Crashproofing the Original NoSQL Key-Value Store

TERENCE KELLY

This episode of Drill Bits unveils a new crash-tolerance mechanism that vaults the venerable `gdbm` database into the league of *transactional* NoSQL data stores. We'll motivate this upgrade by tracing `gdbm`'s history. We'll survey the subtle science of crashproofing, navigating a minefield of traps for the unwary. We'll arrive at a compact and rugged design that leverages modern file-system features, and we'll tour the production-ready implementation of this design and its ergonomic interface.

ENDURING PATTERNS FOR DURABLE DATA

Ken Thompson wrote the original `dbm` amid a storied golden age. Sparked by Unix, software creativity flourished at Bell Labs in the 1970s, producing an ecosystem¹⁰ that remains vibrant decades later. The parent company's ambient technology may have inspired staff to re-imagine data organization, with profound consequences:

IBM OS builders worked in a place where data lived on unit record equipment (80-column cards, 132-column printers, etc.) and built a file system that looked like that. Ken and Dennis worked in a phone company where data traveled in streams over wires, and built a system in which files looked like that.³

BEFORE TERMINALS, WE ALL USED PUNCH CARDS, WHICH WERE ORIGINALLY DEVELOPED TO CONTROL LOOMS. EARLY MAINFRAMES WOULD PRODUCE A SWEATER EACH TIME YOU RAN YOUR CODE. EVENTUALLY WE GOT THEM TO STOP. WE HAD ENOUGH SWEATERS.

<https://xkcd.com/2324/>

Unstructured byte streams from files and pipes forced applications to impose structure. Many opted for the easygoing key-value paradigm—often a good choice if a relational database would be overkill and fixed layout would be too rigid. Enduring key-value patterns from the early years include self-describing text formats² and the built-in associative arrays of AWK and its descendants.

Thompson's `dbm` library provided a *persistent* key-value abstraction.¹⁶ Client programs manipulated pairs of key and value strings via a `store()/fetch()` interface; under the hood `dbm` managed an extensible hash table embedded in a file.

Philip Nelson implemented GNU `gdbm` in 1990. Since then maintainers Justin Downs and Sergey Poznyakoff have updated `gdbm` to exploit new Unix features such as `writenv()` and `mmap()`. Release 1.20 (July 2021) improves performance via a complete rewrite of the hash-table bucket cache. Users and client software maintainers often help with testing.

Thanks to its maturity and ease of use, `gdbm` has become an indispensable workhorse. On some Linux distributions, one in six software packages depends directly or indirectly on `libgdbm`; prominent examples include `man-db` (the manual pages), several Apache modules, and popular interfaces for Python and Perl. A survey of nearly 200,000 Debian systems found that 60 percent regularly use `libgdbm`.⁴

Unfortunately, crash vulnerability, the Achilles' heel of the `dbm` family, has been driving some potential users to alternative data stores ever since the 1990s. An untimely power failure, operating-system kernel panic, or process

crash could maim a `gdbm` database. Applications that must tolerate crashes have therefore used “transactional” NoSQL data stores such as Berkeley DB¹¹ or LevelDB,⁵ or full-blown relational databases supporting ACID transactions. Now, however, you can have your `gdbm` and crash it too.

SUCCEEDING AGAINST FAILURES

Tolerating a crash means restoring persistent data to a *consistent* state that satisfies all relevant correctness criteria in all software layers, including the application. Consistency means more than merely the absence of corruption in lower layers: Consider an imprudently implemented banking application that intends to decrement one account balance and increment another by the same amount. A crash occurs between these two steps, and by chance it has the same effect as an orderly (though ill-timed) shutdown. Data integrity is impeccable at the storage, file-system, and database layers, but the application-level “conservation-of-money” invariant is violated.

Applications prevent crash-induced inconsistency by using *transactions* that perform multiple changes atomically, even in the presence of failures. An application must ensure that both the before and after states are consistent; a transaction mechanism guarantees that one or the other can be recovered following a crash. Unfortunately, reliable implementations are elusive: Mature, widely used transactional data stores don’t always tolerate crashes.¹⁷

Little wonder. Portably implementing failure-atomic

A part from technical difficulties, conflicting priorities also undermine reliability

updates atop Unix-like operating systems is remarkably difficult.¹² File systems provide vague and often weak post-crash guarantees, which furthermore depend on configuration details. The Posix system calls that modify files and synchronously persist the changes to durable media—`write()`/`fsync()` and `mmap()`/`msync()`—are not atomic. For example, an attempted two-byte `write()` may return 1, meaning that the first byte may become persistent at any time but we must try again to `write()` the second byte; meanwhile, of course, a crash may occur. The list of pitfalls goes on: Without explicit ordering constraints from `fsync()` or `msync()`, the operating system may push file modifications from main-memory buffers down to durable storage at any time and in any order. A crash may leave files corrupted but not obviously so. That's just the tip of the iceberg; Pillai et al. describe further challenges.¹³

Apart from technical difficulties, conflicting priorities also undermine reliability. Many data stores offer rich functionality, which requires complex code, which harbors bugs. Worse yet, many databases strive for performance to the detriment of crash tolerance—the main motivation for transactions. With databases as with parachutes, excessive speed compromises safety.

