

Reducing SQLite Journaling Overhead on EXT4 through Reflink-based Atomic Staging

JINHYEOK YANG, KAIST, Republic of Korea

SQLite relies on database-level journaling mechanisms such as rollback journaling and write-ahead logging to guarantee crash consistency. However, when SQLite runs on journaling file systems such as EXT4, database-level journaling and file-system-level journaling coexist. This redundancy causes additional synchronization and write amplification, a phenomenon often referred to as Journaling of Journal.

This work investigates whether atomic-update semantics similar to F2FS Multi-block Atomic Write can be approximated on EXT4. We design a reflink-based atomic staging mechanism that redirects updates to staging inodes and publishes modified extents during commit. The goal is to reduce redundant journaling overhead while preserving old-or-new crash consistency.

We implement a prototype inside EXT4 and evaluate it using SQLite TPC-C workloads. Our prototype achieves 197.2K TpmC on the SQLite TPC-C workload, which is within 4.2% of the non-durable EXT4 upper bound and higher than F2FS Multi-block Atomic Write, which achieves 189.1K TpmC. Compared with conventional SQLite journaling modes on EXT4, the proposed mechanism improves throughput by 12.0% over WAL, 26.2% over TRUNCATE, and 34.1% over DELETE. However, the write-volume results show that the current design still generates 8.0KB of writes per transaction, matching the EXT4 upper-bound configuration but remaining higher than F2FS Multi-block Atomic Write, which generates 1.5KB per transaction. These results show that reflink-based atomic staging on EXT4 can substantially reduce journaling overhead and approach the performance of the unsafe EXT4 upper-bound configuration, but further optimization is required to reduce physical write amplification to the level of native F2FS atomic writes.

1 Introduction

SQLite is the most popular ultra-lightweight, high-performance open-source **Relational Database Management System (RDBMS)**. Since it is included by default in major operating systems such as Android, macOS, and Windows, it is no exaggeration to say that it is currently the most widely used DBMS in the world. Furthermore, its simple structure—which stores all database information in a single file—offers high convenience and portability.

Despite being an ultra-lightweight database, SQLite supports journaling features to ensure a high level of reliability. Specifically, it offers two types of journaling methods: **rollback journaling and write-ahead logging (WAL)**. Both methods share the common feature of using an actual file as a journal or log. Therefore, to ensure crash consistency, SQLite must perform `fsync` on these journal files and the database file, which incurs additional synchronization overhead at the filesystem layer.

The **EXT4 filesystem**, widely used on Linux-based systems, also employs journaling to ensure its own crash consistency. In EXT4, a lower-level layer called JBD2 handles journaling, and in the default ordered mode, it primarily ensures metadata consistency. When SQLite calls `fsync` in this context, the file system is forced to wait for the writeback of dirty data associated with the file and the commit of the JBD2 journal transaction. This process occurs repeatedly not only for database files but also for the journal files generated during the SQLite journaling process; this is known as double journaling, or the JOJ (Journaling of Journaling) problem [3].

F2FS (Flash-Friendly File System) solves this problem in an interesting way. F2FS provides **Multi-Write Atomicity (MAW)**, which ensures that multiple writes occurring within a single atomic write session are applied in an all-or-nothing manner following a crash [1]. By utilizing this, SQLite can delegate the atomicity of multiple page updates to the file system without writing the rollback journal or WAL to disk. This approach naturally leads to the question of whether a similar atomic write mechanism could be provided in the more widely used EXT4.

In this study, we introduce a **reflink-based atomic staging mechanism** to EXT4. When a database update occurs, instead of overwriting the original file directly, only the modified portions are recorded in a temporary staging inode. At the time of commit, the system then atomically publishes the changes stored in the staging inode to the original database file. The goal is to achieve performance improvements while maintaining crash consistency, reducing the costs of application-level journaling and repeated fsync operations previously performed by SQLite, and preserving EXT4's existing structure as much as possible.

2 Background

2.1 EXT4 Journaling

EXT4 is a widely used file system in Linux. EXT4 manages files using inodes, which store both the file's metadata and the disk block locations where the file is stored. The data structure used for this purpose is the **extent tree**, which provides a simple way to represent where each contiguous range of file blocks is physically located.

EXT4 employs journaling to ensure reliability at the operating system level. **JBD2 (Journaling Block Device 2)** handles journaling using a dedicated journal area on the disk. JBD2 groups multiple metadata changes into a single transaction, writes the modified metadata blocks to the journal, and then writes a commit block to indicate that the changes have been completed. This allows the operating system to determine whether the changes are complete in the event of a crash and decide whether to replay them.

