DASH: Database Shadowing for Mobile DBMS

Youjip Won[1] Sundoo Kim[2] Juseong Yun[2] Dam Quang Tuan[2] Jiwon Seo[2]

[1]KAIST, Daejeon, Korea [2]Hanyang University, Seoul, Korea

ywon@kaist.ac.kr {sksioi12|yjs05|damquangtuan|seojiwon}@hanyang.ac.kr

ABSTRACT

In this work, we propose *Database Shadowing*, or *DASH*, which is a new crash recovery technique for SQLite DBMS. DASH is a hybrid mixture of classical shadow paging and logging. DASH addresses four major issues in the current SQLite journal modes: the performance and write amplification issues of the rollback mode and the storage space requirement and tail latency issues of the WAL mode. DASH exploits two unique characteristics of SQLite: the database files are small and the transactions are entirely serialized. DASH consists of three key ingredients *Aggregate Update*, *Atomic Exchange* and *Version Reset*. Aggregate Update eliminates the redundant write overhead and the requirement to maintain multiple snapshots both of which are inherent in the out-of-place update. Atomic Exchange resolves the overhead of updating the locations of individual database pages exploiting order-preserving nature of the metadata update operation in modern filesystem. Version Reset makes the result of the Atomic Exchange durable without relying on expensive filesystem journaling. The salient aspect of DASH lies in its simplicity and compatibility with the legacy. DASH does not require any modifications in the underlying filesystem or the database organization. It requires only 451 LOC to implement. In Cyclomatic Complexity score, which represents software complexity, DASH renders 33% lower (simpler) mark than PERSIST and WAL modes of SQLite. We implement DASH for SQLite on Android and extensively evaluate it on widely used smartphone devices. DASH yields 4× performance gain over PERSIST mode (default journaling mode). Compared to WAL mode (the fastest journaling mode), DASH uses only 2.5% of the storage space on average. The transaction latency of DASH at 99.9% is one fourth of that of WAL mode.

PVLDB Reference Format:
Youjip Won, Sundoo Kim, Juseong Yun, Dam Quang Tuan, Jiwon Seo. Database Shadowing Revisited: From A Mobile Perspective. *PVLDB*, 12(7): 793-806, 2019.
DOI: https://doi.org/10.14778/3317315.3317321

Proceedings of the VLDB Endowment, Vol. 12, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3317315.3317321

1. INTRODUCTION

Crash recovery is a vital part of DBMS design. Algorithms for crash recovery range from naive full-file shadowing [15] to the sophisticated ARIES protocol [38]. Most enterprise DBMS's, e.g., IBM DB2, Informix, Micrsoft SQL and Oracle 8, use ARIES or its variants for efficient concurrency control.

SQLite is one of the most widely used DBMS's. It is deployed on nearly all computing platform such as smartphones (e.g, Android, Tizen, Firefox, and iPhone [52]), distributed filesystems (e.g., Ceph [58] and Gluster filesystem [1]), wearable devices (e.g., smart watch [4, 21]), and automobiles [19, 55]. As a library-based embedded DBMS, SQLite deliberately adopts a basic transaction management and crash recovery scheme. SQLite executes the transactions in a serial manner yielding *no-steal/force* buffer management. It uses page-granularity physical logging for crash recovery. These two design choices make the SQLite transactions suffer from excessive IO overhead; the insertion of a single 100 Byte record in PERSIST mode with the FULL SYNC option (the default option in Android Marshmallow) incurs five `fdatasync()` invocations [30] with 40 KB of writes to the storage (Figure 1). Despite its popularity and significance in all ranges of modern computing platforms, we argue that SQLite has not received proper attention from researchers and practitioners and leaves much to be desired from an IO efficiency perspective.

The inefficiency of IO in SQLite transactions is caused by the un-orchestrated interaction between SQLite and the EXT4 file system. Many studies have addressed this issue [23, 27, 30, 40, 49]. Each of these works has its own drawback, which bars itself from practical deployment in the commodity platform, e.g., space overhead [30], garbage collection overhead [27], overhead of redundant write [49] or non-existent hardware platform [26, 40].

In this work, we propose a new crash recovery technique, called *Database Shadowing* or DASH for short, for SQLite. We observe that the modern mobile DBMS exhibits unique characteristics. The mobile DBMS is very small (typically less than 100 KB), accesses to the mobile DBMS are entirely serialized, and consecutive transactions update many pages in common. Furthermore, in many cases a DBMS is exclusively opened by a dedicated application. These somewhat evident findings have profound implication in designing a crash recovery routine for SQLite. State of art recovery routines are mostly built upon the foundation that the database file is large (a few hundred MB at least) and there are multiple concurrent transactions with complicated dependencies among them. Both of these characteristics make full file shadowing and shadowing paging [32, 62] an infeasible choice for crash recovery in modern DBMS design, the main reason being the excessive space overhead, the overhead of redundant writes and the overhead of maintaining dependencies among the concurrent transactions.

Based upon our observation, we establish three principles in designing a crash recovery scheme for SQLite. First, the space overhead of maintaining an extra copy of a database file is not substantial. Second, there are no concurrent transactions. Third, the filesystem guarantees that the updated filesystem metadata are made durable in the order in which they are updated.

DASH maintains a shadow file for each database file. The shadow file in DASH represents the preceding state of the associated database file, i.e., the one before the the most recent transaction was applied. DASH consists of three key technical ingredients: Aggregate Update, Atomic Exchange and Version Reset. In applying a transaction, DASH aggregates the dirty pages updated by the most recent two transactions and applies them to the shadow file as if they are from a single transaction. We call this method Aggregate Update. After an Aggregate Update, the DBMS atomically switches the two files. Atomic Exchange guarantees the atomicity of the set of `rename()`'s without explicitly enforcing the order in which the results of its `rename()`'s are made durable. Atomic Exchange dispenses with expensive order-preserving primitives such as `fdatasync()` by exploiting the order-preserving aspect of the metadata operation in modern filesystems [28, 35, 43, 53]. Version Reset guarantees the durability of a transaction without invoking bulky filesystem journaling.

Unlike full file shadowing, the database file and the shadow file are not identical in DASH. They differ by those pages updated by the most recently committed transaction. Unlike shadow paging [39], DASH updates the database pages in the shadow file in an in-place manner. In-place update of DASH avoids the overhead of maintaining shadow copies of the database pages in a database file [6, 27]. DASH can be thought as file-granularity shadow paging. It is designed to be free from the inherent drawbacks of full file shadowing and shadow paging. DASH eliminates the redundant writes overhead in full file shadowing with "Aggregate Update", mitigates the overhead of keeping track of the updated locations of the database pages in shadow paging with "Atomic Exchange", and minimizes the overhead of committing a transaction with "Version Reset". The contribution of Database Shadowing can be summarized as follows.

- Aggregate Update, Atomic Exchange and Version Reset collectively reinvent shadow paging and full file shadowing into a new crash recovery scheme, called Database Shadowing. This work demonstrates that Database Shadowing can be a very effective means of crash recovery in the mobile context.
- Database Shadowing effectively relieves the order preserving overhead of rollback journaling of SQLite. Exploiting the *limited* order-preserving nature of the filesystem, Database Shadowing enforces the storage order of the metadata update operations without calling `fdatasync()`.
- DASH exhibits 5× and 30% performance improvement against PERSIST mode (default) and a WAL mode (the fastest mode) in SQLite, respectively. It uses as few as two `fdatasync()` calls whereas PERSIST mode issues five `fdatasync()` calls in a transaction. DASH uses only 2.5% of the storage space of that of WAL mode. It reduces the tail latency of an updated transaction at 99.9% by one fourth against WAL mode.

Table 1: Comparison of journal modes, ◎: very good, ○: good, ×: poor (PST: PERSIST mode, WAL: WAL mode, DASH: SHADOW mode, Thput: transaction throughput, T_{lat}: average transaction latency, T_{Tlat}: tail transaction latency, T_{rcvy}: recovery time)

mode	Thput	T_{lat}	T_{Tlat}	T_{rcvy}	IO Vol	space
PST	×	○	○	○	×	◎
WAL	◎	○	×	×	○	×
DASH	◎	◎	◎	○	◎	○

The beauty of DASH is its simplicity. It is implemented with only 451 LOC. DASH does not require any modifications in the underlying filesystem or the database organization. It does not require any new hardware features, either. The only significant change brought by DASH is its use of a 256 bytes of unused space at the tail end of the database header page in SQLite. Database Shadowing can be immediately deployed on the commodity platform. Table 1 summarizes the comparison of Database Shadowing against PERSIST mode and WAL mode journaling.

The remainder of the paper is organized as follows. Section 2 explains background. Section 3 explains the characteristics of SQLite. Section 4 provides an overview of DASH and its key components. Section 5 explains the details of the implementation. Section 6 explains the optimization technique for the exclusive connection. In section 7, we evaluate DASH and compare it with the existing journaling modes of SQLite. Section 8 summarizes the preceding works. Finally, section 9 concludes the paper.

