



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Master's Thesis

Doubleheader Logging: Eliminating Journal Write
Overhead for Mobile DBMS

Se Hyeon Oh

Department of Computer Science and Engineering

Graduate School of UNIST

2020

Doubleheader Logging: Eliminating Journal Write Overhead for Mobile DBMS

Se Hyeon Oh

Department of Computer Science and Engineering

Graduate School of UNIST


Doubleheader Logging: Eliminating Journal Write Overhead for Mobile DBMS

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Se Hyeon Oh

12/16/2019

Approved by



Advisor

Young-ri Choi


Doubleheader Logging: Eliminating Journal Write Overhead for Mobile DBMS

Se Hyeon Oh

This certifies that the thesis/dissertation of Se Hyeon Oh is approved.

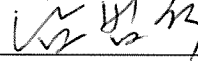
12/16/2019

signature



Advisor: Young-ri Choi

signature



Beomseok Nam: Thesis Committee Member #1

signature



Sam H. Noh: Thesis Committee Member #2

three signatures total in case of masters

Abstract

Various transactional systems use *out-of-place updates* such as logging or copy-on-write mechanisms to update data in a failure-atomic manner. Such out-of-place update methods double the I/O traffic due to back-up copies in the database layer and quadruple the I/O traffic due to the file system journaling. In mobile systems, transaction sizes of mobile apps are known to be tiny and transactions run at low concurrency. For such mobile transactions, legacy out-of-place update methods such as WAL are sub-optimal. In this work, we propose a crash consistent *in-place update* logging method - *doubleheader logging* (DHL) for SQLite. DHL prevents previous consistent records from being lost by performing a copy-on-write inside the database page and co-locating the metadata-only journal information within the page. This is done, in turn, with minimal sacrifice to page utilization. DHL is similar to when journaling is disabled, in the sense that it incurs almost no additional overhead in terms of both I/O and computation. Our experimental results show that DHL outperforms other logging methods such as out-of-place update write-ahead logging (WAL) and in-place update multi-version B-tree (MVBT).

Contents

I	Introduction	1
II	Background and Related Work	4
III	Doubleheader Logging	7
	3.1 In-Page Metadata-only Journal	8
	3.2 Recovery in DHL	11
	3.3 Synchronization: Write Ordering and Durability	12
	3.4 Counting Commit in DHL	13
	3.5 Consistency Guarantee: State Transition of Page in DHL	15
	3.6 Limitations of DHL	16
IV	Experiments	18
	4.1 Experimental Setup	18
	4.2 Experimental Results	18
V	Conclusion	28
	References	29

List of Figures

1	<i>Mobile App Workloads H: Hangout, I: Instagram, Co: Contacts, F: Facebook, G: Gmail, Ca: Calender, K: Kakaotalk</i>	2
2	<i>Slotted-Page Structure with Two Slot Headers</i>	3
3	<i>Walk Through Example of DHL in Action</i>	7
4	<i>Synchronization Steps for Transaction Commit</i>	10
5	<i>Walk Through Example of DHL Counting Commit</i>	13
6	<i>State Transition Diagram (INV: invalid transaction ID, e.g., -1)</i>	15
7	<i>Breakdown of Latency Spent for Insert, Delete and Update Statement in SQLite (AVG. of 1000 Mobibench transactions) O_O: OFF_OPT, C: DHL_CC, T: DHL_TC, M_O: MVBT_OPT, W_O: WAL_OPT, O: OFF, M: MVBT, S: DASH, W: WAL</i>	19
8	<i>Average I/O Volume</i>	20
9	<i>Block Trace of 10 Insert Transactions (Number at Each Block I/O Point Denotes I/O Size in KB)</i>	22
10	<i>Operation Throughput Normalized to WAL as the Number of Operations per Transaction is Varied (AVG. of 1000 Mobibench transactions O: OFF_OPT, C: DHL_CC, T: DHL_TC, M: MVBT_OPT, W: WAL_OPT, S: DASH</i>	23
11	<i>Recovery Time</i>	24
12	<i>CDF of Latencies Spent for Real Workloads</i>	26

I Introduction

In legacy block device storage systems, to retain consistency upon faults, logging or copy-on-write (CoW) techniques have been used to create an isolated snapshot of data and to make changes to it at page granularity. Hence, legacy crash consistency mechanisms perform out-of-place updates and duplicate entire blocks. With these out-of-place mechanisms, Android I/O stack suffers from excessive I/O owing to the *journaling of journal* problem [1], which is a phenomenon where an `fsync()/fdatasync()` issued to journal the SQLite database file triggers another metadata journaling in the file system layer [1–3]. For example, sending a short text message such as “Ok” using a messaging app entails 16 KBytes or 40 KBytes of writes when WAL or PERSIST journal mode is used.

To avoid the journaling of journal problem, various approaches have been proposed to improve the performance of mobile storage systems. For example, new byte-addressable nonvolatile memory (NVM) media has been employed [4–6], the file system has been modified to mitigate the overhead of `fsync()` calls [7–12], and SQLite has been modified to employ other logging methods, such as multi-version B-tree (MVBT) [13], WALDIO (WAL+direct IO) [8], and DASH (DB file shadowing) [14], to reduce the I/O traffic.

Although these works have been shown to mitigate the journaling of journal problem, we believe the previous methods [8, 13, 14] are sub-optimal for mobile workloads. For example, although multi-versioning eliminates the necessity of a separate journal file, the cost of managing and garbage-collecting multiple versions of records are significant and it complicates the management of B+-tree structures, which is tightly coupled with the database engine in SQLite [13, 15]. While multi-versioning is originally designed for highly concurrent database management systems, SQLite makes use of file locks for synchronization and no other embedded database systems support fine-grained concurrency control because there is no significant need for high concurrency in mobile devices. When considering that only one outstanding transaction accesses an entire database file in mobile systems, managing multiple versions of individual records in mobile database systems seems unnecessary overhead.

In mobile devices, it has been reported that a write transaction rarely writes multiple data items because SQLite works in the auto-commit mode by default [4]. In auto-commit mode, each SQL statement becomes an individual transaction that calls `write()` and `fsync()` functions. As was done in previous studies [4, 14, 16], we collected an SQL trace that consists of 323,986 SQL statements from representative mobile applications running on Samsung Galaxy S7 (Android 7.1.2 Nougat and Linux Kernel 3.18.14). Using the trace, we counted how many SQL statements

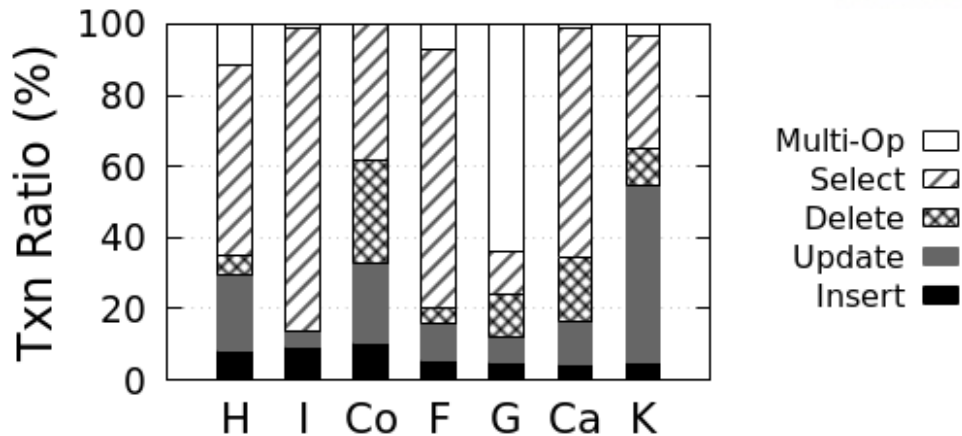


Figure 1: *Mobile App Workloads* H: Hangout, I: Instagram, Co: Contacts, F: Facebook, G: Gmail, Ca: Calendar, K: Kakaotalk

were executed in auto-commit mode. Figure 1 shows that multi-operation transactions account for a very small portion in representative mobile apps except Gmail. As for Gmail, SQLite stores senders, receivers, label names, and mail bodies of a single e-mail into different tables as a single transaction. However, most transactions in the other applications do not batch transactions but commit individual statement automatically. Furthermore, mobile apps frequently open and close DB files, which makes it difficult for WAL to combine and batch a large number of transactions because closing a DB file triggers checkpointing.

Motivated by these observations, we propose a metadata-only journaling scheme termed *doubleheader logging* (DHL) for SQLite, which minimizes, if not eliminates, the logging overhead. DHL is specifically tailored for mobile workloads where concurrency is not a paramount concern and transaction sizes are small. DHL is an in-place update scheme from the perspective of page granularity. However, at the same time, DHL is also an out-of-place update scheme from the perspective of record granularity because it does not overwrite existing consistent records until the write transaction commits. We achieve this goal by leveraging the internal free space of the SQLite database page.

The key idea of DHL is to journal only the metadata of database pages and store updated records, which are not exposed to other transactions until the transaction commits, in the free space of slotted-pages. DHL prevents overwriting the old records in the page until the transaction commits and journals a small amount of metadata, but not the actual data. With the small log size, DHL can co-locate the journal in the slotted-page instead of using a separate rollback journal file or WAL log file, while guaranteeing crash consistency.

