SQL Statement Logging for Making SQLite Truly Lite

Jong-Hyeok Park
Sungkyunkwan University
Suwon 440-746, Korea
akindo19@skku.edu

Gihwan Oh
Sungkyunkwan University
Suwon 440-746, Korea
wurikiji@skku.edu

Sang-Won Lee
Sungkyunkwan University
Suwon 440-746, Korea
swlee@skku.edu

ABSTRACT

The lightweight codebase of SQLite was helpful in making it become the de-facto standard database in most mobile devices, but, at the same time, forced it to take less-complicated transactional schemes, such as physical page logging, journaling, and force commit, which in turn cause excessive write amplification. Thus, the write IO cost in SQLite is not lightweight at all.

In this paper, to make SQLite truly lite in terms of IO efficiency for the transactional support, we propose *SQLite/SSL*, a *per-transaction SQL statement logging* scheme: when a transaction commits, *SQLite/SSL* ensures its durability by storing only SQL statements of small size, thus writing less and performing faster at no compromise of transactional solidity. Our main contribution is to show that, based on the observation that mobile transactions tend to be *short* and exhibit *strong update locality*, logical logging can, though long discarded, become an elegant and perfect fit for SQLite-based mobile applications. Further, we leverage the WAL journal mode in vanilla SQLite as a *transaction-consistent checkpoint* mechanism which is indispensable in any logical logging scheme. In addition, we show for the first time that byte-addressable NVM (non-volatile memory) in host-side can realize the full potential of logical logging because it allows to store fine-grained logs quickly.

We have prototyped *SQLite/SSL* by augmenting vanilla SQLite with a transaction-consistent checkpoint mechanism and a redo-only recovery logic, and have evaluated its performance using a set of synthetic and real workloads. When a real NVM board is used as its log device, *SQLite/SSL* can outperform vanilla SQLite's WAL mode by up to 300x and also outperform the state-of-the-arts SQLite/PPL scheme by several folds in terms of IO time.

PVLDB Reference Format:
Jong-Hyeok Park, Gihwan Oh, Sang-Won Lee. SQL Statement Logging for Making SQLite Truly Lite. *PVLDB*, 11(4): 513 - 525, 2017.
DOI: https://doi.org/10.1145/3164135.3164146

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 4
Copyright 2017 VLDB Endowment 2150-8097/17/12... $ 10.00.
DOI: https://doi.org/10.1145/3164135.3164146

1. INTRODUCTION

Given that SQLite is used as the *de-facto* standard database manager in major mobile platforms, Android, iOS, and Tizen, it is not surprising that popular mobile applications such as Facebook, Twitter, Gmail, and messengers, manage data using SQLite [3]. Further, with the advent of the era of *mobile first*, *Internet of Things* and messenger-style chatbots, it is obvious that users will carry out more computing on mobile platforms unprecedentedly [28], and thus more transactional data will be managed by SQLite.

The pervasive use of SQLite in mobile platforms is mainly due to the development productivity, solid transactional support, and lightweight codebase. But, the compromise for the lightweight codebase, at the same time, forced it to take less-complicated but costlier schemes for the transactional support, such as physical logging at page granularity, redundant journaling, and force commit policy [2, 4]. Hence, this is often cited as the main cause of huge write amplification and tardy response time in mobile applications [17, 26]. In addition, considering more than two-thirds of all writes in smartphones are from SQLite [29], the write amplification by SQLite will, in turn, shorten the lifespan of flash storage in mobile devices.

Considering the ever-growing popularity of SQLite but at the same time its run-time overhead of write operations, it is compelling to make SQLite truly lite in terms of write efficiency. In this paper, for this purpose, we propose a form of logical logging scheme, called *SQLite/SSL*: on a transaction commit, *SQLite/SSL* will write all update SQL statements of the committing transaction persistently and atomically in log device. In this sense, *SQLite/SSL* takes a *per-transaction SQL statement logging* for its transactional support. Thus, it can avoid the overhead of vanilla SQLite: *force-writing every modified page in its entirety redundantly at every commit*. In an ideal case, upon every transaction commit, the single write operation of a small amount of per-transaction SQL statements log in *SQLite/SSL* will replace the redundant write of several or tens of physical pages in vanilla SQLite, thus writing less and performing faster.

In fact, despite its compactness and simplicity, the logical logging approach has been discarded in database community mainly for two reasons [12, 13, 24]. First, there is no efficient *transaction-consistent checkpoint* (in short, TCC) mechanism, which is crucial in realizing any logical logging approach. Second, even though any TCC mechanism does exist (*e.g.*, the shadow page technique [23]), it is quite unrealistic in large multi-user database environments because

of high checkpoint IO cost and intolerable delays for newly incoming transactions during quiescent checkpoints [13].

Then, conversely, could the logical logging approach be practical for a database engine in case the system manages small database in single-user mode and also has a TCC mechanism? Our work on *SQLite/SSL* is motivated by this question as well as a few intriguing observations on SQLite architecture itself, popular mobile applications and their workload characteristics. The first observation is that because the fundamental role of two journaling modes in vanilla SQLite is to propagate multiple pages updated by a committing transaction from buffer cache to the original database atomically [17], a TCC mechanism for logical logging can be easily embodied by slightly extending either journaling mechanism in vanilla SQLite. The second one is that most mobile application run in single-user mode and the size of the database is relatively small. In particular, although concurrent read operations are allowed in some applications, concurrent update transactions are not allowed in any SQLite-based mobile application. The third one is about the characteristics of mobile workloads. The transactions in mobile applications tend to be very *short* (*i.e.*, most transaction consists of one or a few DML statements and each SQL statement accesses a small number of data objects) and, more importantly, exhibits *strong update locality* (*i.e.*, the same logical pages are repeatedly updated by consecutive transactions) [18, 26].

The above observations led us to conclude that logical logging could be an ideal and practical solution for the transactional support in SQLite-based mobile applications. In particular, the characteristic of *short transactions with strong update locality* in mobile applications will allow the logical logging approach to drastically reduce the amount of pages to be written to the storage for the transactional durability: because only a small set of active pages are repeatedly updated by consecutive transactions, those pages will remain buffered at DRAM cache until next checkpoint, at which point of time each page will be written to the storage only once. Consequently, unlike vanilla SQLite which will repeatedly force-write those pages at every commit and thus incur huge write amplification [17, 26], logical logging can amortize repetitive updates to the same page by one write during the checkpoint. In addition, the characteristic will require logical logging to flush only small number of pages at each checkpoint, and thus the checkpointing will not introduce an unacceptable latency spike. Otherwise, if there is no outstanding update locality (*e.g.*, in an extreme case where updates are randomly made against a large set of pages), logical logging will not provide any benefit at all [6]: the eventual effect of logical logging is simply to delay the write operations for the updated pages until the next checkpoint, at which the checkpoint cost will be intolerably high [12].