2. BACKGROUND

2.1 Synopsis

SQLite is a *library-based* DBMS. It accounts for more than 70% of the IOs on the Android platform [23]. The Android Runtime consists of essential libraries such as SQLite, libc, and various media libraries. At its inception, Android smartphones used `yaffs` and `jffs` to manage the FTL-less Flash storage [29]. Recent smartphone products use a generic filesystem such as EXT4 or F2FS to manage Flash storage.

From a concurrency control and crash recovery viewpoint, SQLite must handle similar issues as a distributed DBMS. Each SQLite instance runs in a separate and exclusive address space. In accessing the shared database, there is no communication capability among the SQLite instances or any coordinating entity. To control concurrent accesses, SQLite makes the result of a transaction globally visible after the transaction completes by synchronizing the result of the transaction to the database file. Multiple instances of SQLite that share the same database file coordinate their concurrent activities by storing and retrieving the state of the database to and from *storage.*

SQLite is carefully designed not to rely on OS-specific features other than essential system calls such as `read()`, `write()` and `fdatasync()`. The highly portable, OS-independent and filesyste-agnostic, design philosophy of SQLite was successful in quickly making itself one of the most widely deployed DBMSs. However, this design decision makes SQLite subject to excessive IO behavior [23] and suboptimal performance. For concurrency control, SQLite uses no-steal/force buffer management with strict serial transaction execution. For crash recovery, SQLite uses page granularity physical logging.

2.2 Concurrency Control

SQLite executes the transactions in a serial fashion. A process acquires an exclusive lock on the database file before starting a transaction and releases the lock after the transaction completes. Due to this serial nature, SQLite transactions are conflict-free with one another.

In SQLite, there can be up to three different copies of a database page: one on the storage device, another in the page cache, and a third one in the local buffer of the associated process. SQLite writes the updated database pages back to the database file when the transaction finishes – *force*. SQLite writes the updated database pages to the database file only after the transaction finishes – *no-steal*. This execution mode is often called *Autocommit* mode [51]. Due to the no-steal/force mechanism, the database pages in the page cache are guaranteed to be up-to-date.

SQLite maintains a *version number* at the header of the database file. The version number is used to determine the validity of the database pages in its local buffer. A database page cached in the local buffer of a process becomes obsolete when another process updates it. Distributed systems use sophisticated protocols to synchronize the cache contents [47, 57]. SQLite does not have this luxury. An SQLite instance is responsible for keeping the database pages in its local buffer up-to-date by itself. SQLite increases the version number by one after the database file has been updated as a result of a transaction. The updated version number is written to the file after the transaction finishes. Before starting a transaction, the process reads the version number of the database file from the disk and compares it to the version of the database file in the local buffer. If the versions differ, SQLite re-reads the database pages from the database file before running a transaction.

2.3 Crash Recovery

SQLite provides two types of crash recovery modes: rollback journaling and WAL (write-ahead-logging). In crash recovery, SQLite recovers the system with respect to the state retrieved from storage. It assumes that all cached contents are lost when the crash occurs.

In rollback journaling, SQLite maintains undo records in a rollback journal file. For WAL mode journaling, SQLite maintains redo records in a WAL file. Both the rollback journal file and WAL file consist of a journal header and a sequence of log records. The database page size is 512 $\times 2^l$ byte, $l = 0, \ldots, 7$. The default and maximum page sizes are 4 KB and 64 KB, respectively.

In rollback journal mode, a transaction is executed in three phases: (i) writing the un-altered database pages to the journal file (undo-logging); (ii) updating the database pages and writing them back to the database file (database update); (iii) resetting the journal file (log reset). SQLite resets the journal file to denote that the transaction has successfully committed. SQLite provides three journal modes for rollback journaling: DELETE, TRUNCATE, and PERSIST.

The three rollback journal modes share the first two phases (undo-logging and database update). In the undo-logging phase, SQLite first records the undo logs; then, it updates the journal header with the log count and the updated database version number. Undo logging and header update are interleaved with `fdatasync()`. This is to ensure that the updated header page is made durable only after all log blocks reach the storage surface or are made durable. At the end of the logging phase, SQLite flushes the updated journal header to the disk. In the database update phase, SQLite modifies database pages and database header page. Then, SQLite writes the modified pages back to the database file.

The three rollback journal modes differ in how they reset the journal file. In DELETE mode, SQLite deletes the journal file; in TRUNCATE mode, SQLite truncates the journal file; in PERSIST mode, SQLite fills the journal header with 0's. While the difference in the three modes appears subtle, the difference has critical implications for the transaction performance [30]. In this work, we only address the details of PERSIST mode. PERSIST mode is the default journal mode in recent versions of the Android platform[1] and the fastest one among the three rollback journal modes [12].

In WAL mode, SQLite logs the updated database pages to the WAL file. The logged pages are checkpointed when the number of pages in the WAL file reaches a predefined threshold[2] or when the application terminates. After the checkpoint, SQLite resets the WAL file so that the incoming logs are placed from the beginning of the WAL file.

Assume that we insert a single 100 byte record. SQLite updates two pages in the database file; the database header page and the database node. Figure 1 illustrates the

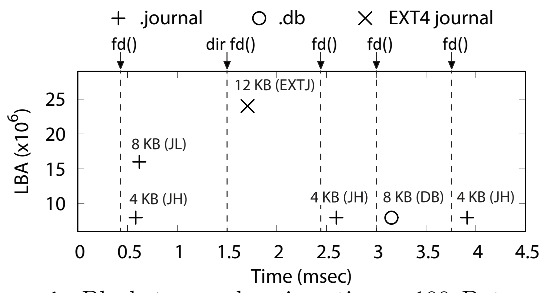


Figure 1: Block trace when inserting a 100 Byte record in SQLite (PERSIST mode). JH: journal header; JL: journal log; EXTJ: filesystem journal transaction; DB: database. The dotted lines denote the invocation of `fdatasync()`.

block trace when inserting a single database record in PERSIST mode. In the logging phase, SQLite writes the journal header, two log pages and the updated journal header again, four pages in all. In the database update phase, SQLite write two pages to the database file. In the log reset phase, SQLite writes the journal header page. Each of the three phases is followed by the `fdatsync()` to make the result of the updates durable. There is one `fdatasync()` call within the logging phase, which interleaves the log writes and journal header write. There is another `fdatasync()` to synchronize the directory that holds the journal file. Calling

[1]since Android 4.1.1

[2]The default is 1000 pages.

`fdatasync()` on the directory, the EXT4 filesystem commits the journal transaction. An EXT4 journal transaction consists of the journal header, one or more log pages and the journal commit block. An EXT4 journal transaction is at least 12 KB. In PERSIST mode, inserting a 100 Byte record amplifies to 40 KB IO in total with five `fdatasync()` calls (Figure 1) in total.

In WAL mode, SQLite appends the two pages to the WAL file and calls `fdatasync()`. The application must reserve a sufficiently large file to avoid frequent checkpoints. If we assume 80 KB database file size, the associated WAL file in its default size (4 MB) is 50× as large as the database file.

3. CHARACTERISTICS OF SQLITE DBMS

We examine the characteristics of SQLite DBMS usage. We collected the statistics from thirteen smartphones, which were used as the primary daily phone of the volunteers. The lengths of the trace collection periods were 2-6 months starting from Aug. 2014 across the devices. The volunteers consisted of nine graduate students, three undergraduate students and one software engineer, 40% of whom were females. For the trace collection, we used Androtrace [31]. For our block level IO analysis, we collected 16 million database transactions across the smartphone devices. They were associated with 846 database files and 211 applications. Previous works that study the block level IO characteristics of SQLite [23, 29, 40] only handle a few user applications such as Twitter, Facebook, or Gmail. In our study, we find that the system-initiated regular logging activities account for more than 40% of the transactions. It is important to analyze the IO characteristics of the system-initiated IO activities.

3.1 Database Size

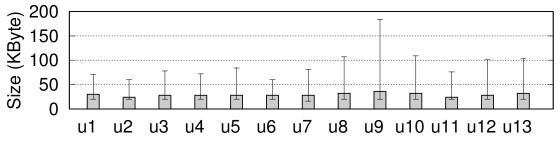


Figure 2: SQLite Database Sizes: The gray bar shows the median size and the range bar on top of the gray bar shows the 25% and 75% quantiles.

The first observation is that the database files are very small; 80% of all database files in 13 smartphones are less than 100 KB, and they account for 20% of the total database size. In legacy DBMS for OLTP, a single database file size reaches 800 GB [48]. On a cloud server, a single key-value store reaches a few GB in size [61]. The smartphones that we examined have as low as 16 GB of storage. The total size of the database files that are smaller than 100 KB accounts for 0.3% of the storage capacity (16 GB). Figure 2 summarizes the size of the database files.