We also develop a novel *counting commit* protocol for DHL that embeds a transaction commit mark and the transaction size to eliminate the need for a separate `fsync()/fdatasync()` to

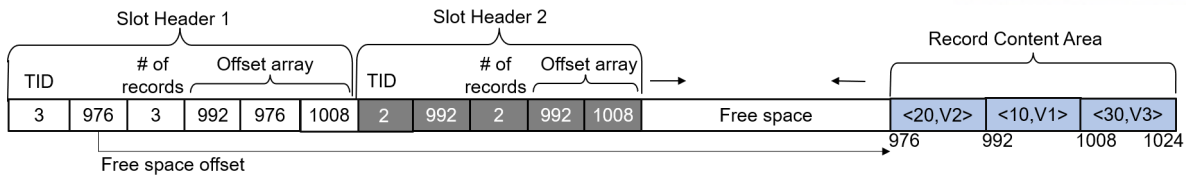


Figure 2: *Slotted-Page Structure with Two Slot Headers*

impose the ordering constraint to dirty page writes. In the counting commit protocol, we count the number of dirty pages flushed by the last transaction and verify that all dirty pages are successfully flushed. By eliminating the ordering constraint, DHL is write-optimal in that it calls `fsync()/fdatasync()` only once and does not duplicate dirty records but only small per-page metadata as the intent of transactions.

We measure the performance of DHL in SQLite on a Samsung Galaxy S7 smartphone. Our performance study shows the following: (i) DHL minimizes the I/O as it calls `write()` exactly the same number of times with the number of updated database pages in a transaction, i.e., one `write()` per each dirty page. (ii) DHL eliminates the overhead of enforcing strict ordering of writes as it requires `fsync()/fdatasync()` only once per transaction. These optimizations combined help resolve the journaling of journal problem as well as reduce wear of flash devices.

The rest of this paper is organized as follows. In Section II, we briefly discuss the background and other research efforts related to this work. In Section III, we present our design and implementation of DHL on SQLite. Section IV presents the performance results and analyses of DHL. Finally, we conclude the paper in Section V.

II Background and Related Work

Slotted-Page: In database systems, a page is essentially the abstraction of a collection of records. With this page abstraction, the buffer cache manager in database systems communicates with I/O devices. To arrange a collection of records in a page, the slotted-page structure [6,17,18] is commonly used in various database systems ranging from mobile DBMSs such as SQLite to server-client DBMSs such as PostgreSQL, InnoDB, and Oracle [6]. The slotted-page structure has a *slot-header* (also referred to as a directory of slots) at the beginning of the page, *free space* in the middle, and a *record content area* at the end of the page. The slot-header contains metadata about the page, e.g., the number of records stored in the page, the end of free space in the page, and an array of offsets that point to records and their lengths.

When a new record is inserted into a slotted-page, space is allocated at the end of the free space extending the record content area. The record's offset is added to the array of record offsets, which grows towards the end of the page. This *level of indirection* in the slotted-page structure allows to minimize the data movement when we insert, modify, or delete records while keeping them in a sorted order. If we insert a key in the middle of sorted keys, the slotted-page structure stores the key in the free space regardless of its value but sorts the array of record offsets by inserting the key's offset in the middle of the array. When we update an existing record in a slotted-page, we overwrite the record and rearrange the offsets according to the updated key value.

Out-of-Place vs. In-Place Logging: In transactional systems, there exist two types of logging approaches, namely, *out-of-place logging* and *in-place logging*. Out-of-place logging methods such as *undo* and *redo* logging store log entries in a separate logging space. In undo logging mode, such as the roll back journal mode in SQLite, i) a transaction creates a copy of the unmodified page to be updated, ii) performs in-place updates (overwrites old pages), and iii) deletes the copy when the transaction commits. In redo logging mode, such as the write-ahead logging (WAL) in SQLite, i) a transaction appends uncommitted dirty pages to a log file, ii) puts a commit mark in the log when the transaction commits, and iii) periodically performs checkpointing in a lazy manner in order to copy the committed pages to the database file in a batch. At the block device level, legacy undo and redo logging methods perform copy-on-write operations at the granularity of a disk block. A problem of copy-on-write operations is that they double the amount of I/O because both the old and new copies that are on different pages need to be written.

In-Page Logging (IPL) is another out-of-place logging scheme designed for flash memory [19]. Although the name implies it is an in-page logging scheme, IPL does not store log entries inside

database pages, but in separate logging space. IPL tackles the erase-before-write limitation of flash memory and exploits the design of the flash translation layer (FTL). In particular, IPL divides each erase unit of flash memory, i.e., a group of pages, into two segments and stores log entries in one segment and data pages in the other segment. Although IPL co-locates log entries and data pages in the same erase unit, log entries are stored separately from data pages. Unlike IPL, DHL is independent of the FTL and it co-locates log entries in data pages. As such, DHL does not duplicate records but copy-on-write per-page metadata in the same block while IPL distinguishes the logging region from the data region and makes copies of records.

Alternatively, write transactions may perform in-place logging by keeping multiple copies of records in the same page. For instance, multi-version concurrency control (MVCC) is one of the most well known in-place logging methods that does not overwrite the original record, but instead creates a new version of the record within the page. One of the benefits of multi-versioning is that it improves the concurrency level as read transactions can access old versions while a write transaction creates a new version. MVCC allows a large number of transactions to access a database table concurrently. However, MVCC is known to hurt space utilization as it stores multiple versions of the same record and therefore, requires garbage collection. More specifically, upon record updates, MVCC writes a page to create a new version of the record instead of overwriting the existing version. Even if the transaction commits and no other transaction accesses the old version, the old version remains in the page until a background garbage collection process detects and deletes it. We note that MVCC is a concurrency control method that provides high concurrency level at the cost of storage and garbage collection overhead. SQLite is a thread-safe embedded database engine that locks the entire database file when it is updated. In embedded database systems, a query normally takes a short time (less than a few milliseconds) [1, 13, 20], and most applications in embedded systems need only low level of concurrency, unlike client-server enterprise database systems. Furthermore, running a periodic background process is not desirable as it may consume unnecessary computing resources and power. In such a mobile environment where storage space is insufficient, legacy MVCC is not a preferred logging method.

Persistent Memory and Mobile DB: In addition to the logging methods for block device storage systems, various efforts have been recently made to leverage the byte-addressability of emerging persistent memory. With persistent memory, out-of-place logging methods do not need to perform copy-on-write operations at the granularity of pages but can be performed at finer granularity [4–6, 21–26]. However, pioneering products, such as Intel Optane DC persistent memory [27], are known to be a storage class memory architected specifically for data center usage, and it is not known when such persistent memory products may be commercially available

for mobile devices. Hence, in this work, we focus on developing a logging scheme for flash memory and embedded database systems.

III Doubleheader Logging

Doubleheader logging (DHL) that we propose is an in-place logging method that performs copy-on-write inside a page by utilizing the slotted-page structure and the free space within the page. Like multi-versioning, it employs transaction IDs to figure out which version is valid. However, unlike MVCC, DHL does not keep multiple versions of records but manages two versions of metadata. Therefore, DHL does not require periodic garbage collection process because running a background process is not an option in mobile devices. DHL is also similar to undo logging in that it makes changes to the original page, and yet, it is different in that DHL does not make a copy of the original page. Also, it is like redo logging in that it journals uncommitted dirty records in free space, and yet, it is different in that the scratch space is within the slotted-page and that there is no checkpointing involved. The key idea of DHL is to keep the old record, the new record, and the commit mark in the same page and write all of them atomically with a single write. Details of how this is done are presented in Section 3.1. As recovery is important for consistency, we discuss recovery in DHL, in detail, in Section 3.2.

