In-Page Shadowing and Two-Version Timestamp Ordering for Mobile DBMSs

Lam-Duy Nguyen
Sungkyunkwan University
Suwon, Korea
duynguyen269@skku.edu

Sang-Won Lee
Sungkyunkwan University
Suwon, Korea
swlee@skku.edu

Beomseok Nam
Sungkyunkwan University
Suwon, Korea
bnam@skku.edu

ABSTRACT

Increasing the concurrency level in mobile database systems has not received much attention, mainly because the concurrency requirements of mobile workloads has been regarded to be low. Contrary to popular belief, mobile workloads require higher concurrency. In this work, we propose novel journaling and concurrency mechanisms for mobile DBMSs, both of which build upon one common concept - *In-Page Shadowing* (IPS). We design and implement a novel In-Page Shadowing recovery method for SQLite to resolve the journaling of journal anomaly, which is known to quadruple the I/O traffic in mobile devices. IPS unions the previous and the next versions of a database page in the same physical page. Using the consolidated two versions of database page, we design *Two-Version Timestamp-Ordering* (2VTO) protocol that enables non-blocking reads as in multi-version concurrency control, but reduces the garbage collection overhead. Designed with mobile environments in mind, IPS and 2VTO are high-performant and resource-efficient transactional solutions. Our performance study shows that IPS and 2VTO outperform state-of-the-art logging methods and an optimistic concurrency control protocol for real mobile workloads.

PVLDB Reference Format:
Lam-Duy Nguyen, Sang-Won Lee, and Beomseok Nam. In-Page Shadowing and Two-Version Timestamp Ordering for Mobile DBMSs . PVLDB, 15(11): 2402 - 2414, 2022.
doi:10.14778/3551793.3551802

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DICL/sqlite.

1 INTRODUCTION

With the growth of IoT connectivity and wireless display technology, mobile devices are now replacing desktop computers for a variety of performance-intensive tasks. With this trend, multithreading has become a must for mobile apps to meet the performance expectations[1, 10, 21, 25]. To improve the performance and usability, mobile processors have transitioned to multi-core processors since 2010. This is because power efficiency is paramount for mobile devices and multi-core processors use less battery power for the same amount of computation than single-core processors with higher clock frequencies.

For mobile apps, SQLite has become the de facto standard storage interface. SQLite is a serverless library database engine that provides transactional support for numerous mobile apps. However, SQLite provides poor concurrency in multithreaded apps because it uses coarse-grained file-based locking and allows at most one thread to acquire an exclusive lock on the entire database file. Although a growing number of modern mobile apps use multi-threading, SQLite serializes concurrent accesses to database files, degrading the performance of multithreaded mobile apps. The concurrency level required by mobile apps is not as high as server-client environments, but our study shows that the file-based locking in SQLite is too coarse-grained for core Android mobile apps and it fails to leverage IO parallelism and multi-core processors.

This shortcoming is an undesirable consequence of taking a simplistic approach to transaction atomicity, isolation, and durability. Recall that, as an embedded database system, SQLite is designed to simplify its code, to minimize the memory footprint, and to reduce the use of battery and computing resources. Therefore, SQLite is a set of in-process library functions and does not require background server processes or threads. Without using background server processes, SQLite relies on the operating system's file locking to serialize concurrent writes to database tables. Specifically, SQLite has to acquire exclusive locks on WAL and database files to ensure atomic propagation of multiple dirty pages to the database. As a result, SQLite poses two performance problems. First, concurrency is poor because the dependency on the WAL/database file allows only one writer transaction at a time. Second, the *force policy* exacerbates the write amplification problem of NAND flash[18, 26], i.e., if SQLite writes a log or journal using the force policy, the underlying Linux file system triggers another journaling to log its file system metadata. This phenomenon of *journaling of journal* has been reported to quadruple the I/O traffic, degrading the I/O performance of the Android I/O stack[13].

In order to alleviate the journaling of journal problem and improve SQLite's I/O performance, various efforts have been made to reduce the number of calls to `fsync()` in the SQLite layer (e.g., database file shadowing (DASH)[34], multi-version B-tree[16], and doubleheader logging (DHL)[24]). In addition, several efforts have been made to reduce the `fsync()` overhead in the file system layer[19, 31]. Although these previous works have been shown to be effective in mitigating the journaling of journal problem, they neglect the concurrency control management, despite the close relationship between database logging and concurrency control. That is, each log entry must be applied in serializable order [22].

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551802

In this work, we advocate the *in-page logging* scheme, which not only resolves the journaling of journal anomaly but also improves the concurrency level of SQLite for resource-constrained mobile devices. Specifically, we present *In-Page Shadowing* (IPS) that places the per-record redo log entry at the internal free space of the corresponding database page itself. IPS consolidates an original page and its shadow copy into a single physical page. Thereby, unlike conventional shadow paging techniques that perform page-level copy-on-writes, IPS eliminates the need for separate log pages (and files) and thus completely removes the logging I/O overhead. Furthermore, by providing two logical versions with a single physical page, IPS enables a novel multi-version concurrency control scheme, *Two-Version Timestamp-Ordering* (2VTO), which allows concurrent read transactions to proceed without being blocked. In addition to improving the concurrency level of SQLite, 2VTO reduces the garbage collection overhead. To sum up, IPS and 2VTO are resource-efficient transactional solutions suitable for mobile environment. Contributions of this study are summarized as follows.

- First, by consolidating the unmodified version of a database page and its shadow page on a single physical page, IPS resolves the journaling of journal problem, achieving the single-write journaling and thus reducing the IO traffic in half.
- Second, a positive side effect of IPS in terms of concurrency control is that IPS does not relocate committed records and metadata, enabling concurrent non-blocking reads. To support a moderate level of concurrency, we propose *Two-Version Concurrency Control* (2VTO), a lightweight page-level concurrency control protocol for mobile database systems. 2VTO only manages two versions of database pages, allowing non-blocking reads and simple but efficient rollback operations.
- Third, we propose a novel starvation avoidance mechanism for 2VTO, named *Abortee-or-First-Updater Wins* (AFUW) rule. Classic First Writer/Updater Wins (FCW/FUW) rules abort the same transactions repeatedly, which is undesirable and thus should be refrained in resource-constrained mobile devices. AFUW rule is a pessimistic starvation avoidance mechanism that limits the number of aborts per transaction to at most one.

The rest of this paper is organized as follows. In Section 2, we discuss the background and motivations. In Section 3, we present the design and implementation of IPS. In Section 4, we describe 2VTO concurrency control protocol and AFUW starvation avoidance rule. In Section 5, we evaluate the performance of IPS and 2VTO with AFUW using synthetic and real mobile workloads. Finally, we conclude the paper in Section 6.

2 BACKGROUND AND MOTIVATIONS

2.1 Slotted-Page Structure and SQLite

The slotted-page structure [28, 30] is a page format commonly used to store a set of variable-length records. The slotted-page structure adds a *level of indirection* in database pages to manage variable-length records (in sorted order in SQLite) while minimizing data movement upon record insertions, updates, or deletions. The slotted-page structure has a metadata region, called *slot-header*, at the beginning of page, *free space* in the middle, and an array of variable-length records (slots), called *records content area* at the end of page. The slot-header contains metadata about the database page, i.e., the number of records stored in the page (`numSlots`), the end of free space (`free_offset`), and an array of offsets (`arrSlots[]`) that point to records and their lengths (similar to Figure 2).

In SQLite, a transaction updates a database table at page granularity, and the changes made to dirty pages are *forced* to be written to disks. Even if a transaction inserts a small 8-byte record into a database page, it creates a copy of the page and inserts the record into one of the two copies while the other copy (original database file or a journal file) remains intact [14]. Such logging at page granularity not only results in duplicate writes of unmodified records inside the page, but also doubles the number of `fsync()` calls.

Although SQLite updates database tables at page granularity, SQLite prevents concurrent read transactions from accessing clean pages if a write transaction is writing to another page in the same database file. This is because SQLite uses file-based locking to control concurrent accesses, even for multi-threaded environments.

2.2 Journaling of Journal Anomaly

Traditional database systems and file systems have used logging (journaling) or copy-on-write (CoW, shadowing) to create modifiable, isolated snapshots of data. Logging and CoW are particularly expensive in SQLite since the force policy is used with the Linux file system[13, 16, 19]. Since a large portion of write transactions in mobile apps run in *auto-commit mode*, each insert, update, or delete statement becomes an individual transaction. In this mode, each write query calls expensive `fsync()` system calls to synchronize its small changes to a write-ahead log (WAL) or rollback journal file, and then the database file at page granularity. Linux file systems then double the database logging overhead because it does journaling to protect the file system metadata, i.e., it allocates a free block for the new log page and journals the corresponding inode information. As a result, storing a short message such as "Hi" in WAL mode results in sixteen dirty pages (16 KB) in the Linux filesystems[13]. Specifically, SQLite writes at least two pages i.e., one for the log file and the other for the database file (i.e., 2x amplification). For each file, the file system writes at least one metadata journal page (i.e., 4x amplification). To aggravate the problem, the Flash Translation Layer in flash devices amplifies the internal write due to garbage collection. This problem is called *journaling of journal anomaly*[13].