3.2 Database Access

We observe three characteristics of SQLite database accesses. First, the SQLite transactions have a high duplication ratio. The duplication ratio of the transaction T_n is the fraction of database pages updated by T_n that were updated by T_{n-1}, i.e., $\frac{|T_n \cap T_{n-1}|}{|T_n|}$. Figure 3 illustrates the duplication ratio of the transactions for `snet_files_info.db`. We select four users who use `snet_files_info.db` the most. For user

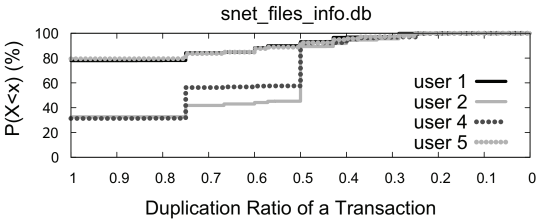


Figure 3: Cumulative percentage of transactions with corresponding duplication ratio in `snet_files_info.db`.

1, 80% of the transactions update the same database pages as the preceding transaction. For user 4 and user 5, 90% of the transactions share more than half of the database pages with the preceding transaction. In the `snet_files_info.db` database, the average duplication ratio of a transaction is 90%.

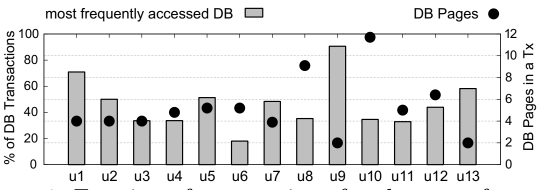


Figure 4: Fraction of transactions for the most frequently accessed database and the average transaction size (in pages)

Second, the database accesses are highly skewed and the transaction sizes are small (Figure 4). We examine the most frequently accessed database for each user and compute the ratio of its transaction count to the total transaction count used by all databases. In all cases, the database transactions for the most frequently accessed database account for more than 40% of the total database transactions. The most frequently accessed databases include `snet_files_info.db`, `es0.db`, `accounts.db` and `savepoint.db`. These databases are updated by automatic system activities and not by the user. On average, a transaction updates three to five database pages. The transactions for the system security check, `snet_file_info.db` are particularly significant [13, 25]. In four devices, the transactions to `snet_files_info.db` account for 40% of the database transactions.

Third, on average, 24% of the database files are exclusively used by dedicated applications.

4. DATABASE SHADOWING

4.1 Overview

Database Shadowing is designed for the following conditions: (i) the database file is relatively small, so the space overhead of maintaining a shadow copy of the database file is not substantial, (ii) the transactions are strictly serialized so that the DBMS does not have to maintain multiple versions of a database page in memory to represent the dependencies among the concurrent transactions and (iii) consecutive transactions share a substantial number of database pages.

The idea of Database Shadowing is simple and straightforward. DASH defines a shadow file for each database file; the roles of the shadow and database files switch on each transaction commit. The database file contains the most recent state written by the latest committed transaction T_n

(and all previous transactions); the shadow file contains the state written by the previous transaction or T_{n-1} (and the previous ones). When one executes a new transaction T_{n+1}, the updates are written to the shadow file with the updates from T_n. It is called *Aggregate Update*. After T_{n+1} commits, the DBMS switches the database and shadow files so that the newly designated database file represents the most recent state of the database. It is called *Atomic Exchange*. Figure 5 illustrates the concept of a transaction in DASH.

SQLite stores the version number at the header page of the database file. Let us assume that the version number of database file and shadow file are $n - 1$ and 0, respectively. When a transaction successfully completes, the version number of the database file and the version number of the shadow file are set to n and 0, respectively. At the beginning of a new transaction T_{n+1}, the DBMS sets the version number of the shadow file to $n + 1$, which signifies that T_{n+1} is being executed. When executing transaction T_{n+1}, the DBMS identifies the updated pages of preceding transaction T_n by referring to the Global Page List (described in Section 4.2) of the database file. The DBMS writes the pages updated by T_n and T_{n+1} to the shadow file. After that, the transaction is committed by switching the two files and setting the version number in the shadow file to 0. After the commit, the version number of the database is $n + 1$. The version numbers in the shadow and the database files are used to check the consistency of the two files if a failure occurs.

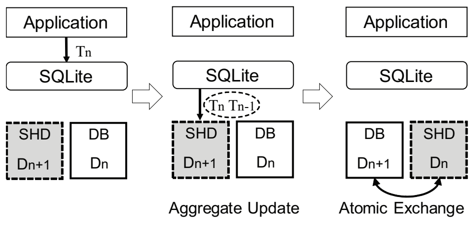


Figure 5: Transaction in Database Shadowing: a Concept

DASH is a hybrid mixture of shadow paging, full file shadowing and logging [7, 39]. Most DBMS's use logging [18] rather than shadow paging. The overhead of maintaining multiple versions for a database page makes the shadowing paging a sub-optimal choice for crash recovery method. SQLite is free from this inherent overhead of shadow paging because of the serial nature of the transaction execution.

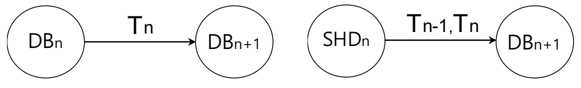
A database page in DASH has two fixed locations: one in the database file and the other in the shadow file. DASH resembles shadow paging in that DASH updates the shadow file and leaves the original database file intact when applying a transaction. DASH resembles logging for the following reason. For a database file (or a shadow file), the location of the database page is fixed and the database pages are overwritten with the updated content. In case of system crash, DASH recovers the database as logging does; the recovery module identifies the list of modified blocks, reads the unaltered images of the associated database pages and reflects them back to the original location.

The shadow file in Database Shadowing plays a different role from the shadow file in full file shadowing. Full file shadowing applies each transaction to both database file and shadow file to maintain the shadow file as an identical copy of the associated database file. Database Shadowing applies the transaction only to the shadow file and applies two consecutive transactions in an aggregate manner. The redundant updates in full file shadowing is mostly relieved in DASH.

Multi-Version Concurrency Control (MVCC) maintains multiple versions of the updated database page in a database file [6]. In multi-version concurrency control, the DBMS maintains the lifespan of the individual versions of the database pages so that the incoming read request is directed to the correct version of the associated page. Multi-version concurrency control needs to maintain the complex dependencies among the transactions for recovery. Like multi-version concurrency control, Database Shadowing maintains the two most recent versions of the database page. Unlike multi-version concurrency control, Database Shadowing does not maintain the complex dependency among the concurrent transactions, since the transactions are completely serialized.

4.2 Aggregate Update

Aggregate Update is the crux of Database Shadowing. By applying two consecutive transactions at a time to the shadow file, the DBMS can make the shadow file represent the most recent state of the database file without much overhead. In Aggregate Update, the DBMS reads the database pages that need to be updated by the incoming transaction, modifies them, aggregates them with the updated pages associated with the most recently committed transaction and writes them together to the shadow file.



(a) Single Transaction **(b)** Aggregate Update
Figure 6: Applying a Transaction

Let DB and SHD be the database and the associated shadow database, respectively. DB_0 is the initial state of the database. Let $T_0, T_1, T_2, \ldots, T_n$ be the history of transaction executions where $T_i (i = 0, 1, \ldots)$ is the i^{th} transaction. DB_{i+1} denotes the state of the database after T$_i$ has been applied. Applying transaction T_i corresponds to storing the database pages modified by T_i to the database file, which changes the state of the database from D_i to D_{i+1} (Figure 6a). The state of database that can be obtained from a sequence of transactions can be formulated as follows; $DB_{i-k} \xrightarrow[T_{i-k},\ldots,T_i]{} DB_{i+1}$.

The shadow database represents the preceding state of the associated database, DB_n; $SHD_n \equiv DB_{n-1}$. Let T_{n-1} and T_n be the most recently committed transaction and the incoming one, respectively. Applying T_n to D_n is equivalent to applying T_{n-1} and T_n to SHD_n (Figure 6b). After Aggregate Update, the shadow file represents D_{n+1}. Figure 7 illustrates an example Aggregate Update that updates the pages indexed in the B+-tree. Transactions T_i and T_{i+1} update $\{P_1, P_2, P_6\}$ and $\{P_1, P_7\}$, respectively. In Aggregate Update, the DBMS writes $\{P_1, P_2, P_6, P_7\}$ to the shadow file.

We introduce Global Page List to address the situation where two consecutive transactions are associated with different processes. If the following transaction is associated with a different process from the preceding one, the process that issues the following transaction cannot determine the updated pages associated with the preceding transaction

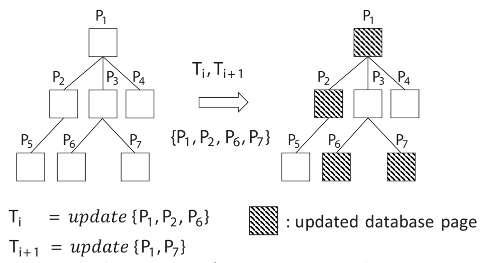


Figure 7: Aggregate Update

since they reside in different address space. For transaction T_n, the DBMS records the page ID's updated by T_n in the Global Page List. We locate Global Page List at the tail end of the database header page. The database header page is 4 KB, of which only 100 bytes at the beginning is used to store the database metadata; the remainder is left unused. We use this unused space for storing Global Page List. When a transaction starts, the DBMS examines the Global Page List in the header page of the database file and reads the associated pages if they do not exist in the local buffer or if the database pages in the local buffer are obsolete. SQLite reads and writes the database header page at the beginning and end of the transaction, respectively. The reading and writing of the Global Page List are piggybacked on the existing operations. Maintaining the Global Page List does not incur any additional storage space or IO.