The key contributions of this work are summarized as follows:

- Based on observations made about SQLite and popular mobile workloads running on SQLite, we recognize that the logical logging approach is a perfect fit for the transactional support in SQLite-based mobile applications. To our best knowledge, this is the first work that utilizes non-volatile memory (NVM) so as to make logical logging realizes its full potential: fine-grained logical logs can be quickly written with byte-addressable NVM without the overhead of standard I/O stack.
- We have designed and implemented a new mobile database manager, *SQLite/SSL*, on a *real* NVM board by augmenting vanilla SQLite minimally and compatibly. It optimizes data management for mobile applications by replacing redundant page writes with fine-grained SQL log writes. For transactional atomicity and recoverability upon system crashes under its *no-force* commit policy, *SQLite/SSL* provides a *transaction-consistent checkpoint* mechanism and a *redo-only recovery* logic.
- *SQLite/SSL* has been evaluated empirically with real traces obtained from popular mobile applications as well as a synthetic benchmark. We have observed that *SQLite/SSL* can improve the performance of mobile applications by an order of magnitude.

2. BACKGROUND AND MOTIVATIONS

Clearly, logical logging is the best approach in guaranteeing the durability of committing transactions in terms of log compactness since one single log record (*e.g.*, one SQL statement) could correspond to several or tens of physical page updates or Aries-like physiological log records [12]. Though elegant, however, any logical logging approach has not been successful mainly for two reasons. First, there has not been an efficient transaction-consistent checkpoint solution. Second, there has not been any major killer database application with which logical logging can fit well.

From this point of view, this section will describe why logical logging is a perfect fit for mobile applications running on SQLite database. Concretely, as background and motivation of our work, we will review the SQLite architecture and the characteristics of mobile workloads in depth, particularly pointing out that SQLite provides TCC and most mobile applications are running in single user mode and their transactions tend to be short and with strong update locality. In addition, we will review the performance characteristics of a byte-addressable non-volatile memory and explain why it can make logical logging more attractive than ever.

2.1 SQLite Architecture

SQLite is a software library that implements a serverless transactional SQL database engine [3], and thus mobile applications are linked to the SQLite library to utilize its database management functions. In SQLite, tables and indexes are managed in a single database file on top of a underlying file system such as `ext4`. In order to provide the solid transactional atomicity and durability while keeping its codebase lightweight as well as portable on a wide spectrum of platforms, SQLite takes less-complicated but costlier recovery schemes. The architectural features of recovery in SQLite are summarized below.

For the durability of committing transactions, SQLite adopts the force policy for buffer management: when a transaction commits, it force-writes all the pages updated by the transaction in their entireties to a stable storage by calling the `fsync` command. In addition, since the atomic propagation of one or more updated pages is not guaranteed by the underlying operating system and storage device, SQLite relies on redundant journaling mechanisms for the atomicity of committing transactions: *rollback* mode [2] and *write-ahead log* mode [4]. In *rollback* mode, if a transaction is about to update a page, the original content of the page is copied to the rollback journal file before updating

it in the database, so that the change can always be undone if the transaction aborts. In this regard, *rollback* mode takes a *undo-based journaling*. In contrast, *write-ahead log* mode (hereafter, WAL mode for short) takes a *redo-based journaling*. In WAL mode, pages updated by a committing transaction are appended to the WAL journal file while their old page copies remain intact in the original database. The change is then later propagated to the database by checkpoint. Once safely written in the WAL file, any committed change can be redone by copying the recent page copies from the WAL file to the original database.

In either mode, the less-complicated transactional scheme taken by SQLite causes the costlier run-time IO overhead since every page updated by every committing transaction should be redundantly force-written in its entirety. Further, given that the actual amount of changes in each page made by mobile SQLite transactions is generally very small [18, 26], the *redundant force-writing of updated pages* is the root cause of the huge write amplification in SQLite [15, 17, 29].

The journaling mechanisms of SQLite, meanwhile, offer one attractive aspect in implementing a logical logging approach. For example, the checkpointing and recovery schemes in WAL mode guarantee the atomic propagation of a set of all pages updated by committed transactions to the original database despite crash. Therefore, the original database under the WAL mode will always remain in transaction-consistent state. In this sense, the WAL mode provides a transaction-consistent checkpoint (TCC) mechanism which is indispensable in any logical logging approach including *SQLite/SSL* [12, 13, 24]. This architectural aspect of SQLite allows *SQLite/SSL* to adopt the logical logging of *per-transaction SQL statements*, where all modified pages by recently committed transactions are checkpointed in a transaction-consistent manner.

2.2 Mobile Workload Characteristics

Now let us explain the transactional characteristics of mobile applications running on SQLite and their adverse effects on write amplification on vanilla SQLite, and discuss the opportunities they provide for logical logging.

Short Transactions with Strong Update Locality

As mentioned above, popular mobile applications manage their data using SQLite. And mobile transactions in those applications have a few unique characteristics. First of all, they tend to be very short, mostly running in the autocommit mode, where a transaction consists of a single SQL statement [29]. More importantly, each transaction usually modifies very small amount of data. For instance, the database workload in a mobile messenger application is mostly *small insertions* and, once stored, most of the messages are seldom deleted or update [18, 26]. The second, and more important, one is that mobile transactions exhibit strong update locality. Whenever a new record is inserted into a SQLite database table (which is also organized as a B^+-tree), it is inserted into the rightmost leaf node of the table B^+-tree because a surrogate key automatically created by SQLite is stored together as part of the record and the surrogate keys are increasing monotonically. Consequently, the *same* leaf node of the table will be consecutively updated until that node becomes full when successive insertions are made to the table. Interestingly, the secondary indexes from mobile applications also show high update locality [26],

Opportunities for Logical Logging

To better understand the opportunities that the characteristic of "*short transaction with high update locality*" in mobile applications provides in terms of logical logging, we ran SQLite traces collected while running six popular mobile applications and, for each trace, measured several update-related metrics and summarized the results in Table 1. Refer to Section 5.2 for the detailed description of those applications. The second and third columns in Table 1 show the total number of logical page writes made by update transactions (`A`) and the total number of distinct pages updated by update transactions (`B`) in each trace, respectively. The fourth column shows the total number of update transactions in each trace (`C`). The fifth column shows the average number of logical page writes made by each update transaction (`D`), which is calculated by dividing `A` by `C`. Lastly, the sixth column shows the average number of overwrites per updated pages (`E`), which is calculated by dividing `A` by `B`.