While this journaling in EXT4 is primarily intended to ensure filesystem metadata consistency, it does not completely eliminate the need for the journaling in SQLite, which will be discussed later. Since the transaction units guaranteed by EXT4 differ from those guaranteed by SQLite, a distinct journaling method is required. However, this process leads to the so-called **JOJ (Journaling of Journal) problem**, where the operating system itself must journal the journaling process, making it a major cause of performance degradation [3].

2.2 SQLite Journaling

Databases typically use journaling as a technique to ensure crash consistency. This technique ensures that, in the event of an unexpected failure, the contents of the file remain intact and can be correctly restored to their pre-modification state or a specific point in time. SQLite supports multiple journaling modes, including **Rollback Journaling** (for operations such as DELETE and TRUNCATE) and **WAL Journaling**.

Simply put, **Rollback Journaling** is a method that saves the pre-modification content to a file and preserves it until the modification is complete. Before starting a modification, SQLite creates a journaling file in advance and copies the existing content into it. By safely syncing this journal to disk before making any changes, it is guaranteed that, under any circumstances, either the original file or the journal file will always retain its original state as synced to disk. Once the changes are safely completed, the journal file is either deleted or invalidated, depending on the processing method.

WAL Journaling was designed to address the shortcomings of rollback journaling. Rollback journaling, which first fsync's the journal file to disk and then fsync's the original file, inherently involves a high computational load. It also has the drawback of holding locks (such as EXCLUSIVE) that block reads for extended periods, which can adversely affect other readers. To address these issues, WAL journaling introduced a WAL log file to which changes are appended. Changes to new pages are appended to the WAL log file, and for readers, the progress of these changes is recorded in a shared memory file. The advantage is that merging with the actual original file is performed at the checkpoint, thereby reducing the fsync load. However, the issues of the continuously growing WAL file and the necessity of performing fsync at the checkpoint remain.

Prior studies have shown that both rollback journaling and WAL journaling still incur substantial overhead due to repeated synchronization and filesystem-level journaling interactions [1, 3]. Cheon and Won reported that, when inserting a single 100 B record on EXT4, SQLite actually updates only 12 KB of database pages, but Rollback

Journaling in PERSIST mode generates a total of 48 KB of write I/O because of duplicated journal writes and EXT4 Journaling-of-Journal (JOJ) overhead [1, 3]. Even WAL mode, despite reducing synchronization cost and improving concurrency, still produces 36 KB of total write I/O, which is approximately 2× larger than the actual data update size due to filesystem journaling and checkpoint overhead. Furthermore, WAL mode continuously appends updates to the WAL file and periodically performs checkpoints, which introduce additional read/write operations and fsync overhead. The paper also notes that WAL files may grow up to 4 MB per database connection, causing non-negligible storage overhead. These observations suggest that existing SQLite journaling mechanisms still suffer from considerable write amplification and synchronization overhead despite various optimizations.

2.3 F2FS Multi-block Atomic Write

Unlike EXT4, F2FS employs a log-structured file system architecture [2]. While EXT4 modifies data by continuously overwriting the original location, F2FS writes data to new locations and maintains consistency through checkpoints and roll-forward recovery rather than journaling. F2FS divides the entire disk into 2MB segments and manages metadata using the SIT (Segment Information Table), NAT (Node Address Table), and SSA (Segment Summary Area). Unlike EXT4's extent tree, the F2FS NAT uses inodes, direct inodes, and indirect inodes to map these locations to the NAT. This resolves the "wandering tree problem," where write amplification occurred because all metadata, including the parent metadata, had to be updated after a modification.

F2FS provides a feature called **MAW (Multi-block Atomic Write)** [1]. This feature ensures the "all-or-nothing" nature of SQLite's journaling at the file system level. Instead of committing writes directly to the file, it collects new data through F2FS's **COW (Copy-On-Write)** path and makes the update visible atomically at commit time. While this is a representative method for solving JOJ in F2FS, the issue of relying solely on a specific filesystem's features remains.

2.4 COW(Copy-On-Write) Method

Unlike the conventional method of modifying a file directly, **COW (Copy-On-Write)** is a structure in which a modified version is first written to a different block or inode, and only the metadata pointer is updated at the end. This method makes it easy to guarantee atomicity.

Reflink is one of the techniques that utilizes this COW feature. If a file is copied to create a new file, this technique initially allows the two files to share the same blocks, and when modifications are made, only the modified portions are moved to different blocks.