Let us provide an example (Figure 8). Assume that Process 1 and Process 2 update the same database file. Process 1 updates p_1 and p_2. Process 2 updates p_1 and p_3. p_1 is the header page of the database file. When Process 1 finishes its transaction, the Global Page List on the disk contains p_1 and p_2. When Process 2 starts the transaction, it first reads the database header page and identifies the dirty pages created by the previous transaction, which are p_1 and p_2, in the Global Page List. Process 2 reads p_1 and p_2 into its user space from the disk. If p_1 and p_2 are in the page cache (in most cases they will be), reading them from the disk will not cause any disk IO. Process 2, then, updates p_1 and p_3. SQLite updates the header page in its local buffer so that the Global Page List contains $\{p_1, p_3\}$. Finally, it writes p_1, p_2 and p_3 to the shadow file. After Aggregate Update, the Global Page List in the database header contains p_1 and p_3.

Aggregate Update simultaneously commits multiple (two) transactions as in group commit [14, 17] or compound transaction [54]. However, Aggregate Update should be distinguished from these techniques. Group commit and compound transaction were developed to improve transaction throughput. Aggregate Update is developed to make the shadow file represent the most recent state of database without updating both the database file and shadow file. Aggregate Update is not for improving transaction throughput. Group commit and compound transaction trade transaction latency and durability, respectively, for transaction throughput. Unlike group commit and compound transaction, Aggregate Update guarantees the durability of the transaction and improves transaction latency over the existing journaling modes of SQLite.

Aggregate Update writes a transaction's update twice to a shadow database: the first time after it commits and the second time after the next transaction commits. The overhead of redundant update in Aggregate Update may not be as substantial as it appears compared to when the individual transactions are separately committed, for two reasons. First, the consecutive transactions exhibit strong temporal locality, i.e. $|T_n T_{n+1}| \ll |T_n| + |T_{n+1}|$. Second, the flush overhead accounts for the dominant fraction of the transaction commit and is relatively insensitive to the number of database pages that are flushed.

4.3 Atomic Exchange

Atomic Exchange switches the shadow file and database file atomically with respect to a system failure. Atomic Exchange exchanges the hard links of the associated directory entries. We implement Atomic Exchange with three `rename()` calls [2]. `rename(char* old, char* new)` atomically changes the name of the hard link from *old* to *new*. Figure 9 illustrates the pseudo code for Atomic Exchange. The Atomic Exchange operation appears trivial. However, the rationale behind it deserves elaboration.

The simplest method to implement Atomic Exchange is to use the recently proposed `renameat2()`[3] [34]. Although it simplifies the implementation, `renameat2()` fails to satisfy a design constraint of SQLite: Database Shadowing should be OS-independent and filesystem-agnostic so that it can be backward-compatible with any existing SQLite versions. The `renameat2()` is only available in recent versions of Linux and not available in other OS's, e.g., FreeBSD and iOS.

For the atomicity of Atomic Exchange, the results of `rename()` calls must be made durable in the order in which they are invoked. Otherwise the recovery routine cannot identify the state of the system at the time of the crash and cannot recover the system to the one that satisfies the atomicity. In Figure 9, `rename(db, shd)` is called after `rename(shd, tmp)`. If the system crashes while the result of `rename(shd, tmp)` is in flight and the result of `rename(db, shd)` has become durable, then we lose the contents of the shadow file.

Modern filesystems do not guarantee the order in which the results of the set of operations are made durable [9, 60]. The application explicitly calls `fdatasync()` when it needs to enforce the order in which the results of the update operations are made durable. For example, in the logging phase of a PERSIST mode transaction, SQLite calls `fdatasync()` after it completes writing the log blocks and before it writes the updated journal header to the log file. This step is to ensure that the updated journal header is made durable only after the log blocks have become durable. The `fdatasync()` is one of the most expensive system calls [30]. It blocks the caller and exposes the caller to a raw Flash cell programming delay. Fortunately, modern filesystems are not entirely orderless; for metadata operations, modern filesystems do preserve the order in which the results of the operations are made durable [35, 43, 53]. The directory entry is filesystem metadata. Exploiting the limited order-preserving nature of modern filesystems, the proposed Atomic Exchange does not invoke any costly `fdatasync()` and yet guarantees that the results of `rename()`'s are made durable in order.

The correctness can be shown as follows. The files that are subject to the `rename()` calls can be either in the same directory or in different directories. In both cases, the results of `rename()`'s are guaranteed to be made durable in order. In the first case (a same directory), the first `rename()` operation acquires an exclusive lock for the directory entry. When the

[3]available since Linux kernel 3.16

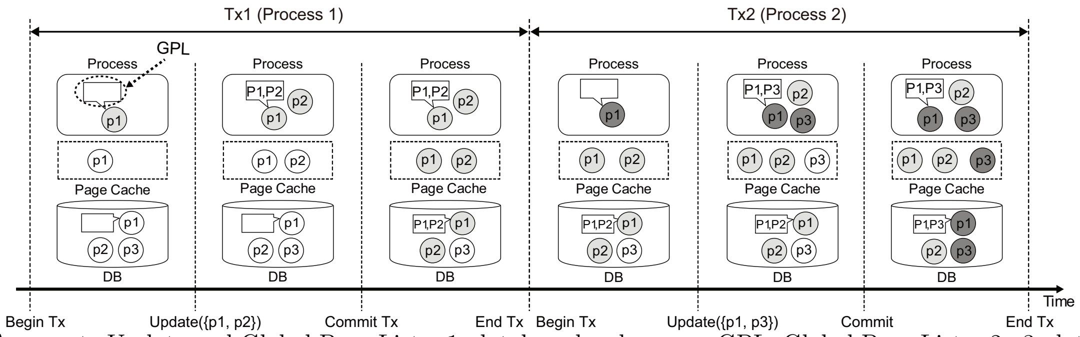


Figure 8: Aggregate Update and Global Page List; $p1$: database header page; GPL: Global Page List; p2,p3: database pages

```
void Atomic_Exchange(char* shd, char* db) {
    char *tmp="tmpfile" ;
    rename(shd, tmp);
    rename(db, shd);
    rename(tmp, db);
}
```

Figure 9: Pseudo code for Atomic Exchange

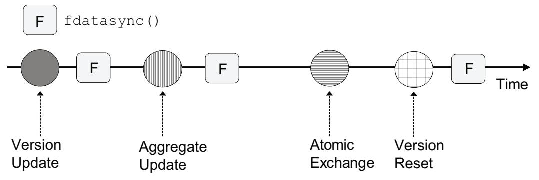


Figure 10: A transaction in Database Shadowing

two consecutive rename() operations update the same directory block, the consecutive rename calls update the directory block serially. The result of the rename()'s operations are guaranteed to be made durable in order. Now consider the second case, i.e., `rename()` calls updating different directory blocks. The `rename()` calls are guaranteed to be made durable in order in journaling filesystems such as EXT4 or log-structured filesystems such as F2FS. In those filesystems, the metadata are updated in the order that they are issued. In journaling filesystems or log-structured filesystems an application inserts the updated metadata blocks to the journal transaction or to the log as they are updated. The journaling filesystem commits the filesystem journal transaction atomically. The log-structured filesystem commits the logs in memory in FIFO order in an atomic manner. Due to this nature, Atomic Exchange guarantees that the results of the rename operations are made durable in order [60].

The recovery algorithm is as follows. If the crash occurs during the Atomic Exchange, the filesystem falls into one of three states: there are (i) `a.db` and `a.shd`, (ii) `a.db` and `tmp`, or (iii) `tmp` and `a.shd`. In case (i), there is nothing to be done. In case (ii), the crash has occurred after the results of the first `rename()` were made durable. The recovery routine replays Atomic Exchange starting from the second `rename()`. In case (iii), the crash has occurred after the results of the first and second `rename()` calls became durable. The recovery routine replays the last `rename()` for recovery. In all cases, the filesystem is recovered to contain `a.shd` and `a.db`. The atomicity is well preserved.

The atomicity of each `rename()`, the order-preserving nature of the sequence of `rename()` calls, and the associated recovery module collectively and successfully guarantee the *atomicity* of Atomic Exchange.

5. IMPLEMENTATION

5.1 A Transaction in Database Shadowing

When the shadow file does not exist, SQLite creates one by cloning the database file. This happens when the database file is first created, when the journal mode is changed to Database Shadowing from other modes, or when the recovery routine removes the shadow file. The shadow file has the same Global Page List value as the database file. If the shadow file is successfully cloned, the version number in its header is initialized to 0 and the Global Page List is empty. If Global Page List in the shadow file is empty, it indicates that the database file and the shadow file are identical. When the Global Page List in the shadow file is empty, SQLite applies the transaction to the database file instead of performing Aggregate Update to the shadow file.