From Table 1, we can make a few important observations on why logical logging is an attractive alternative for mobile applications. First, as shown in column `D`, each transaction in all applications except `Twitter`, mostly running in autocommit mode, updates three to eight pages on average. In this case, it is well known that logical logging is especially useful by replacing multiple pages writes with one SQL statement log [6]. The second observation is that, across all the traces used, the actual number of all distinct pages updated in each trace (column `B`) is relatively very small, compared to the total number of pages written in each trace (column `A`). In addition, from column `E`, we know that one same logical page is repetitively overwritten when each trace was run using vanilla SQLite in the WAL mode. In one extreme case of `AndroBench`, each page is overwritten on average almost up to 150 times. This confirms that database workload in mobile applications is mostly small updates. Therefore, taking into account that the default size of buffer cache in SQLite (*i.e.*, 1,000 pages) is large enough to buffer all the pages updated by many consecutive transactions, there is no compelling reason to take the force commit policy as long as the durability of each committing transaction can be guaranteed in another way (*e.g.*, SQL statement logging). Therefore, by taking logical logging approach and thus buffering updated page in DRAM, instead of force-writing them upon every commit, a multitude of successive page writes to the same logical page can be avoided. The third observation is about checkpoint and recovery. Since only small number of active pages will be updated by many consecutive transactions and those pages can be buffered in the cache, those pages can be checkpointed in a transaction-consistent way without causing unacceptable latency spike. In addition, compared to the force commit policy in vanilla SQLite, the WAL file will be filled up at a much slower rate under logical logging approach because of its write buffering effect. Therefore, checkpoint operations are called much less frequently than vanilla SQLite. In addition, in terms of recovery time, the number of pages to be recovered from crashes is limited so that the recovery can be completed only with small IOs.

2.3 Phase Change Memory (PCM)

One obvious benefit of logical logging over other logging techniques such as physical logging and *Aries*-style physiological logging is its compactness of log. However, if the log

Table 1: Analysis of Update Patterns in Mobile Application Traces

Trace	Total # of page writes (A)	Total # of distinct pages updated (B)	Total # of update TX (C)	Page writes / TX (D = A/C)	Avg. overwrites / page (E = A/B)
AndroBench	10,407	37	3,077	3.38	281.3
Gmail	6,041	190	704	8.58	31.8
KakaoTalk	7,835	178	2,187	3.58	44.0
Facebook	3,717	476	1,194	3.11	7.8
Browser	5,232	613	1,350	3.88	8.5
Twitter	11,083	278	7,907	1.40	39.9

data is stored in files on the secondary block storage, the advantages of writing small logical log will be offset mainly by the I/O stack. Specifically, because the I/O stack takes about 20,000 instructions to issue and complete a 4KB IO request under standard Linux, its overhead can exceed the hardware access time in fast storage devices such as flash memory SSDs [7]. Therefore, the best way to making the most of logical logging is to use a persistent memory abstraction with DIMM interface. By doing so, we can avoid the latency of I/O stack and also minimize the write amplification at the flash memory storage layer.

A contemporary PCM product can write 4 bytes in 7.5 *us* while a TLC NAND flash memory chip takes as long as 1,500 *us* in writing 8KB [8, 16]. A similar read and write speed was observed in other PCM products as well [19, 22, 25]. From this, we confirm that current PCM technology is, though not delivering its promised performance as yet especially for write operations [21], absolutely superior to flash memory for fine-grained writes (*e.g.*, less than several hundred bytes). Thus, it is obvious that PCM is an ideal and practical log device for small-sized logical logs.

Considering that the PCM technology is still in its infancy in the commercial market and its price is quite higher (at least, 10 times expensive as of now) than that of flash memory, it is unlikely that PCM will supplant flash memory in foreseeable future. Instead, we expect that while flash memory device is used as main storage, a small amount of PCM will, in the form like the UMS board, be complementarily used as a special purpose device. In this paper, we will show that the availability of byte-addressable PCM as log device is key to making logical logging realize its full potential. Meanwhile, the lifespan of NVM is in general quite longer than that of flash memory: NVM can be overwritten at least 10^6 times [21]. Therefore, when NVM is used as log device for logical logging, its lifespan, though limited in theory, would not be a limiting issue in practice.

In this paper, as the PCM device, we will use a prototype development board that allows PCM to be accessed via DIMM interface [22]. This prototype will be called a unified memory system (UMS), as both DRAM and PCM can be accessed through the same DIMM interface [26]. With this board, an application can write a small amount of data (*e.g.*, a logical log record), much smaller than a page, persistently to PCM through the DIMM interface.

3. RELATED WORKS

By taking a logical logging approach to leverage the characteristics of mobile applications and also by exploiting NVM in storing fine-grained logs quickly, *SQLite/SSL* can, at no compromise of its transactional solidity, minimize the amount of data written to the flash storage. In this regard, three types of existing work are related to *SQLite/SSL*. Each work is briefly reviewed and compared with *SQLite/SSL* below.

3.1 Logical Logging

At least two database systems have taken logical logging approach: operation logging in System/R [11] and command logging in VoltDB [24]. In operation logging, the before- and after-value of one or more records updated by an update SQL statement are logged. In command logging, the stored procedure name with its actual parameters is the unit of logging. Although each scheme differs in the format of log and the layer where log is captured, these two schemes and *SQLite/SSL* are common in that they try to minimize the size of log data for faster durability. In this respect, the idea of *SQLite/SSL* is not new.

However, *SQLite/SSL* has made three contributions distinguishable from the previous studies. First, we show for the first time that *SQLite/SSL*, a variant of logical logging, can be a perfect fit for SQLite-based mobile applications. In fact, large multi-user databases have been main-stream in database community, and there was no practical solution for transaction-consistent checkpoint [11, 13]. For this reason, logical logging had been rejected from the database community until VoltDB's command logging and its asynchronous transaction-consistent checkpoint technique is recently proposed [24]. Second, *SQLite/SSL* present a new way to implement transaction-consistent checkpoint by leveraging the existing WAL checkpoint mechanism in vanilla SQLite, which is different from the existing ones [11, 24]. Third, *SQLite/SSL* is the first work to show that logical logging can realize its full potential when combined with byte-addressable NVM. As shown in Section 5, the performance of *SQLite/SSL* can boost by changing its log device from flash storage to host-side *"real"* PCM device.

3.2 NVM-based Logging

In order to exploit the fast durability and byte-addressability of NVMs, many NVM-based logging schemes have recently been proposed [5, 9, 10, 20, 26]. Among them, SQLite/PPL [26] and NVWAL [20] are closest to *SQLite/SSL* in that they utilize NVM to boost the SQLite performance. Whenever pages are updated by a transaction, the changes are captured in either physio-logical log [26] or physical-differential log [20], and, later when the transaction commits, the logs are flushed to NVM. However, *SQLite/SSL* is in stark contrast with these schemes in that while they capture *per-page differential logs*, *SQLite/SSL* takes *SQL statement logging*. Therefore, *SQLite/SSL* will be

obviously superior to them mainly due to its log compactness, especially in SQLite-based mobile applications. Considering the time taken to write in NVM is proportional to the amount of data to transfer, more log means longer commit latency. More importantly, given the same size of PCM, larger log data will trigger more frequent checkpoints. For this reason, as will be shown in Section 5, *SQLite/SSL* outperforms SQLite/PPL by several folds in many cases.

3.3 Flash-optimized Single-Write Journaling

One of the main roles of SQLite RBJ and WAL journaling is to atomically propagate multiple pages updated by a transaction to the storage. However, the atomicity comes at the cost of redundant writes [17]. This double-write journaling is one of the major factors explaining the huge write amplification in SQLite databases. To achieve the write atomicity of multiple pages at no cost of redundant writes, two novel schemes, X-FTL [17] and SHARE [27], have been recently proposed for flash storage from the database community. Though quite novel, they should force-write all physical pages updated by every committing transaction and thus will cause excessive write amplifications in SQLite-based mobile applications. In addition, they assume a flash storage with a special interface and accordingly require some changes in OS kernel stack. In contrast, *SQLite/SSL* drastically reduces the amount of data directed to the flash storage by storing only SQL statements as log in NVM and by taking a periodic checkpoint.