2.3 Concurrency Requirements in Mobile Apps

Numerous mobile apps run multiple threads that access the same database tables. Figure 1 shows the characteristics of SQLite transactions (434,164 SQL statements) that we collected from representative mobile apps running on a Samsung Galaxy S10 (Android 10), as was done in the previous studies [14, 23, 34]. To measure the ratio of read-write and write-write conflicts between transactions, we tagged each SQL statement with the thread ID and analyzed if multi-query transactions from different threads overlap in time.

In the trace, contrary to popular belief, we find the demand for concurrency in mobile apps is not negligible mainly due to *Google Play Services*. Google Play Services allow third-party Android apps to utilize Google APIs for various Google services, such as authentication, contact lookups, access to user privacy settings, location-based services, and many more[2, 7, 20]. Our traces shows that each application depends to varying degrees on Google Play

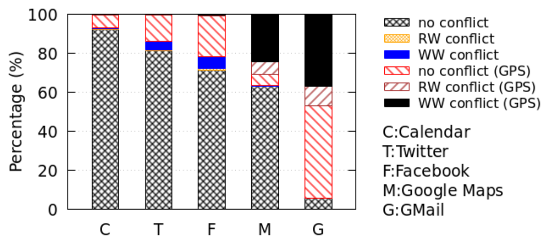


Figure 1: Conflicts in Concurrent Mobile Workloads

Services. Figure 1 shows that Gmail delegates more than 90% of transactions to Google Play Services, and about 47% of transactions conflict (37% write conflicts and 10% read conflicts), and Google Maps also delegates about 47% of transactions to Google Play Services. Being used by most mobile apps, Google Play Services is reported to be the process that submits the most SQL queries to SQLite in Android devices [15]. The SQL traces we collected show the same results as about 80.4% of SQL transactions are from Google Play Services. The number of concurrent transactions in Google Play Services is higher than the other apps (i.e., up to 5 vs. up to 2). Therefore, the database lock contention in Google Play Services is high; overall, 48.85% of Google Play Services transactions conflict. Given that Google Play Services is a core service in Android, our observation indicates the need to design a new concurrency control protocol for SQLite.

2.4 Concurrency Control for Mobile Databases

For scalable enterprise database systems, multi-version concurrency control (MVCC), and its variants such as MVTO [29], MVOCC [17], MV2PL [6], SSI [8], and SSN [32], are commonly used because they enable non-blocking access and provide high concurrency, i.e., writers do not block readers, and vice versa.

However, mobile computing devices pose unprecedented challenges to database systems because of their resource-constrained environments. In particular, power consumption is one of the most significant challenges in mobile computing devices, and a small number of cores is another [9, 33]. In mobile computing devices, the CPU is one of the largest power consumers, along with the backlight and display components as well as cellular network components. Carroll et al. [9] showed that CPUs consume more than 10× power than NAND flash storage for I/O intensive benchmarks, and computation-intensive workloads consume 2× more power than the backlight at maximum brightness level[9]. This result indicates that running background processes or threads that constantly consume CPU cycles should be avoided whenever possible.

Classic MVCC protocols require background threads for garbage collection (GC) to detect and delete obsolete versions. Therefore, MVCC, which requires garbage collectors, is not suitable for mobile devices. Besides this, if a database table has a large number of versions scattered over different pages, queries have to traverse a long version chain, which incurs more disk I/O and worsens query response time. As mentioned earlier, the number of concurrent transactions in Android mobile apps is not as high as server-client enterprise database systems. Therefore, the classic MVCC is inappropriate for mobile devices in terms of performance and energy.

3 IN-PAGE SHADOWING

In order to reduce the number of writes and the I/O traffic, *In-Page Shadowing* (IPS) obviates costly copy-on-write or journaling, but performs in-place updates at page granularity. In particular, IPS appends uncommitted records to the free space on the page, allowing undo operations and guaranteeing consistency. In this sense, IPS takes an out-of-place update approach at record granularity. DHL [24] is another logging method that performs in-place updates at page granularity but performs out-of-place updates at record granularity. In this section, we present how IPS works. The similarities and differences between DHL and IPS will be discussed in Section 4.1.

3.1 Data Structure for In-Page Shadowing

IPS requires three additional metadata in the slotted-page structure, as shown in Figure 2, i.e., (1) *transaction ID* (`TID`) of the transaction that modified the page (i.e., the last modified timestamp), (2) two versions of *number of slots (records) in use* (`numSlots[0]` and `numSlots[1]`), and (3) two versions of *tombstone bitmaps* (`TB[0]` and `TB[1]`).

In SQLite, each transaction is assigned with a monotonically increasing transaction ID (TID). TID is generated by reading and incrementing the *File Change Counter* (henceforth referred to as FCC) which is stored in the first page (database header page) of a database file. In SQLite, FCC plays the role of the commit timestamp (TS_{commit}) of the latest write transaction. Upon commit of every successful write transaction, SQLite increments the FCC and uses it to serialize concurrent transactions. In each page, `numSlots[0]` and `numSlots[1]` take turns representing the current and the previous number of valid slots, i.e., one of the two `numSlots` becomes the journal of the other. Tombstone bitmaps (`TB[0]` and `TB[1]`) are used for deletions, i.e., each bit indicates whether its corresponding slot is valid or not. Similar to `numSlots[]`, the two tombstone bitmaps take turns becoming the journal of the other. To indicate which index of `numSlots[]` and `TB[]` is more recent, we reserve one bit (denoted as `F`) in the TID field, and flip the bit flag every time a write transaction updates the page. For example, if `F` is `0`, subsequent transactions consider `numSlots[0]` and `TB[0]` are more recent than `numSlots[1]` and `TB[1]`. If `F` is 1, it is the opposite case.

However, the more recent `numSlots` and `TB` are not always committed ones, i.e., even if a write transaction updates and flushes a database page to a file, it may fail to update another page or put a commit mark (i.e., increment the FCC). Therefore, the bit flag alone does not provide enough information to determine which `numSlots` and `TB` are more recently committed metadata. When a transaction accesses a database page, IPS determines whether the more recent `numSlots` and `TB` are committed metadata or not by comparing the page's TID (last modified timestamp) against the transaction's start timestamp (TS_{start}), which is the FCC the transaction read from the database header page when it was scheduled, as shown in Algorithm 1. In the case the page's TID is older than TS_{start}, the bit flag (`F`) in the TID is considered valid and we use `numSlots[F]` and `TB[F]` to construct a logical view of the page in the buffer cache. In the opposite case, i.e., the page's TID is younger than TS_{start}, the query ignores the more recent `numSlots[F]` and `TB[F]` because the transaction that updated the page has not yet

Algorithm 1 Version Selection in IPS

```
1:  function SELECTVERSIONFORREAD(page)
2:      if $page.TID > FCC$ then – page is dirty
3:          – Uncommitted TXN has modified $\{numSlots, TB\}[page.F]$
4:          return $\neg page.F$ – read the old version
5:      end if
6:      return $page.F$ – read the more recent version
7:  end function
8:  function SELECTVERSIONFORWRITE(txn, page)
9:      $assert(page.TID \leq txn.TS_{commit})$
10:     – $\{numSlots, TB\}[F]$ are new journals
11:     $page.numSlots[\neg(page.F)] \leftarrow page.numSlots[page.F]$
12:     $page.TB[\neg(page.F)] \leftarrow page.TB[page.F]$
13:     $< page.F, page.TID > \leftarrow < \neg page.F, txn.TS_{commit} >$
14:     return $\neg(page.F)$
15: end function
```

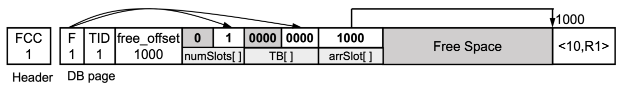
put a commit mark (i.e., increment the FCC in the database header page). Therefore, the bit flag is considered invalid, and we use the journaled metadata, i.e., `numSlots[¬F]`) and `TB[¬F]`. Using the versioned metadata, IPS provides two consecutive versions of pages, which enables *Two-Version Concurrency Control*, as will be detailed in Section 4.

3.2 Database Operations in In-Page Shadowing

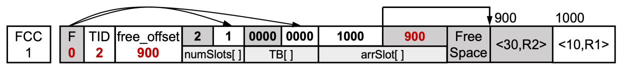
In the following, we describe how IPS updates a single-slotted page for `insert`, `delete`, and `update` queries, and how IPS updates multiple pages for B+tree rebalancing operations. Then, we discuss how IPS guarantees failure-atomicity and supports *rollback* operations.

