus may be correlated.

Second, the File Tracer connects the blocks being updated to the higher-level files and directories they store. For a given file system, the meaning of each block is well-defined (e.g., for ext3 the location of the inode bitmap is fixed after formatting, and the blocks belonging to a particular file are defined via the corresponding inode block). The File Tracer first identifies the inode number for each file (including regular files, directories, and the filesystem journal). Then, it extracts the corresponding data block numbers for each inode. In this way, the File Tracer generates a mapping (the blk2file mapping) that identifies the ownership of each block in the file system. Because each database file usually has a well-defined functionality (e.g., UNDO/REDO log files), the blk2file mapping lets us identify which I/O requests are modifying, say, the UNDO log versus the main index file. We can also identify which updates are to filesystem metadata and which are to the filesystem journal. This trace, together with the op2cmd mapping above, gives us an excellent picture of the behavior of the database at the I/O level.

The third type of trace is generated by the Call Tracer, which instruments the workload and the database and records all function calls and system calls invoked. Each call is recorded with an invoke timestamp, a return timestamp, and its thread ID. This information not only directly reveals the semantics of the databases, but also helps in understanding the lower-level I/O traffic. For example, it allows us to tell if a particular I/O was explicitly requested by the database (e.g., by correlating `fsync` and `msync` calls with their respective files) or if it was initiated by the file system (e.g., dirty blocks from another file or ordinary dirty block write back).

Finally, all of the traces, including the block trace, op2cmd mapping, blk2file mapping, and the call trace, are supplied to the Trace Combiner. The block trace and call trace are combined based on the timestamps associated with each entry. For example, based on the timestamps when a `fsync` call started and finished, and the timestamp when a SCSI WRITE command is received in between, we associate the blocks transferred in the WRITE command with the `fsync` call. Note that in a multi-threaded environment, the calls from different threads (which can be identified by the associated thread IDs) are usually interleaved. However, for each synchronous I/O request (e.g., `fsync`), the blocks transferred are normally grouped together without interference from other requests via a write barrier. So in practice we can always associate the blocks with the corresponding synchronous calls. Besides combining the block trace and the call trace, the op2cmd mapping and the blk2file mapping are further combined into the final trace based on the LBA of the blocks. In this way, we generate a multi-layer trace that spans everything from the highest-level semantics to the lowest-level block accesses, which greatly facilitates analysis and diagnosis. We show examples of the multi-layer traces in Section 5.1.

4.2 Replay for Testing

After the record phase, the replayer leverages the iSCSI layer to replay the collected I/O block trace with injected faults, tests whether the database can preserve the ACID-properties, and helps to further diagnose the root causes if a violation is found.

4.2.1 Block Replayer

Figure 3(b) shows the workflow of the replay-for-testing phase. Although our replayer can inject worse errors (e.g., corruption, flying writes, illegally dropped writes), we focus on a "clean loss of power" fault model. Under this fault model, all data blocks transferred before the power cut are successfully committed to the media, while others are lost. The Replayer first chooses a fault point based on the injection policy (see Section 4.2.2). By starting with a RAM disk image of the block device at the beginning of the workload, we produce a post-fault image by selectively replaying the operations recorded in the Worker's block trace. This post-fault image can then

op#	LBA	file
...	...	...
35	***1038***	***x.db***
36	2347	x.db
37	2351	x.db
...	...	...
49	1038	x.db

(a) P_{rep}

op#	LBA	file
61	3142	x.db
62	3146	x.db
63	***2081***	***x.db***
64	***5191***	***x.db***
65	1025	x.db
66	1029	x.db

(b) P_{jump}

op#	LBA	cmd#
152	1070	42
153	***1106***	***43***
154	***1110***	***43***
155	***1114***	***43***
156	1118	43
157	1765	44

(c) P_{head}

op#	LBA	file
245	5545	x.db
246	5646	x.db
247	***5545***	***x.db***
248	***8351***	***fs-j***
249	***8352***	***fs-j***
250	8356	fs-j

(d) P_{tran}

op#	LBA	file	syscall
331	1012	x.db	msync(x.db)
...	...	...	...
460	6841	x.log	fsync(x.log)
461	9598	fs-j	fsync(x.log)
462	9602	fs-j	fsync(x.log)
463	***1012***	***x.db***	***fsync(x.log)***

(e) P_{mmap}

Figure 4: Five patterns of vulnerable points based on real traces: (a) repetitive LBA, (b) jumping LBA, (c) head of command, (d) transition between files, and (e) unintended update to `mmap`'ed block. op# means the sequence number of the transfer operation for a data block, cmd# means the SCSI command's sequence number, `x.db` is a blinded database file name, `x.log` is a blinded database log file name, and `fs-j` means filesystem journal. The red italic lines mean a power fault injected immediately after that operation may result in failures (a–d), or a fault injected after later operations may result in failures (e). For simplicity, only relevant tracing fields are shown.

be mounted for the Checker to search for ACID violations. Because our power faults are simulated, we can reliably replay the same block trace and fault deterministically as many times as needed.

4.2.2 Fault Injection Policy

For effective testing, we design two fault injection policies that specify where to inject power faults given a workload's block trace.

Policy 1: Exhaustive. Under our fault model, each block transfer operation is atomic, although multi-block SCSI operations are not. The exhaustive policy injects a fault after every block transfer operation, systematically testing all possible power-fault scenarios for a workload.

Although exhaustive testing is thorough, for complicated workloads it may take days to complete. Randomly sampling the fault injection points may help to reduce the testing time, but also reduce the fault coverage. Hence we propose a more principled policy to select the most vulnerable points to inject faults.