4. DESIGN OF *SQLite/SSL*

In this section, we present a new mobile database manager called *SQLite/SSL* that logs only SQL statements upon commit, thus achieving its transactional atomicity and durability in a truly lightweight manner. For the realization of the statement logging strategy in *SQLite/SSL*, we have augmented vanilla SQLite with a few new features and also have modified its existing modules minimally. This section presents the design overview of *SQLite/SSL*, describes its overall architecture, shown in Figure 1, and provides the detailed description of modules and data structures (in gray color) added to or modified from vanilla SQLite.

4.1 Design Overview

Design objectives The design objectives of *SQLite/SSL* are threefold. First, while embodying its new functionalities, *SQLite/SSL* takes full advantage of the existing proven features in vanilla SQLite so as to make only minimal changes and thus keep its codebase as reliable as vanilla SQLite. For instance, the WAL mode in vanilla SQLite was leveraged to embody a *transaction-consistent checkpoint* in *SQLite/SSL*. Second, *SQLite/SSL* should be able to keep its recovery logic as simple and efficient as vanilla SQLite. In fact, as is described below, *SQLite/SSL* introduces an additional data structure for logging SQL statements, *SLA*, which in turn can, upon crashes, lead to the numerous failure combinations of *SLA* and the existing WAL journal. Therefore, we deliberately chose to take a simple checkpoint and recovery logic at the cost of some performance overhead. Third, *SQLite/SSL* aims at making the implementation logic of logging SQL statements in *SLA* as generic as possible. In addition to PCM, the emerging NVDIMM [14] as well as the existing flash storage can also be used as the log device for *SQLite/SSL*. Therefore, as is detailed later, we use the *mmap* and *msync* calls to achieve both the device independence and the byte-addressability in storing statement logs irrespective of the log devices.

New data structures *SQLite/SSL* introduce two new key data structures: *statement log area* (*SLA*) and *statement log buffer* (*SLB*). For an active transaction, all the updating SQL statements, in addition to `transaction_begin`, `commit`, and `abort`, are captured and buffered in *SLB* (Step 1 in Figure 1). When a transaction is about to normally commit, all the updating statement logs of the transaction which are buffered in *SLB* will be flushed to *SLA* (Step 2 in Figure 1). Note that because the system can crash while flushing logs from *SLB* to *SLA*, all logs of a committing transaction should be atomically flushed to *SLA*. When *SLA* is managed in the byte-addressable NVM with DIMM interface, the `mmap` call will avoid the overhead of I/O stack. In contrast, when *SLA* is managed in block storage device, not in UMS board, the `msync()` command should be further called to make the log of *SQLite/SSL* durable in *statement-log-file*, as illustrated in Figure 1 (Step 2.1). In this case, the log write in *SQLite/SSL* will follow the standard I/O stack of file systems. The `mmap` interface is chosen as a unified abstraction to access any byte-addressable NVM so that *SQLite/SSL* should be able to work without any changes in its code even when any storage media is used as *SLA*. Since every modern file system supports the `mmap` interface, both a specific DRAM area and file residing on flash storage can be accessed using a single `mmap` interface.

Durability, atomicity, and recovery *SQLite/SSL* differs from the vanilla SQLite mainly in the way transactional durability and atomicity are guaranteed. When a transaction commits, *SQLite/SSL* guarantees its durability by force-writing all update SQL statements of the committing transaction in *SLA*. Note that under *SQLite/SSL* all the updated pages by the committing transaction are buffered in DRAM cache. When the log data reaches a threshold in *SLA*, the checkpoint process is triggered, by which every pages dirtified by the committed transactions but still buffered in DRAM cache are propagated to its permanent location in original databases. In terms of transactional durability, a multitude of successive page writes against the same logical pages by successive committing transactions in vanilla SQLite is replaced by one page write per each logical page at checkpoint in *SQLite/SSL*. Therefore, *SQLite/SSL* can achieve faster durability with much less write amplification than vanilla SQLite, especially when the byte-addressable PCM is used as the storage media for *SLA*.

For transactional atomicity, *SQLite/SSL* basically takes the same approach with vanilla SQLite. Both systems rely on the WAL journal to ensure that all the pages updated by committed transactions are atomically propagated to the original database. However, while vanilla SQLite force-writes all updated pages to the WAL journal upon every commit, the buffer manager in *SQLite/SSL* has been modified to take the *no-force* commit policy so that all pages updated by committing transactions remain buffered in DRAM cache, as depicted in the left side of Figure 1. Those pages will be, upon next checkpoint, first journaled in the WAL file and then written to the original database. This transaction-consistent checkpoint in *SQLite/SSL* will

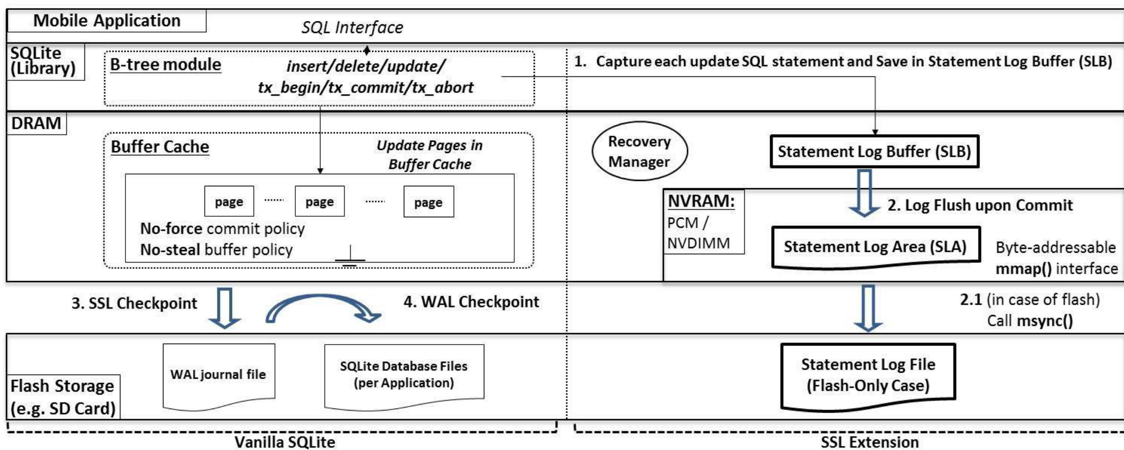


Figure 1: *SQLite/SSL*: Architecture

guarantee the transactional atomicity against unexpected crashes. Please refer to section 4.2 for details.

Upon crash, database can be recovered to the state where the last transaction successfully committed by re-executing all valid SQL statements recorded in *SLA* against the original database. Note that under *SQLite/SSL* the original database remains unchanged since the last checkpoint because any updated page is not allowed to propagate to the original database until the next checkpoint.