3 Motivation

The main motivation of this work is to reduce the cost caused by redundant journaling, known as the Journaling of Journal (JOJ) problem. When SQLite runs on EXT4, each transaction may involve several expensive persistence steps:

- (1) writing SQLite journal records,
- (2) synchronizing the SQLite journal,
- (3) overwriting database pages,
- (4) synchronizing the database file, and
- (5) committing EXT4 metadata through JBD2.

This sequence creates unnecessary write amplification because the database and the file system both attempt to provide consistency.

To address this issue, F2FS adopted the MAW method described above. However, F2FS is not as widely deployed as EXT4, and SQLite's batch-atomic-write optimization relies on a filesystem-specific feature that is currently not supported on EXT4. Therefore, to resolve this issue, we aim to support COW-reflink-based MAW on EXT4 by

emulating this process. However, EXT4 was not originally designed as a copy-on-write or log-structured file system. Therefore, supporting atomic multi-block updates efficiently on EXT4 is challenging.

4 Design and Implementation

In this study, we implemented a COW-reflink-based MAW on top of EXT4. This new mode is designed to provide old-or-new crash consistency for SQLite-style multi-page updates without requiring SQLite journaling or sync waits, and is implemented by emulating the structure of F2FS.

4.1 Design Goal

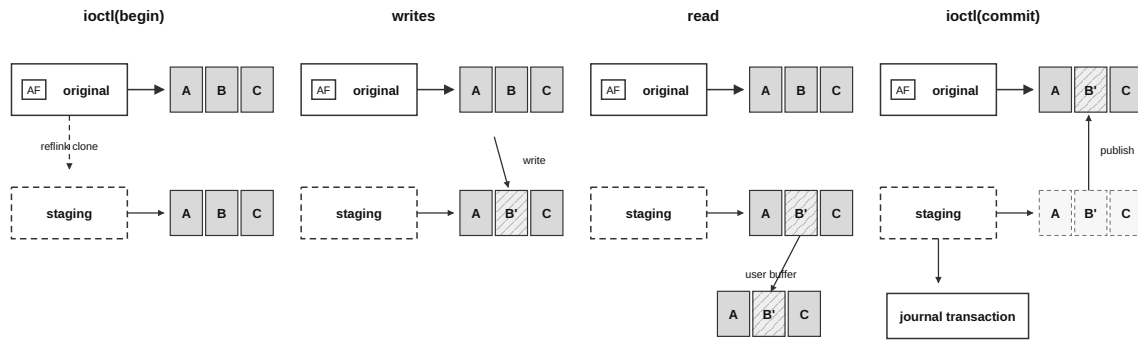


Fig. 1. COW based EXT4-Reflink Overview.

In EXT4, this implementation was achieved through a staging inode structure. The original file is copied via a reflink clone, and modifications made during an atomic session are handled using the COW method, where only the modified portions are allocated to new blocks.

One key point to note is that reflink clones offer a significant performance advantage over actual deep copies. While a deep copy requires copying the entire contents of a file, a reflink clone only needs to set metadata to share blocks initially, resulting in a noticeable reduction in overhead.

If the owner reads while such an atomic session is active, the mapping is switched to the staging inode only for that read, making it appear as if the original DB file has been modified. Consequently, the owner can read the new page they just wrote, while external readers still read the existing inode and thus see the old state.

Writes occurring within the atomic session are redirected to write to the new mapping in the staging inode rather than the original one. This approach has the advantage of reusing the standard ext4 buffered write mechanism. However, it introduces many restrictions, such as unsorted writes, because the logical states of the original and staging inodes must always remain identical.

A commit occurs when the atomic session ends. During the commit, the system collects the extents in the staged inode that are not currently shared with the original. At this point, splitting occurs at the EOF boundary; for extents newly appended beyond the EOF, a pre-zeroed placeholder is created to ensure consistency. Ultimately, the commit is finalized by simply swapping the extents, which changes only the block mapping. Since this operation occurs at the metadata level, it is significantly more efficient than a standard copy. The process concludes by detaching the staging inode.

5 Evaluation

5.1 Experimental Setup

To compare actual performance, we designed an experiment to examine how performance varies across different file systems and journaling modes under the same TPC-C workload. To compare the performance differences between our implemented EXT4-Reflink and the MAW mode of the existing F2FS, as well as the WAL and Rollback Journaling modes of the existing EXT4, we use the standard TPC-C workload to compare the TPS (Transactions Per Second), TpmC, commit tail latency (p99), and write volume per transaction for each case.