Policy 2: Pattern-based ranking. By studying the multi-layer traces of two databases (TokyoCabinet and MariaDB) with exhaustive fault injection, we extracted five vulnerable patterns where a power fault occurring likely leads to ACID violations. In particular, we first identify the correlation between the fault injection points and the ACID violations observed. Then, for each fault point leading to a violation, we analyze the context information recorded in the trace around the fault point and summarize the patterns of vulnerable injection points.

Figure 4 shows examples of the five patterns based on the real violations observed in our early experiments:

Pattern A: repetitive LBA (P_{rep}). For example, in Figure 4(a) op#35 and op#49 both write to LBA 1038, which implies that 1038 may be a fixed location of important metadata. The parameter for this pattern is the repetition threshold.

Pattern B: jumping LBA sequence (P_{jump}). In Figure 4(b), the operations before op#63 access a large contiguous region (e.g., op#62 , op#61, and earlier operations which are not shown), and the operations after op#64 are also contiguous. The LBAs of op#63 and op#64 are far away from that of the neighbor operations and are jumping forward (e.g., from 2081 to 5191) or backward (e.g., from 5191 to 1025). This may imply switching operation or complex data structure updates (e.g., after appending new nodes, update the metadata of a B+ tree stored at the head of a file). The parameters of this pattern include jumping distance and jumping direction.

Pattern C: head of a SCSI command (P_{head}). Each SCSI command may transfer multiple blocks. For example, in Figure 4(c), op#153–156 all belong to cmd#43. If the fault is injected after op#153, 154, or 155, the error will be triggered. The reason may be that the blocks transferred in that SCSI command need to be written atomically, which is blocked by these fault points. The parameter of this pattern is the minimal length of the head command.

Pattern D: transition between files (P_{tran}). In Figure 4(d), the transition is between a database file (`x.db`) and the filesystem journal (`fs-j`). This pattern may imply an interaction between database and file system (e.g., delete a log file after commit) that requires special coding. The pattern also includes transitions among database files because each database file usually has a specific function (e.g., UNDO/REDO logs) and the transition may imply some complex operations involving multiple files.

Pattern E: unintended update to `mmap`'ed blocks (P_{mmap}). `mmap` maps the I/O buffers for a portion of a file

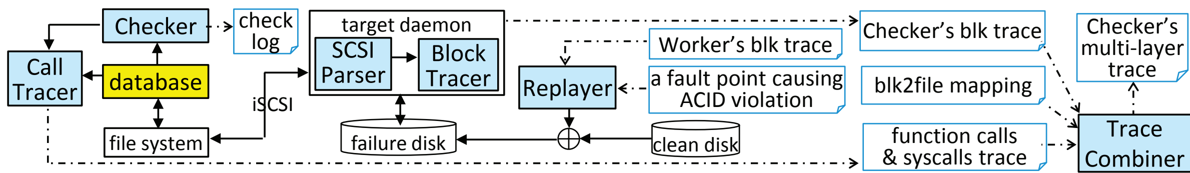

Figure 5: Replay for diagnosis

into the application's address space and thus allows the application to write to the file cache directly. This mechanism is attractive to database systems because it allows the database to circumvent double-buffering of the file cache when running on top of a file system [22]. As shown in Figure 4(e), `x.db` is a `mmap`'ed file, and LBA 1012 is in the `mmap`'ed region. Based on the system call trace, we find that op#331 is caused by calling `msync` on the memory map of `x.db`, which is an explicit request of the database. On the other hand, op#463, which also updates LBA 1012, is unintended because it happens when calling `fsync` on another file (`x.log`). Other causes for such implicit updates could be the periodic write back of dirty pages or dirty page write back due to memory pressure. In the real trace, injecting a fault immediately after the implicit update (i.e., op#463 in this case) may not necessarily cause ACID violations. Instead, injecting faults later may cause failures. So the pattern considers all operations between the implicit update and the next explicit update, and uses sampling to select fault injection points within the range. The parameter of this pattern is the sampling rate.

In summary, we extract the five block-level patterns from the real failure logs of two databases. Three of them (P_{rep}, P_{jump}, and P_{head}) are independent of file systems and OSes, two (P_{tran} and P_{mmap}) require knowledge of the filesystem structure, and P_{mmap} is further associated with system calls. Intuitively, these patterns capture the critical points in the I/O traffic (e.g., updates to some fixed location of metadata or transition between major functions) so we use them to guide fault injection. Specifically, after obtaining the traces from the record phase, we check them against the patterns. For each fault injection point, we see if it matches any of the five patterns, and score each injection point based on how many of the patterns it matches. Because a fault is always injected after a block is transferred, we use the corresponding transfer operation to name the fault injection point. For example, in Figure 4(a) op#35 and op#49 match the repetitive pattern (assuming the repetition threshold is 2), so each of them has the score of 1, while op#36 and op#37 remain 0. An operation can gain a score of more than 1 if it matches multiple patterns. For example, an operation may be simultaneously a repetitive operation (P_{rep}), the first operation of a command (P_{head}), and represent a transition between files (P_{tran}), yielding a score of 3. After scoring the operations, the framework injects power faults at the operations with the highest score. By skipping injection points with low scores, pattern-based ranking reduces the number of fault injection rounds needed to uncover failures. We show real examples of the patterns and the effectiveness and efficiency of this fault injection policy in Section 5.