4.2 Added Functions

For the realization of *SQLite/SSL*, vanilla SQLite is augmented mainly with four functional modules (`log capturer`, `log writer`, `transaction-consistent checkpoint`, and `recovery manager`) as is illustrated in Figure 1. The functional modules are described below in more detail.

Log capturer In the vanilla SQLite, when a new SQL statement is issued from an active transaction, it is first parsed at the Virtual Database Engine(VDBE) layer. For each updating statement (*i.e.*, `INSERT, DELETE`, or `UPDATE`) which has passed the parsing step, `log capturer` buffers the statement into *SLB* in sequence. To capture update statements at the VDBE layer is very crucial in making our *SQLite/SSL deterministic*. If pure SQL statements with functions or parameters varying over time are captured and logged, the database state recovered from crashes by re-executing those SQL statements will not be deterministic. Fortunately, every parsed SQL statement at the VDBE layer has deterministic values as its parameters, and therefore the recovery in *SQLite/SSL* will be always deterministic. In addition, the transaction-consistent checkpoint scheme in *SQLite/SSL* can guarantee any physical structural changes (*e.g.*, index splits) which might be non-deterministic to be propagated to the original database in a transaction-consistent manner. Therefore, *SQLite/SSL* can maintain its original database always deterministic even with its SQL statement logging scheme.

Meanwhile, *SQLite/SSL* does not either capture or log any `SELECT` statements. We made this design choice for two reasons. First, SELECT statement does not update any data page. Second, and more importantly, all updating SQL statements in all the mobile applications we observed have no dependency on any preceding `SELECT` statement in terms of their parameter values. In addition, it is obvious that no logging of SELECT statements will keep *SLA* more compact, which will in turn trigger less checkpoints.

The format of a log record in *SQLite/SSL* is shown in Figure 2. The `Length` field stores the length of the following SQL Statement. This field was introduced to distinguish each SQL statement while reading the SQL statements from *SLA* during the recovery process. The `SQL Statement` field stores an SQL statement to be logged. In addition, the four byte `CRC` (cyclic redundancy check) field is added to ensure the atomicity of writing a log record longer than 32 bytes because the UMS board we used does not guarantee the atomicity [22]. Upon reading each log record during recovery, a CRC calculation is repeated and, when new CRC value does not match with the CRC value stored in the log, the log record is regarded as invalid and the recovery process stops.

Storing SQL statements sequentially is the key to knowing the beginning and end of a transaction. Therefore, it allows to handle both `autocommit` and `batchcommit` modes in a unified way. From the start of the transaction, the SQL statement is stored in the statement log buffer, and the processing depends on the behavior of the transaction (`commit`, `rollback`, `abort`). In the case of commit, it is guaranteed to be stored durable in *SLA* against the SQL statement log that has been buffered. In the case of rollback, the buffered statement log is simply discarded from *SLB*.

Length	SQL Statement	CRC

Figure 2: Log format in *SQLite/SSL*

Log writer When a transaction commits, the `log writer` is responsible for writing all the update SQL statement logs of the transaction persistently to *SLA*. When PCM is used as *SLA*, all the logs are copied from *SLB* in DRAM to *SLA* in PCM and then the `CLFLUSH` command is called to ensure the log durability. Meanwhile, when flash

memory is used as *SLA*, the durability of the log write is ensured by invoking the `msync` call to *SLA*.

Transaction consistent checkpoint For faster commit, *SQLite/SSL* takes the *no-force* commit policy. Therefore, to recover the database after a crash, *SQLite/SSL* will have to replay all the logged SQL statements in *SLA* against old database each time, which would be very time-consuming. Hence, all the updated pages by committed transactions should be regularly checkpointed to reduce the recovery time. As is depicted in Figure 1, the checkpoint process in *SQLite/SSL* consists of two sub-checkpoints: *SSL-checkpoint* and *WAL-checkpoint*. ***SSL-checkpoint*** is triggered when the current transaction successfully commits and, at the same time, either the amount of logs in *SLA* reaches to a pre-determined threshold (*i.e.*, 70%) or the number of dirty pages in the buffer cache is larger than 1024. While a checkpoint is in progress, no new transaction can start, likewise vanilla SQLite. But, the cost of *quiescent* checkpoints in both SQLite versions, as will be shown in Section 5, is acceptable consistently across all the real workloads we tested.

During *SSL-checkpoint*, all the pages once updated since the last checkpoint but still buffered in DRAM cache, which are maintained by the `pCheckpoint` list, will be written to the WAL journal (Step 3 in Figure 1). Once all the pages ever dirtified since the last checkpoint are written to the WAL journal, it is guaranteed that the effect of all the transactions executed since the last checkpoint has been made durable at the WAL journal. This, in turn, means that all the SQL statement logs in *SLA* can be safely truncated. As an implementation mechanism to truncate *SLA*, a field `log_size`, which represents the total amount of SQL statement logs in byte, is managed at the head of *SLA* file, and its value is reset to zero and all the old logs are reset by calling the `mset` interface. This atomic reset of *SLA* represents the success of *SSL-checkpoint*. It is crucial to reset *SLA* as the final step of *SSL-checkpoint* since, during the recovery upon a system crash, *SQLite/SSL* will leverage the status of *SLA* to identify the exact crash state.

Immediately after *SSL-checkpoint* is finished, ***WAL-checkpoint*** is triggered, which works exactly same as in vanilla SQLite: all the latest pages in the WAL journal are copied to the original database and then the WAL journal is reset by truncating all the page copies in it (Step 4 in Figure 1). In vanilla SQLite, meanwhile, *WAL-checkpoint* is not triggered on every transaction commit. Instead, it happens only when the number of pages in the WAL journal reaches the threshold (1,000 pages by default). One benefit of this lazy checkpoint is the write buffering effect: *i.e.*, when same pages are repeatedly written because of the update locality, only the most recent version of each page need to be copied from the WAL journal to the original database. Because of this write buffering effect, *write-ahead log* can outperform *rollback* in many cases [4, 17]. However, while designing *SQLite/SSL*, we instead decided to take immediate *WAL-checkpoint* right after *SSL-checkpoint* for two reasons. First, the effect of write buffering by the WAL journal is, if any, not such high in *SQLite/SSL* because the update locality will be absorbed mainly by the DRAM cache and thus the same page is not likely to be re-written to the WAL journal. Second, and more importantly, the lazy *WAL-checkpoint* will make the recovery logic more complicated.

Once the checkpoint is successfully completed in *SQLite/SSL*, the original database is now in a transaction-consistent state: it represents a consistent snapshot of the database at the time the last transaction commits, and does not contain any uncommitted changes. In this respect, *SQLite/SSL* accomplishes its *transaction-consistent checkpoint* by carrying out two sub-checkpoints in sequence. Even when a crash is encountered during the checkpoint, *SQLite/SSL* can, along with its *redo-only* recovery mechanism, guarantee that all updates from committed transactions since the last checkpoint be propagated to the original databases in all-or-nothing manner. Note that although *SQLite/SSL* takes the *steal* policy from the perspective of buffer replacement and thus some dirty pages would be replaced out from DRAM cache to the WAL journal, *SQLite/SSL* can be regarded as to take the *no-steal* policy in effect from the perspective of recovery because prior to *WAL-checkpoint*, any dirty page in the WAL journal is not allowed to propagate to its home location in original database. Therefore, the undo recovery is not required in *SQLite/SSL* since any update made by non-committed transactions is not allowed to propagate to the original database.