A transaction in Database Shadowing consists of three phases; (i) Version Update, (ii) Aggregate Update and Atomic Exchange, and (iii) Version Reset. Figure 10 schematically illustrates these phases. We discussed the details of the Aggregate Update and Atomic Exchange in Section 4. In this section, we will discuss Version Update and Version Reset, which play a critical role for the atomicity and durability of a transaction in Database Shadowing.

Version Update and Version Reset are employed to denote that a transaction has begun and finished, respectively. The first phase of a transaction is "Version Update". Its objective is to denote that the transaction is now in-flight. There are three main tasks in this phase: (i) check whether the preceding transaction has successfully committed, (ii) check whether the local copies of the database pages are up-to-date and (iii) update the version number of the shadow file. In Database Shadowing, the version number is used for two objectives, (i) to determine whether the preceding transaction has successfully finished and (ii) to determine whether the local copy of the database pages are up-to-date.

First, SQLite reads the version number of the shadow file and checks whether it is zero. If so, SQLite concludes that the preceding transaction has successfully finished. Otherwise, it runs the recovery routine. Second, it examines whether the version number of the database file in its local buffer matches the version number of the database file on disk. If they do not match, SQLite re-reads the database pages that must be modified from the disk. Third, SQLite increases the version number extracted from the database file by 1 and writes the updated version number to the shadow file along with the associated Global Page List.

Second, SQLite applies an *Aggregate Update* to the shadow file and performs *Atomic Exchange.*

The third phase of the transaction is "Version Reset". In this phase, SQLite sets the version number of the shadow file to zero and updates the associated header block of the shadow file. The objective of Version Reset is to guarantee the durability of a transaction and denote that the transaction has successfully finished. Version Reset is equivalent to writing a commit block to storage in a logging-based crash recovery scheme. Algorithm 1 illustrates the detailed steps of executing a transaction in Database Shadowing.

Algorithm 1: Transaction in DASH
```
1  function Apply($T_n$, $a.db$)
2      Version_Update($a.shd$) ;
3      fdatasync($a.shd$) ;
4      Aggregate_Update() ;
5      fdatasync($a.shd$) ;
6      Atomic_Exchange($a.shd, a.db$) ;
7      Version_Reset($a.shd$) ;
8      fdatasync($a.shd$) ;
9      return ;
10 end
```

In two places, DASH must enforce the order in which the results of the updates are made durable. First, we must ensure that the updated version number is made durable at the shadow file before any result of the Aggregate Update is made durable. Otherwise, the database may be recovered to an inconsistent state after an unexpected system failure. To do this, SQLite calls `fdatasync()` (Line 3 in Algorithm 1). Assume that the system crashes after some of the updated pages by Aggregate Update are partially committed to the shadow file while the header page with the updated version number is still in-flight. In the recovery phase, the DBMS recovery module will mistakenly conclude that the preceding transaction was successfully completed. The DBMS continues to execute and applies Aggregate Update to the corrupt shadow database. The transaction in Database Shadowing leads the database file in an inconsistent state.

Second, between Aggregate Update and Atomic Exchange, we must ensure that Atomic Exchange only begins after the results of Aggregate Update are made durable. To do this, SQLite calls `fdatasync()` after Aggregate Update (Line 5 in Algorithm 1). Therefore, the recovery module knows that the result of Aggregate Update has been made durable if it finds that either Atomic Exchange or Version Reset has finished. By same token, because of `fdatasync()` after Aggregate Update, the results of Atomic Exchange and Version Reset can be made durable out-of-order and yet the crash recovery routine determines if the Aggregate Update has finished successfully. The details of the crash recovery will be explained in section 5.3. As the last step of the transaction, we call `fdatasync()` to make the result of Version Reset durable. There are three `fdatasync()`'s in a transaction in Database Shadowing. The first two (Line 3 and Line 5) are to control the order in which the associated blocks are made durable. The third one (Line 8) is to ensure the durability of a transaction.

5.2 The Role of Version Reset

Version Reset is designed to minimize the IO overhead associated with guaranteeing the durability of a transaction. It should deserve an elaboration. In Database Shadowing, writing the commit mark of a transaction is equivalent to make the result of Atomic Exchange durable.

The easiest way to make the result of Atomic Exchange durable is to call `fsync(a.shd)`. By calling `fsync(a.shd)` immediately after Atomic Exchange, the DBMS can make the directory entries updated by Atomic Exchange durable. Under this mechanism, the DBMS compares the version numbers of the database file and the shadow file and determines whether the transaction has successfully finished or not. If the version number of the database file is greater than the version number of the shadow file by one, the DBMS concludes that the preceding transaction has successfully finished. Otherwise, the DBMS concludes that the preceding transaction was incomplete. However, `fsync()` is one of the most expensive system calls. It triggers bulky filesystem journaling. It leaves the caller under a number of context switches and makes the caller wait for two flush operations: one before writing the commit block and one after writing the commit block of the filesystem journal transaction [60]. In addition, compound journaling of EXT4 may create unexpected dependencies between irrelevant requests and may leave the caller under anomalous delay [54].

To address the overhead of `fsync()`, we develop an indirect and more efficient route, "Version-Reset," to mark the successful completion of the database transaction. Version Reset sets the version number of the shadow file to zero. It updates a single block, the header block, of the shadow file. With Version Reset, it suffices to flush the single block to mark the successful completion of the database transaction; the DBMS calls `fdatasync()` to make the result of Version Reset durable. Version Reset dispenses with the expensive filesystem journaling (`fsync()`) in guaranteeing the durability of a transaction.

When using Version Reset to commit a transaction, the recovery routine of Database Shadowing examines the version number of the shadow file (or database file) if it is 0. If it is 0, the DBMS knows that the preceding transaction has finished successfully. For correctness, we show that Database Shadowing guarantees the durability of the Atomic Exchange with Version Reset. First, assume that the crash recovery routine finds that the version number of the shadow file is 0. Then, the crash has occurred either before Version Update began or after Version Reset has successfully finished. Thus, the results of the Aggregate update and Atomic Exchange have been made durable. The durability of a transaction is preserved. Second, assume that the crash recovery routine finds that the version number of the database file is 0. The following has occurred at the time of crash: Atomic Exchange has swapped the hard links of the database file and the shadow file, and Version Reset has set the version of the newly designated shadow file (old database file) to zero. Both of these have happened in memory. After that, Database Shadowing calls `fdatasync()` (Line 8, Algorithm 1). `fdatasync()` flushes the header block of the newly designated shadow file. The crash has occurred while the updated directory blocks associated with Atomic Exchange are still in-flight. The recovery module finds that the version number of the database file is zero since the updated hard links are lost. In this case, the recovery routine performs Atomic

Exchange again to redo the transaction. The durability of a transaction is preserved.

5.3 Crash Recovery

The recovery module acquires an exclusive lock on the database and shadow files before starting a recovery. When a number of SQLite instances run the recovery routine on the same database file simultaneously, the first one that acquires the exclusive lock performs the recovery. The other ones find that the database file is in a consistent state.

We represent the state of the database with a pair of version numbers on the disk : the version numbers of the database file and shadow file. The state space consists of four states : $\mathcal{S} = \{< n, 0 >, < n, n+1 >, < n+1, n >, < 0, n >\}$, where the first and second values are the version numbers of the database and shadow file, respectively. Figure 11 illustrates the state transition diagram to execute a transaction. Initially, the state is at $< n, 0 >$. After the transaction completes the state becomes $< n + 1, 0 >$. The version number of the shadow file is 0 before the beginning and after the end of a transaction.

Let us describe the state transition in detail. The initial state is $< n, 0 >$. After Version Update, the state changes to $< n, n + 1 >$. If the results of Atomic Exchange and Version Reset are made durable in order, the state of database changes to $< n+1, n >$ (the result of Atomic Exchange) and subsequently to $< n + 1, 0 >$ (the result of Version Reset). If they are made durable out of order, the state of database first changes to $< 0, n + 1 >$ and $< n + 1, 0 >$.

Creating a shadow file is crafted to imitate the process of executing a normal transaction in Database Shadowing. In this way, when a crash occurs while a shadow file is created, the recovery routine can apply the same recovery procedure as if the crash happened during the normal transaction execution. Assume that the database file is first created. The version number and Global Page List of the database file are initialized to one and empty, respectively. Then, the DBMS populates the database pages at the database file and creates the empty shadow file. The version number of the shadow file is initialized to two as if Version Update is applied; at the same time, the Global Page List in its header is initialized to empty. Once this completes, the database pages in the database file are copied to the shadow file. After the copy completes, we set the version number in the shadow file to zero as if Version Reset is applied. Creating a shadow file now completes. We do not need to apply Atomic Exchange in creating the shadow file because the database file and the shadow file have identical database pages.