4.3 Replay for Diagnosis

Although identifying that a database has a bug is useful, diagnosing the underlying root cause is necessary in order to fix it. Figure 5 shows the workflow of our diagnosis phase. Similar to the steps in replay for testing, the replayer replays the Worker's block trace up to the fault point (identified in the replay for testing phase) that lead to the ACID failure. Again, the Checker connects to the database and verifies the database's state. However, for diagnosis we activate the full multi-layer tracing. Moreover, the blocks read by the database in this phase are also collected because they are closely related to the recovery activity. The Checker's block trace, blocks-to-files mapping (collected during the record phase), and the function calls and system calls trace are further combined into Checker's multi-layer trace. Together with the check log of the integrity checker itself, these make identifying the root cause of the failure much easier. Further exploring the behavior of the system for close-by fault points that do not lead to failure [45] can also help. We discuss diagnosis based on the multi-layer traces in more detail in Section 5.1.

5 Evaluation

We built our prototype based on the Linux SCSI target framework [6]. The Call Tracer is implemented using PIN [8]. The File Tracer is built on `e2fsprogs` [3] and XFS utilities [12]. We use RHEL 6[1] as both the iSCSI target and initiator to run the databases, except that we use Windows 7 Enterprise to run the databases using NTFS.

We apply the prototype to eight widely-used databases, including three open-source embedded

[1] All results were verified with Debian 6, and per time constraints a subset were verified with Ubuntu 12.04 LTS.

databases (TokyoCabinet, LightningDB, and SQLite), one open-source OLTP SQL server (MariaDB), one proprietary licensed[2] key-value store (KVS-A), and three proprietary licensed OLTP SQL databases (SQL-A, SQL-B, and SQL-C). All run on Linux except SQL-C, which runs on NTFS.

Since none of the tested databases fully support raw block device access,[3] the file system could be another factor in the failure behavior. Hence for our Linux experiments, we tested the databases on two different file systems: the well-understood and commonly deployed ext3, and the more robust XFS. We have not yet fully implemented the Call Tracer and the File Tracer for Windows systems but there are no core technical obstacles in implementing these components using Windows-based tooling [7, 9]. Also, for the proprietary databases without debugging symbols, we supply limited support for diagnosis (but full support for testing).

5.1 Case Studies

In this subsection, we discuss three real ACID-violation cases found in three databases, and show how the multi-layer traces helped us quickly diagnose their root causes.

5.1.1 TokyoCabinet

When testing TokyoCabinet under Workload 4, the Checker detects the violations of atomicity (a transaction is partially committed), durability (some rows are irretrievable), and consistency (the retrievable rows by the two query methods are different). These violations are non-deterministic—they may or may not manifest themselves under the same workload—and the failure symptoms vary depending on the fault injection point, making diagnosis challenging. The patterns applicable to this case include P_{jump}, P_{head}, P_{tran}, and P_{mmap}.

For a failing run, we collect the Checker's multi-layer trace (Figure 6(b)). For comparison purposes, we also collect the Checker's trace for a bug-free run (Figure 6(a)). By comparing the two traces, we can easily see that `tchdbwalrestore` is not invoked in the failing run. In the parent function (`tchdbopenimpl`) there is a read of 256 bytes from the database file `x.tcb` in both traces, but the content read is different by one bit (i.e., 108 vs. 118 in op#1). Further study of the data structures defined in the source code reveals that the first 256 bytes contain a flag (`hdb->flags`), which determines whether to

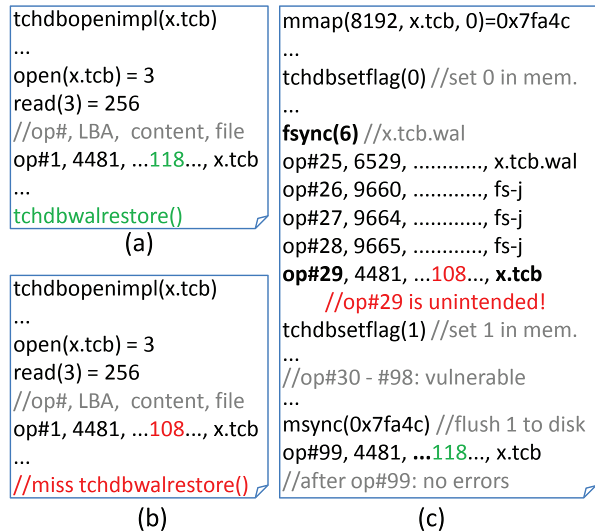

Figure 6: Example of multi-layer traces adapted from the real traces of TokyoCabinet. (a) Checker's trace when no violations were found. (b) Checker's trace when ACID violations were found. (c) Worker's trace around the power fault points leading to ACID violations. LBA, and address) are reduced and only relevant fields and lines are shown.

invoke `tchdbwalrestore` on startup. The one bit difference in op#1 implies that some write to the beginning of `x.tcb` during the workload causes this ACID violation.

We then look at the Worker's multi-layer trace near the power-fault injection points that manifest this failure (Figure 6(c)). The majority of the faults within op#30–98 cause ACID violations, while power losses after op#99 do not cause any trouble. So the first clue is op#99 changes the behavior. Examining the trace, we notice that the beginning of `x.tcb` is `mmap`'ed, and that op#99 is caused by an explicit `msync` on `x.tcb` and sets the content to 118. By further examining the writes to `x.tcb` before op#30–98, we find that op#29 also updates `x.tcb` by setting the content to 108. However, this block update is unintended: an `fsync` on the write-ahead log `x.tcb.wal` triggers the OS to also write out the dirty block of `x.tcb`.