Recovery manager In vanilla SQLite, when the system restarts, the existence of the WAL journal file indicates a crash, and the database can be recovered simply by copying every page with a corresponding commit record in the journal file to the original database. Meanwhile, since new data structure *SLA* and an additional step of *SSL-checkpoint* are introduced, *SQLite/SSL* can encounter more crash scenarios. That is, it can fail at any point in time while flushing log to *SLA*, carrying out *SSL-checkpoint* or *WAL-checkpoint*. For this reason, the recovery module in *SQLite/SSL* has been carefully designed to cope with all the various failure cases, as will be detailed in Section 4.4.

4.3 Database Operations in *SQLite/SSL*

With the added data structures and functions described above, *SQLite/SSL* performs basic database operations such as `read`, `write`, `commit`, `abort`, and `normal shutdown` differently from vanilla SQLite. This section describes how those basic operations are performed in *SQLite/SSL*.

4.3.1 Read

The read operation in *SQLite/SSL* works in the same way with that in vanilla SQLite. On a page hit, the page frame found in the buffer pool can be returned because the page in the buffer pool is always up to date. On a page fault, a data page needs to be fetched from flash memory, whose up-to-date copy may reside in either the WAL journal or the original database file.

4.3.2 Update and Commit

The B^+-tree module of vanilla SQLite processes an update statement by inserting a new entry to or, deleting or updating an existing one from leaf nodes of a table and its secondary indexes. When a page is first updated by a transaction, its identifier is appended to the `pDirty` list. *SQLite/SSL* differs from vanilla SQLite in that, after processing an update statement, the `log capturer` captures a log of the statement and adds it to *SLB*.

Upon commit, the vanilla SQLite relies on the *force* policy to write all the dirty pages, which are listed in `pDirty`, to

the WAL journal immediately. Therefore, the commit time overhead is substantial because each dirty page is written twice physically (including the write operations incurred by checkpointing in the `WAL` journal mode) and a write barrier operation (by a `fsync` call) is executed at least once. After completing the flush operation, the `pDirty` list is reset to empty. In contrast, *SQLite/SSL* does not immediately force-write pages updated by the committing transaction. Instead, the durability of a committing transaction is ensured by logging all its update statements to the *SLA* persistently. Thus, because small amount of logical SQL statements is written as log in *SLA* in byte unit, the commit time overhead in *SQLite/SSL*, compared to vanilla SQLite, can drastically reduce, especially when the byte-addressable NVM is used as the device for *SLA*.

Note that, before resetting the `pDirty` list after having flushed SQL statement log, *SQLite/SSL* makes the snapshot of each page listed in `pDirty` and adds it to `pCheckpoint`. The `pCheckpoint` list is newly introduced in *SQLite/SSL* for the purpose of keeping track of all the pages which are updated at least once since the last checkpoint and thus need to be flushed upon next checkpoint. If a page is already listed in `pCheckpoint`, *SQLite/SSL* just copies its recent committed version to its snapshot page. The goal of maintaining separate snapshot of every updated page in `pCheckpoint` is to preserve the effects on each page ever made by all the committed transactions in preparation for transaction aborts. This issue will be detailed in Section 4.3.3.

Whenever new pages are updated by the committing transactions, the `pCheckpoint` list will be ever growing. However, taking the spatial locality of updates in mobile transactions into account, the length of the `pCheckpoint` list is usually kept very small (*e.g.*, several tens) until the next checkpoint at which it will be reset.

4.3.3 Abort

When a transaction aborts, vanilla SQLite simply discards all the pages updated by the transaction from the buffer by calling the `pcacheDrop` function for each page listed in `pDirty`. By doing this, any effect on each page updated by the aborting transaction can be completely removed. When any following transaction needs to access the dropped page again, it is fetched from either the WAL journal or original database. Vanilla SQLite can take this simple approach because of its *force* commit policy. That is, because any update on a page made by the preceding transactions which have successfully committed is made persistent either in the WAL journal or in the original database, the effect of in-memory undo can be achieved by reading the page from the storage. If the *force* commit policy is not taken, more complex undo recovery scheme should be devised.

Upon abort, likewise vanilla SQLite, *SQLite/SSL* also drops the pages updated by the aborting transaction from the buffer cache. In addition, all the buffered SQL statements from the aborting transactions will be discarded from *SLB*. But some of the pages being dropped might be included in `pCheckpoint`. For each page listed in `pCheckpoint`, *SQLite/SSL* will copy its most recent snapshot from `pCheckpoint` back to the page, thus reverting the page to the transaction-consistent state just before the aborting transaction started. By doing so, *SQLite/SSL* carries out the *in-memory undo* against the pages modified by the aborting transaction. This rather simple in-memory undo scheme was deliberately taken to avoid developing the complex undo logic like in Aries and also to prevent the excessive IO overhead due to the *force* policy in vanilla SQLite.

4.3.4 Normal Shutdown

When an application terminates, it will invoke the shutdown routine of the *SQLite/SSL* library. When the routine is called, it first triggers the checkpoint, and deletes WAL journal and *SLA* files in turn. Note that it is critical to keep the order of file deletions for the correct recovery in *SQLite/SSL* because an unexpected crash can happen during the normal shutdown and the existence of *SLA* upon restart indicates that system crashed.

4.4 Recovery

A crash may occur due to power loss or system crash during program execution. On a system reboot, vanilla SQLite has its own way to detect failure. It first checks whether the WAL journal file exists, and the existence of the file indicates that the system terminated abnormally. In this case, vanilla SQLite will copy the most recent version of each page in the WAL journal to the original database. Note that this recovery logic is idempotent in that it can recover the same consistent database despite repetitive crashes during the recovery. *SQLite/SSL* has extended this simple recovery logic in vanilla SQLite so that it can cope with all the various crash cases introduced by the new data structures such as *SLA* and *SSL-checkpoint*.

On a reboot, *SQLite/SSL* first checks the existence of the *SLA* file. If the file is not found, it means that the system has terminated normally and thus the normal operation can start without further recovery action. In contrast, the existence of the *SLA* file indicates that the system crashed. In this case, based on the existence of WAL journal as well as *SLA* file and their status (*i.e.*, *reset* or *in-use*), *SQLite/SSL* can identify the step at which it failed in the previous execution. Recall from Section 4.2 that the checkpointing in *SQLite/SSL* propagates all updated pages from the buffer cache to the original database along with the following four steps in sequence: 1) flushing dirty pages from buffer cache to the WAL journal, 2) resetting *SLA*, 3) copying all the pages from the WAL journal to the original database, and 4) resetting the WAL journal. The system can crash in any of these steps. In addition, the system can also crash before a *SSL-checkpoint* is triggered (that is, prior to step 1). Now let us explain how *SQLite/SSL* can identify the crashes into four cases according to the combinations of the existence and status of *SLA* and WAL journal file, and what actions it takes to recover from each case.

