SQLean: Database Acceleration via Atomic File Update

Rajat Verma, Anton Ajay Mendez, Sandya Mannarswamy, Terence P. Kelly, Brad Morrey

Hewlett Packard Labs
HPL-2015-103

Keyword(s):
SQLite; Android; failure-atomic msync; performance; databases; storage; file systems

Abstract:
Application software must not allow untimely crashes to harm user data. Yet remarkably, most file systems do not support failure-atomic application data update. Complex databases are therefore used to protect application data from failure. Such databases suffer poor performance because they duplicate the underlying file system's metadata protection mechanisms. Unlike conventional file systems, HP's Advanced File System (AdvFS) supports efficient failure-atomic file updates. Applications can use this feature directly to achieve crash resilience with reduced complexity and improved performance. The benefits multiply when we improve popular infrastructure, e.g., the widely used SQLite database at the foundation of the Android data stack. Our modified SQLite database, "SQLean," exploits AdvFS's atomic file update capability to improve performance by more than 2×.

External Posting Date: December 14, 2015 [Fulltext]
Internal Posting Date: December 14, 2015 [Fulltext]

SQLean: Database Acceleration via Atomic File Update

Rajat Verma,[1] Anton Ajay Mendez,[1] Stan Park,[2] Sandya Mannarswamy,[1] Terence Kelly,[2] Brad Morrey[2]

Hewlett-Packard Enterprise [1]Storage Division [2]Labs

{rajatv, ajay.mendez, stan.park, terence.p.kelly, brad.morrey}@hpe.com

Abstract

Application software must not allow untimely crashes to harm user data. Yet remarkably, most file systems do not support failure-atomic application data update. Complex databases are therefore used to protect application data from failure. Such databases suffer poor performance because they duplicate the underlying file system's metadata protection mechanisms. Unlike conventional file systems, HP's Advanced File System (AdvFS) supports efficient failure-atomic file updates. Applications can use this feature directly to achieve crash resilience with reduced complexity and improved performance. The benefits multiply when we improve popular infrastructure, e.g., the widely used SQLite database at the foundation of the Android data stack. Our modified SQLite database, "SQLean," exploits AdvFS's atomic file update capability to improve performance by more than $2\times$.

Problem statement

Applications must protect user data from corruption or destruction by failures such as process crashes, OS kernel panics, and power outages. Applications ultimately store data in file systems, which in turn store data on durable media (e.g., hard disks or solid state drives). Remarkably, however, conventional file systems do not provide convenient and efficient mechanisms for failure-atomic updates of user data in files. Many applications therefore interpose a transactional database between application logic and file systems. For example, the popular SQLite database is used in applications ranging from Web browsers to avionics [7] and is the data-management foundation of the vast majority of applications on Android consumer devices [5].

Databases between applications and file systems can impair both reliability and performance. The inherent complexity of a database built atop the Spartan facilities of conventional file systems provides fertile ground for bugs; one recent study of the most widely used commercial and open source databases found that *none* correctly tolerate sudden crashes [8]. Poor database performance results from duplication of effort: Transactional databases employ logging/journaling mechanisms to protect *application data* from crashes, while beneath them the file system uses similar mechanisms to protect *file system metadata*. This redundant "journaling of journal" increases storage write traffic and substantially degrades application performance [3].

Fortunately, the problem is localized at the database/file system boundary. Furthermore the popularity of transactional databases is heavily skewed, offering tremendous leverage: If we streamline a single highly popular transactional database, the benefits will propagate effortlessly upward to the multitude of applications that run atop it.

Our solution

Our solution eliminates the root cause of the problem: redundant transactional mechanisms in both the database layer and underlying file system. Specifically, we have added to a production file system simple, efficient, and comprehensive support for failure-atomic *application data* update. This allows us to *remove* transaction support from a modified database, which relies instead on the transactional capabilities of the file system.