3.2.1 Insert. All slotted-pages that SQLite uses for database files and index files are B-tree pages, so SQLite keeps records in sorted order by their keys and overwrites the existing array of offsets as well as records. However, IPS updates pages in an append-only manner. That is, appended records are written to the free space area regardless of key order, so existing records and metadata are not modified. If we append a new record and its offset in insertion order rather than key order, we can easily roll back the latest transaction by truncating the appended record and offset. Another key rationale for relaxing the sort constraint is that the position of committed records should not be changed by write transactions to allow non-blocking reads (i.e., 2VTO). That is, if a write transaction is allowed to move records around, concurrent read transactions must wait. But, if committed records do not change their positions, concurrent read transactions can access other records while a write transaction is storing a new record in the same page. For concurrent write transactions, IPS requires that an exclusive lock be held on the updated page until the write transaction commits. Otherwise, subsequent write transactions may overwrite the journaled page view and make recovery impossible.

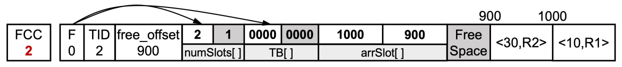
Suppose a transaction inserts a new record `<30,R2>` into a page shown in Figure 2(a). The new record is written to the free space extending the record content area towards the beginning of the page. Accordingly, the new record's offset is appended to the `arrSlots[]` and the `free_offset` is updated to 900, as illustrated in Figure 2(b). The existing key 10 is smaller than the inserted key 30, but the offset of `<30,R2>` (900 in the example) is appended at the end without considering the order of the keys.



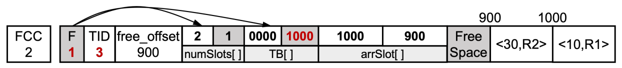
(a) Slotted-Page with a Single Record `<10,R1>`



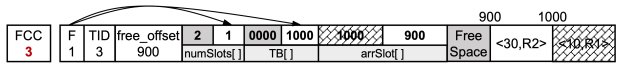
(b) Insertion of Record `<30,R2>`



(c) Transaction Commits Incrementing FCC



(d) Deletion of Record `<10,R1>`



(e) Transaction Commits Incrementing FCC

Figure 2: Walk Through Example of In-Page Shadowing

The overhead of sorting keys can be imposed either when records are written to the page, or when the page is loaded into the buffer cache. The vanilla SQLite takes the former approach, whereas IPS takes the latter approach. When a page is loaded into the buffer cache, IPS sorts keys and constructs a *logical view*, i.e., binary search tree (BST), for the more recent committed version.

When a write transaction acquires an exclusive lock on a page, IPS creates a copy of the logical view so that the write transaction can change it while concurrent read transactions access the previous logical view. To improve the concurrency level of 2VTO, IPS does not discard the old logical view for concurrent read transactions until the write transaction becomes the oldest transaction and commits, or until the old logical view is evicted from the cache due to memory pressure. With the logical views, no modification to SQLite database operations is required except they use BSTs instead of sorted arrays.

3.2.2 Delete and Update. For deletion, IPS uses tombstone bitmaps. That is, it sets the deletion bit for a deleted record in the latest tombstone bitmap. Suppose a transaction deletes `<10,R1>` from the page shown in Figure 2(c). By checking the bit flag, the transaction finds which `numSlots` and `TB` will become the more recent metadata. In the example, `numSlots[1]` and `TB[1]` are the committed metadata. Hence, the write transaction decrements `numSlots[0]` and sets the first bit of `TB[0]` to mark the deletion of the first record, i.e., the record pointed by `arrSlot[0]`. Even if the page is written to the database file before the transaction commits, the deleted record `<10,R1>` can be found by using the previous `numSlots` and `TB` because IPS does not overwrite the previous snapshot. Therefore, upon a transaction abort, we can revert the page to the previously consistent state by simply flipping the bit flag.

For an update, IPS does not overwrite an existing record, but appends its new version to the free space and sets the tombstone bit

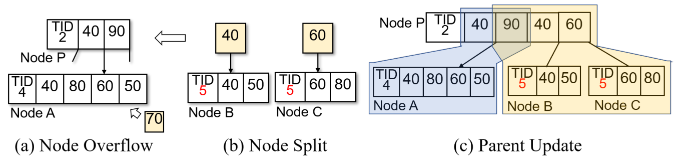


Figure 3: B+tree Node Split with IPS

for the previous version of the updated record in TB. Multiple update or delete operations can result in deleted records accumulating on a page, which need to be garbage collected. Unlike traditional MVCC, IPS collocates multiple versions of the same records in the same physical page. By clustering multiple versions in the same page, the garbage collection overhead can be reduced. Also, unlike MVCC, IPS does not eagerly garbage collect those obsolete records even if no longer needed by concurrent transactions. Instead, they are lazily garbage collected when the page overflows. The rationale behind this decision is that garbage collection overhead can be minimized by doing the garbage collection in batches instead of garbage collecting each record one at a time.

When a page overflows, IPS garbage collects obsolete records via copy-on-write. Because a page split causes to allocate and write a new page via copy-on-write anyway, garbage collection at the time of page overflow masks its overhead with the page split overhead. When a page overflows, IPS checks whether the node has any obsolete records to delete. If there is any, IPS creates a copy of the node with only valid records without creating a new sibling page. The throughput of lazy garbage collection is 3.2% higher than that of eager garbage collection in our experiments. Despite the performance benefits, this *lazy garbage collection* has the disadvantage that the page is fragmented internally and page overflows more frequently. In our experiments, IPS causes about 2.9% of write queries to page overflow, whereas only 1.3% of write queries make pages overflow in WAL mode. However, our experiments show that the advantages of increasing the concurrency level outweigh the disadvantages caused by internal fragmentation.

3.2.3 *Multiple Page Writes: B+tree Node Split.*

A transaction often writes multiple database pages. Even auto-commit transactions that insert a single record may update multiple pages if a B+tree node splits. Consider the B+tree node split example shown in Figure 3 that describes how IPS updates multiple pages atomically.

In the example, we assume each page can hold up to four key-value records. Suppose we insert key 70 into Node A. Since there is no empty slot, we create two new nodes - Node B and Node C to redistribute the keys across the two new nodes as shown in Figure 3(b). Then, we insert the keys and pointers of the two child nodes into the parent node. Instead of overwriting the pointer for Node A, we append the two key-pointer pairs to the free space in a log-structured fashion, and update `numSlots`, TB, and TID of the parent node as shown in Figure 3(c). After updating all three nodes (Node B, Node C, and Node P), we increment the FCC in the header page of the database file.

Node P provides two logical views - one for the transactions older than TID 5 and the other view for younger transactions. Even if multiple pages are updated, the previous consistent pages (i.e., pages consistent in version 4 in the example) can be reconstructed when the system crashes or a transaction aborts before increasing the FCC. Once the transaction successfully commits, the obsolete journal version (in the example, Node A) is no longer needed. Thus, the write transaction deletes Node A from the database file. However, IPS does not eagerly delete the obsolete `key[0]` 40 from Node P. Instead, IPS defers garbage collection until Node P overflows.

3.2.4 *Rollback.*

If a transaction aborts, the vanilla SQLite removes its dirty pages from the buffer cache. In contrast, IPS does not drop dirty pages from the buffer cache because they are not just dirty pages but also clean pages accessed by concurrent read transactions. To rollback the aborted transaction, IPS (1) drops its dirty logical views (BSTs), (2) inverts the bit flag (F) of the pages, and (3) restores the previous TID of the pages. To restore the previous TID, IPS keeps the previous TID of each dirty page along with old consistent logical views in memory. By reusing the cached pages for subsequent and concurrent transactions, IPS not only avoids reading the same pages from disks but also improves the concurrency level.

If a system crashes while a transaction is committing, uncommitted dirty pages can be written to a database file. Besides, IPS employs *steal/force* buffer manager policy. Hence, a stolen dirty page can be also written to the database with an uncommitted TID. Nonetheless, since UNDO information is in the page itself, it is recoverable and the UNDO information (i.e., the journal version of the page) will not be lost because the transaction that updated the stolen page will keep holding an exclusive lock until it commits. For UNDO operations, the last modified timestamp (TID) of the uncommitted pages is reset to zero (since we lost the previous TID) and the bit flag (F) is flipped. The recovery process does not need to restore the previous TID correctly because subsequent transactions will check whether its start timestamp is more recent than the timestamp of restored pages.

While the *no-force* buffer management policy is known to outperform the *force* policy, the overhead of the WAL implementation in SQLite is not less than that of other SQLite journal modes that use force policy [26]. This is because the no-force policy complicates enforcing durability and SQLite is a lightweight embedded database system that should avoid such complexity. When a transaction commits, the SQLite WAL mode flushes all its dirty pages to the WAL file as-is. Therefore, the efficiency and complexity of IPS's buffer management are similar to that of WAL.