The whole picture becomes more clear with the collected trace of high-level function calls. It turns out that at the beginning of each transaction (`tchdbtranbegin()`, not shown), the flag (`hdb->flags`) is set to 0 (`tchdbsetflag(0)`), and then set to 1 (`tchdbsetflag(1)`) after syncing the initial `x.tcb.wal` to disk (`fsync(6)`). If the synchronization of `x.tcb.wal` with disk is successful, the flag 0 should be invisible to disk. In rare cases, however, the `fsync` on `x.tcb.wal` causes an unintended flush of the flag 0 to `x.tcb` on disk (as captured by op#29). In this scenario, if

[2] Due to the litigious nature of many database vendors (see "The DeWitt Clause" [1]), we are unable to identify the commercial databases by name. We assure readers that we tested well recognized, well regarded, and mainstream software.

[3] Nearly all modern databases run through the file system. Of the major commercial OLTP vendors, Oracle has removed support for raw storage devices [33], IBM has deprecated it for DB2 [23], and Microsoft strongly discourages raw partitioning for SQL Server [26].

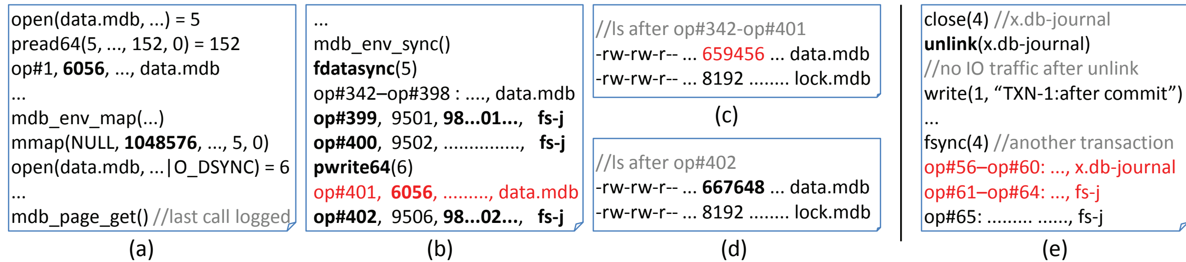

Figure 7: Examples of multi-layer traces adapted from real traces of LightningDB (a–d) and SQLite (e): (a) LightningDB Checker's trace, (b) Worker's trace around the bug-triggering fault point (op#401), (c) Checker's check log showing the size of `data.mdb` after op#342–401, (d) Checker's check log showing the size of `data.mdb` after op#402, and (e) SQLite Worker's trace around the fault points (op#56–64) that cause durability violations.

a power fault is injected at the points between op#29 and 99, it will interrupt the transaction and the post-fault disk image has the flag set to 0. Upon the recovery from the post-fault disk image, the database will mistakenly think `x.tcb.wal` has not been initialized (because the on-disk flag is 0), and skip the `tchdbwalrestore()` procedure. Consequently, an incomplete transaction is exposed to the user.

The unintended updates to the `mmap`'ed file could be caused for two reasons. One is the flushing of dirty pages by kernel threads due to memory pressure or timer expiry (e.g., `dirty_writeback_centisecs`), the other is the internal dirty page flushing routines of file systems. Since we can see the `fsync` call that causes the unintended update, it is clear that the fact that ext3 and XFS are aggressive in flushing blocks when there is a sync is to blame. However, regardless if it is due to the kernel or the file system, the programmer's incorrect assumption that a change to a `mmap`'ed files will not be asynchronously written back is the underlying root cause.

One solution to this problem is using failure-atomic `msync` [34], which prevents the non-explicit write back to the `mmap`'ed file.

5.1.2 LightningDB

When testing LightningDB on ext3 under Workload 4, the Checker, which links the database library into its address space, crashes on certain queries. The applicable patterns include P_{rep}, P_{head}, and P_{tran}.

The Checker's multi-layer trace (Figure 7(a)) shows that before the crash there are two `pread64`s (the 2nd one is omitted in the figure) from the head (LBA 6056) of the database file `data.mdb`; also, `data.mdb` is `mmap`'ed into memory. The size of the mapping is 1,048,576 bytes (256 4 KB-pages), which exceeds the actual length of the file. The last function logged before the crash is `mdb_page_get`, which returns the address of a page. The LightningDB documents [5] and source code reveal that the first two pages of this file are metapages, which maintain the valid page information of the internal B+ tree, and that the mapping is 256 pages by default for performance reasons. Given this information, we suspect that the crash is caused by referencing a `mmap`'ed page that is valid based on the metapages but lies beyond the end of the backing file.

The Worker's trace (Figure 7(b)) and the Checker's check logs ((c) and (d)) verify our hypothesis. In this example, a power fault after op#401 leads to the crash, while a fault any other time (e.g., after op#342–400 and op#402) causes no problem. As shown in (b), op#401 is an update to the head (LBA 6056) of `data.mdb`, which maintains the valid page information. However, after applying op#401, the size of `data.mdb` did not get updated in the file system (Figure 7 (c)). Only after op#402, which is an update to the filesystem metadata, is the length of the file increased to 667,648 bytes (Figure 7 (d)), and the memory-mapping is safe to access from then on.

Based on the traces we can further infer that op#399, 400, and 402 form a filesystem journal transaction that updates the length metadata for `data.mdb`. The content of op#399 (i.e., "98...01", which is 98393bc001 in the real trace) matches the magic number of the ext3 journal and tells us that it is the first block (i.e., the descriptor block) of the journal transaction. op#402 (with content "98...02") is the last block (i.e., the commit block) of the journal transaction. op#400 is a journaled length update that matches the format of the inode, the superblock, the group descriptor, and the data block bitmap, all of which need to be updated when new blocks are allocated to lengthen a file. This is why the length update is invisible in the file system until after op#402: without the commit block, the journal transaction is incomplete and will never be checkpointed into the main file system.