5.1.1 Experiment Environment. We ran experiments on a Supermicro SYS-1029U-TR4 server with a Supermicro X11DPU motherboard and two Intel Xeon Gold 5220R processors (24 cores per socket, 48 physical cores / 96 hardware threads total). The host was equipped with 192 GiB of ECC DDR4 RDIMM memory configured at 2666 MT/s, plus Intel Optane DC persistent memory modules installed in the platform. The benchmark target device, exposed to the guest as `/dev/nvme0n1`, was a Samsung SSD 970 EVO 500GB NVMe SSD passed through via VFIO from host PCI device `0000:d8:00.0` over a PCIe Gen3 x4 link.

5.1.2 Experiment Variables. We conducted the experiment under a single-writer condition. We configured a total of four warehouses in TPC-C to determine the transaction workload. To ensure a consistent environment, we minimized the impact of previous experiments by performing the same process of discarding, mkfs, mounting, and benchmarking the same physical NVMe device for each experiment. At the same time, we maintained a ramp-up time of 30 seconds and a cooldown time of 20 seconds for each run to minimize the impact of CPU state. Under these conditions, we conducted measurements for 120 seconds per experiment.

5.2 Crash Consistency Test

We also performed a QEMU-based crash consistency test to validate the correctness of the modified EXT4 kernel. The test creates an EXT4 disk image, boots it with QEMU, performs atomic updates on a target file, and forcibly terminates QEMU at predefined crash points using `kill -9`. The same disk image is then rebooted and verified.

The verifier checks that the recovered file contains either the complete old version or the complete new version. Any mixed state across pages or torn state inside a 4KB page is treated as a failure. We tested several crash points, including pre-commit, post-commit-before-fsync, mid-commit publication, mixed overwrite-and-append cases, abort cases, and reflink-enabled cases. In the remaining logs, all tested scenarios resulted in PASS_OLD, meaning that the file always recovered to the complete old version without exposing partially published or torn data. This result supports the correctness of the atomic staging and publish path used in our prototype.

5.3 Performance Results

All configurations in Figure 2 are evaluated under SQLite `synchronous=OFF`. In this setting, `ext4-off` should be interpreted only as an unsafe upper-bound baseline because it disables SQLite-level synchronization without a filesystem-level atomic session. In contrast, `f2fs-maw` and `ext4-reflink` use filesystem-level atomic sessions, so `synchronous=OFF` removes redundant SQLite-level synchronization rather than removing the atomicity mechanism itself.

Figure 2 summarizes the transaction throughput and I/O volume of each configuration under the TPC-C workload. The baseline `ext4-off` shows the highest throughput, achieving 205.8K TpmC. The proposed `ext4-reflink` configuration achieves 197.2K TpmC, which is close to the `ext4-off` upper bound. It also outperforms `f2fs-maw`, which achieves 189.1K TpmC. In contrast, conventional SQLite journaling modes on Ext4 show lower throughput: `ext4-wal`, `ext4-truncate`, and `ext4-delete` achieve 176.0K, 156.3K, and 147.1K TpmC, respectively.

The I/O volume result shows a different trend. `f2fs-maw` generates the lowest write volume, with only 1.5KB per transaction. On the other hand, both `ext4-off` and `ext4-reflink` generate 8.0KB per transaction. The

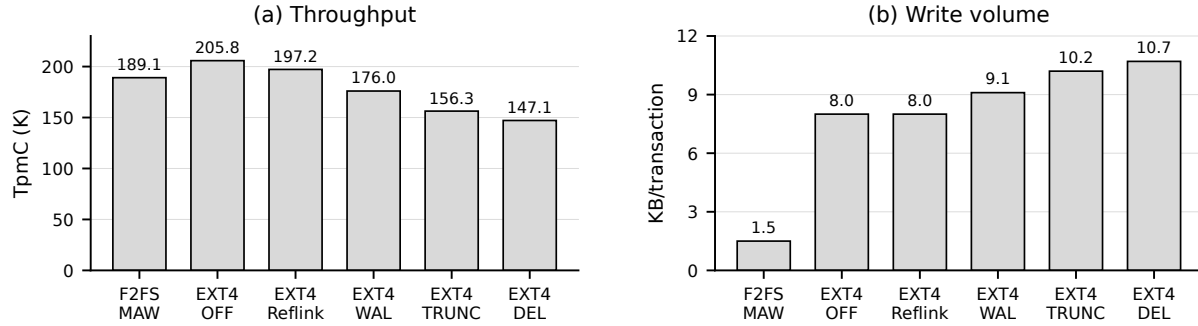


Fig. 2. TPC-C throughput and write volume.

traditional SQLite journaling modes generate even higher write volume, with ext4-wal, ext4-truncate, and ext4-delete producing 9.1KB, 10.2KB, and 10.7KB per transaction, respectively.