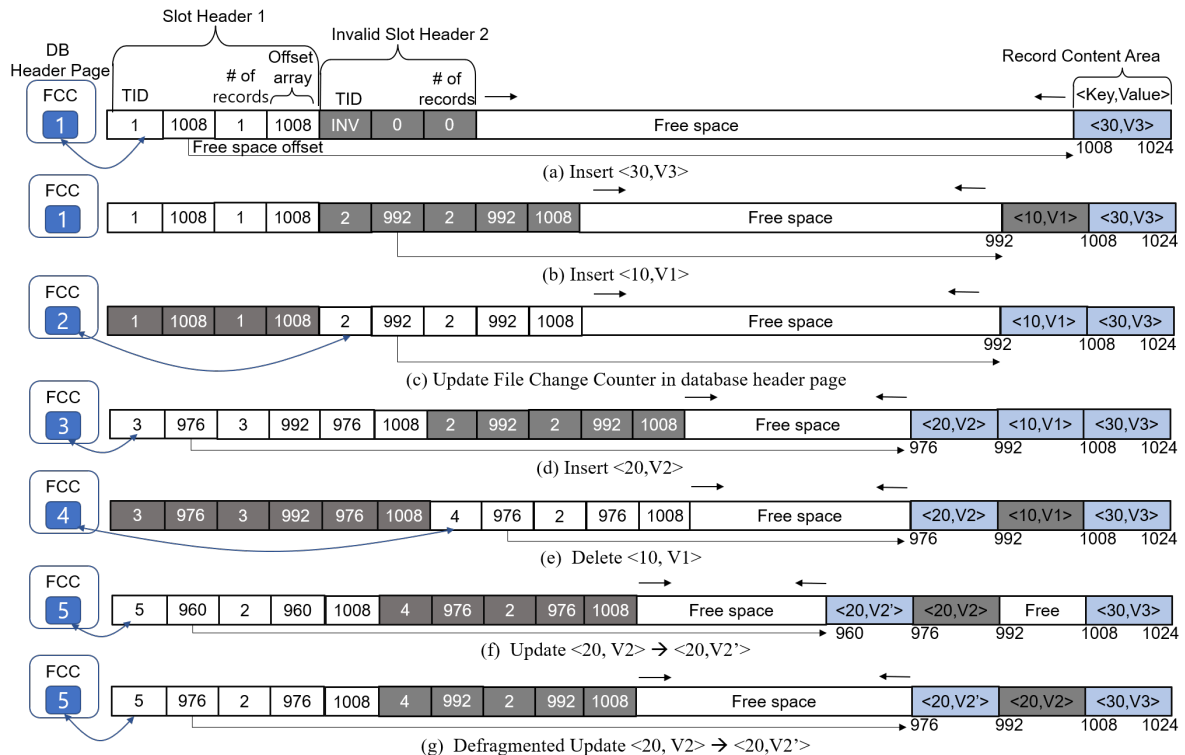


Figure 3: Walk Through Example of DHL in Action

3.1 In-Page Metadata-only Journal

In essence, the key idea of DHL is to write the intent (that is, in our case, the new record) as in redo logging, but write it to the original page as in undo. For this, we propose a technique that we call *in-page metadata-only journaling*. In particular, we propose to have two versions of metadata (double slot-headers, hence, doubleheader, in short) in the slotted-page, as shown in Figure 2. As we will explain later, one of the two slot-headers becomes the journal of the other. Hence, the term metadata-only journaling.

The key idea in maintaining consistency with DHL is being able to identify the more recent and/or committed slot-header of the two through which the valid records in the page will be accessed. For this, each slot-header is tagged with the ID of the transaction, which is assigned in monotonically increasing order, that modified the page. In SQLite, the first page of a database file (database header page) stores global metadata pertaining to the database table, which includes the File Change Counter, henceforth referred to as FCC. FCC is incremented upon every successful write transaction and used for concurrency control. In DHL, we use FCC in assigning a transaction ID to tag slot-headers. In addition to the transaction ID, each slot-header stores, as depicted in Figure 2, the free space offset value, the number of records stored in the page, and the offset array that manages the location of the records in the page. Note that aside from the addition of the transaction ID, each DHL slot-header is identical to the slot-header of the stock slotted-page structure.

In the following, we use the example depicted in Figure 3 to walk through the detailed workings of DHL for each database operation. Before so doing, one important point must be emphasized. That is, in DHL, just like conventional slotted-page structure management, access to the page of interest, whether for reads or writes, occur in volatile buffer cache. Hence, until the changes are reflected into storage, all changes are temporary and the original page can always be recovered from storage. Thus, in essence, recovery from faults is no different from legacy mechanisms used for stock SQLite. DHL does not use any more buffer space than other logging methods. In contrast, WAL uses more memory space to construct WAL frame headers, while MVCC uses more memory and disk space to manage multiple versions. Note that memory is a scarce resource, and more so for mobile devices.

Insert: Assume that the system has been initialized and thus, both slot-headers are initialized and invalid. Let us now consider the case where a transaction creates a page with a single record as depicted in Figure 3(a). As in stock SQLite, the transaction stores the record at the end of the page and updates the first slot-header accordingly. Note that here the TID

for Slot Header 1 is 1. As this transaction ends, the FCC value in the database header page is also set to 1 (through the stock slotted-page mechanism in SQLite) and then persisted. It is this incremented and persisted FCC value that serves as a commit mark.

Now, assume we have a subsequent, new insert operation. The new record is written in the free space (address 992) and the invalid slot-header (Slot Header 2) is overwritten with a new slot-header that is updated appropriately from the current valid slot header (Slot Header 1) (that is, number of records incremented and offset array contents updated with the new record offset). In particular, note that the TID is 2, which is obtained by reading and incrementing the FCC value (which, at the time of updating Slot Header 2, is 1). Figure 3(b) depicts the results. Note again that at this point FCC is still 1 and so, the second insert has not yet been committed. We emphasize again that this is all happening in the volatile buffer cache.

Now to commit the second insert, we first `write()` (page flush) the slotted-page. Note that only single write is required to update both the slot-header and the new record as both are in the same slotted-page. Then, another `write()` is needed to persist the incremented (from 1 to 2) FCC value in the database header to commit the insert, until which time Slot Header 2 and $\langle 10, V1 \rangle$ is still invalid. Figures 3(b) and (c) show the states before and after commit, respectively.

Through similar steps, we show in Figure 3(d) the results of another record being inserted into the page. We see slot-headers taking turns overwriting and validating the previous invalid slot-header (the shaded slot-headers in the figure) with the updated metadata.

Commit: As just described, a transaction commits only with the validation of the TID that updates one of the slot-headers, which, in SQLite, is achieved with the increase of FCC in the header page. Note that persisting FCC requires an extra page `write()` and `fsync()`.

Reference: Upon a reference of a page by a transaction, we compare the transaction IDs of the two slot-headers against the ID of the most recently committed transaction. Note that the more recent slot-header may not always be the consistent one as the transaction that wrote the slot-header may not have yet committed. In Figure 3(b), the more recent slot-header (Slot Header 2) and the new record are not yet exposed to other transactions as the FCC value is smaller than the TID of the slot-header. If the FCC is greater than or equal to the TIDs of both slot-headers, the slot-header with the greater value is the valid one. The number of records and the offset values in this valid slot-header are used to find the record of interest.

Update and Delete: While the insert transaction stores a new record in the free space of the page, the update transaction overwrites an existing record and the delete transaction shifts records to merge the free space. Such an in-place update is not acceptable in DHL because

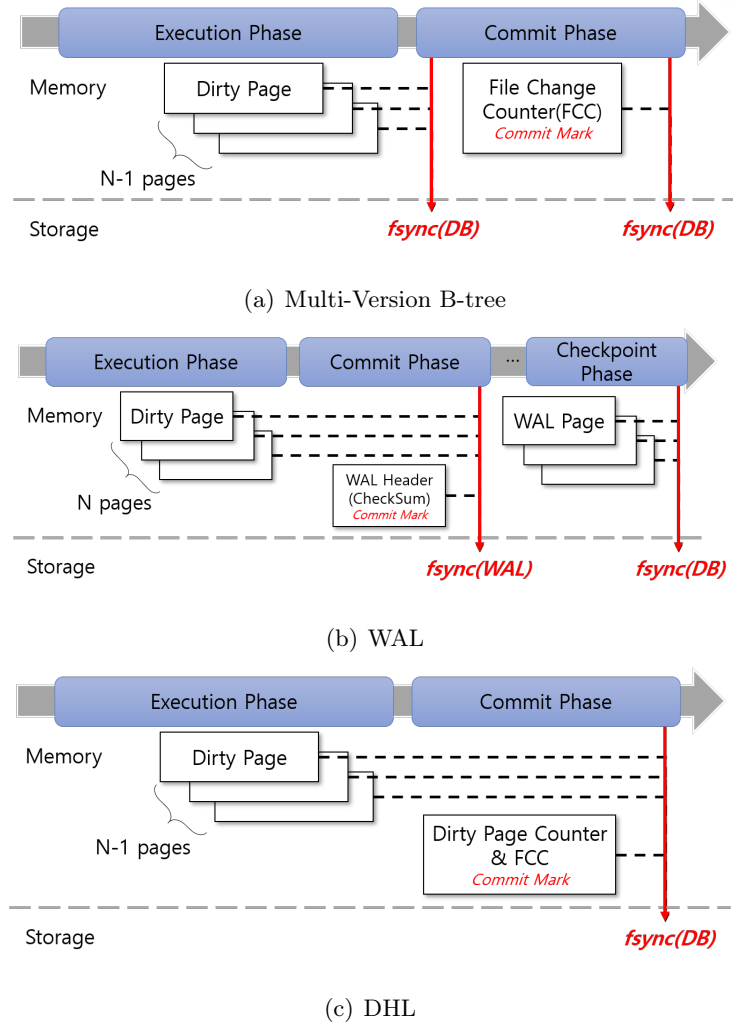


Figure 4: *Synchronization Steps for Transaction Commit*

it needs to keep the old record until the transaction commits. To remedy this issue, DHL uses the free space as an isolated scratch space for write transactions and performs an in-page copy-on-write operation as follows:

(1) Deletes: Assume that Figure 3(d) is the current committed state of the page and we intend to delete record $\langle 10, V1 \rangle$. We first perform an in-page copy-on-write of the valid slot-header Slot Header 1 to Slot Header 2 in volatile buffer cache. Then, appropriate changes are made: the number of records is decremented, an element of the offset array is removed, and the TID set to 4. When the transaction is ready to commit, DHL calls `write()` to output the buffer cache. Notice that all these changes are not reflected until a commit occurs. Hence, if a fault occurs during or after these changes, but before the commit, the page will simply ignore these changes as FCC will be 3, and not 4. Figure 3(e) depicts the state after commit, that is, the page is made persistent and FCC is persisted to 4, which will happen in this particular order. Note that with deletes, the record content area cannot be compacted as it must be recoverable

if a fault occurs in between the page and the FCC being persisted. Hence, this incurs internal fragmentation. However, this can be easily resolved with a subsequent write transaction. Since records in volatile buffer cache are free to move around in the page and their offset values can be updated accordingly, records invalidated by a transaction T can be garbage collected by a subsequent write transaction whose transaction ID is greater than $T + 1$.