Note that op#401 itself does not increase the file size since it is written to the head of the file. Instead, the file is extended by op#342–398, which are caused by appending B+ tree pages in the memory (via `lseeks` and

`pwrites` before the `mdb_env_sync` call) and then calling `fdatasync` on `data.mdb`. On ext3 with the default "ordered" journaling mode, the file data is forced directly out to the main file system prior to its metadata being committed to the journal. This is why we observe the journaling of the length update (op#399, 400, and 402) after the file data updates(op#342–398).

The fact that the journal commit block (op#402) is flushed with the next `pwrite64` in the same thread means `fdatasync` on ext3 does not wait for the completion of journaling (similar behavior has been observed on ext4). LightningDB's aggressive `mmap`'ing and referencing pages without verifying the backing file finally trigger the bug. We have verified that changing `fdatasync` to `fsync` fixes the problem. Another platform-independent solution is checking the consistency between the metapage and the actual size of the file before accessing.

Because we wanted to measure how effective our system is in an unbiased manner, we waited to look at LightningDB until after we had finalized and integrated all of our components. It took 3 hours to learn the LightningDB APIs and port the workloads to it, and once we had the results of testing, it took another 3 hours to diagnose and understand the root cause. As discussed in more detail in Section 5.3, it takes a bit under 8 hours to do exhaustive testing for LightningDB, and less than 21 minutes for pattern-based testing. Given that we had no experience with LightningDB before starting, we feel this shows that our system is very effective in finding and diagnosing bugs in databases.

5.1.3 SQLite

When testing SQLite under any of the four workloads, the Checker finds that a transaction committed before a power fault is lost in the recovered database. The applicable patterns include P_{rep}, P_{head}, and P_{tran}.

Figure 7(e) shows the Worker's multi-layer trace. At the end of a transaction (`TXN-1`), the database closes and unlinks the log file (`x.db-journal`), and returns to the user immediately as implied by the "after commit" message. However, the `unlink` system call only modifies the in-memory data structures of the file system. This behavior is correctly captured in the trace—i.e., no I/O traffic was recorded after `unlink`. The in-memory modification caused by `unlink` remains volatile until after completing the first `fsync` system call in the next transaction, which flushes all in-memory updates to the disk. The SQLite documents [10] indicate that SQLite uses an UNDO log by default. As a result, when a power fault occurs after returning from a transaction but before completing the next `fsync` (i.e., before op#65), the UNDO log of the transaction remains visible on the disk (instead of being unlinked). When restarting the database, the committed transaction is rolled back unnecessarily, which makes the transaction non-durable.

One solution for this case is to insert an additional `fsync` system call immediately after the `unlink`, and do not return to the user until the `fsync` completes. Note that this case was manifested when we ran our experiments under the default DELETE mode of SQLite. The potential non-durable behavior is, surprisingly, known to the developers [2]. We ran a few additional experiments under the WAL mode as suggested by the developers, and found an unknown error leading to atomicity violation. We do not include the atomicity violation in Table 1 (Section 5.2) since we can reproduce it even without injecting a power fault.

5.2 Result Summary

Table 1 summarizes the ACID violations observed under workloads 1–3 (W-1 through W-3) and three different configurations of workload 4 (W-4.1 through W-4.3). W-4.1 uses 2 threads, 10 transactions per thread, each transaction writes to 10 work rows, and the total number of work rows is 1,000. W-4.2 increases the number of threads to 10, the number of work rows being written per transaction to 20, and the total number of work rows to 4,000. W-4.3 further increases the number of work rows written per transaction to 80.

Table 1 shows that 12 out of 15 database/filesystem combinations experienced at least one type of ACID violation under injected power faults. Under relatively simple workloads (i.e., W-1, W-2, and W-3), 11 database/filesystem combinations experienced one or two types of ACID violations. For example, SQL-B violated the C and D properties under workload 3 on both file systems. On the other hand, some databases can handle the power faults better than others. For example, LightningDB does not show any failures under power faults with the first three workloads. Another interesting observation is that power faulting KVS-A causes hangs, preventing the Checker from further execution in a few cases. We cannot access the data and so cannot clearly identify which sort of ACID violation this should be categorized as.

Under the most stressful workload more violations were found. For example, TokyoCabinet violates the atomicity, consistency, and durability properties, and LightningDB violates durability under the most stressful configuration (W-4.3).

The last four columns of Table 1 show for each type of ACID violation the percentage of power faults that cause that violation among all power faults injected under the exhaustive policy, averaged over all workloads. An interesting observation is that the percentage of violations for XFS is always smaller than that for ext3 (except for SQL-B, which shows a similar percentage on both file