3.2.5 *Failure-Atomic Multiple Pages Updates.*

In IPS taking the steal policy, a transaction is allowed to flush its dirty pages to disks before commit. To undo such prematurely written pages and reduce the number of `fsync()`/`fdatasync()` calls, IPS employs the *counting commit* protocol [24]. Because `fsync()`/`fdatasync()` does not enforce the order in which multiple pages are written, there is no guarantee that database header pages will be flushed after all data pages are flushed.

In the counting commit protocol, a transaction counts the number of dirty pages updated by the transaction (i.e., the transaction size) and writes the number in the database header page as a commit mark along with the FCC. Then, it calls `fsync()` for the dirty pages and the header page. To recover from failures, the counting commit protocol requires that the recovery process scans the entire database file to count the number of dirty pages with a transaction

ID greater than or equal to the FCC. If there are pages with a transaction ID greater than the FCC, it is either because they are stolen pages or because a transaction with TID greater than the FCC has aborted before writing a commit mark in the header page. In either case, the recovery process needs to roll them back to the previous version. If there is no page with a transaction ID greater than the FCC, the FCC is considered the TID of the latest write transaction, and the number of pages with the same transaction ID as the FCC is compared to the size of the transaction written along with the FCC in the header page. If the two numbers are equal, no recovery is necessary. Otherwise, the latest write transaction is considered to have failed to flush all the dirty pages it wrote. Hence, the latest write transaction needs to be rolled back, and the bit flag (F) of the dirty pages with a TID equal to the FCC is flipped to restore their previous versions. Since 2VTO does not allow multiple transactions to commit at the same time, the FCC increases monotonically and transactions with a TID smaller than the FCC are guaranteed to have been committed.

4 TWO-VERSION TIMESTAMP ORDERING

IPS updates metadata and records in an append-only fashion without relocating them. This property provides an opportunity to design a new concurrency control protocol that seamlessly integrates journaling and multi-version concurrency control (MVCC) for mobile DBMS.

To improve the concurrency level of SQLite, we develop a fine-grained page-level concurrency control protocol called Two-Version Timestamp-Ordering (2VTO). 2VTO, a variant of Multi-Version Timestamp-Ordering (MVTO) protocol, manages two versions of each database page for multi-threaded apps. For concurrent access from multiple processes, the operating systems' file-based locking is still used because SQLite is an embedded library DBMS. By managing only two versions, 2VTO eliminates the need for a background garbage collection mechanism.

4.1 IPS vs. DHL for Non-Blocking Reads

DHL (DoubleHeader Logging) [24] is another in-page journaling scheme that provides two logical views of a single slotted-page. DHL is similar to IPS in that it uses the free space of the slotted-page structure as a journal space for page metadata, i.e., it manages two page headers including the offset arrays and one page header behaves as a journal of the other.

The key difference between IPS and DHL is that DHL duplicates the offset array to maintain the old and new sorted orders. Because each array size changes dynamically, DHL relocates the two arrays and it makes non-blocking reads impossible. For example, if a write transaction is shifting the offset arrays for insertions, concurrent read transactions may access incorrect records. This constraint makes DHL unsuitable for 2VTO.

4.2 2VTO: Two-Version Timestamp Ordering

Using the two logical views of each page provided by IPS, 2VTO follows the isolation specification of SQLite [4]. That is, read transactions must see an unchanging *snapshot* of the database file as it existed at the time when the read transaction started. Thus, any partial change made by a concurrent writer that has not been committed is invisible to the reader. Multiple writers can access the same database file concurrently, but they must update disjoint sets of pages. If a write transaction splits a page, the writer garbage-collects the old page in 2VTO. If an old transaction needs to access the garbage-collected page, the old transaction is aborted. We note that old releases of Oracle also took the same approach, i.e., they abort a transaction if undo logs it needs to access have been overwritten by concurrent writers.

As in MVTO, 2VTO assigns transactions two timestamps - TS_{start} and TS_{commit} when they start, which is used to determine the serializability order. Implementing 2VTO in SQLite, TS_{start} is set to the FCC when a transaction starts, i.e., a transaction can read data written by a transaction older or equal to its TS_{start}. TS_{commit} is also assigned to write transactions during initialization, and determines the commit order of write transactions. For example, if two write transactions T_i and T_j are scheduled when FCC is fcc, their start timestamps ($T_i.TS_{start}$ and $T_j.TS_{start}$) are set to fcc, but their commit timestamps are incremented atomically in the order they are scheduled, i.e., $T_i.TS_{commit} = fcc+1$, $T_j.TS_{commit} = fcc+2$. 2VTO commits T_i before T_j regardless of how long each transaction runs. That is, a write transaction can commit only if all other previously scheduled write transactions have already been committed.

4.2.1 *Conflict Resolution.*

IPS requires one of the two logical views to play the role of journal. Therefore, 2VTO does not allow a write transaction to update database pages modified by other uncommitted write transactions, as each page must be updated exclusively by at most one transaction until the transaction commits.

2VTO does not prevent a younger write transaction from updating a database page before an older write transaction accesses it. Therefore, cyclic write-write conflicts can occur in 2VTO. 2VTO resolves the cyclic dependency using the commit timestamps, and thus, deadlock does not occur. Specifically, if a write-write conflict occurs, 2VTO gives priority to the first updater and aborts subsequent transaction that fails to acquire a lock, as in the traditional First-Updater-Wins [5, 11]. To resolve write-write conflicts, various database systems including Oracle, MySQL, and PostgreSQL have implemented *First Committer Wins* (FCW) or its variant *First Updater Wins* (FUW) rule [5, 27, 28]. If a write transaction makes an update that conflicts with another concurrent write transaction, FCW and FUW commit the first committer and first writer, respectively, and the other transactions are aborted.

For read-write conflicts, 2VTO ensures that a read transaction finds all committed records that existed at the time the transaction started, despite IPS only provides two versions. This is because 2VTO does not allow more than one concurrent write transaction to update the page until it is committed. Suppose a transaction T_i reads the current consistent logical view of a page ($LV[F]$) while a younger write transaction T_j (i.e., $T_i.TS_{commit} < T_j.TS_{commit}$) updates the same page and creates the next logical view ($LV[\neg F]$). Unless T_i commits, 2VTO does not allow T_j to commit. Therefore, the current consistent logical view of the page ($LV[F]$) is guaranteed to be available for all transactions older than the write transaction. When the write transaction commits, the previous logical view ($LV[F]$) is not needed by other transactions and can be overwritten by a subsequent write transaction.

Algorithm 2 Abortee-or-First-Updater Wins Algorithm

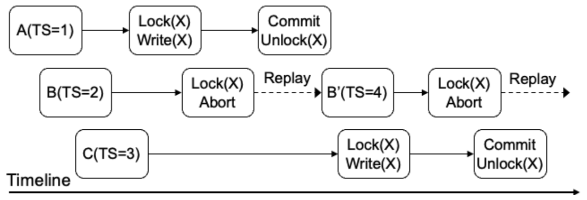
```
function ABORT(txn)
    $StopExecution(txn)$
    $txn.TS_{commit} \leftarrow NextCommitTimestamp()$
    while $txn.TS_{commit} > cur\_oldest\_TS$ do                ▷ ForcedSleep
        $cond\_var.wait(txn)$
    end while
    $txn.TS_{start} \leftarrow txn.TS_{commit}$
    $Restart(txn)$
end function
function EXCLUSIVE_REQUEST(txn, page)
L1: $lock\_acquired \leftarrow GetExclusiveLock(txn, page)$
    if $lock\_acquired == False$ then                          ▷ Step (1)
        if $txn.TS_{commit} == cur\_oldest\_TS$ then           ▷ Step (1-1)
            $Abort(GetCurrentLockOwner(page))$
            goto L1
        end if
        $Abort(txn)$                                           ▷ Step (1-2)
    end if                                                     ▷ Step (2)
    if $page.TID > txn.TS_{start}$ then                        ▷ Step (2-1)
        $Abort(txn)$
    end if
    – Otherwise, txn proceeds normally                         ▷ Step (2-2)
end function
```

4.3 Abortee-or-First-Updater Wins

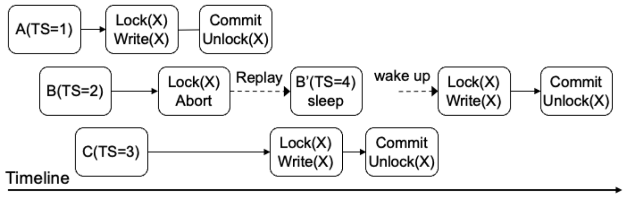
To prevent any deadlock and also to reduce the number of replays of aborted transactions, which waste a significant amount of computing resources and battery power, 2VTO employs a variant of FUW - *Abortee-or-First-Updater Win* (AFUW) rule that limits the number of aborts per transaction to at most one.

FUW uses exclusive locks to detect conflicts early. In contrast, FCW calls expensive `memcpy()` to create a thread-local copy of each object and checks write-write conflicts at commit time. Out of these two rules, FUW is more suitable for IPS than FCW because IPS also requires write transactions to hold locks until they commit, i.e., the two logical views provided by IPS must be protected using an exclusive lock until the transaction commits. Similar to FUW, AFUW aborts one of two conflicting transactions to prevent deadlocks.