(2) Updates: Assume that Figure 3(e) is the current committed state of the page and we intend to update record $\langle 20, V2 \rangle$ to $\langle 20, V2' \rangle$. Similarly to deletes, for updates, we perform in-page copy-on-write of the valid slot-header, and appropriate changes are made. Again, these changes are being done in volatile buffer cache and will be permanently reflected only upon a commit. Hence, fault before the commit will recover to the old slot-header as the FCC is not yet updated. Figure 3(f) shows the state of the page after the commit. Note that instead of overwriting the existing record, the updated record is written in free space, as in inserts. This is because, similarly to delete, we need to recover properly when a fault occurs between the page and the FCC becoming persistent. This, in effect, also incurs internal fragmentation.

3.2 Recovery in DHL

Various system failures such as power loss can occur during DHL execution. If the system crashes as we insert a new record, there can be three outcomes, i.e., (1) the system crashes before we flush the dirty page, (2) the system crashes after we flush the dirty page but before the transaction commits, i.e., the (updated) FCC is not persisted, (3) the system crashes after the transaction commits. First, if the system crashes before the dirty page is written to persistent storage, both the record and its updated slot-header have not been reflected on storage. In this case, the recovery process does nothing. In the second case where the system crashes after the dirty page is written but before the transaction commits, the record is pointed to by the updated slot-header but its TID is not valid, i.e., the transaction did not increase the FCC. To recover from this failure, a recovery process must verify the TIDs of slot-headers and invalidate the updated slot-header whose TID is greater than the FCC. Suppose the system crashes in the state shown in Figure 3(b). To recover from the failure, we scan the page and invalidate the second slot-header by setting the TID of the second slot-header to INV. It is noteworthy that this recovery algorithm does not copy any record but changes a single metadata. Finally, if the system crashes after the transaction updates the page and increases the FCC, the transaction has committed. Thus, recovery is not necessary.

Recovery in DHL has to scan the entire database file as it needs to find the pages that have slot-headers with TID greater than the FCC. However, it updates only a few pages updated by

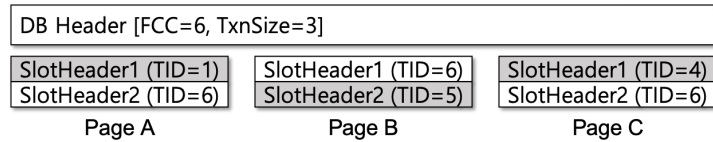
the aborted transactions. Considering that reads are much faster than writes in most storage devices, the recovery time depends on how many uncommitted slot-headers are written to the file. This recovery method is similar to that of LS-MVBT [13], which also scans all pages for sanity check but updates only a few pages that should be rolled back to a consistent version. Such a brute-force recovery method is known to be faster than the legacy recovery method using WAL in mobile applications because database tables in mobile systems are often very small [13]. In Section IV, we measure the recovery time of DHL.

3.3 Synchronization: Write Ordering and Durability

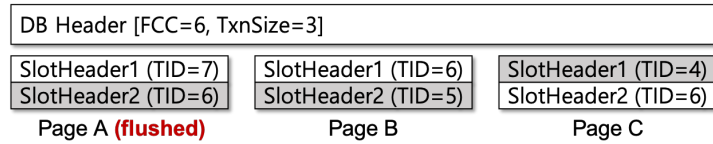
To guarantee transactional consistency, a commit mark must be written after all other dirty pages in the transaction are flushed. To impose the ordering constraint, `fsync()/fdatasync()` needs to be called before a `write()` is called for a commit mark. As illustrated in Figure 4(a) and (b). MVCC calls the first `fsync()/fdatasync()` to enforce the write ordering between dirty pages and a commit mark, and the second `fsync()/fdatasync()` to persist the transaction and notify the client that the transaction has committed. While MVCC is designed for highly concurrent database systems, not all applications require such high concurrency level at the cost of additional storage overhead and garbage collection. In particular, in embedded systems such as mobile devices, running a garbage collection process in the background is not an option. Thus, proposals such as LS-MVBT, a multi-version-based recovery mode for SQLite, employs a lazy garbage collection mechanism [13].

Now, if a database system guarantees isolation between transactions using reader/writer locks and thus, no two transactions concurrently modify the same page, the recovery process simply has to choose one of the two intentions (slot-headers) in the page, i.e., the one with the higher or lower TID. Therefore, keeping only two versions of each page, unlike MVCC that manages multiple versions of each individual record, is sufficient to guarantee failure atomicity.

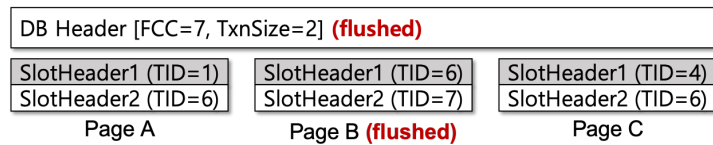
WAL mode in SQLite, on the other hand, takes an optimistic approach. Instead of imposing the ordering constraint between dirty pages and a commit mark, it flushes dirty pages, a commit mark, and their checksum bytes altogether via a single `fsync()/fdatasync()`. Although checksums can detect inconsistent log entries in most cases, there is a small possibility of missing inconsistent states [28].



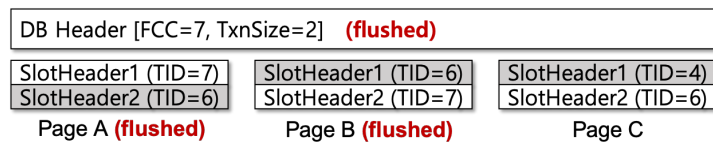
(a) Initial State



(b) Partially Synchronized State



(c) Another Partially Synchronized State



(d) Final State When Transaction (TID=7) Commits

Figure 5: Walk Through Example of DHL Counting Commit

3.4 Counting Commit in DHL

To eliminate the need for a separate flush of a commit mark, cyclic commit protocols such as Simple Cyclic Commit (SCC) and Back Pointer Cyclic Commit (BPCC) have been proposed [29]. In cyclic commit protocols, a transaction is committed only if all dirty records are written and their pointers create a cycle. That is, if any of the dirty pages are not flushed, the pointers will not form a cycle and the transaction is considered to have aborted. In DHL, we take a similar but simpler approach. Instead of creating a cycle, we store the *transaction size* (the number of dirty pages to be flushed together) along with FCC, as shown in Figure 4(c). We refer to this commit protocol as *counting commit*.

The counting commit protocol works as follows. SQLite uses the FCC as the write TID and a simple file-based locking mechanism is used for concurrency control. Therefore, no two concurrent write transactions can update the database file at the same time. Given such low concurrency, which is typical in embedded systems, we can embed the FCC and the number of updated dirty pages in one of the dirty pages. Let us consider the example shown in Figure 5(a), where the database file has three pages A, B, and C. The most recently committed TID is 6 and

the transaction has updated all three pages.

Let us suppose that the next transaction updates pages A and B. Since the current FCC is 6, the new TID becomes 7. According to the DHL algorithm described earlier, one of the slot headers in the two pages are updated. As a commit mark, the new FCC value 7 and the number of dirty pages in the transaction, which is 2, must be written to the database header page, as shown in Figure 5(d). Since DHL does not call `fsync()/fdatasync()` to enforce write ordering, there is no guarantee which dirty page will be flushed first. Nevertheless, DHL still guarantees that it will rollback to the previous consistent state upon a crash. Specifically, there are two cases to consider. First, suppose the system crashes when some and possibly all dirty data pages were flushed but the database header page that stores the FCC and the transaction size was not flushed. An example of such cases is shown in Figure 5(b). When the system recovers, DHL reads the database header page to find the largest FCC, which in this case is 6. Then, the recovery process scans the entire database file to count the number of slot headers whose transaction ID is equal to the FCC, 6. If the number of found slot headers is equal to the transaction size written in the header page, the transaction is valid. While scanning the database file, the recovery process marks any slot header invalid if its TID is greater than the FCC of the header page. In this example, the first slot header (TID=7) in page A will be marked invalid since the transaction 7 aborted before putting a commit mark in the header page.