DB	FS	W-1	W-2	W-3	W-4.1	W-4.2	W-4.3	A	C	I	D
TokyoCabinet	ext3	D	D	D	A C D	A C D	A C D	0.15	0.14	0	16.05
	XFS	—	D	D	A C D	D	A C D	<0.01	0.01	0	4.38
MariaDB	ext3	D	D	D	D	D	D	0	0	0	1.36
	XFS	D	D	D	D	D	D	0	0	0	0.49
LightningDB	ext3	—	—	—	—	—	D	0	0	0	0.05
	XFS	—	—	—	—	—	—	0	0	0	0
SQLite	ext3	D	D	—	D	D	D	0	0	0	19.15
	XFS	—	—	D	D	D	D	0	0	0	10.60
KVS-A	ext3	—	—	Hang*	—	—	—	0	0	0	0
	XFS	—	—	—	—	—	—	0	0	0	0
SQL-A	ext3	D	D	D	D	D	D	0	0	0	3.31
	XFS	D	D	D	D	D	D	0	0	0	0.92
SQL-B	ext3	D	D	C D	C D	C D	C D	0	8.96	0	3.24
	XFS	C D	D	C D	C D	C D	C D	0	7.77	0	3.90
SQL-C	NTFS	D	D	D	D	D	D	0	0	0	8.08

Table 1: ACID violations observed under workloads 1–3 (W-1 through W-3) and three configurations of workload 4 (W-4.1 through W-4.3). "—" means no failure was observed. The last four columns show for each type of ACID violation the percentage of power faults that cause that violation among all power faults injected under the exhaustive policy, averaged over all workloads. *The checker never reported errors for KVS-A, but in some cases power loss caused a hang in the database code during recovery afterwards. This could potentially be categorized as either an I or a D error; regardless the database is not usable.

systems). This may indicate that XFS is more robust for databases compared to ext3.

Some violations are difficult to trigger. For example, LightningDB violates durability under only 0.05% of the power faults injected under the exhaustive policy. Here the exhaustive approach is not very efficient, and a random sampling approach would be likely to miss the error.

Overall, violation of durability is the most prevalent failure, being found in 7 out of the 8 tested databases and ranging from 0.05% up to 19.15% among all power faults injected. A common type of durability violation is a transaction committed before the power fault being missing after recovery. TokyoCabinet, SQLite, and SQL-B have this failure behavior.

Another common type of durability violation is partial table corruption. Examples include non-retrievable rows, rows retrievable but with corrupted data, or a database crash when touching certain rows. TokyoCabinet, LightningDB, and SQL-B exhibit such failures.

The third type of durability violation is failing to connect to the database upon restart from the post-fault disk image. As a result, the whole table was non-durable. MariaDB, SQL-A, SQL-B, and SQL-C have demonstrated this failure behavior. Our best efforts at manually recovering from this condition failed, except that for MariaDB there is an additional recovery procedure that can allow full recovery. This suggests that for MariaDB the data is likely intact, but the default recovery procedure failed to recognize it upon restart. Although that may be a reasonable strategy for arbitrary image corruption, we do not feel this is completely acceptable behavior under the easy fault model we apply.

5.3 Effectiveness of Patterns

We now evaluate the effectiveness of our pattern-based ranking algorithm at identifying the most vulnerable fault injection points. The five patterns we use (see Section 4.2.2) were extracted based on the ACID violations observed in TokyoCabinet and MariaDB with the exhaustive policy, and we apply them to all 8 tested databases. For SQL-C, we apply only the filesystem-independent and OS-independent patterns (i.e., P_{rep}, P_{jump}, and P_{head}).

Table 2 compares the pattern-based policy with the exhaustive policy under W-4.1 and W-4.3 on ext3 (the results on XFS and under other workloads are similar). Overall, the pattern-based policy is very effective. Injecting power faults at points with scores exceeding 2 can manifest all the types of ACID violations detected by exhaustive testing, except in one or two cases per configuration. For SQL-A, the points with scores exceeding 2 do not suffice; using the score 2 points in addition, however, does suffice to manifest the ACID violations.

The patterns we identified using analysis from only 2 of the databases generalized well to the other 6. Especially for LightningDB, we performed all of the work-

DB	W-4.1		W-4.3	
	match?	top?	match?	top?
TokyoCabinet	Y	Y	Y*	Y
MariaDB	Y	Y	Y	Y
LightningDB	—	—	Y	Y
SQLite	Y	Y	Y	Y
KVS-A	—	—	—	—
SQL-A	Y	N	Y	N
SQL-B	Y	N	Y*	Y
SQL-C	Y	Y	Y	Y

Table 2: Comparison of exhaustive and pattern-based fault injection policies. Y in the match? column means injecting faults at the operations identified by the pattern policy can expose the same types of ACID violations as exposed under the exhaustive policy. Y in the top? column means that power faults need to be injected only at the score 3 or higher points. "—" means no error is detected under both polices. *Extrapolated from a partial run, given the long time frame.

DB	Exhaustive	Pattern	%
TokyoCabinet	41,625	7,084	17.0%
MariaDB	1,013	14	1.4%
LightningDB	5,570	171	3.1%
SQLite	438	23	5.3%
KVS-A	4,193	69	1.7%
SQL-A	1,082	53*	4.9%
SQL-B	20,200	936	4.6%
SQL-C	313	2	0.6%
Average	—	—	4.8%

Table 3: Comparison of our two policies in terms of the number of fault injection points under W-4.3. The pattern-based policy includes only points with scores exceeding 2 *except for SQL-A, which includes points with scores exceeding 1.

DB	Exhaustive	Pattern	%
TokyoCabinet	12d 1h*	2d 0h	16.6%
MariaDB	3h 27m	3m 2s	1.5%
LightningDB	7h 56m	20m 44s	4.4%
SQLite	13m 12s	0m 42s	5.3%
KVS-A	5h 17m	5m 32s	1.7%
SQL-A	3h 33m	10m 37s	5.0%
SQL-B	71d 1h*	2d 9h	3.4%
SQL-C	3h 23m	2m 34s	5.1%
Average	—	—	5.4%

Table 4: Comparison of our two policies in terms of replay for testing time under W-4.3. *Estimated based on progress from a 3-day run.

loads, testing, and diagnosis presented in Section 5.1.2 without any further iteration on the framework. Yet we were still able to catch a bug that occurs in only 0.05% of the runs.