5.4 Analysis

The result indicates that the proposed ext4-reflink design can approach the throughput of the non-durable ext4-off configuration. The throughput gap between ext4-off and ext4-reflink is only about 4.2%, which suggests that the reflink-based staging mechanism does not introduce significant runtime overhead in the tested workload. Moreover, it achieves higher throughput than f2fs-maw, showing that the proposed Ext4-based design can be competitive with F2FS multi-write atomicity in terms of transaction processing performance.

However, the I/O volume result reveals a limitation of the current design. Although ext4-reflink improves throughput, its write volume remains the same as ext4-off. This means that the current implementation reduces the overhead of SQLite journaling paths, but does not yet reduce the physical write volume to the level of F2FS. In contrast, f2fs-maw shows much lower I/O volume because it provides native multi-write atomicity and allows SQLite to avoid application-level journaling with less write amplification.

Therefore, the main benefit of the current ext4-reflink implementation is performance rather than write-volume reduction. The result suggests that the proposed approach successfully removes much of the overhead caused by conventional journaling modes, but still inherits the block-mapping and metadata-update costs of Ext4.

6 Discussion

These results show that providing multi-write atomicity inside the file system can significantly improve SQLite performance by reducing the need for application-level journaling. Conventional SQLite journaling modes such as DELETE, TRUNCATE, and WAL suffer from additional writes caused by journal creation, journal synchronization, database updates, and journal invalidation. As a result, they show both lower throughput and higher I/O volume compared to the atomic-write-based configurations.

The comparison between ext4-reflink and f2fs-maw highlights an important trade-off. ext4-reflink achieves higher throughput, but it does not reduce I/O volume as effectively as F2FS. This is because F2FS provides atomicity through its native log-structured and copy-on-write design, while Ext4 must emulate atomic publication through staging and metadata remapping. Consequently, Ext4 still requires updates to block mappings and related metadata, which limits its ability to reduce write amplification.

Another important point is that ext4-off should be interpreted as an upper-bound performance result. Since it disables SQLite synchronization without a filesystem-level atomic-write mechanism, it does not provide the same crash-consistency semantics as f2fs-maw or ext4-reflink. In contrast, f2fs-maw and ext4-reflink rely on

filesystem-level atomic sessions, so `synchronous=OFF` mainly removes redundant SQLite-level synchronization rather than removing the atomicity mechanism itself.

Overall, the current result suggests that an Ext4-based multi-write atomicity mechanism is promising, but further optimization is required to reduce I/O volume. A more efficient design should avoid unnecessary metadata updates during commit and reduce the cost of publishing staged writes to the original database file.

7 Conclusion

In this work, we evaluated the performance and I/O behavior of an Ext4-based reflink COW approach for supporting SQLite-style multi-write atomicity. The experimental results show that `ext4-reflink` achieves 197.2K TpmC, which is close to the 205.8K TpmC of `ext4-off` and higher than the 189.1K TpmC of `f2fs-maw`. This demonstrates that the proposed design can provide competitive transaction throughput with low runtime overhead.

However, the I/O volume result shows that the current design still generates 8.0KB of writes per transaction, which is much larger than the 1.5KB per transaction of `f2fs-maw`. This indicates that the current Ext4-based implementation improves throughput but does not yet fully eliminate write amplification. The remaining overhead mainly comes from the need to publish staged updates through Ext4 metadata changes.

Therefore, the proposed approach is effective as a performance-oriented step toward multi-write atomicity on Ext4, but additional work is needed to reduce physical write volume. Future work should focus on minimizing metadata updates during atomic commit and designing a more lightweight publish mechanism for staged database writes.

Acknowledgments

First, I would like to express my gratitude to my advisor, Dr. Lee Jae-Hwan, for his invaluable assistance throughout this research project. I would also like to thank Master's student Kim Min-Joo and the many KAIST undergraduate students who offered their support and encouragement, making this research possible. Above all, I would like to extend my deepest gratitude to Professor Kwon Young-Jin for providing me with this research opportunity.

References

- [1] Seungyong Cheon and Youjip Won. 2017. Exploiting Multi-Block Atomic Write in SQLite Transaction. In *Proceedings of the International Conference on High Performance Compilation, Computing and Communications (HP3C '17)*. Association for Computing Machinery, New York, NY, USA, 23–27. doi:10.1145/3069593.3069616
- [2] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. USENIX Association, 273–286.
- [3] Kai Shen, Stan Park, and Meng Zhu. 2014. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*. USENIX Association, 287–293.