Second, suppose the system crashes after the header page with the FCC and the transaction size is flushed, but not all other dirty pages were flushed, a case of which is depicted in Figure 5(c). This case is possible since the write ordering is not guaranteed in file systems unless `fsync()` is called twice. When the system recovers, we find the largest FCC in the header page, in this case 7, and count how many slot headers have TIDs equal to the largest FCC. If the transaction size is greater than the number of slot headers that we found, we know that the system crashed before all dirty pages were flushed. In the Figure 5(c) example, the transaction size is 2, but we only find 1 page with FCC value 7. To rollback transaction 7, we count the number of slot headers whose transaction ID is 6, in this case 3, and restore the database header page. To rollback transaction 7, we restore the database header page by decreasing the FCC by one and restoring the transaction size. Note that a slot header whose TID is T is never overwritten by the next transaction whose TID is $T + 1$ in DHL. Therefore, even if transaction $T + 1$ aborts, the number of slot headers written by its previous transaction T is guaranteed to be found. Therefore, decreasing FCC and restoring the transaction size atomically in the database header page guarantees failure-atomicity of transactions and eliminates the need for a separate `fsync()/fdatasync()` for the commit mark.

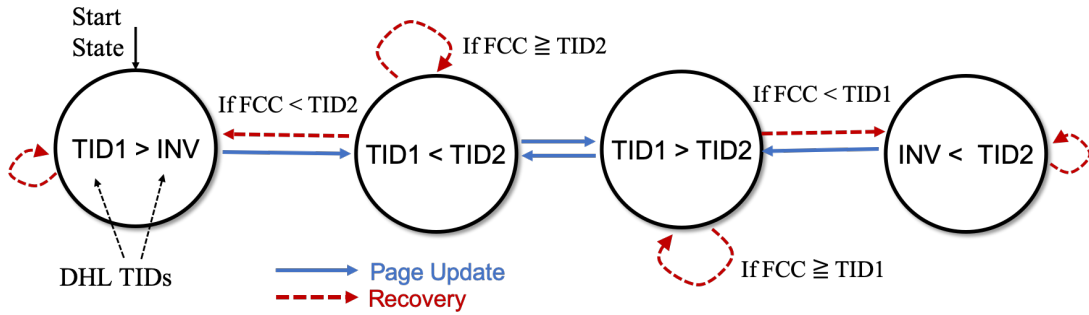


Figure 6: State Transition Diagram (*INV*: invalid transaction ID, e.g., -1)

3.5 Consistency Guarantee: State Transition of Page in DHL

Figure 6 illustrates the states that a DB page can have during its lifetime. Transition edges represent atomic write events under which a DB page changes its state and the rollback activities undertaken during the recovery upon system failures. A page update transition (solid arrow) denotes a write operation, i.e., insertion, deletion, or update of a record in a DB page. A recovery transition (dashed arrow) denotes a recovery operation that invalidates an aborted transaction ID.

Theorem 1 *At least one of the two TIDs in a DHL page is the TID of a committed transaction.*

Proof 1 *In DHL, the TID is monotonically increased by write transactions. Since DHL does not allow write transactions to access dirty pages written by an aborted transaction, a DHL page cannot have two TIDs of aborted transactions. If a DHL page has a TID of an aborted transaction, the other smaller TID must be the TID of a committed transaction.*

According to the theorem, transactions can always access a consistent state of the page. If both TIDs in a DHL page are of committed transactions, we use the more recent higher TID and overwrite the other smaller TID as shown in Figure 6.

Theorem 2 *If both TIDs of a DHL page are transaction IDs of committed transactions ($TID_{old} < TID_{valid}$), the old slot header $slot_header(TID_{old})$ can be overwritten without compromising consistency.*

Proof 2 *As there can only be one outstanding write transaction, rollback does not require two previous consistent states but only the most recent consistent state. Therefore, the slot header of the older transaction $slot_header(TID_{old})$ is not necessary for recovery and, therefore, can be overwritten without compromising consistency. Read transactions also do not need the old slot header because read transactions are serialized by write locks.*

3.6 Limitations of DHL

There are a few limitations to DHL, namely, decreased page utilization, low concurrency, and random writes. We now elaborate on these three issues.

Page utilization: One drawback of DHL is that it needs to store two slot-headers in each page thereby decreasing page utilization. In SQLite, each offset in the slot-header occupies two bytes. If the record size is 8 bytes, a legacy 4KB slotted-page can hold a maximum of 340 records, but DHL does not allow more than 291 records because of the additional slot-header. In other words, when the record size is 8, 16, 32, or 64 bytes, DHL decreases page utilization by 14% (340 vs. 291), 9% (204 vs. 185), 5% (113 vs. 107), and 3% (60 vs. 58), respectively. We do not have a solution to this drawback and consider this issue as a trade-off for improved performance.

Low concurrency: As DHL allows only two slot-headers, unlike MVCC, multiple transactions cannot access the same pages at the same time. Therefore, DHL can be used along with per-page locking methods but not with per-record locking methods. In order for DHL to allow multiple write transactions using lock-based protocols, it would have to manage a list of concurrent write transactions, and write transactions must not overwrite the slot headers updated by other concurrent transactions. However, as was previously discussed, SQLite allows only one exclusive lock on the entire database file rather than individual tables, not to mention individual pages or records. Such coarse-grained locks in SQLite is acceptable because mobile apps rarely run at high concurrency. Thus, our current DHL implementation only allows serial write transactions and does not support concurrency as in other logging methods of stock SQLite.

Random writes: Another potential performance drawback of in-place logging is that it performs random writes, unlike write-ahead logging. WAL mode is known to achieve write efficiency as it amortizes multiple writes into sequential writes. However, when the number of dirty pages to be written is not large and each page write is followed by `fsync()/fdatasync()`, the performance gap between sequential and random writes is not significant and it has even been reported that the performance of the two could be reversed on EXT4 and F2FS [1]. WAL performs sequential writes when it appends WAL frames to the end of the log file. However, it increases the log file size and incurs an update of the filesystem metadata. Because such metadata updates also issue random writes in the filesystem layer, WAL is known to aggravate the journaling of journal problem [1,8]. Furthermore, checkpointing also performs random writes. Therefore, unless a transaction updates a very large number of pages as in an enterprise DB, WAL does not benefit from sequential writes in mobile systems because transactions in mobile

systems are tiny due to the auto-commit [1, 4, 8].

To benefit from sequential writes on NAND flash memory, SQLite needs to run on a file system that results in sequential writes such as a log-structured file system [30, 31] rather than a pseudo-sequential WAL mode. We believe the root cause of the random write problem should be resolved by file systems, not by SQLite. In log-structured file systems, DHL will also perform sequential writes.

Another problem of WAL mode in SQLite is that it calls only a single `fsync()/fdatasync()` similar to our counting commit. We note that MySQL also calls a single `fsync()/fdatasync()` when its WAL is updated. However, we note that WAL mode with a single `fsync()/fdatasync()` is potentially vulnerable to the inconsistency problem because of the probability of undetected errors caused by the checksum bytes. As reported by Zheng et al. [28], WAL mode in SQLite often leads to atomicity violations even without injecting a power fault. This is because write ordering is not guaranteed with a single `fsync()/fdatasync()`. That is, a commit mark of a transaction must be put only after all dirty WAL frames are flushed to storage. This ordering can be enforced only by separately synchronizing the frames and the commit mark.

IV Experiments

We evaluate the performance of DHL against WAL and OFF modes in stock SQLite 3.7. As WAL mode is known to be significantly faster than legacy rollback journal modes such as DELETE and TRUNCATE in all scenarios, we do not show the performance of the rollback journal modes. We set the checkpointing interval of WAL mode to 1000 dirty pages, which is the default in SQLite. We also compare the performance of DHL with DASH [14] and Multi-Version B-tree (MVBT) [13]. DASH [14] is one of the state-of-the-art logging methods for mobile DBMS, which was implemented with 451 lines of code. DASH maintains two database files, one of which behaves as a shadow file. For DHL and MVBT, we implemented approximately less than 1000 lines of code and 8000 lines of code, respectively.

4.1 Experimental Setup

We evaluate the performance of DHL on a Galaxy S7, which has a Samsung Exynos 8 Octa 8890 Processor (2.3Ghz Quad-Core Exynos M1 Mongoose and 1.6Ghz Quad-Core ARM Cortex-A53), 4GB of DDR memory and 32GB UFS 2.0, formatted with the EXT4 file system. We run Mobibench [32] to evaluate the logging methods. Mobibench measures the average insert/delete/update times of 1000 transactions, each of which accesses one or two 128-byte records using a randomly generated key. Unlike TPC benchmarks, Mobibench focuses on the performance of non-concurrent auto-commit transactions. We also evaluate DHL using the real mobile app workloads we collected and presented in Section I. All results reported in this section are averages of five runs.

4.2 Experimental Results

Performance Breakdown

Figure 7 breaks down the query latency into `computation` time, the time elapsed for `write` system calls, and the `fsync` overhead. We observe that the performance of SQLite is dominated by the I/O time in the mobile platform.

The stock WAL mode is much slower than the journal OFF mode because WAL fails to benefit from sequential writes due to tiny transactions. Although WAL writes WAL frames in append-only manner, each WAL frame increases the log file size. Since the increased file size must be reflected to filesystem metadata, the filesystem performs metadata journaling. Thus, contrary to popular belief, WAL mode performs random writes as we will show in Figure 9(a).