To prevent the same transaction from being aborted multiple times, AFUW forces an aborted transaction to sleep until it becomes the oldest transaction so that it has the highest priority and prevents other transactions from blocking it. When a transaction aborts, it calls *wait* on a condition variable so that its immediately preceding transaction can wake it up. When a transaction wakes up, it checks the transaction manager object, which contains metadata for all active transactions, to ensure that it is the oldest transaction. Alternatively, transactions can call `usleep()` in a while loop and periodically query the list of IDs. Once an aborted transaction becomes the oldest transaction, it runs with the highest priority so that it can steal exclusive locks from other conflicting transactions and abort them. Unlike *Wound-Wait* deadlock prevention rule, AFUW allows only the oldest transaction to wound other transactions. When a transaction is aborted and rescheduled as the oldest transaction, the transaction's start and commit timestamps (TS_{start} and TS_{commit}) will have the same value. This property ensures that aborted transactions under AFUW always read the latest committed data and commit successfully, i.e., AFUW compensates for the problem of FUW on mobile devices as it makes FUW fall back to a serial schedule if FUW fails to make a concurrent serializable schedule using two logical views.



(a) First-Updater-Wins



(b) Abortee-or-First-Updater-Wins

Figure 4: First-Updater- vs. Abortee-or-First-Updater-Wins

Algorithm 2 shows how AFUW works. Suppose a write transaction T_i requests a write lock on a database page to update. (1) If the write lock for the page is held by another transaction T_j, the lock request will be rejected and two cases must be considered. First, (1-1) if T_i is currently the oldest transaction, i.e. $T_i.TS_{commit}$ is smallest amongst all running transactions, T_i has the highest priority and it can abort T_j to steal the write lock. Second, (1-2) if T_i is not the oldest transaction, T_i aborts and sleeps until it becomes the oldest transaction. When it becomes the oldest transaction, it replays its operations with the highest priority and steals write locks from any other concurrent transactions if necessary. Thereby, AFUW guarantees that a transaction is aborted at most once. (2) If the lock is acquired, 2VTO checks if the page has been updated by another concurrent transaction, i.e. $page.TID > T_i.TS_{start}$. (2-1) If it is, transaction T_i aborts. (2-2) Otherwise, the transaction T_i is allowed to modify the page to create a new logical view while keeping the old journal view in the same page using IPS.

Figure 4 is an example illustrating the difference between AFUW and FUW. In the example shown in Figure 4(a), FUW aborts transaction B twice because both transactions A and C acquire exclusive locks on the object before transaction B. In contrast, AFUW makes transaction B sleep until all the previous transactions (in the example, transaction A and C) commit. Once transaction B wakes up, it may abort other younger transactions if it has to steal exclusive locks from them regardless of whether they are first writers or not.

4.3.1 *Overhead of AFUW: Forced Sleep.*

FUW is more optimistic than AFUW in that FUW expects that a rescheduled transaction will not conflict again. However, as we discussed in Section 2, resource-constrained embedded database systems such as SQLite should not repeat replaying the same transaction over and over due to battery usage. Therefore, AFUW takes a rather pessimistic approach to limit the number of aborts. If there are many concurrent transactions that conflict with each other, AFUW will abort and reschedule most of them one by one, as in a serial schedule. As a result, the concurrency level of AFUW can be lower than that of FUW. However, as we have discussed, the level of concurrency requirements of mobile apps is not as high as that of enterprise database systems.

In exchange for limiting the number of aborts to one, AFUW forces transactions to sleep. In addition, 2VTO forces a transaction, though ready to commit, to wait for its previous transactions to commit, as in other timestamp-ordering protocols [35]. In Section 5, we will show that the forced sleep overhead of AFUW is lower than the rollback/replay overhead of SQLite's implementation of optimistic concurrency control. This is in part because mobile workloads mostly consist of single query transactions in auto-commit mode, and in part because transactions that conflict with other transactions continue to conflict with subsequent transactions in the mobile workload [26].

5 EVALUATION

We evaluate the performance of IPS against state-of-the-art logging schemes - DASH [34] and DHL [24], as well as the stock WAL and OFF modes of SQLite 3.7. To evaluate the concurrency level of 2VTO, we enable the *shared-cache* mode, which allows multiple connections to the same database to share the buffer cache. All presented results are the average of five runs. DASH [34] is a state-of-the-art logging method that mitigates the journaling of journal problem by shadowing the entire database file. Both DHL and DASH rely on file-based locking. However, DHL can be improved to use 2VTO by the position of the offset arrays fixed. For example, the first offset array can be preallocated to its maximum size, taking into account the case where the largest number of smallest 4-byte records can fit on a page. Alternatively, the two offset arrays can be interleaved to reduce the size of pre-allocated space. However, in either approach, DHL duplicates offset array elements and store fewer records per page compared to IPS. We implemented a variant of DHL that preallocates the first offset array to make it work with 2VTO, and evaluate its performance (denoted as `DHL+2VTO`) for concurrent workloads.

5.1 Experimental Setup

Our testbed is a Samsung Galaxy S10, which runs Android 10 (Linux Kernel 4.14) on two 2.7 Ghz Exynos M4 processors, 8GB memory, and 256GB UFS 2.1 formatted with the EXT4 file system with `data-ordered` journal mode. We fixed the frequency to the maximum 2.7 GHz to reduce the standard deviation of the experiments.

We use two types of workloads for performance evaluation. First, we run Mobibench [12], a microbenchmark commonly used to evaluate the performance of the mobile database systems [16, 19, 24, 34]. Mobibench runs a single thread that that inserts, deletes, or updates 128-byte records using randomly generated keys. To evaluate the concurrency level of SQLite, we made small modifications to Mobibench to run multiple concurrent client threads. We populate a database table with 5000 records (552KB DB table). Then, we run a variable number of client threads, each submitting additional 5000 queries in auto-commit mode. We generate keys in both uniform and Zipfian distributions to evaluate the effect of potential conflicts between concurrent transactions. In addition to the micro-benchmark, we evaluate IPS and 2VTO using SQL statement traces that we collected from five representative mobile apps: Facebook, Twitter, Google Play Services, Google Calendar, and Google Maps.

5.2 Evaluation of In-Page Shadowing

5.2.1 Latency Breakdown with Mobibench. In the first set of experiments shown in Figure 5, we run single-threaded Mobibench that submits 5000 queries in a batch, and break down the average query latency into 1) computation time, 2) the time taken for `write()` system call, and 3) the time taken for `fsync()` system call.

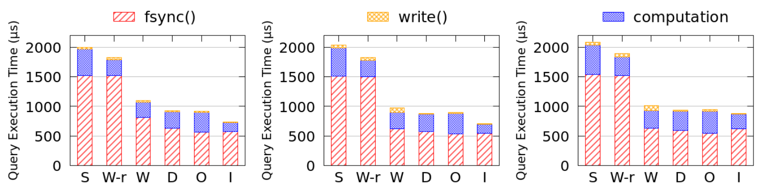
Our experiments show that WAL is slower than DHL, OFF, and IPS modes because of file system metadata journaling. Contrary to popular belief, WAL does not perform sequential writes but random writes because each transaction appends new dirty pages, increasing the size of a log file. Therefore, the file system journals the file system metadata (i.e., inode and file size updates) [13, 19, 24].

We evaluate WAL using two different journal size limit configurations. If `journal_size_limit` is set to -1 (unlimited) as in the default configuration of SQLite, checkpointing does not truncate the WAL file but overwrites it to avoid the metadata journaling. However, a drawback of this default configuration is that it wastes the storage space. Therefore, our trace shows that all mobile apps in our testbed smartphone (e.g., Twitter, Facebook, Gmail, Calendar, and Maps) set the `journal_size_limit` (i.e., reserved journal space) to 512 KB such that they truncate WAL files and reuse 512 KB. If a WAL file grows larger than the limit, WAL suffers from the file system metadata journaling. We denote the performance of the latter configuration (i.e., 512 KB reserved journal space as in real workload) as `W-r`. Figure 5 shows that `W`, the default WAL mode with unlimited journal size, shows performance superior to `W-r` and comparable to the DHL mode because it mitigates the journaling of journal problem. However, WAL mode periodically checkpoints dirty pages from WAL files to database files and amplifies the amount of I/O and increases the `fsync()` time. As a result, Figure 5(a) shows that the `fsync()` overhead of `W` is 28% higher than that of DHL, OFF, and IPS modes.

DASH exhibits the worst performance in the experiments. This is because DASH manages two database files. As the two files take turns in behaving as a shadow file, DASH manages a recent history of which dirty page was written to which file. Using the history, DASH writes the pages updated by the latest two transactions to a new shadow file. It not only increases the computation time, but also flushes a larger number of dirty pages than other logging modes that commit only one transaction.