***SLA* = reset & No-WAL-file** This combination indicates that the system crashed right after deleting WAL file during the normal shutdown. Therefore, the system can resume simply after creating WAL journal file.

***SLA* = reset & WAL = reset** This combination indicates that the system crashed just after system initialization or checkpointing and no new transaction has not committed. Thus, *SQLite/SSL* can resume without further action.

***SLA* = reset & WAL = in-use** This indicates that the system crashed during the step 3 of checkpointing. In this case, *SQLite/SSL* will complete the recovery by copying the most recent version of each page in the WAL journal to the original database, and then resetting the WAL file.

***SLA* = in-use** When *SLA* exists and is in *in-use* status, it indicates, regardless of the WAL journal, that system crashed prior to or during the step 1 of checkpointing. This means that the effects of all committed transactions since the last checkpoint were made durable in *SLA* but not propagated to the original database yet. In this case, *SQLite/SSL* re-executes all the valid SQL statements from *SLA* in sequence against the original database, and then call its checkpoint to propagate all the pages to the original database. In this sense, the recovery process in *SQLite/SSL* can be regarded as *redo-only*. Recall that, as explained in Section 4.2, although a dirty page is allowed to be replaced out from DRAM buffer to the WAL journal before a checkpoint, the page is not allowed to propagate to its home locations in the original database until next checkpoint. In this respect, *SQLite/SSL* can be regarded to take the *no-steal* policy in effect from the perspective of recovery while it does the *steal* policy from the perspective of buffer replacement. Therefore, the undo recovery is not necessary for *SQLite/SSL*.

Finally, let us discuss the above four cases in *SQLite/SSL* in terms of recovery time. For the first two cases, the recovery time would be negligible. Also, for the third one, the recovery time in *SQLite/SSL* would be almost same to that in vanilla SQLite. But, the recovery time in the last case would not be marginal. In fact, the re-execution of SQL statements in *SLA* will incur many read operations and CPU overhead, and the recovery time is proportional to the number of SQL logs in *SLA*. But this prolonged recovery time is a compromise for its faster normal time performance.

5. PERFORMANCE EVALUATION

In this section, we present the results of empirical evaluation of *SQLite/SSL* and analyze its impact on the performance of mobile applications. We tested five real traces and one synthetic trace with the *SQLite/SSL* on the UMS board. To evaluate the performance effect of the *SLA* log device, we carried out the same experiment on the UMS board using flash memory SD card and PCM, respectively. For comparison, we also tested the same workloads on the UMS board with the vanilla SQLite in the *WAL* journal mode [4] and *SQLite/PPL* [26]. Also we carried out the same test with the *SQLite/SSL* and vanilla SQLite on a commodity PC with SD card as its storage device.

5.1 Experimental Setup

All the experiments were conducted with the UMS board and a commodity PC with flash memory SD card. The UMS board [22] is based on Xilinx Zynq-7030 equipped with a dual ARM Cortex-A9 1GHz processor, 1GB DDR3 533Mhz DRAM, 512MB LPDDR2-N PCM and a flash SD card slot. The host OS is a Linux system with 3.9.0 Xilinx kernel, and we used `ext4` file system in the `ordered` journaling mode. The version of vanilla SQLite used in this work is 3.13.0, and the size of database page is set to 4KB to match the page size of the underlying file system.

In order to evaluate the effect of *SQLite/SSL* on a commodity PC with fast CPU performance, a set of experiments was carried out using Linux system with 4.6 kernel running on Intel core i7-3770 3.40GHz processor and 12GB DRAM. We used the same `ext4` file system and the flash memory SD card storage as in the UMS board.

5.2 Workloads from Mobile Applications

For the evaluation, we used real traces from five popular mobile applications, all of which uses SQLite for data management: `KakaoTalk` messenger, `Gmail`, `Facebook`, `Twitter` and `Web Browser`. These traces were obtained by running the applications on a Nexus7 tablet with Android 4.1.2 Jelly Bean [26]. In addition, a publicly available mobile benchmark program, `AndroBench` [1], was used.

AndroBench is an update-intensive workload that consists of 3 different types of SQL statements performed on a single table with 17 attributes. The workload includes 1,024 insertions, 1,024 updates, and 1,024 deletes [1].

Gmail includes common operations such as saving new messages, reading from and searching for keywords in the inbox. It relies on SQLite to capture and store everything related to messages such as senders, receivers, label names and mail bodies in the *mailstore* database file. Therefore, this trace includes a large number of insert statements, and most of the SQLite transactions are run in the batch mode.

KakaoTalk is a popular mobile messenger application in Korea, which is similar to other messengers such as Whatsapp, Viber, and iMessenger. It stores the text messages in the *kakaotalk* database file. In the KakaoTalk trace, most transactions are processed in the autocommit mode.

Facebook was obtained from a Facebook application that reads news feed, sends messages and uploads photo files. Among 11 files created by the Facebook application, `fb.db` was accessed most frequently by many SQL statements. The other files were used to manage the information about users, threads, and bookmarks. Similarly to Gmail, this trace includes a large number of insert statements.

Browser was obtained while the Android web browser read online newspapers, surfed several portals and online shopping sites, and SNS sites. The web browser uses SQLite to manage the browsing history, bookmarks, the titles and thumbnails of fetched web pages using six database files.

Twitter As a social networking service, Twitter enable users to send and receive a short text message called tweet that is no longer than 140 bytes. Twitter manages text messages in 21 tables and 9 indexes distributed over seventeen database files, and most of the SQLite transactions process text messages in the autocommit mode.

In order to provide better insights into understanding the performance difference between vanilla SQLite and *SQLite/SSL*, we collected several metrics from each trace, and summarized them in Table 2. The second and third column represents the number of database files and the total size of database in each trace, respectively. The fourth and fifth column shows the distribution of transactions and the detail of the transactions broken down into SQL statement executed in the batch mode (enclosed by `begin` and `commit/abort`) and auto-commit mode, respectively. The sixth column shows the average number of logical page writes requested by a committing transactions in each trace when the database was run in the vanilla SQLite WAL mode. Lastly, the seventh column shows the average of total size of all update SQL statements per transaction in each trace. By comparing the average page writes per transaction (*i.e.*, the sixth column) and the average size of SQL statement per transaction (*i.e.*, the seventh column) from Table 2, we can expect that *SQLite/SSL* will show much faster commit latency than vanilla SQLite.