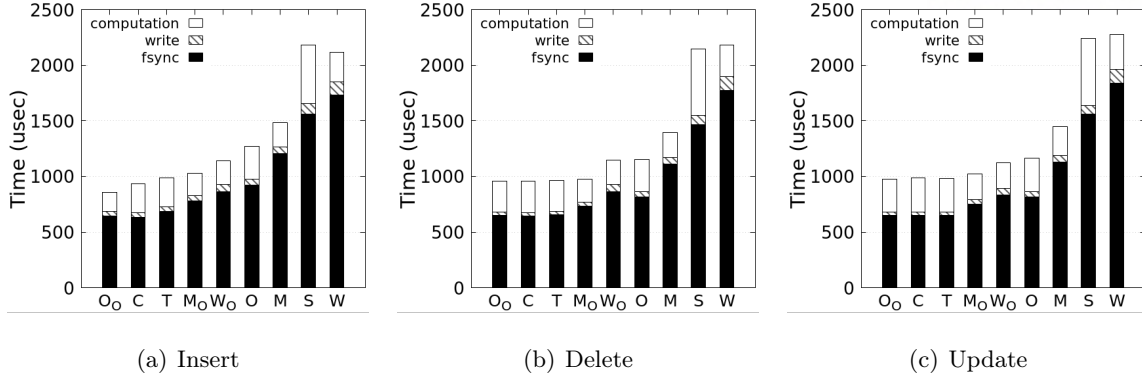


Figure 7: *Breakdown of Latency Spent for Insert, Delete and Update Statement in SQLite (AVG. of 1000 Mobibench transactions)* O_o: OFF_OPT, C: DHL_CC, T: DHL_TC, M_o: MVBT_OPT, W_o: WAL_OPT, O: OFF, M: MVBT, S: DASH, W: WAL

Interestingly, DASH shows similar performance with WAL mode in our experiments. In DASH mode, each transaction obtains a list of dirty pages from the database header page and it compares the list against the list of the previous transaction in the shadow database file. If there are missing dirty pages, they are written to the shadow file along with the new dirty pages updated by the current transaction. Then, DASH swaps the names of the two files per transaction so that the roles of the shadow and database files switch. We found this approach helps reduce the I/O volume especially when subsequent transactions access the same pages repeatedly, but its high computation time offsets the benefits.

MVBT outperforms the WAL mode because it eliminates the need for a separate journal file and weaves version-based recovery information into the database file itself. Since MVBT performs in-place updates, it updates filesystem metadata less frequently than WAL mode. Therefore, the `write()` and `fsync()` time of MVBT is shorter than that of WAL mode.

DHL with the traditional commit, which calls `fsync()` before writing a commit mark, and counting commit, are denoted DHL_TC and DHL_CC, respectively. Overall, both DHL_TC and DHL_CC consistently outperform all other logging methods. Note that the performance of DHL_TC and DHL_CC are no different in these experiments because each transaction updates only one page. An interesting observation in Figure 7 is that DHL outperforms journal OFF mode. This is because of the *metadata embedding*. As we implement DHL in SQLite, we make use of metadata embedding as suggested by Kim et al. [13]. In metadata embedding, the FCC is stored in the most recently updated database page instead of the database header page [13]. Metadata embedding has the effect of reducing the number of dirty pages to be flushed by one. The cost for metadata embedding is that the recovery process has to scan the entire database

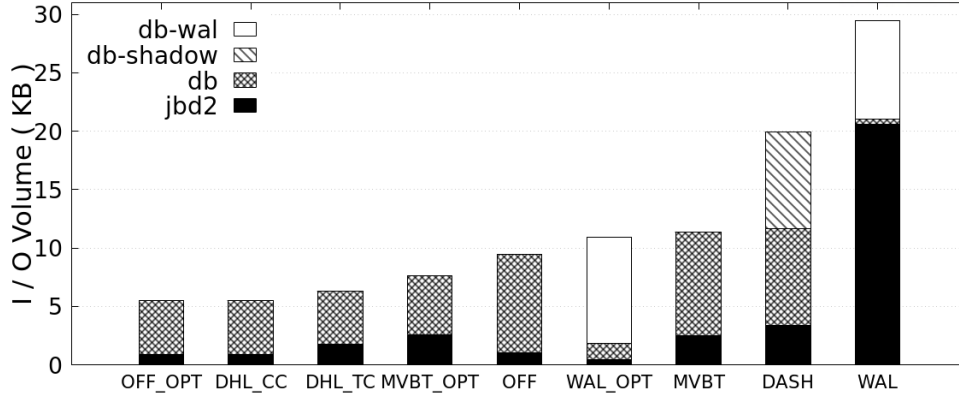


Figure 8: Average I/O Volume

table to find the most recently updated page. Since DHL must scan all the database tables anyway to correct potentially inconsistent pages, the overhead of metadata embedding in DHL is negligible. Note that DHL can be used either with or without metadata embedding.

For fair comparison, we implement metadata embedding in MVBT and journal OFF mode, whose performance are denoted as `MVBT_OPT` and `OFF_OPT`. For WAL mode, we optimized its performance by pre-allocating 100 log pages and setting the checkpointing interval to the same number of dirty pages so that each transaction does not increase the WAL file size at all, and thereby it can avoid filesystem metadata updates although it may waste some disk space per database file. We denote the optimized WAL mode as `WAL_OPT`.

Write Traffic

Figure 8 shows the average disk I/O volume for a single insertion for the same experiments shown in Figure 7. In the stock journal OFF mode, a database page and the database header page need to be updated per transaction. However, in `OFF_OPT` mode, the FCC is updated into the last modified database page. Therefore, `OFF_OPT` mode writes only a single data page to the database file (.db) for each insertion and reduces the number of block accesses by half. Although SQLite does not write a journal file in the journal OFF mode, the EXT4 file system updates the metadata regarding the database file, such as file size and modified time, in the EXT4 metadata journal region (jbd2). Therefore, the total number of bytes written to the block device storage in the journal OFF mode and `OFF_OPT` mode are about 10KB and 6KB, respectively.

In the stock WAL mode, each insertion writes a WAL frame header and a WAL frame (a dirty page), which spans two or three pages in the log file. That is, each insertion updates at least two pages. Hence, WAL mode appends approximately 2 pages (8KB) on average to the WAL file per insertion. As the size of WAL file continuously changes in WAL mode, the I/O

volume to the EXT4 metadata journal region (jbd2) accounts for as much as 68% of the total I/O volume.

In `WAL_OPT` mode, the I/O volume to the EXT4 metadata journal region (jbd2) is dramatically decreased due to the pre-allocation optimization. The total I/O volume of `WAL_OPT` is even comparable to that of journal `OFF` mode. However, due to the additional WAL frame header required for each dirty page, the total I/O volume of `WAL_OPT` is $1.9\times$ higher than that of `OFF_OPT`. Note that the I/O volume to the database file is almost negligible in both WAL modes. This is because WAL mode in SQLite periodically flushes the dirty pages in the volatile buffer cache, but not the WAL frames in the log, when it performs checkpointing. In other words, the number of dirty pages written to the database file is not dependent on the number of WAL frames in the log, but dependent on the number of dirty pages in the buffer cache. Moreover, checkpointing occurs only once for every checkpointing interval. Hence, the I/O volume caused by the checkpointing process is amortized and does not account for more than 17% of the total I/O.

In both DHL, only one page (4KB) is written to the database file per transaction as it writes a single dirty page without a redundant copy, that is, there is no external log file. Even so, recovery is guaranteed because it embeds recovery information into the data page itself. Although `MVBT_OPT` also does not use an external log file, the I/O volume of `MVBT_OPT` is slightly higher than that of DHL because of the large number of B-tree node splits caused by its low page utilization.

Block Trace

Figure 9 shows a more detailed analysis of the I/O traffic to the block device when we run ten transactions, each of which inserts two records. The y -axis indicates the LBA (logical block address), i.e., which file is accessed when. The numbers below each point shows how many consecutive KBytes are accessed, and the vertical line shows when the last transaction commits.

In WAL mode, each transaction writes two 4KB WAL frame pages and an additional 24-byte WAL frame header. As a result, WAL mode flushes at least three pages (12KB) to `.db-wal` file per transaction. Since each transaction increases the WAL file size, numerous metadata blocks are written to the EXT4 journal. We note that WAL mode calls `fsync()` only once per transaction, i.e., stock WAL mode does not guarantee the ordering of commit mark and WAL frames. To prevent this consistency problem, two `fsync()`s must be called, which will aggravate the journaling of journal problem. When we close the database file at the end of the experiments, the checkpointing process updates the database file (`.db`), i.e., 52KB are written to `.db` file.