CRASH-TOLERANCE PROTOCOL

Crash tolerance in `gdbm` begins with a naïve protocol that is obviously slow and subtly flawed. Fixing the flaws shows that crashproofing is tricky but need not be esoteric. Fixing the performance problem leverages an existing solution to an unrelated problem.

TABLE 1: DOCUMENTATION FOR MODERN SYSTEM CALLS AND UTILITIES

URLs append suffixes below to https://man7.org/linux/man-pages/		
man2/close.2.html	man3/malloc.3.html	man3/raise.3.html
man3/free.3.html	man2/mmap.2.html	man3/realpath.3.html
man2/fsync.2.html	man2/msync.2.html	man2/write.2.html
man2/ioctl_ficlone.2.html	man2/open.2.html	man2/writev.2.html
man1/cp.1.html	man1/dd.1.html	man8/filefrag.8.html

A `gdbm` database resides in a *data file*. Traditionally, applications call `gdbm_sync()` to persist a database from volatile memory down through the file system onto durable storage media. This slow operation makes sense only when the database is consistent. Therefore, the existing `gdbm_sync()` function serves an additional purpose when crash tolerance is enabled: It tells `gdbm` that the database is consistent and suitable for crash recovery.

Our first (incorrect) attempt at crash tolerance is simply to copy the data file immediately after `gdbm_sync()`, and then `fsync()` the copy; post-crash recovery uses the most recent copy. This approach is broken in two subtle ways: First, copies might be lost in a crash because their directory entries were not persisted. Second, after a crash it's uncertain whether the most recent copy-and-`fsync()` completed successfully; the copy might be malformed.

To solve the first problem, we ensure that a copy is visible in the file system after a crash by calling `fsync()` on the copy itself *and* on every *directory* on the `realpath()` between the copy and the root. To solve the second problem, we mark copies containing consistent data using their permission bits. To make a copy we: `open()` it *write-only*; fill it with the contents of the `gdbm` data file; `fsync()` it; then

change its permission bits to *read-only*; and finally `fsync()` the copy again to persist its permission bits. Crucially, file systems ensure that the metadata operation in the last step *is* failure-atomic and that the two `fsync()` calls persist data in program order. Post-crash recovery uses the most recent *readable* copy, which is sure to be consistent and persistent because it became readable *after* it was successfully filled with a consistent database and `fsync()`'d.

While correct, the protocol sketched above is impractical because it physically copies the entire data file on every `gdbm_sync()`. Copying a large data file to commit a few minor changes would be wasteful. The performance cost should be bounded not by the size of the entire database [$O(\text{data})$] but by the size of *changes* between consecutive `gdbm_sync()` calls [$O(\text{delta})$].

Fortunately, modern file systems support an efficient copy operation: *Reflink cloning* was motivated by virtual machines, which employ virtual disks that reside in files in a hypervisor's file system. Cloning reduces the overhead of multiple files containing near-identical copies of a virtual disk. Cloning doesn't physically copy storage blocks when a logical copy is made. Instead, the new clone shares data blocks with the original file. A copy-on-write mechanism ensures that subsequent changes to either file do not affect the other. The overall cost in time and storage grows with the changes to both files after the clone is made: $O(\text{delta})$, not $O(\text{data})$.

Figure 1 illustrates creating then changing a clone. On a Linux system with 4 KiB file-system blocks, I filled a 100-block file with random bytes, then cloned it with the commands

1

FIGURE 1: EXTENT MAPS OF ORIGINAL FILE AND CLONE

```
dd if=/dev/urandom bs=4096 count=100 > random
cp --reflink=always random random.copy
```

Figure 1a shows how the files map logical to physical blocks, courtesy the `filefrag` utility. Each file contains a

single 100-block *extent* [range of blocks], and both files map the same logical extent to the same physical extent (i.e., they share storage). Next I modified logical blocks 7 and 47 of the clone, for which copy-on-write allocated two new physical blocks (figure 1b). Both files now have five extents: three large shared extents and two private single-block extents; 98 percent of each file is backed by shared physical blocks.