OFF mode disables database logging. Thus, it does not suffer from the journaling of journal problem and reduces the number of writes in half. As a trade-off, OFF mode does not support rollback. The `fsync()` overhead of DHL and IPS is similar to that of OFF mode while they allow transactions to rollback since both logging schemes perform in-place logging. As we discussed in Section 3.2.2, IPS has the internal fragmentation problem. Therefore, the `fsync()` overhead of IPS is slightly but less than 1% higher than that of OFF mode. The `fsync()` overhead of DHL is about 10% higher than that of IPS. This is because DHL has 2x larger metadata, so it has fewer node fanouts and splits more often than IPS.

The computation time of IPS is much lower than that of DHL and OFF modes because IPS allows non-blocking reads and eliminates



(a) Insert Latency (b) Delete Latency (c) Update Latency

Figure 5: Latency Breakdown of Mobibench Queries
S: DASH, W-r: WAL-512KB, W: WAL-unlimited, D: DHL, O: OFF, I: IPS

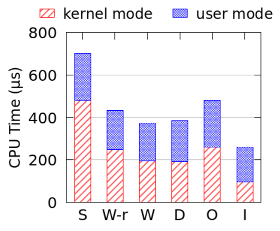


Figure 6: CPU Time

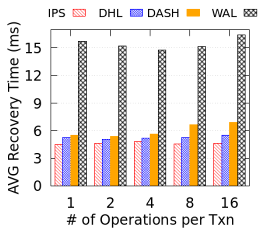


Figure 7: Recovery Time

the overhead of acquiring shared locks, whereas OFF and DHL require every transaction to acquire shared locks. Therefore, IPS invokes fewer system calls and also triggers fewer context switches than DHL and OFF. As a result, IPS spends less time in the kernel, as shown in Figure 6, which breaks down the computation time of an individual insert query into kernel time and user-level time.

Figure 5(b) shows IPS outperforms all other logging modes for delete queries. This is because IPS performs deletions by flipping the tombstone bits and defers B+tree rebalancing operations until the page overflows. That is, if a page overflows due to inserts, the deleted records and under-utilized pages are garbage collected via copy-on-write in IPS. Therefore, the tree rebalancing computation overhead is only included in inserts, but not in deletions.

While IPS outperforms DHL and OFF modes by 22% and 20% for inserts and deletions, Figure 5(c) shows that IPS is only 5% faster than DHL and OFF modes for updates because internal fragmentation issues are more problematic in update queries. Unlike DHL and OFF modes, IPS does not overwrite obsolete records until the page overflows.

5.2.2 *Recovery Performance.*

In the experiments shown in Figure 7, we measure the latency for crash recovery. We warm up a database table with 1500 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 to sixteen. The size of each record is 128 bytes.

Overall, the recovery time of WAL is much longer (16 msec) than the other logging modes due to the overhead of copying dirty pages from the WAL file to the database file. In addition, this checkpointing process updates the file system metadata, i.e., inode table and block bitmap, to truncate the WAL file. Although the recovery process of DASH is much faster than WAL, its recovery time is longer than DHL and IPS because it has to make the shadow file and the database file identical, i.e., DASH has to copy the pages written by the last two, not just one, transactions to one of the two files. Therefore, as the transaction size becomes larger, its overhead increases. The recovery process of DHL compares two transaction IDs against the FCC value in all pages, and overwrites invalid metadata with valid one using `memcpy()`. In contrast, IPS flips only a dirty bit flag to rollback. By avoiding the `memcpy()` overhead, the recovery process of IPS is slightly faster than DHL.

5.2.3 *Storage Overhead.*

In terms of storage overhead, DASH is the worst because it shadows the entire database file. WAL isn't much better than DASH because it writes additional metadata, i.e., WAL header per each dirty page and WAL index, and the WAL file often grows larger than small database files. In contrast, IPS and DHL place recovery information inside slotted-pages and require negligible storage overhead.

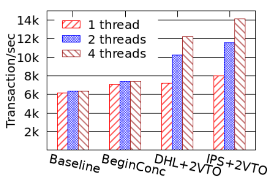
5.3 Evaluation of 2VTO

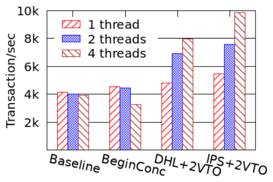
In this section, we evaluate the concurrency of SQLite using multi-threaded Mobibench with varying numbers of client threads submitting 5000 transactions each. We set the checkpoint interval to 100 transactions (i.e., `wal_autocheckpoint=100`) and the journal size limit to 512 KB as set in the mobile app trace. We vary the number of threads to a maximum of four because at most five threads submit queries concurrently in the trace. We compare 2VTO with IPS against the vanilla SQLite in WAL mode. In vanilla SQLite, if multiple write transactions are scheduled, only one of them is granted the exclusive lock and the other transactions abort. However, due to the growing demand for higher concurrency, *BEGIN CONCURRENT* branch (SQLite 3.37) for WAL mode is under development [3]. BEGIN CONCURRENT branch employs an optimistic page-level-locking and prevents conflicting transactions from being committed. That is, each commit operation checks whether any database page, which the transaction has modified, was altered by other concurrent transactions. If any conflict is found, the transaction cannot commit and therefore aborts. Otherwise, it commits. Although BEGIN CONCURRENT branch is not merged into the main, it shares the same goal with 2VTO. Hence, we compare 2VTO against BEGIN CONCURRENT.

We do not show the performance of DASH because it is even slower than WAL, and is not designed for concurrent workloads. The original DHL also does not support concurrent transactions, but we implemented a variant of DHL with fixed positions of offset array elements so that DHL can use 2VTO. We denote it as `DHL+2VTO`.

5.3.1 *Concurrent Throughput.*

In the experiment shown in Figure 8, we run Mobibench to populate the database with 5000 records (552 KB) and submit another 5000 auto-commit queries per client thread. The percentage of write transactions (UPDATE statements) is varied to increase conflicts between client threads. `Baseline` denotes the performance of the default SQLite in WAL mode using file-based locks. We use the *shared-cache* mode for `Baseline` because `Baseline` performs better in shared-cache mode than *private-cache* mode. This is because the shared-cache mode allows multiple client threads to share the same schema cache, which reduces the memory usage and I/O. For BEGIN CONCURRENT, denoted as `BeginConc`, we use the private-cache mode because the private cache allows each client thread to work on its own private cache and defers


(a) Write Ratio=10%


(b) Write Ratio=20%

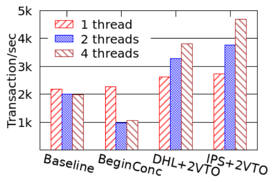

(c) Write Ratio=50%

Figure 8: Concurrent Throughput with Various Locking Schemes

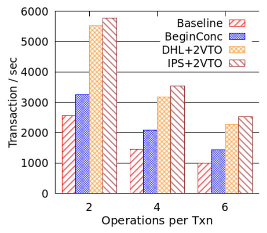


Figure 9: Txn Size Effect

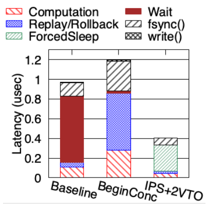


Figure 10: Latency Breakdown

locking until a commit is executed. As such, `BeginConc` performs better in private-cache mode.

Figure 8(a) shows that `Baseline` and `BeginConc` gain a very small throughput improvement from a larger number of threads even when the conflict is low (i.e., write ratio is 10%). This is because their locking schemes depend on the WAL file. That is, even if `BeginConc` employs fine-grained optimistic page-level locking and improves the concurrency level during computation, write transactions compete with each other and only one write transaction can append to the very end of the WAL file at any time, i.e., only one write transaction can write and commit. Thus, the throughput of `BeginConc` is not quite different from that of `Baseline`. If the write ratio is higher than 10%, transactions conflict more often and the overall throughput degrades with a larger number of client threads because optimistic `BeginConc` rolls back and replays transactions frequently. In contrast, DHL+2VTO and IPS+2VTO scale to 4 threads and exhibit more than 2x higher throughput than `Baseline` as they reduce the number of replays and benefit from non-blocking reads of internal tree nodes when traversing B-trees. Although DHL+2VTO scales, its throughput is up to 23.7% lower than IPS+2VTO, mainly due to low page utilization.

5.3.2 *Physical Battery Consumption Test.* To evaluate the battery usage efficiency of 2VTO, we ran physical experiments to measure how much power each logging mode consumes. For each experiment, we emptied the battery of our testbed smartphone completely, and charged it only for exactly 3 minutes. Then, we repeated the same workload used for the experiments shown in Figure 8(c) until the phone turns off with no battery left. On average, `BeginConc` and `Baseline` executed approximately 140K and 200K transactions until the phone drained the battery and shut down in 105 and 90 seconds, respectively. In contrast, IPS+2VTO ran about 400K transactions before the phone turns off in 90 seconds. That is, IPS+2VTO executes approximately 2x and 2.8x number of transactions than vanilla SQLite and `BeginConc` with the same battery charge, respectively.