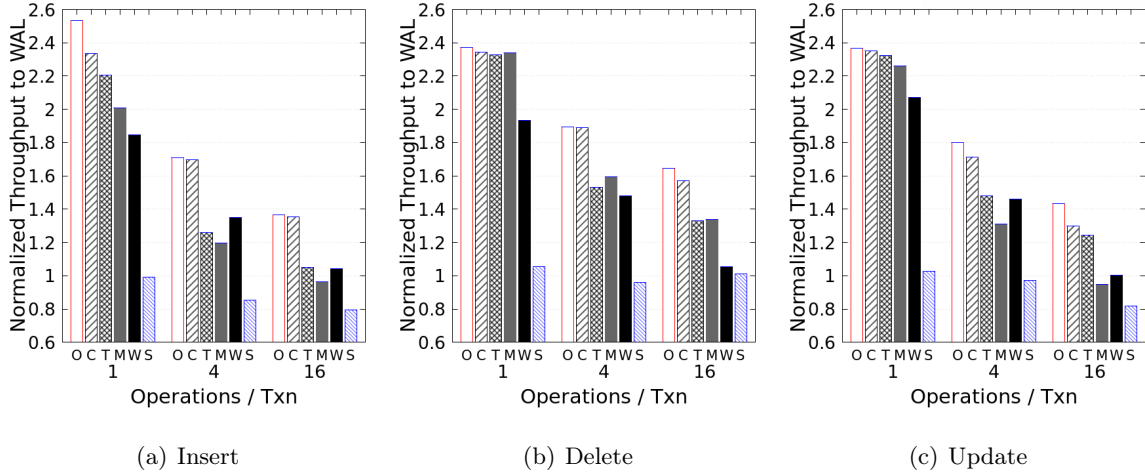


Figure 10: *Operation Throughput Normalized to WAL as the Number of Operations per Transaction is Varied (AVG. of 1000 Mobibench transactions O: OFF_OPT, C: DHL_CC, T: DHL_TC, M: MVBT_OPT, W: WAL_OPT, S: DASH*

utilization, MVBT also suffers from garbage collection. In Figure 9(d), we observe that garbage collection generates additional I/O (i.e., 36KB at 15 msec).

DHL_TC calls `fsync()` twice per transaction as in MVBT. I.e., `fsync()` is called per 4KB page. Similar to MVBT_OPT, DHL also avoids updating the database header page and each transaction updates only two pages, which makes the performance of DHL_TC similar to that of MVBT_OPT. This result shows that DHL_TC effectively eliminates the root cause of the journaling of journal, but through a much simpler approach than MVBT.

In DHL_CC mode, the counting commit protocol eliminates an extra `fsync()`. Therefore, Figure 9(f) shows that `fsync()` is called per 8KB page as each transaction inserts two records. Comparing Figure 9(f) to (g), block trace of DHL_CC is almost identical to that of OFF_OPT. This result shows that DHL_CC is write-optimal.

Transaction Size

In the experiments shown in Figure 10, we vary the transaction size - number of inserts, updates, and deletes per transaction and show the throughput normalized to WAL. As we call more query statements in each transaction, a larger number of pages become dirty and the overhead of `fsync()` increases. As a transaction runs more queries, the performance gap between logging methods decreases because the relative performance of WAL improves with a larger number of writes. This is because the filesystem metadata journaling occurs per transaction, not per operation and the filesystem metadata journaling overhead is amortized over multiple operations. However, as we discussed in Section 3.6, sequential writes have no significant advantage over

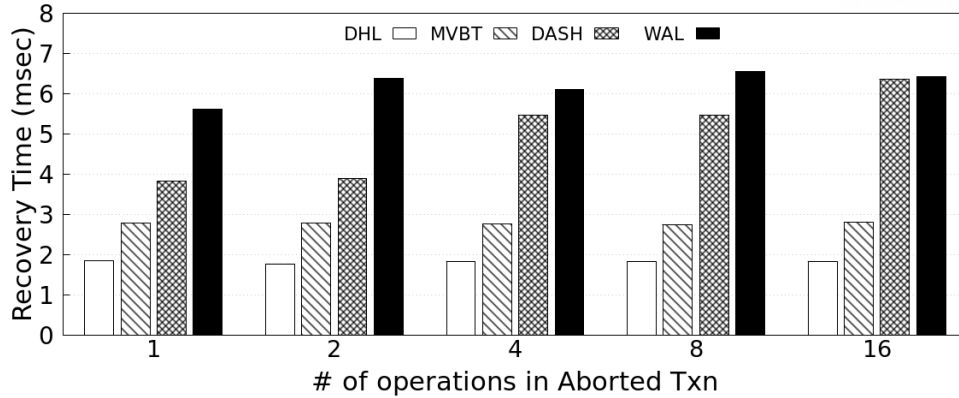


Figure 11: *Recovery Time*

random writes due to metadata journaling. It is noteworthy that `WAL_OPT` shows up to $2\times$ higher throughput than the stock `WAL` mode as it significantly reduces the metadata journaling overhead. In the experiments shown in Figure 10, other logging methods except `DASH` consistently outperform `WAL` mode when a transaction writes no more than 16 pages, which is typical in embedded database systems [1, 13].

When each transaction inserts two data items, the throughput of `DHL_CC` is up to 33% higher than that of `DHL_TC` because the counting commit protocol eliminates the extra `fsync()`. As a transaction inserts more data items, the reduced `fsync()` overhead is amortized over more insertions and the difference in throughput is reduced.

Throughout the experiments, the throughput of `DHL_CC` is consistently similar to that of `OFF_OPT`. For update transactions, `DHL_CC` shows slightly lower throughput than `OFF_OPT` because of the defragmentation overhead, which is purely computational. It is also noteworthy that the performance gap between `WAL_OPT` and `DHL_CC` widens as the transaction size increases because checkpointing is triggered more frequently.

`MVBT` shows similar throughput with `DHL_TC` for delete transactions, but lower throughput than `DHL_TC` for insert and update transactions because of low page utilization and garbage collection overhead.

Recovery

In the experiments shown in Figure 11, we measure the latency for crash recovery. Since database tables used in mobile apps are often very small [4, 14], we warm up a database table with 1200 records and then inject a fault while an insert transaction is running. We vary the number of inserted records in the faulty transaction from one page to sixteen pages. When the database is accessed by a subsequent transaction, each scheme triggers a recovery process.

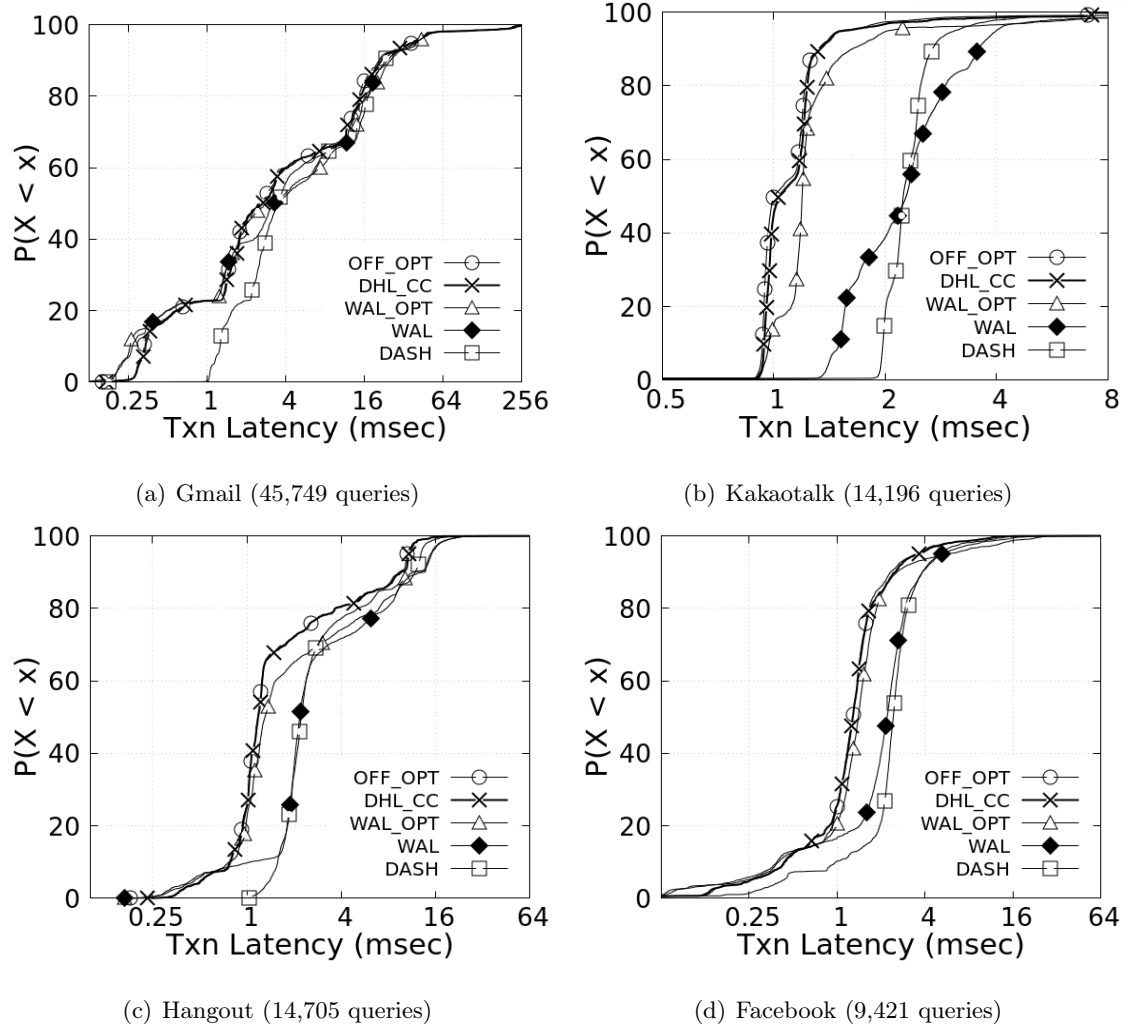
Our experiments shown in Figure 11 show that when a transaction aborts while inserting 2, 4, 8, or 16 records of 128 bytes, its recovery takes about 6.3 msec in WAL mode, which is about $3.5\times$ longer than the recovery latency of DHL (1.8 msec). The recovery process in WAL mode not only checkpoints committed transactions but also updates the inode table and block bitmap for the WAL file. In the experiments, the number of WAL frames in the log file is about 455. However, only a small portion of the WAL frames are checkpointed to the database file because of locality, i.e., many subsequent transactions access the same database page. Note that the recovery time in WAL mode takes only 6.3 msec while an auto-commit insert transaction takes about 2.1 msec on average, as shown in Figure 10(a). Although the recovery overhead of WAL mode is not significant, WAL is outperformed by other modes, i.e., DASH, MVBT, and DHL.