5.4 Efficiency of Patterns

We further evaluate the efficiency of our pattern-based policy in terms of the number of fault injection points and the execution time for testing databases with the power faults injected at these points. Table 3 compares the number of fault injection points under the exhaustive and pattern-based policies for W-4.3 on ext3 (the results for the other cases are similar). Under this setting only the points with scores exceeding 2 are needed to manifest the same types of ACID violations as the exhaustive policy (except for SQL-A, which requires points with scores exceeding 1). Compared to the exhaustive policy, the pattern-based policy reduces the number of fault injection points greatly, with an average 21x reduction, while manifesting the same types of ACID violations.

Table 4 further compares the two policies in terms of the execution time required in the replay for testing. Note that the pattern-based policy is very efficient, with an average 19x reduction in execution time compared to the exhaustive policy. For example, we estimate (based on letting it run for 3 days) that exhaustive testing of SQL-B would take over 2 months, while with the pattern-based policy, the testing completed in about 2 days.

6 Comparison to EXPLODE

EXPLODE [44] is most closely related to our work. It uses ingenious *in situ* model checking to exhaust the state of storage systems to expose bugs, mostly in file systems. Part of our framework is similar: we also use a RAM disk to emulate a block device, and record the resulting trace. However, our fault models are different. EXPLODE simulates a crash model where data may be corrupted before being propagated to disk, and where buffer writes are aggressively reordered. This is quite harsh to upper-level software. Our fault model focuses on faults where the blame more squarely lies on the higher-level software (e.g., databases) rather than on the OS kernel or hardware device. Besides, unlike EXPLODE, we provide explicit suggestions for workloads that are likely to uncover issues in databases, and explicit tracing support to pin found errors to underlying bugs.

When applied to our problem—i.e., testing and diagnosing databases under power fault—EXPLODE has

some limitations. First, the number of manually-defined choice points is limited. The size of the current write set between two choices could be huge, which is especially likely under heavy database transactions. It would be prohibitively expensive, if not impossible, for a model checking tool to exhaust every subset or permutation of the write set on every path. As is, EXPLODE has such a large set of states to explore that in practice it terminates when the user loses patience [44]. Meanwhile, many of the simulated crash images could contain harsh corruption that is unrealistic under power fault. Second, EXPLODE enforces deterministic thread scheduling. This factor, together with the coarse-grained choices, may hide certain types of bugs involving concurrent transactions, a common case in databases. For instance, in a database using `mmap`, the pages maintained in the write set in EXPLODE is likely different from a native run. Specifically, a thread T1 may call `fsync` (which causes the write set to shrink) in its transaction, but due to the enforced scheduling, thread T2 may lose its chance to update the pages before the shrink. Since EXPLODE generates crash images whenever the write set shrinks, it cannot generate an image (and manifest a bug) that requires updates from both T1 and T2. Third, even if a bug is triggered, pinpointing the root cause in a database is still challenging given the code size and complexity. Finally, EXPLODE is built on the Linux kernel, which does not allow testing Windows databases.

On the other hand, by exploring the state space, EXPLODE could exercise different code paths and make corner cases appear as often as common ones, which is a merit of model checking. Thus, we view EXPLODE-like approaches as complementary.

7 Related Work

Testing databases Previous work mostly focuses on functional testing of databases rather than on resilience to external faults. Slutz [38] generates millions of valid SQL queries, and then compares the results from multiple databases; if one database disagrees then that indicates a probable bug. Chays *et al.* [14] proposes a set of tools that automatically generate schema-compliant queries for testing. To improve test coverage, Bati *et al.* [13] propose a genetic approach for generating random test cases for database engines. All of these approaches focus on database bugs in fault-free operation, rather than when power is lost.

Subramanian *et al.* [39] examines the effects of disk corruption on MySQL. Unlike [39], we study the effects of power faults on a range of databases, including closed-source ones, and assume an easy, "perfect" block device under power fault.

Reliability analysis of storage software Similar to EXPLODE [44], MODIST [43] applies model checking to distributed systems and evaluates replicated Berkeley DB. RapiLog [21] analyzes the durability of databases and simplifies the logging by leveraging a formally-verified kernel and synthesized driver. Again, we view these formal methods as complementary. Thanumalayan *et al.* [35] proposes an abstract persistent model (APM) of filesystem properties and studies the effects on application consistency after simulating crashes in the filesystem model. Unlike their modeling approach, we test databases running on real file systems and do not intentionally manipulate the order or content of the blocks. The NoFS [17] shows how a file system can be designed to maintain consistency in the face of crashes; we presume that a database written using the NoFS techniques would be more resilient than those we tested. The IRON file system [36] implements additional redundancy and recovery methods in order to better survive various failures. Again, similar ideas could be applied to databases, although a direct mapping from concepts such as inodes and superblocks to their database equivalents may be nontrivial.

Reliability analysis of storage hardware Several studies have looked at the failure behavior of storage hardware, from spinning magnetic disks [15, 32, 37] to flash memory [19, 20, 40, 46]. Schroeder *et al.* [37] considers in-the-wild failure probabilities, while Chen *et al.* [15] considers how RAID improves the durability and reliability of storage systems. Nightingale *et al.* [32] analyzes hardware failures on PCs including disk subsystem failures. However, none of these studies looks at how the software using the hardware actually responds to faults.