5.3.3 *Transaction Latency in Concurrent Workload.* In Figure 10, we breakdown the query response time of three concurrency modes for the experiments that submit 20% write queries using four client threads. `Baseline` suffers from high latency waiting for locks due to the coarse-grained file locking. In particular, it spends 69% of its execution time waiting for file-based locks. On the other hand, `BeginConc` spends 48% of its execution time on rollback operations. Taking an optimistic approach, `BeginConc` defers conflict checks until transactions are ready to commit. Because the database table

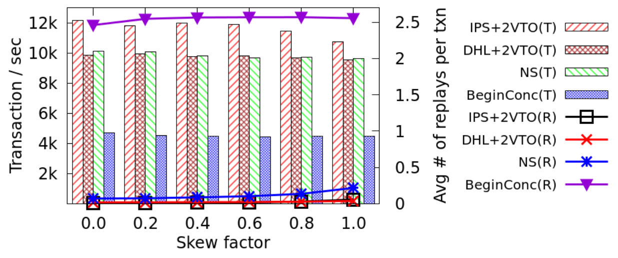


Figure 11: Impact of Conflicts
NS: IPS+2VTO without Forced Sleep, **(T)**: Throughput, **(R)**: # of Replays

has only about 140 pages in the experiments, even with 20% of write transactions, transactions conflict and abort frequently. Due to its optimistic concurrency control, `BeginConc` keeps repeating and aborting the same transactions. As a result, its `Replay/Rollback` overhead is high, and `Computation` time is also much higher than the other two modes. Note that `fsync()` overhead of `BeginConc` is also higher than the other two modes. This is because it employs *private-cache* and checkpointing occurs more frequently. On the other hand, IPS+2VTO shows much lower query response times than the other modes because IPS+2VTO limits the number of aborts. The largest portion of the query latency in IPS+2VTO is `ForcedSleep`, which is the time transactions spend sleeping, i.e., if a transaction is aborted or ready to commit, it sleeps until it becomes the oldest transaction. In the experiments, `ForcedSleep` accounts for 66% of the average latency. It should be noted that although IPS+2VTO spends a significant amount of time sleeping, the CPU cores are busy processing other transactions.

5.3.4 *Effect of Transaction Size.* In the experiments shown in Figure 9, we generate transactions of various sizes using Mobibench. We run four client threads, each submitting 5000 transactions. Keys are in uniform distribution, and 85% of SQL statements are SELECT statements regardless of transaction size. As the transaction size increases, the transaction throughput decreases, but 2VTO consistently outperforms `Baseline` and `BeginConc`. Interestingly, unlike auto-commit transactions, `BeginConc` exhibits much better performance than `Baseline`. `BeginConc` shows 27.3%, 43.8%, and 43.9% higher throughputs than `Baseline` when the transaction size is two, four, and six, respectively. This is because of the private-cache mode. That is, as a transaction accesses a larger number of pages, the wait time for page-level locks increases in the shared-cache mode. In particular, the lock wait time in the private-cache mode is only 1/8 compared to that of the shared-cache mode.

Table 1: Workload Characteristics

T: Number of Threads, **DS**: Initial DB Size in KB

Apps	Txn Size	Read	Write(Update)	T	DS
GPS	9.82	58.70%	41.3(0.1)%	4	244
Facebook	1.86	27.98%	72.0(36.2)%	2	88
Twitter	4.24	66.71%	33.3(19.0)%	2	396
Calendar	1.26	99.61%	0.4(0.2)%	2	156
Maps	1.41	91.14%	8.8(5.5)%	2	40

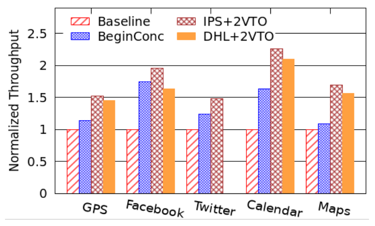


Figure 12: Concurrent Throughput with Real Workload

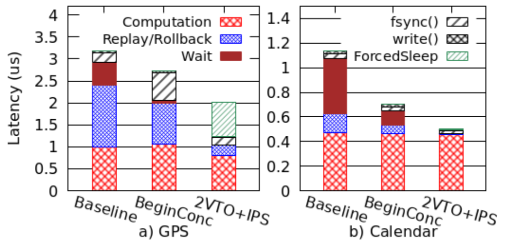


Figure 13: Real Workload Latency Breakdown

Another interesting observation is that the replay/rollback cost of `BeginConc` in the private-cache mode is smaller than the shared-cache mode as the transaction size grows, i.e., only 1/4 compared to that of `BeginConc` and `Baseline` both in the shared-cache mode. This result is different from Figure 10 because long running transactions in the shared-cache mode conflict more frequently and rollback in the shared-cache mode requires coordination with other concurrent transactions over shared data structures.

5.3.5 Skew Effect. In the experiments shown in Figure 11, we vary the Zipfian skew factor and measure its impact on the number of aborted transactions. We run four client threads, each of which submits 5000 transactions, and the write ratio is 15%. When keys are generated in uniform distribution (skew factor=0), DHL+2VTO and IPS+2VTO abort about 1% of the transactions. As the skew factor increases, the number of aborted transactions increases, and when the skew factor is 1.0, about 5.7% transactions abort in IPS+2VTO. As a result, the throughput degrades by 13%. DHL+2VTO aborts a slightly fewer number of transactions than IPS+2VTO because its page utilization is low and popular records are distributed across more pages. As a result, the throughput of DHL+2VTO is only reduced by 5%, but its throughput is still lower than that of IPS+2VTO. Disabling forced sleep in IPS+2VTO causes aborted transactions to conflict with other transactions up to 3.8x more frequently. As a result, its throughput (denoted as `NS(T)`) degrades by at least 11.5% compared to IPS+2VTO. We note that `BeginConc` mode is rather insensitive to the skew factor, i.e., its throughput degrades only 4.7%. This is because `BeginConc` suffers from replaying a large number of aborted transactions repeatedly even when keys are in uniform distribution, and its throughput is already much lower than that of IPS+2VTO. On average, `BeginConc` aborts each transaction more than 2.4 times even in uniform distribution. When the skew factor is 1.0, the average number of aborts increases to 2.5, and the throughput degrades by 4.7%.

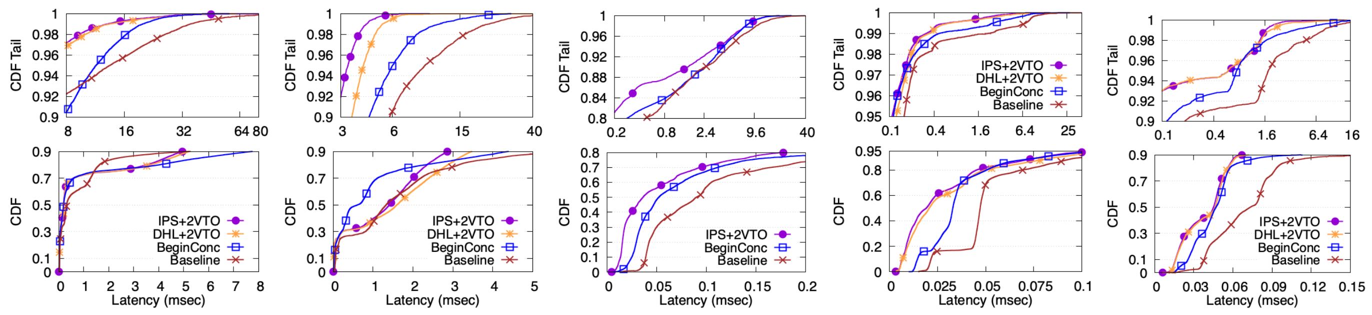
5.4 Evaluation with Real Workloads

Finally, we evaluate the performance of IPS and 2VTO using the SQL traces that we collected from mobile apps. For BEGIN CONCURRENT branch, we replace all `BEGIN` statements in the trace to `BEGIN CONCURRENT` statements to enable the enhanced concurrency control mode. We did not modify other types of `BEGIN` statements (e.g. `BEGIN EXCLUSIVE`) for correctness.

Table 1 shows different characteristics of five workloads we use. Since Gmail is heavily dependent on Google Play Services (GPS) and its performance is similar to that of GPS, we do not present the performance of Gmail. The average number of operations per transaction in GPS workload is 9.825, and 41% of transactions contain either INSERT or DELETE queries. Facebook is the most write-intensive workload as 72% of its transactions are write transactions and about a half of write transactions contain UPDATE queries. Calendar is a read-intensive workload, i.e., 99.6% of its transactions are reads. Google Maps is also read-intensive but 8.8% of transactions contain write queries. We use four threads for GPS, i.e., one less number of threads than the actual number of threads used in the trace, because GPS runs less than 5 threads for 99.98% of the transactions in our trace.