The file system containing our novel transactional user-data update mechanism is HP's Advanced File System (AdvFS), a descendant of DEC's Tru64 file system that has been ported to Linux. AdvFS will be the foundation of several HP Storage products starting in 2015. AdvFS now includes two failure-atomic mechanisms for updating application data—*not* merely file system metadata. The first mechanism operates upon individual files. If a file is opened with a special new flag, `open(O_ATOMIC)`, then the file is guaranteed to reflect the most recent successful file synchronization system call, regardless of failures. `O_ATOMIC` thus provides programmers with a simple and convenient way to update files failure atomically using either the `write()`/`fsync()` or `mmap()`/`msync()` families of interfaces (or both). Rather remarkably, nearly all conventional file systems provide no such guarantee, nor

any other convenient mechanism for failure-atomically updating files. AdvFS implements single-file failure-atomic updates by leveraging its efficient support for *clones* (writable snapshots of individual files). AdvFS's second failure-atomic mechanism updates *multiple* files. A novel "`syncv()`" call allows higher-level software to specify *bundles* of `fsync()` and `msync()` operations on different files; each bundle of updates occurs atomically, regardless of failures. Our FAST 2015 paper describes in detail AdvFS's novel failure-atomic file data update mechanisms and how they can fortify applications against crashes [6].

We have modified the popular SQLite database to exploit AdvFS's new support for transactional file update. We chose SQLite because it is very widely used in applications ranging from desktop software (e.g., the Chrome and Firefox Web browsers) to avionics (on Airbus A350 XWB aircraft); scripting languages including Python and Tcl/Tk include extensive support for SQLite [7]. Furthermore, nearly all Android applications that maintain persistent state employ SQLite [5].

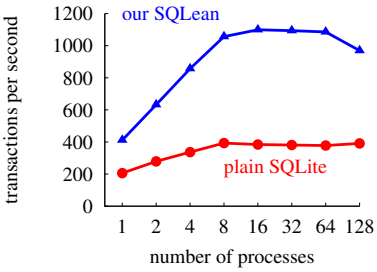
SQLite stores an entire database in a single file, which made it very easy to supplant SQLite's original transactional mechanisms: We simply change the *one line of code* where SQLite opens a database file, adding AdvFS's new `O_ATOMIC` flag; we then disable SQLite's own transactional machinery with a *single command* to the database ("`PRAGMA journal_mode=OFF`"). The net result, which we call "SQLean," is an SQLite database that relies entirely upon AdvFS's simple and efficient failure-atomic file update mechanism.

Evidence the solution works

We have previously reported extensive experiments demonstrating the correctness and efficiency of AdvFS's failure-atomic file update mechanisms [6]. In this paper we describe an analogous evaluation of SQLean: We verify that SQLean protects application data from failure and we compare our SQLean with the original SQLite in terms of performance and implementation complexity.

Our correctness and performance tests employ a simple TPC-B-like transactional workload that repeatedly transfers money between bank accounts. The database must ensure that transactions are atomic; e.g., failures must neither create nor destroy money. We evaluate SQLean's correctness by subjecting a running database to two kinds of failures, injected OS kernel panics and whole-system power interruptions, and we verify "conservation of money" following each failure. A scriptable external power supply enables us to cut electric power to a running computer, precisely as if the machine were physically unplugged. We inject kernel panics using a facility in the Linux `/proc/` pseudo file system. Our SQLean recovered successfully from 2,048 power interruptions and 480 kernel panics.

Our performance comparisons between SQLean and the original SQLite measure both light-load transaction latency and peak transaction throughput under heavy load. We ran our tests on an HP server with twelve 1.8 GHz Xeon E5-2450L cores and 92 GB of DRAM; its RAID controller had a 1 GB battery-backed cache configured as 90% write cache and a 1 TB 7200 RPM SAS hard drive. In terms of per-transaction latency under light load, our SQLean is more than 2.5× faster than SQLite by several measures: 1.28 ms vs. 3.63 ms mean latency, 1.52 ms vs. 3.33 ms 99th percentile latency, and 1.80 ms vs. 4.65 ms maximum latency. As shown at right, peak throughput under heavy load is also more than 2.8× higher for our SQLean versus the original SQLite.