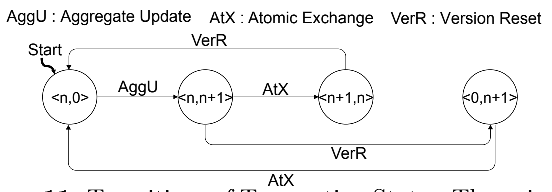


Figure 11: Transitions of Transaction States. The pair represents <*version number* of the database file, *version number* of the shadow file>

When the crash occurs, the DBMS will be in one of the four states. Aggregate Update is a critical step that determines the recovery action. If Aggregate Update has not successfully finished, the recovery module undoes the transaction. The recovery module removes the corrupt shadow file and reconstructs the shadow file to be identical to the database file. If Aggregate Update has successfully finished, the recovery module replays the remaining steps, which are either Atomic Exchange or Version Reset, to finish the transaction. The recovery procedure associated with each database state is as follows.

- $< n, 0 >$: The system has crashed while the database is in a consistent state. There is nothing to be done.
- $< n, n+1 >$: The system has crashed after Version Update has finished and before Aggregate Update completes. This includes the case that a crash happens when the shadow file is initially being created. For both cases the recovery routine reconstructs the shadow file from the database file. We expedite the reconstruction process using *partial reconstruction*, which is explained below. (*undo*)
- $< n+1, n >$: The system has crashed after the Atomic Exchange has finished. The recovery module performs Version Reset to finish the transaction. (*redo*)
- $< 0, n + 1 >$: The system has crashed after the Version Reset has finished. The recovery module performs Atomic Exchange to finish the transaction. (*redo*)

We develop *partial reconstruction* to make the shadow file identical to the database file without copying the entire database file. The recovery module uses partial reconstruction when it finds that the database is in state $< n, n + 1 >$. In partial reconstruction, the recovery module examines the Global Page Lists of the database file and shadow file. The page ID's in these two Global Page Lists represent all pages where the two files differ. SQLite copies the associated database pages from the database file to the shadow file. Note that in the special case when the state is $< 1, 2 >$, that is, the crash occurs during the initial creation of the shadow file, we copy the entire database pages in the database file to the shadow file. After these blocks are made durable, SQLite updates the version number of the shadow file to 0.

When the system crashes again during undo or redo, the recovery module recovers as if the crash has occurred during the normal transaction execution. When the system crashes during partial reconstruction, the recovery module will find the DBMS at $< n, n + 1 >$. Then, it performs the partial reconstruction again. In logging-based crash recovery, DBMS uses separate routines, e.g., CLR(compensation log record) [38] or `revoke` [16], to handle crashes that occur during the log-replay. Unlike logging-based crash recovery, Database Shadowing does not require a separate crash recovery routine.

5.4 Implementation Complexity

We emphasize the simplicity and portability when designing DASH. DASH does not require an additional library other than the ones being used by stock SQLite. DASH requires 424 LOCs to implement the entire functionality including the recovery. For comparison, PERSIST mode requires 618 LOCs and WAL requires 581 LOCs to implement. By adding DASH, the size of the SQLite binary increases by 5KB from 621KB to 626KB. We measure the complexity of DASH with Cyclomatic Complexity [36]. The complexity score of DASH is 115, which is 33% lower than that of PERSIST (177) and that of WAL (172).

6. EXCLUSIVE CONNECTION

If the database file is exclusively used by a dedicated application, i.e., an *Exclusive Connection*, the database pages in the local buffer are guaranteed to be up-to-date and the transaction execution in Database Shadowing can be significantly simplified. In Exclusive Connection, it is unnecessary to check if the preceding transaction has successfully finished every time when a transaction starts. Instead, SQLite suffices to check whether the preceding connection has successfully closed when it opens a database connection. In DASH, when SQLite closes the database connection, SQLite zero-fills the entire database header (100 byte) of the shadow file. We call it *Header Reset*. Note that Version Reset sets the version number (4 byte) of the shadow file to zero. When SQLite opens a database connection, it examines whether the header of the shadow file has been reset properly. If the preceding connection has not been properly closed, the recovery routine examines the state of the database at the time of crash and performs appropriate recovery action. In Exclusive Connection, we also eliminate checking if the data blocks at the local buffer always are up-to-date when the transaction starts.

In Database Shadowing, the recovery module determines the state of the DB solely based upon the version numbers of the database file and shadow file. In Exclusive Connection, the header of the shadow file is also used to determine the state of the DB. The `fdatasync()` that follows the Version Update (line 3 at Algorithm 1) is used to ensure that both database file and shadow file are consistent when the recovery module encounters $< n, 0 >$. In Exclusive Connection, we eliminate this `fdatasync()`. Instead, SQLite examines the header of the shadow file to determine the consistency of the database file and shadow file when it encounters $< n, 0 >$. If the state is $< n, 0 >$ with the preceding connection being properly closed, SQLite determines that both files are in a consistent state. If the state is $< n, 0 >$ and the preceding connection has not been properly closed, SQLite determines that the Aggregate Update may have not been successfully finished. Taking the pessimistic measure, SQLite removes the existing shadow file and creates a new shadow file via cloning the database file.

We finish the section with a brief analysis on Database Shadowing. Assume that a transaction updates n database node pages. PERSIST mode in SQLite is full of redundancies. In each transaction, it writes all updated database pages twice (once in the database file and once in the journal file) and the header page of the rollback journal file three times. In addition, it calls `fdatasync()` to flush the directory block. In all, total $2n + 8$ pages are written with five `fdatasync()` calls. Database Shadowing is free from updating the journal header. It exploits the order-preserving aspect of the filesystem and is relieved from the excessive `fdatasync()` calls. Assuming 50% duplication ratio in consecutive transactions, Database Shadowing writes $1.5n+1$ blocks in a transaction with two or three `fdatasync()` calls.

7. EXPERIMENT

We implemented Database Shadowing in SQLite [52]. We added two journaling modes: SHADOW and SHADOW-X for Database Shadowing and Database Shadowing with Exclusive Connection, respectively. We rebuild the Android platform of Galaxy S7 Edge with our modified SQLite (Android Marshmallow, Linux Kernel 3.18) [3]. We use Mobibench to generate the workload [23]. In Mobibench, we perform the database transaction 10,000 times in each experiment. We vary the number of operations in a transaction from 1 to 20. The record size is 100 byte [23]. We do not create any index. We compare the performance of four journal modes: PERSIST, WAL, SHADOW and SHADOW-X. When there are multiple operations in a transaction, each operation is applied to different tables in the database file.

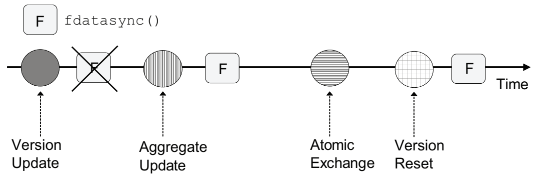
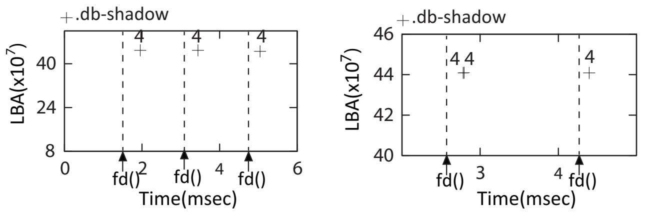


Figure 12: A transaction in the Exclusive Connection

7.1 IO Volume



(a) SHADOW **(b)** SHADOW-X

Figure 13: Block Level IO: Inserting 100 byte record to the SQLite database (Unit: KB)

We observe the block-level IO patterns in SHADOW and SHADOW-X when we insert a single 100 byte record into a database table. The result is shown in Figure 13. In SHADOW mode, 12 KB of data are written to disk in total with three fdatasync() calls (denoted as fd() in the figure). In SHADOW-X mode, SQLite makes two `fdatasync()` calls. In PERSIST mode, inserting a single 100 byte record incurs 40 KB IO with five flush operations (see Figure 1). Database Shadowing reduces IO volume by a factor of four and flush operations by a factor of three.

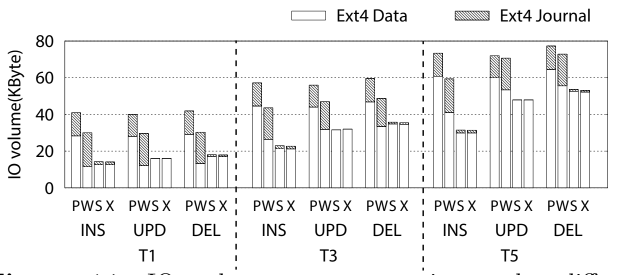


Figure 14: IO volume per transaction under different journal modes: P:PERSIST; W:WAL; S:SHADOW; X:SHADOW-X

Figure 14 compares the average IO volume per database transaction in four journal modes; we generate 10,000 transactions. For the comparison we run transactions with 1, 3, and 5 operations per transaction (denoted as T1, T3, T5). SHADOW mode reduces the IO volume per transaction by as much as 1/3 compared to PERSIST mode (T1 case).