Cloning is currently available on three file systems: OCFS2 (Oracle Cluster File System version 2), Btrfs, and XFS. For convenience we may install one of these reflink-capable file systems *inside an ordinary file* on a different file system such as ext4; the `gdbm` crash-tolerance documentation explains how.¹⁵ Critical operations such as cloning and `fsync()` will work as reliably as if the reflink-capable file system had been installed directly on dedicated storage.⁶

Crashproofing via cloning offers different performance tradeoffs than transactions based on write-ahead logging. Reflink copy-on-write tracks changes to blocks, which are typically 4 KiB, whereas many transaction systems track changes at finer granularity. Workload affects performance differently in these approaches.

IMPLEMENTATION DETAILS

To prototype crash-tolerant `gdbm` I changed four source files in release 1.20, adding fewer than 200 lines of code in total. I tested my prototype against injected crashes; an earlier implementation of the same basic protocol survived tens of thousands of power failures.⁹ Original `gdbm` author Phil Nelson and maintainer Sergey Poznyakoff reviewed my protocol design. Poznyakoff reviewed my code, repeated my

key-value store

haiku

crashproof KVS?
add ioctl(FICLONE)
to gdbm

tests, and merged my prototype into release 1.21.

Applications “opt in” to crash tolerance by passing `gdbm_failure_atomic()` a pointer to a `gdbm` database and the names of two *snapshot files* that will hold clones of the main data file; all three files must reside on the same reflink-capable file system. The `gdbm_failure_atomic()` call creates two empty write-only snapshot files with

the given names and `fsync()`s them and their enclosing directories up to the root. After `gdbm_failure_atomic()` returns, subsequent calls to `gdbm_sync()` use `ioctl(FICLONE)` to deposit clones of the main database file in one or the other snapshot file according to the protocol described earlier, alternating between the two.

Following a crash, applications recover by calling `gdbm_latest_snapshot()` to identify the most recently modified snapshot file. This file, which contains a consistent database reflecting the most recent successful `gdbm_sync()` call, simply replaces the main `gdbm` data file.

Because `gdbm` doesn’t use write-ahead logs, there’s no need to “garbage collect” logs that pile up during failure-free operation and no need to replay logs to reconstruct the database during post-crash recovery.

Crash tolerance depends on the proper functioning of system calls and of file and storage systems; all bets are off if, say, a storage device misbehaves after a power failure.¹⁸ Furthermore the `gdbm` library can’t protect an application from corrupting its own database—a substantial risk for C/C++ programs but a small risk for the many Python and Perl programs that use `gdbm`.

It’s easy to understand crash-tolerant `gdbm` and use it

correctly because it simply strengthens the semantics of a familiar interface: `gdbm_sync()` has always meant “persist the database”; crash tolerance adds, “and failure-atomically clone it too.” All changes to a `gdbm` database between consecutive `gdbm_sync()` calls constitute a transaction. By contrast, a `begin()`- and `end()`-transaction interface is more error prone: Bookends such as `malloc()/free()`, `lock()/unlock()`, and `open()/close()` are easy to mismatch.

DRILLING DEEPER

Like packing your own parachute, rolling your own crash tolerance is foolhardy without proper training and supervision. An off-the-shelf transaction system is usually better—if it’s fit for purpose.¹⁷ To be a smart shopper and a smart user, study the design and implementation of failure-atomic update mechanisms. Learn how to audit and break them. Understand how they depend on operating systems, file systems, and storage.^{12,13,18} If you rely on transactions, train as you fight: Test your entire hardware-software stack against the failures that it must survive in production.⁹

Bits

Grab `gdbm` release 1.21 from <https://ftp.gnu.org/gnu/gdbm/gdbm-1.21.tar.gz>. In file `gdbmsync.c`, check out the implementations of user-visible function `gdbm_failure_atomic()` and internal function `_gdbm_snapshot()`, called after `gdbm_sync()` finishes its traditional job. If you need a crash-tolerant key-value store, consider the new `gdbm` after reading all relevant documentation.^{14,15}