Our SQLean is simpler than the original SQLite because the former dispenses with the latter's complex transactional mechanisms. We quantify the reduction in effective complexity by compiling both SQLite and SQLean to measure run-time code coverage and then running our transactional workload. Of 522 functions invoked by SQLite at run time, 12% are *not* invoked by SQLean; these functions perform database journaling and related activities, which SQLean does not require. The reduction in lines of code executed is in the same ballpark: 12.7%.

We also traced system calls and measured block storage device activity during execution to understand the root cause of the performance difference. SQLite performs 2.57× more `write()` system calls, and 3× more `fsync()` calls than SQLean. However, SQLite writes only 1.4× more 512 byte sectors, indicating that the performance hit comes from the synchronous I/O activity.

Competitive approaches

AdvFS's failure-atomic file update mechanisms generalize the research prototype that inspired them: failure-atomic `msync()` (FAMS) implemented in a modified Linux kernel [4]. AdvFS's `O_ATOMIC` mechanism generalizes FAMS by supporting `fsync()` as well `msync()`; AdvFS's `syncv()` extends FAMS to multiple files. AdvFS's mechanisms inherit all of the ergonomic attractions of FAMS: It is remarkably easy to make legacy software crash-resilient by "sliding FAMS beneath it." For example, a user-space implementation of FAMS enabled HP Indigo digital printing presses to become crash-resilient, reducing recovery times following power outages by orders of magnitude [1]. However, by implementing failure-atomic file update in the *file system*, AdvFS obtains several important advantages over prior approaches: Unlike both the kernel and user-space implementations of FAMS, AdvFS's `O_ATOMIC` avoids double-writes of data to storage. Whereas the size of transactions supported by the kernel-based FAMS prototype was limited by the size of the file system journal, transaction size on AdvFS's clone-based implementation is limited only by free space in the file system. Finally, AdvFS's file system implementation is much easier to adopt than a kernel implementation because it runs on unmodified Linux kernels.

FusionIO provides a failure-atomic `writev()` mechanism for use with their high-performance flash-based storage devices [2]. AdvFS's file-system-based implementation requires no special hardware and runs on any block storage device or storage array. Additional comparisons of AdvFS with previous hardware- and software-based atomic data update mechanisms are available in our FAST 2015 paper [6].

Current status & next steps

The AdvFS file system with `O_ATOMIC` and `syncv()` features is fully implemented, thoroughly tested, and is on Plan-of-Record to ship to customers in 2015 with HP Storage appliance products. AdvFS and SQLean are freely available within HP and we welcome collaboration with colleagues interested in these technologies. Particularly promising opportunities employ AdvFS's failure-atomic file update features in a wider range of use cases. The most radical possibilities involve not *accelerating* database layers but *eliminating* them from software stacks. Prior research has shown that failure-atomic `msync()` (FAMS) can be the foundation of an alternative software stack centered on persistent heaps and other higher-level abstractions [4, 6]. Applications running on this alternative software stack manipulate durable data *directly, in-place, in-memory* via ordinary LOAD and STORE instructions, using FAMS to transactionally update the persistent heap. This style of programming offers substantial benefits in terms of simplicity and performance, and it closely approximates the most natural way of programming emerging byte-addressable non-volatile memory (NVM), e.g., HP's Memristor. AdvFS's generalized FAMS therefore provides a smooth transition path from today's programming practices to those suitable for tomorrow's NVM.

References

[1] A. Blattner et al., "Generic Crash-Resilient Storage for Indigo," HP Labs Tech Report HPL-2013-75, [PDF]

[2] FusionIO, "NVM Primitives Library" (see `nvm_batch_atomic_operations`), [PDF]

[3] S. Jeong et al., "I/O Stack Optimization for Smartphones," USENIX Annual Tech 2013, [PDF]

[4] S. Park et al., "Failure-Atomic `msync()`," EuroSys 2013, [PDF]

[5] R. Spahn et al., "Pebbles: Fine-Grained Data Management Abstractions," OSDI 2014, [PDF]

[6] R. Verma et al., "Failure-Atomic Updates of Application Data in a Linux File System," FAST 2015, [URL]

[7] Well-Known Users of SQLite, `https://www.sqlite.org/famous.html`

[8] M. Zheng et al., "Torturing Databases for Fun and Profit," OSDI 2014, [PDF]