7.2 Transaction Throughput

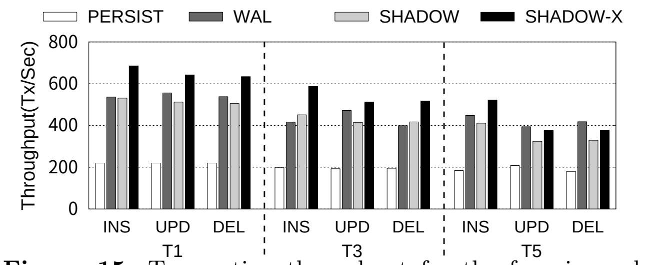


Figure 15: Transaction throughput for the four journal modes, (INS:Insert, UPD:Update, DEL:Delete)

We compare the transaction throughput in four transaction modes: PERSIST, WAL, SHADOW, and SHADOW-X. For the comparison we run transactions with 1, 3, and 5 operations per transaction (denoted as T1, T3 and T5). The number of operations in a transaction is selected based on the fact that a single SQLite transaction in mobile applications updates on average three database tables [12]. We executed 10,000 transactions for the evaluation with a database record size of 100 bytes. A transaction performs the insertion, deletion, and update. The insert operation appends the new record at the end of the database file. The update and delete operations randomly select the record.

Figure 15 shows the transaction throughput in different journal modes. For insert transactions, SHADOW mode has at least twice the throughput of PERSIST mode in T1, T3, and T5. Compared to WAL mode, the SHADOW mode shows a similar throughput. and SHADOW-X performs 25% better. This performance improvement is attributed to the reduced flush operations (`fdatasync()`) in a transaction from three in SHADOW mode to two in SHADOW-X.

7.3 Transaction Latency

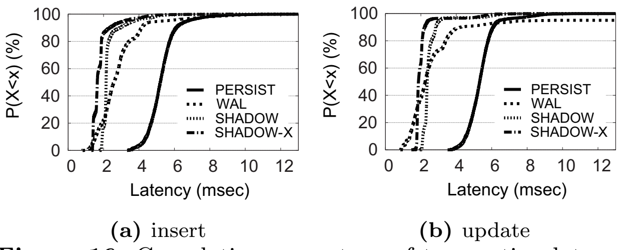


Figure 16: Cumulative percentage of transaction latency. Each transaction contains three operations.

We evaluate transaction latencies in four journaling modes. for insert and update transactions. Figure 16 shows the CDF of the transaction latencies under four journal modes. Table 2 illustrates the average and tail latencies of the database transactions under four journal modes. The performance trends remain similar when we vary the numbers of operations in a transaction.

For the average latency, both SHADOW mode and WAL mode exhibit the best behavior. The average latency in SHADOW-X mode is approximately 20% smaller than that of SHADOW mode. PERSIST mode exhibits the worst behavior in average latency. For insert operations, the average transaction latency in SHADOW mode is 40% that of PERSIST mode.

Table 2: Quartile statistics: inserting and updating three pages (OP: operation; PER: Persist; SHD: Shadow; SHD-X: Shadow-exclusive)

OP	Insert (msec)			Update (msec)		
Mode	Avg.	Med.	99.9%	Avg.	Med.	99.9%
PER	5.2	5.1	10.6	5.3	5.3	10.0
WAL	2.4	2.3	10.1	2.1	1.8	36.9
SHD	2.2	2.1	5.9	2.4	2.3	6.9
SHD-X	1.7	1.6	5.5	2.0	1.9	6.3

SHADOW mode excels in tail behavior. Its tail latency at 99.9% is 60% that of PERSIST mode. The tail latency of an update transaction in SHADOW mode at 99.9% is one fourth that of WAL mode. The poor tail latency of WAL mode is due to the checkpoint overhead. Before starting a transaction in WAL mode, SQLite first checkpoints the database pages in WAL file if the number of log pages reaches the predefined threshold. Since SQLite is fundamentally a library-based DBMS, it is inevitable that SQLite handles the checkpoint in the foreground. The transactions in SQLite occasionally can be exposed to excessive delay when SQLite must first checkpoint the logs to the database file before starting a transaction. The tail latency is governed by the number of database pages that are checkpointed. In SQLite, an insert transaction appends the new record at the end of the database file. Since a single database page accommodates multiple database records, a number of insert transactions update the same database page. Append-only nature of the insert operation makes a large fraction of log pages in WAL file invalid. Thus, only a small fraction of the log pages are checkpointed to the database file. Update transaction has different characteristics. In update workload, SQLite randomly selects the database records to be updated. Update transaction renders much fewer invalid log pages in the WAL file. In WAL mode, an update transaction is subject to much longer tail latency than an insert transaction, 36.9 msec (update) vs. 10.1 msec (insert) as in Table 2. To reduce the tail latency of a transaction, we can limit the number of database pages that a WAL file can hold, e.g., from 1,000 to 250 pages. However, it will increase the average transaction latency due to more frequent checkpoint. We like to leave the detailed analysis on the effect of the WAL file size over the transaction latency to the separate context.

7.4 Recovery Overhead

We examine the latency for crash recovery. We vary the number of updated pages in a transaction from one page to twenty pages. In this experiment, we assume that the duplication ratio of a transaction is 50%.

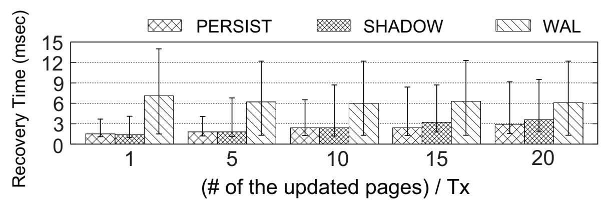


Figure 17: Recovery time: average, minimum and maximum

Figure 17 shows the average, minimum, and maximum recovery times for the three journal modes. PERSIST mode yields the best recovery time in all cases. SHADOW mode performs nearly as good as PERSIST mode. For larger transaction sizes, PERSIST mode exhibits slightly shorter recovery latency than DASH. WAL mode exhibits the worst recovery overhead among the three journal modes. For a transaction with five records, the recovery time in SHADOW mode is 1/3 of that in WAL mode. In our experiment, we assume that the system crashes when the WAL file is approximately 50% full (500 pages).

In PERSIST mode, the worst-case recovery time is 2× of the average recovery time. Undoing a transaction may involve not only recovering the old image of the database pages but also shrinking the database file. Shrinking a file updates the inode table and block bitmap. Due to the overhead of journaling the updated metadata, the recovery latency can increase substantially.

7.5 Overhead of Aggregate Update

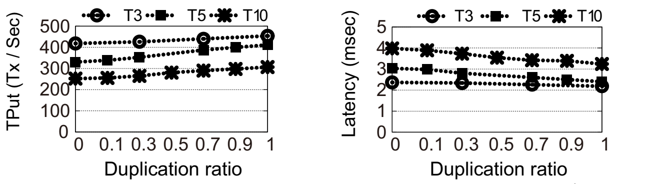


Figure 18: Overhead of Aggregate Update (TPut: Throughput, Tn: transaction with n insert operations)

We examine the overhead of Aggregate Update. We vary the number of operations in a transaction, $N = 3, 5, 10$ and the "duplication ratio" varies from 0.0 to 1.0.

The transaction throughput decreases by 5% when approximately half of the database pages are duplicated in comparision with when a transaction is applied in the non-aggregated manner (Figure 18). When a transaction has ten operations, the transaction latency increases by 8% when the duplication ratio is 0%, i.e. the transaction does not share anything compared to when the duplication ratio is 100%.

The result of this experiment provides an important ground for Database Shadowing. In Database Shadowing, every transaction is executed twice. This experiment shows that the redundancy caused by the Aggregate Update does not have significant performance overhead. Even when the consecutive transactions do not share any pages, the performance overhead of Aggregate Update is less than 8% against when the individual transactions are executed separately.

8. RELATED WORK

There are three main approaches to improve the excessive IO behavior of SQLite. The first approach modifies SQLite or the underlying filesystem, while physical page granularity logging of SQLite remains intact. Jeong et al. [23] proposed optimization techniques in the Android IO stack, such as using `fdatasync()` instead of `fsync()`, polling-based IO instead of interrupt-driven IO, and F2FS instead of EXT4. Lue et al. proposed to maintain the journal file in DRAM [33]. Shen et al. [49] proposed to use the EXT4 Data journaling mode to improve transaction throughput. Kim et al. [27] proposed to use Multi-version B-trees and achieved 70% throughput gain against WAL mode. Lee et al. [30] proposed to use direct IO and asynchronous commit for SQLite transactions. They achieved 5× performance against the WAL mode. [50] proposed to perform LSM tree for SQLite for better transaction performance. Despite the significant performance benefit, both [27, 30] have major limitations for practical deployment. [27] requires a modification on the database organization of SQLite and can render excessive tail latency due to the garbage collection overhead. [30] does not guarantee the durability of individual transactions and requires a modification of the underlying Linux kernel.