8 Conclusions

We have shown that even ostensibly well-tested databases can lose data. This should be a wake-up call for any author of storage systems software: undirected testing is not enough. Thorough testing requires purpose-built workloads designed to highlight failures, as well as fault injection targeted at those situations in which storage system designers are likely to make mistakes. We can offer no panacea; creating failure-proof storage software is hard. But unless careful attention is paid to correctness, we will continue to cluck our tongues and sigh, while users will continue to cry.

9 Acknowledgments

The authors would like to thank Terrence Kelly, David Andersen (their shepherd), and the anonymous reviewers for their invaluable feedback. This research is partially supported by NSF grants #CCF-0953759 (CAREER Award), #CCF-1218358, and #CCF-1319705, and by a gift from HP.

References

[1] DeWitt Clause. http://en.wikipedia.org/wiki/David_DeWitt.

[2] Discussion on a potential bug on SQLite forum. http://sqlite.1065341.n5.nabble.com/Potential-bug-in-crash-recovery-code-unlink-and-friends-are-not-synchronous-td68885.html.

[3] E2fsprogs: Ext2/3/4 filesystems utilities. http://e2fsprogs.sourceforge.net.

[4] iSCSI wiki. https://en.wikipedia.org/wiki/ISCSI.

[5] LightningDB documents. http://symas.com/mdb/#docs.

[6] Linux SCSI target framework (tgt). http://stgt.sourceforge.net/.

[7] NFI: NTFS file sector information utility. http://support.microsoft.com/kb/253066/en-us/.

[8] PIN — a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool/.

[9] PIN for windows. https://software.intel.com/en-us/articles/pintool-downloads/.

[10] SQLite documents. http://www.sqlite.org/docs.html.

[11] SQLite testing website. http://www.sqlite.org/testing.html.

[12] XFS file system utilities. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/xfsothers.html.

[13] BATI, H., GIAKOUMAKIS, L., HERBERT, S., AND SURNA, A. A genetic approach for random testing of database systems. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB '07, VLDB Endowment, pp. 1243–1251.

[14] CHAYS, D., DENG, Y., FRANKL, P. G., DAN, S., VOKOLOS, F. I., AND WEYUKER, E. J. An agenda for testing relational database applications. In *Proceedings of Software Testing, Verification and Reliability* (March 2004), pp. 17–44.

[15] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June 1994), 145–185.

[16] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 74–83.

[17] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)* (San Jose, California, February 2012).

[18] CNN. Manufacturer blames super bowl outage on incorrect setting. http://www.cnn.com/2013/02/08/us/superdome-power-outage/, 2013.

[19] GABRYS, R., YAAKOBI, E., GRUPP, L. M., SWANSON, S., AND DOLECEK, L. Tackling intracell variability in TLC flash through tensor product codes. In *ISIT'12* (2012), pp. 1000–1004.

[20] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.

[21] HEISER, G., LE SUEUR, E., DANIS, A., BUDZYNOWSKI, A., SALOMIE, T.-L., AND ALONSO, G. RapiLog: Reducing System Complexity Through Verification. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 323–336.

[22] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2 (2007), 141–259.

[23] IBM. Database logging using raw devices is deprecated. http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.rn.doc/doc/c0023086.htm, 2006.

[24] LEACH, A. Level 3's UPS burnout sends websites down in flames. http://www.theregister.co.uk/2012/07/10/data_centre_power_cut/, 2012.

[25] MCMILLAN, R. Amazon blames generators for blackout that crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazon_explains/, 2012.

[26] MICROSOFT. NTFS vs. FAT vs. raw partitions. http://technet.microsoft.com/en-us/library/cc966414.aspx, 2007.

[27] MILLER, R. Human error cited in hosting.com outage. http://www.datacenterknowledge.com/archives/2012/07/28/human-error-cited-hosting-com-outage/, 2012.

[28] MILLER, R. Power outage hits London data center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/, 2012.

[29] MILLER, R. Data center outage cited in visa downtime across canada. http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/, 2013.

[30] MILLER, R. Power outage knocks DreamHost customers offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/, 2013.

[31] NBC. Power outage affects thousands, including columbus hospital. http://columbus.gotnewswire.com/news/power-outage-affects-thousands-including-columbus-hospital, 2014.

[32] NIGHTINGALE, E. B., DOUCEUR, J. R., AND ORGOVAN, V. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 343–356.

[33] ORACLE. Desupport raw storage device. http://docs.oracle.com/cd/E16655_01/server.121/e17642/deprecated.htm, 2013.

[34] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 225–238.

[35] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)* (October 2014).

[36] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)* (Brighton, United Kingdom, October 2005), pp. 206–220.

[37] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)* (2007).

[38] SLUTZ, D. R. Massive stochastic testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1998), VLDB '98, Morgan Kaufmann Publishers Inc., pp. 618–622.

[39] SUBRAMANIAN, S., ZHANG, Y., VAIDYANATHAN, R., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND NAUGHTON, J. F. Impact of disk corruption on open-source DBMS. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 509–520.

[40] TSENG, H.-W., GRUPP, L. M., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)* (2011).

[41] VERGE, J. Internap data center outage takes down Livestream and StackExchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[42] WOLFFRADT, R. S. V. Fire in your data center: No power, no access, now what? http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html, 2014.

[43] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, pp. 213–228.

[44] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)* (November 2006), pp. 131–146.

[45] ZELLER, A. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, USA, 2002), SIGSOFT '02/FSE-10, ACM, pp. 1–10.

[46] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)* (2013).