The recovery time of DHL is even shorter than that of DASH and MVBT because MVBT checks the version of every individual record including dead records and DASH has to reconstruct the shadow file from the database file, but DHL compares only two transaction IDs against the FCC value.

Real Workloads

In the experiments shown in Figure 12, we measure the transaction latencies of real workloads - the SQL trace that we collected from representative mobile applications - Gmail, Facebook, Hangout, and Kakaotalk. These four SQL traces have different characteristics. As shown in Figure 1, more than 60% of the queries in Gmail are multi-operation transactions and the average number of operations per transaction is 21.9. In Facebook, about 70% of the queries are select transactions in auto-commit mode, i.e., single select queries. Hangout also has about 50% single select queries, but about 20% of the queries are update transactions in auto-commit mode. In KakaoTalk, an instant messaging app, more than 50% of the queries are update transactions in auto-commit mode. Since the logging methods do not affect select query latencies, we do not show the latencies of select queries in auto-commit mode, but only present the latencies of select queries in multi-operation and write transactions. We do not show the performance of MVBT because the B-tree implementation in SQLite is tightly coupled with the database engine and our MVBT implementation fails to process complex real workloads. Also, we do not show the results of DHL_TC for easier readability, but note that its latencies are slightly higher than those of DHL_CC.

Overall, DHL_CC and OFF_OPT show comparable performance and they consistently show the best behavior whereas stock WAL mode and DASH suffer from high tail latency because of journaling of journal overhead and large I/O traffic.

Figure 12: *CDF of Latencies Spent for Real Workloads*

Although WAL_OPT improves significantly over WAL, it still falls short of DHL_CC. We note that the presented performance of WAL_OPT is close to the ideal performance of WAL_OPT because we let WAL_OPT pre-allocate the number of log pages necessary for each checkpointing. Note that we do not want WAL_OPT to pre-allocate a larger number of log pages than the checkpointing interval because it will unnecessarily write unused log pages to a block device resulting in increased I/O traffic. In contrast, if we choose to pre-allocate a smaller number of log pages than the checkpointing interval, WAL_OPT has to periodically increase the log file size, which will result in filesystem journaling resulting in increased query response time due to random writes. In our workloads, the average checkpointing intervals for Gmail, KakaoTalk, Hangout, and Facebook are approximately 84, 147, 23, and 220 pages, respectively. These checkpointing intervals can vary depending on users' usage patterns because SQLite triggers checkpointing if a user closes a mobile app. Therefore, we varied the number of pre-allocated log pages according to each app's average checkpointing interval such that it can avoid filesystem journaling and benefit from

pure sequential writes. Although the checkpointing interval of SQLite is not deterministic and there is such a trade-off between filesystem journaling overhead and unnecessary pre-allocation overhead, WAL_OPT has to choose a fixed pre-allocation size. However, with a fixed pre-allocation size, the performance of WAL_OPT will be worse than those presented in Figure 12.

For the Gmail workload where multi-operation transactions are dominant and only about 20% of the queries are single write transactions, the latencies of all logging methods, except DASH, are similar. This result is consistent with the results shown in Figure 10. As the transaction size increases, the performance gap between logging methods decreases.

For the KakaoTalk, Hangout, and Facebook workloads, where single write transactions are dominant, DHL_CC and OFF_OPT outperform other logging methods. In particular, DHL_CC shows superior performance to other logging methods especially in KakaoTalk. This is because the transaction size in the KakaoTalk workload is much smaller than the other workloads, i.e., the average number of operations per transaction in KakaoTalk is only 1.76 whereas they are 3.99 and 2.98 in Hangout and Facebook, respectively.

V Conclusion

In this work, we proposed a novel metadata-only journaling scheme for mobile database systems, which we call doubleheader logging (DHL). DHL enables in-place updates without redundant writes caused by external logging or copy-on-write methods. DHL co-locates minimal recovery information and data on the same page, thereby avoiding redundant copies while, at the same time, guaranteeing crash consistency.

Our performance study shows that DHL outperforms WAL mode, DASH, and Multi-Version B-trees in Samsung Galaxy 7 smartphone. DHL is a write-optimal recovery method in the sense that the number of calls to `write()` is no higher than that of journal OFF mode. Regardless of the underlying file system, DHL effectively eliminates the need for an external log file and the root cause of the journaling of journal problem.

References

- [1] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, “I/O stack optimization for smartphones,” in *Proceedings of the USENIX Annual Technical Conference*, 2013.
- [2] H. Kim, N. Agrawal, and C. Ungureanu, “Revisiting storage for smartphones,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.
- [3] K. Lee and Y. Won, “Smart layers and dumb result: Io characterization of an android-based smartphone,” in *Proceedings of the 12th International Conference on Embedded Software (EMSOFT 2012)*, 2012.
- [4] G. Oh, S. Kim, S.-W. Lee, and B. Moon, “SQLite optimization with phase change memory for mobile applications,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1454–1465, 2015.
- [5] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, “NVWAL: Exploiting NVRAM in write-ahead logging,” in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [6] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, “Failure-atomic slotted paging for persistent memory,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [7] K. Shen, S. Park, and M. Zhu, “Journaling of journal is (almost) free,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.
- [8] 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 *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015.
- [9] J. Ren, C.-J. M. Liang, Y. Wu, and T. Moscibroda, “Memory-centric data storage for mobile systems,” in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2015, pp. 599–611.

- [10] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, “Weardrive: Fast and energy-efficient storage for wearables.” in *USENIX Annual Technical Conference*, 2015, pp. 613–625.
- [11] D. Park and D. Shin, “iJournaling: Fine-grained journaling for improving the latency of fsync system call,” in *2017 USENIX Annual Technical Conference*, 2017, pp. 787–798.
- [12] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, “Barrier-enabled IO stack for flash storage,” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, 2018, pp. 211–226. [Online]. Available: <https://www.usenix.org/conference/fast18/presentation/won>
- [13] 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 *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.
- [14] Y. Won, S. Kim, J. Yun, D. Q. Tuan, and J. Seo, “Dash: Database shadowing for mobile dbms,” *Proceedings of the VLDB Endowment*, vol. 12, no. 7, pp. 793–806, 2019.
- [15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, “Consistent and durable data structures for non-volatile byte-addressable memory,” in *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.
- [16] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, “X-FTL: transactional FTL for SQLite databases,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 97–108.
- [17] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. New York, NY, USA: McGraw-Hill, Inc., 2005.
- [18] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. McGraw-Hill, 2005.
- [19] S.-W. Lee and B. Moon, “Design of flash-based dbms: An in-page logging approach,” in *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.
- [20] “Sqlite,” <http://www.sqlite.org/>.
- [21] J. Arulraj, M. Perron, and A. Pavlo, “Write-behind logging,” *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

- [22] J. Xu and S. Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories,” in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
- [23] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee, “Better I/O through byte-addressable, persistent memory,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.
- [24] E. Lee, H. Bahn, and S. H. Noh, “Unioning of the buffer cache and journaling layers with non-volatile memory,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.
- [25] J.-H. Park, G. Oh, and S.-W. Lee, “SQL statement logging for making SQLite truly lite,” *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 513–525, 2017.
- [26] Y. Zhang and S. Swanson, “A study of application performance with non-volatile main memory,” in *Proceedings of the 31st International Conference on Massive Storage Systems (MSST)*, 2015.
- [27] “Intel and Micron produce breakthrough memory technology,” <https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology>.
- [28] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, “Torturing databases for fun and profit,” in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Oct. 2014, pp. 449–464.
- [29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, “Transactional flash,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 147–160. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1855741.1855752>
- [30] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.
- [31] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2fs: A new file system for flash storage,” in *13th USENIX Conference on File and Storage Technologies*, 2015, pp. 273–286.
- [32] “Mobibench,” <https://github.com/ESOS-Lab/Mobibench>.