Drills

1. Standardized *performance* tests for databases abound; examples include the TPC benchmarks. Try to find crash-tolerance tests.
2. Find an application that already uses `gdbm`. Modify it to exploit crash tolerance.
3. Find software that uses some other transactional key-value store and modify it to use crash-tolerant `gdbm`. Test both against realistic failures.⁹ If both pass, compare performance. Construct workloads for which each is faster than the other. Measure sustainable long-term steady states; don't allow write-ahead logs to pile up.
4. If a distant customer asks a bank to transfer money from a savings account to a checking account, what's the relationship between the balance transfer transaction and the bank's reply informing the customer that the money has been moved? What can go wrong if the reply and the transaction aren't properly coordinated?
5. Reliability and security are closely related. What principles from the economics of computer security¹ apply to reliability? For example, who should evaluate crash-tolerance claims: database vendors, independent researchers, or end users?
6. What are the pros and cons of using `raise(SIGKILL)` to "abort a transaction" in `gdbm`?
7. One way to build a persistent hash table is to begin with an in-memory hash table [for example, a C++ STL `<map>`], and make it persistent.⁸ Adding crash tolerance as described earlier isn't difficult.⁷ Compare that approach to transactional key-value stores such as `gdbm`. Which supports a more natural programming style?

<https://xkcd.com/2347/>

Acknowledgments

GDBM author Phil Nelson, GDBM maintainer Sergey Poznyakoff, and *Programming Pearls* author Jon Bentley provided valuable feedback on drafts of this column. Lionel Debroux answered questions about key-value stores. XFS developer Christoph Hellwig answered questions about reflink cloning.

References

1. Anderson, R. 2001. Why information security is hard—an economic perspective. In *Annual Computer Security Applications Conference [ACSAC]*, December; <https://www.acsac.org/2001/papers/110.pdf>.
2. Bentley, J. 1987. Programming pearls: self-describing data. *Communications of the ACM* 30(6), 479–483; <https://doi.org/10.1145/214762.315733>. [Chapter 4 in *More Programming Pearls*, Addison-Wesley, 1988].
3. Bentley, J. August 2019. E-mail to colleague. Ken Thompson and Dennis Ritchie wrote the first UNIX operating system.
4. Debian popularity contest. July 2021; <https://popcon.debian.org/>.
5. Ghemawat, S., Dean, J. 2021. LevelDB; <https://github.com/google/leveldb>.
6. Hellwig, C. July 2021. Personal communication. Requirements for block devices and file-system semantics guarantee that filesystem-in-file installations function properly.
7. Kelly, T. 2019. Good old-fashioned persistent memory. *Usenix ;login:* 44(4), 29–34; https://www.usenix.org/system/files/login/articles/login_winter19_08_kelly.pdf.

8. Kelly, T. 2019. Persistent memory programming on conventional hardware. *acmqueue* 17[4]; <https://queue.acm.org/detail.cfm?id=3358957>.
9. Kelly, T. 2020. Is persistent memory persistent? *Communications of the ACM*, 63[9], 48-54; <https://dl.acm.org/doi/10.1145/3397882>. To avoid paywall: *acmqueue* 18[2]; <https://queue.acm.org/detail.cfm?id=3400902>.
10. Kernighan, B. W., Pike, R. 1984. *The UNIX Programming Environment*. Prentice Hall.
11. Oracle Berkeley DB. July 2021; <https://www.oracle.com/database/berkeley-dbl>.
12. Pillai, T. S., et al. 2014. 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]*; <https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-pillai.pdf>.
13. Pillai, T. S., et al. 2015. Crash consistency. *acmqueue* 13[7]; <https://dl.acm.org/doi/pdf/10.1145/2800695.2801719>.
14. Poznyakoff, S. 2021. Crash-tolerant gdbm [September]. Release 1.21 [September 2021]. Source tarball: <https://ftp.gnu.org/gnu/gdbm/gdbm-1.21.tar.gz>. Diff against 1.20: <https://git.gnu.org.ua/gdbm.git/rawdiff/?id=v1.21&id2=v1.20>
15. Poznyakoff, S. 2021. Crash tolerance. gdbm Manual [July]; <https://www.gnu.org.ua/software/gdbm/current/Crash-Tolerance.html>.
16. Unix dbm manual; http://man.cat-v.org/unix_7th/3/dbm.
17. Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E. S., Zhao, B. W., Singh, S. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th*

Usenix Symposium on Operating Systems Design and Implementation (OSDI); https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-zheng_mai.pdf.

[Note that an errata sheet is provided separately.]

18. Zheng, M., Tucek, J., Qin, F., Lillibridge, M. 2013. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th Usenix Conference on File and Storage Technologies (FAST)*; <https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf>.

At the drop zone, former skydiver Terence Kelly (tpkelly@acm.org) went by the nickname “Crash.”

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.

SHAPE THE FUTURE OF COMPUTING!

Join ACM today at acm.org/join

**BE CREATIVE. STAY CONNECTED.
KEEP INVENTING.**

Association for
Computing Machinery

Advancing Computing as a Science & Profession