5.4.1 Throughput. Overall, Figure 12 illustrates that `IPS+2VTO` consistently outperforms `Baseline` and `BeginConc` in all workloads due to its fine-grained concurrency control and non-blocking reads. For write-intensive workloads (GPS, Twitter, and Facebook), the throughput improvement comes mostly from fine-grained locking granularity. Specifically, for Facebook workload, `BeginConc` shows comparable performance with IPS+2VTO (i.e. only 4.4% lower throughput than `BeginConc`) and is even more performant than DHL+2VTO. This is because Facebook workload consists of a large number of UPDATE queries. That is, IPS suffers from internal fragmentation for UPDATE queries, as discussed in Section 5.2. For GPS and Twitter, IPS+2VTO shows 31% and 19% higher throughput than `BeginConc`. We could not run DHL+2VTO for Twitter due to implementation issues with emulating DHL. For read-intensive workloads (Calendar and Maps), `IPS+2VTO` and `BeginConc` benefit from page-level locking, and `IPS+2VTO` shows up to 89% higher throughputs than `Baseline`.

5.4.2 Latency Breakdown. Figure 13 shows the latency breakdown for write-intensive GPS and read-intensive Calendar workloads. We note that the results are slightly different from the multithreaded Mobibench results. Unlike the Mobibench that submits auto-commit transactions consisting of a single query, the real transactions are long running as they consist of multiple queries. On average, a GPS transaction consists of 9.825 read and write queries. As a result, `Baseline` aborts a large number of GPS transactions. Aborted transactions need to rollback the changes they made to the metadata in the shared buffer cache (e.g., `BtShared` in SQLite). In addition, long running transactions often execute read queries before write queries, which need to be replayed every time they are aborted. Therefore, Figure 13 shows that `Replay/Rollback` time accounts for more than 43% of transaction latency in `Baseline`. On the other hand, `BeginConc` employs fine-grained locks to avoid write-write conflicts. Therefore, its `Replay/Rollback` time and `Wait` time are



(a) Google Play Services (b) Facebook (c) Twitter (d) Google Calendar (e) Google Maps

Figure 14: CDF of Latencies Spent for Real Workloads

smaller than those of `Baseline`. The number of aborted transactions in 2VTO is only 1/12 of that of `Baseline` because long running transactions that conflict are serialized by AFUW. As such, its `Replay/Rollback` overhead is much smaller than `Baseline` and `BeginConc`. Instead, IPS+2VTO makes transactions sleep for about about 40% of its execution time. Since IPS+2VTO does not use CPU cycles during the forced sleep, IPS+2VTO executes 23.7% and 20% less instructions than `BeginConc` and `Baseline` does, respectively.

For read-intensive Calendar workload, write-write conflicts rarely occur and fewer transactions abort. Therefore, `BeginConc` shows much better performance than `Baseline` due to higher concurrency. With fewer write-write conflicts, the overhead of `ForcedSleep` is almost negligible in IPS+2VTO. Also, read transactions benefit from non-blocking reads. Thus, the scheduling overhead of IPS+2VTO accounts for less than 4% of the transaction response time.

In the experiments shown in Figure 14, we measure the transaction latencies of real workloads. Note that we do not include the queueing delay in the latency. In Google Play Services workload, about 58.7% of the transactions are read-only. Therefore, about 50% of the transactions show similar latencies for all concurrency modes although IPS+2VTO and DHL+2VTO show lower latencies than the others. However, it is noteworthy that, for 72th~90th percentile latencies, IPS+2VTO, DHL+2VTO, and `BeginConc` perform worse than file-based locking (i.e., `Baseline`). This is because of the Rollback/Replay overhead. I.e., about 18% of write transactions using per-page locks are aborted due to write-write conflicts, whereas file-based locking does not. However, these aborted transactions are rescheduled and commit fast enough to have lower latencies than the 90th percentile latency. The 90th percentile and higher tail latencies show different results. IPS+2VTO and DHL+2VTO have the lowest tail latencies of all others because they limit the number of aborts per transaction to one. In addition, the tail latencies of `BeginConc` and `Baseline` are higher because they perform periodic checkpointing operations.

Facebook is the most write-intensive workload, but its transactions consist of UPDATE statements and its transaction size is not as large as GPS, as shown in Table 1. Due to the internal fragmentation issue, the 30th~80th percentile latencies of IPS+2VTO and DHL+2VTO are higher than those of `BeginConc`. However, the 90th percentile and higher tail latencies of IPS+2VTO are much lower than the others, as in GPS workload. DHL+2VTO has higher tail latencies than IPS+2VTO due to low page utilization. Twitter is another write-intensive workload, in which 19% of transactions contain UPDATE queries. However, the transaction size is larger than Facebook. Therefore, the effect of the internal fragmentation issue of IPS is offset by its higher concurrency, and it outperforms the others. In terms of the 90th and higher tail latencies, IPS+2VTO is similar to the others. This is because Twitter workload has many large transactions that rarely conflict. For read-intensive Google Calendar and Maps, small read transactions in IPS+2VTO and DHL+2VTO benefit from non-blocking reads.

6 CONCLUSION

In this work, we develop In-Page Shadowing (IPS) that resolves the journaling of journal anomaly and enables 2VTO (*Two-Version Timestamp-Ordering*) protocol to support a moderate level of concurrency in mobile devices. 2VTO with AFUW (Abortee-or-First Updater Wins) rule limits the number of aborted transactions to one for resource-constrained mobile devices. Our performance study using synthetic and real workloads shows that IPS successfully reduces the IO traffic and outperforms the state-of-the-art logging schemes such as DASH and DHL. Our study also shows that 2VTO significantly reduces the number of aborted transactions while improving the concurrency level. Thereby, 2VTO outperforms an optimistic concurrency control protocol - *Begin Concurrent*. For future work, we intend to explore the possibility of extending IPS for MVCC for higher concurrency. To be specific, if we address the challenges of managing an arbitrary number of version-related metadata without the help of a background garbage collection thread, higher concurrency is not going to be impossible.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and feedback. We would also like to give special thanks to Kisung Lee at Samsung Electronics for motivating this work. This research was supported in part by Samsung Electronics, and also by the R&D program of IITP (grant No. 2015-0-00314, 2018-0-00549, and 2021-0-01817) and Electronics and Telecommunications Research Institute (ETRI) (grant No. 20ZS1310). The corresponding author is Beomseok Nam.

REFERENCES

[1] Android Processes and Threads Overview. *https://developer.android.com/guide/components/processes-and-threads*.

[2] Google Play Services. *https://developers.google.com/android.*

[3] SQLite Begin-concurrent Work-in-progress. *https://sqlite.org/src/doc/begin-concurrent/doc/begin_concurrent.md.*

[4] SQLite Isolation. *https://www.sqlite.org/isolation.html*.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] Visutr Boonnateephisit. *Andros Store: The Open-Source Android Application Store*. PhD thesis, AIT, 2019.

[8] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[9] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, page 21, USA, 2010. USENIX Association.

[10] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. *ACM SIGPLAN Notices*, 48(8):271–280, 2013.

[11] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[12] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. Androstep: Android storage performance analysis tool. *Software Engineering 2013-Workshopband*, 2013. *https://github.com/ESOS-Lab/Mobibench*.

[13] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[14] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-ftl: transactional ftl for sqlite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2013.

[15] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. Pocket data: The need for tpc-mobile. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, pages 8–25, Cham, 2016. Springer International Publishing.

[16] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip 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.

[17] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[18] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 55–66, 2007.

[19] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 235–247, July 2015.

[20] Douglas J. Leith and Stephen Farrell. Contact tracing app privacy: What data is shared by europe's gaen contact tracing apps. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[21] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024, 2014.

[22] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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.

[23] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1454–1465, August 2015.

[24] Sehyeon Oh, Wook-Hee Kim, Jihye Seo, Hyeonho Song, Sam H. Noh, and Beomseok Nam. Doubleheader logging: Eliminating journal write overhead for mobile dbms. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1237–1248. IEEE Computer Society, apr 2020.

[25] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380, 2009.

[26] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. Sql statement logging for making sqlite truly lite. *Proceedings of the VLDB Endowment (PVLDB)*, 11(4):513–525, 2017.

[27] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12), 2012.

[28] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2005.

[29] David Patrick Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, 1978.

[30] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam 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.

[31] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.

[32] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, 2017.

[33] Gregory F. Welch. A survey of power management techniques in mobile computing operating systems. *ACM SIGOPS Operating Systems Review*, 29(4):47–56, oct 1995.

[34] Youjip Won, Sundoo Kim, Juseong Yun, Dam Quang Tuan, and Jiwon Seo. Dash: Database shadowing for mobile dbms. *Proceedings of the VLDB Endowment (PVLDB)*, 12(7):793–806, 2019.

[35] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment (PVLDB)*, 10(7):781–792, March 2017.