The second approach proposes to use byte-addressable NVRAM as a logging device. The works in this category develop various byte-granularity logging schemes for SQLite [26, 40, 42]. Oh et al. [40] use record-level logging and Kim et al. [26] use physical differential logging for SQLite. Both of these works are physical logging techniques. Lee et al. [42] propose a transaction-level logical logging technique for SQLite. All these works [26, 40, 42] are different from the other NVRAM-based logging techniques [5, 8, 22, 56]. The NVRAM based logging schemes for SQLite specifically address the issues in page-granularity physical logging with no-steal/force buffer management scheme, whereas the others address the issues in ARIES-style physio-logical logging for an enterprise DBMS.

The third category of works propose to use a transactional filesystem that frees SQLite DBMS from explicit logging. They include transactional Flash storage, e.g., X-FTL [24]and transactional filesystems such as SHARE [41], ANViL [59], and CFS [37]. These works require new hardware interfaces or new system calls and substantial changes in the existing SSD firmware or in the filesystem.

Database Shadowing distinguishes itself from the aforementioned works in that it does not rely on any new hardware, new OS primitives or any changes in the DBMS organization while effectively addressing the significant issues in SQLite journaling.

Shadow paging [32, 62] which has not been well accepted in the DBMS community manifests itself as a filesystem technique, e.g. COW-based filesystem [20, 44, 45, 46] and version-based metadata management in the filesystem [10, 11]. This is because the filesystem does not allow concurrent transactions and therefore maintaining the multiple versions of a file page does not entail substantial overhead.

9. CONCLUSION

In this work, we identify a few unique characteristics in SQLite workload and develop a new crash recovery scheme, Database Shadowing which exploits the characteristics of the SQLite DBMS. Aggregate Update, Atomic Exchange and Version Reset collectively make Database Shadowing an effective crash recovery technique for SQLite. When the database file is small and database transactions are serialized, Database Shadowing well addresses the issues in SQLite such as transaction throughput, average and tail latencies of the transaction, recovery time, IO volume and storage space requirements for journaling.

Acknowledgement: We like to thank anonymous reviewers for their valuable feedback. Special thanks go to Lei Chen for his invaluable help in preparing the final manuscript of this work. This work is in part funded by Basic Research Lab Program (NRF-2017R1A4A1015498) and Expert Lab Program (IITP-2018-0-00549).

10. REFERENCES

[1] An introduction to gluster architecture. `http://docs.gluster.org`.

[2] Rename system call. `http://man7.org/linux/man-pages/man2/rename.2.html`.

[3] Samsung galaxy s7 edge specification. `http://www.samsung.com/in/smartphones/galaxy-s7/hardware/`.

[4] Samsung gear s3. `http://www.samsung.com/global/galaxy/gear-s3/`.

[5] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proc. of ACM SIGMOD 2015*, pages 707–722. ACM, 2015.

[6] P. A. Bernstein and N. Goodman. Multiversion concurrency control-theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[8] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[9] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proc. of ACM SOSP 2013*, Farmington, PA, USA, Nov 2013.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[11] A. Craig, G. Soules, J. Goodson, and G. Strunk. Metadata Efficiency in Versioning File Systems. In *Proc. of USENIX FAST 2003*, San Francisco, CA, USA, 2003.

[12] T. Q. Dam, S. Cheon, and Y. Won. On the io characteristics of the sqlite transactions. In *Proc. of IEEE MOBILESoft 2016*, pages 214–224, 2016.

[13] developer.android.com. Safetynet. `https://developer.android.com/training/safetynet/verify-apps`.

[14] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. of ACM SIGMOD 1984*, pages 1–8, New York, NY, USA, 1984.

[15] A. G. Fraser. Integrity of a Mass Storage Filing System. *The Computer Journal*, 12(1):1–5, 1969.

[16] D. Fryer, M. Qin, J. Sun, K. W. Lee, A. D. Brown, and A. Goel. Checking the integrity of transactional mechanisms. *ACM Transactions on Storage (TOS)*, 10(4):17, 2014.

[17] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[18] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[19] I. Heuner. Persistent subsystem. `https://at.project.genivi.org`.

[20] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *Proc. of USENIX winter 1994*, volume 94, 1994.

[21] https://wearos.google.com/. Anrdoid wear. `http://www.android.com/wear/`.

[22] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *PVLDB*, 8(4):389–400, 2014.

[23] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC 2013*, Berkeley, CA, USA, 2013.

[24] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min. X-FTL: transactional ftl for sqlite databases. In *Proc. of ACM SIGMOD 2013*, pages 97–108. ACM, 2013.

[25] T. Kim, H. Ha, S. Choi, J. Jung, and B.-G. Chun. Breaking ad-hoc runtime integrity protection mechanisms in android financial apps. In *Proc. of ACM ASIACCS 2017*, pages 179–192, 2017.

[26] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting nvram in write-ahead logging. In *Proc. of ACM ASPLOS 2016*, volume 50, pages 385–398, New York, NY, USA, Mar. 2016.

[27] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving journaling of journal anomaly in Android I/O: Multi-version B-tree with lazy split. In *Proc. of USENIX FAST 2014*, Santa Clara, CA, USA, 2014.

[28] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proc. of USENIX FAST 2015)*, pages 273–286, Santa Clara, CA, USA, 2015.

[29] K. Lee and Y. Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proc. of ACM EMSOFT 2012*, pages 23–32. ACM, 2012.

[30] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proc. of USENIX ATC 2015*, Santa Clara, CA, USA, 2015.

[31] E. Lim, S. Lee, and Y. Won. Androtrace: Framework for tracing and analyzing ios on android. In *Proc. of ACM INFLOW 2015*, pages 3:1–3:8, New York, NY, USA, 2015.

[32] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, 1977.

[33] H. Luo, L. Tian, and H. Jiang. qNVRAM: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *Proc. of USENIX HotStorage 2014*, Philadelphia, PA, USA, 2014.

[34] lwn.net. Exchanging two files. `https://lwn.net/Articles/569134/`.

[35] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of the Linux symposium 2007*, pages 21–33, 2007.

[36] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[37] C. Min and W.-H. Kang. Lightweight application-level crash consistency on transactional flash storage. In *Proc. of USENIX ATC 2015*, pages 221–234, 2015.

[38] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[39] D. S. Munro, R. C. Connor, R. Morrison, S. Scheuerl, and D. W. Stemple. Concurrent shadow paging in the Flask architecture. pages 16–42, 1995.

[40] G. Oh, S. Kim, S.-W. Lee, and B. Moon. SQLite optimization with phase change memory for mobile applications. *PVLDB*, 8(12):1454–1465, 2015.

[41] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee. Share interface in flash storage for relational and nosql databases. In *Proc. of ACM SIGMOD 2016*, pages 343–354, New York, NY, USA, 2016.

[42] J.-H. Park, G. Oh, and S. W. Lee. Sql statement logging for making sqlite truly lite. *PVLDB*, 11(4):513–525, 2017.

[43] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. on Storage (TOS)*, 9(3):9, 2013.

[44] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[45] O. Rodeh and A. Teperman. zFS-a scalable distributed file system using object disks. In *Proc. of IEEE MSST 2003*, pages 207–218, 2003.

[46] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[47] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, 39(4):447–459, 1990.

[48] Scalzo. *Oracle DBA Guide to Data Warehousing and Star Schemas*. Prentice Hall Professional Technical Reference, 2003.

[49] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proc. of USENIX FAST 2014*, Santa Clara, CA, USA, 2014.

[50] Y. Shi, Z. Shen, and Z. Shao. Sqlitekv: An efficient lsm-tree-based sqlite-like database engine for mobile devices. In *Proc. of IEEE ASP-DAC 2018*, pages 28–33. IEEE, 2018.

[51] SQLite.org. Transaction control at the sql level. `https://www.sqlite.org/lockingv3.html`.

[52] SQLite.org. Well-known users of sqlite. `https://www.sqlite.org/famous.html`.

[53] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *Proc. of USENIX ATC 1996*, volume 15, 1996.

[54] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *Proc.of The Fourth Annual Linux Expo*, Durham, NC, USA, May 1998.

[55] G. Vukov, M. Kovačević, B. Kovačević, and T. Maruna. One solution of event data recorder in car on android operating system. In *Proc. of IEEE TELFOR 2016*, pages 1–4, 2016.

[56] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[57] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proc. of ACM ASPLOS 1989*, pages 243–256, New York, NY, USA, 1989.

[58] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI 2006*, pages 307–320, Seattle, WA, USA, 2006.

[59] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. ANViL: advanced virtualization for modern non-volatile memory devices. In *Proc. of USENIX FAST 2015*, pages 111–118, Santa clara, CA, USA, 2015.

[60] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled IO stack for flash storage. In *Proc. of USENIX FAST 2018*, pages 211–226, Oakland, CA, 2018.

[61] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. *PVLDB*, 10(4):301–312, 2016.

[62] T. Ylönen. Concurrent Shadow Paging: A new direction for database research. 1992.