Table 2: Analysis of Mobile Application Traces

Trace	# of DB files	DB Size (MB)	Total # of TXs (Batch+Auto)	Total # of SQLs (Batch+Auto)	Page writes / TX	Avg. size of update SQL stmt. / TX (B)
AndroBench	1	0.19	3,081 (2+3,079)	3,082 (3+3,079)	3.38	215
Gmail	1	0.74	984 (806+178)	10,597 (10,419+178)	8.58	1,913
KakaoTalk	1	0.45	4,342 (432+3,910)	8,469 (4,559+3,910)	3.58	1,094
Facebook	11	1.95	1,281 (262+1,019)	3,082 (2,063+1,019)	3.11	1,094
Browser	6	2.51	1,522 (1,439+29)	4,493 (4,464+29)	3.88	8,304
Twitter	17	6.08	2,022 (17+2,005)	10,291 (448+2,005)	1.40	506

5.3 Performance Analysis

5.3.1 Baseline Performance

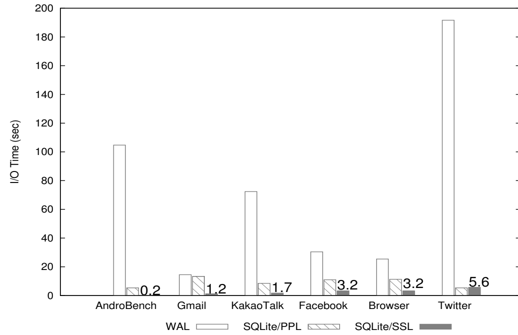
We measured the performance of *SQLite/SSL*, *SQLite/PPL*, and vanilla SQLite (in WAL mode), respectively, by replaying the six traces on the UMS board using PCM as *SLA* log device. Also, in order to evaluate the effect of *SQLite/SSL* when the traditional block storage device is used as *SLA* log device and a faster CPU is available, we measured the performance of vanilla SQLite and *SQLite/SSL* by replaying the traces on a commodity PC using flash memory SD card as *SLA* log device. The results are presented in Figure 3. Figure 3 shows the I/O time taken to each workload completely. From Figure 3(a), we see that, when PCM is used as *SLA* log device, *SQLite/SSL* can outperform *SQLite/PPL* and vanilla SQLite by up to 27 and 300 times, respectively. From Figure 3(b), we see that, even when flash memory SD card is used as *SLA* log device, *SQLite/SSL* can outperform vanilla SQLite by up to 6 times. Overall, the performance results presented in Figure 3 confirm two main points: 1) the logical logging approach itself, without the help of NVM device, can give significant performance improvement to SQLite-based mobile applications (Figure 3(b)), and 2) *SQLite/SSL* can realize its full potential when PCM is used as its *SLA* log device (Figure 3(a)).

In addition to Figure 3, Table 3 drills down the I/O activities further for the traces. We separated the number of the page writes requested by the SQLite and the file system. As expected, *SQLite/SSL* wrote a far smaller number of data pages to flash memory than vanilla SQLite and *SQLite/PPL*.

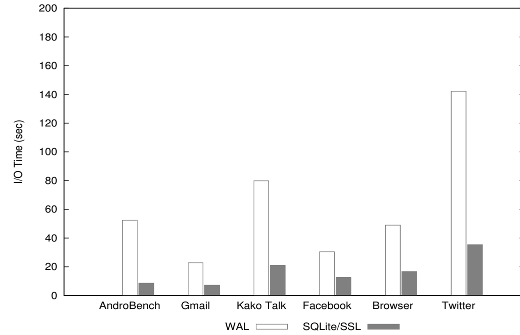
Let us first discuss the performance results in Figure 3(a). The considerable performance gain of *SQLite/SSL* in Figure 3 is direct reflection of reductions in the number of write operations against flash storage by *SQLite/SSL* (the fourth row in Table 3, `SSL-PCM (UMS)`). *SQLite/SSL* delays the page write until *SLA* becomes full. Therefore, multitude of updates against same pages by consecutive transactions will be buffered in `DRAM` cache and each page will be written only once at checkpoint. On the other hand, in the case of vanilla SQLite, because it repetitively force-writes all the pages updated by consecutively committing transactions, the number of writes (*i.e.*, the second row in Table 3) was much higher than that done by *SQLite/SSL*. Finally, in the case of *SQLite/PPL*, multitude of updates against same pages can be collected as physio-logical logs and saved in PCM log sector and later the logs are merged into the corresponding data pages [26]. However, the amount of physio-logical log in *SQLite/PPL* is quite larger than that of compact SQL statement log in *SQLite/SSL*. This implies that, when the same size of PCM was used as log device, more frequent checkpoints are required in *SQLite/PPL* than in *SQLite/SSL*. Recall that one update SQL statement will update several pages of table and its secondary indexes. For this reason, the number of page writes in *SQLite/PPL* (the third row in Table 3) is larger than that in *SQLite/SSL* (the fourth row in Table 3) by up to several times. The least performance gain was observed in the `Browser` trace. This is because, as shown in the last column of Table 2, the average length of SQL statement logs per transaction in the trace is relatively quite large, which implies frequent checkpoints.

Let us then analyze the performance results of vanilla SQLite and *SQLite/SSL* in Figure 3(b), which were obtained by running the six traces on a commodity PC with SD card as the *SLA* log device. As expected, for each trace, the IO time taken by vanilla SQLite is almost same to that in the UMS board. Also, it is not surprising to see that the IO time taken by *SQLite/SSL* is quite higher than that in the UMS because the log device was changed from byte-addressable and fast PCM to slow SD card with block interface. But the main point to note from Figure 3(b) is that the performance gap between vanilla SQLite and *SQLite/SSL* is substantial even when traditional SD card with block interface is used as the *SLA* log device. This considerable performance difference between two schemes can be explained as follows. Because *SQLite/SSL* takes the logical logging approach, it will cause just one page write upon every commit. But, in the case of vanilla SQLite, as shown in the sixth column of Table 2, multiple pages should be force-written upon every commit. Thus, the number of page writes in vanilla SQLite (*i.e.*, the second row in Table 3) is quite larger than that in *SQLite/SSL* (*i.e.*, the fourth row in Table 3) consistently over all the traces used. In addition, while running six mobile workloads on top of UMS board, we also measured the average transactional latency, the frequency of checkpoints, the average time taken to complete checkpoints, and the average number of dirty pages to be flushed per checkpoint and present the results in Table 4.

Let us discuss first about the transactional latency. From the second row in Table 4, it is clear that *SQLite/SSL* significantly outperforms vanilla SQLite across all the workloads. To be more specific, the relative performance gap of the average transactional latency between *SQLite/SSL* and vanilla SQLite is almost in proportion to that of the total runtime between them in Figure 3(a). In terms of the checkpoint frequency (the third row in Table 4), vanilla SQLite triggers quite more checkpoints than *SQLite/SSL* for all workloads. The force commit policy in vanilla SQLite will fill up quickly the WAL journal area and thus causes frequent checkpoints. Even though the number of checkpoints in the last three traces including `Facebook`, `Browser`, and `Twitter` are more than tens in *SQLite/SSL* mode, most of check-



(a) UMS board (PCM as *SLA* log device)



(b) PC (SD card as *SLA* log device)

Figure 3: SQLite Performance: Vanilla SQLite (WAL mode) vs. *SQLite/PPL* vs. *SQLite/SSL*

Table 3: I/O Count (# of Physical Pages Written in Flash Storage)

Mode	AndroBench		Gmail		KakaoTalk		Facebook		Browser		Twitter	
	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS
WAL (UMS)	7,700	619	7,093	2,055	10,236	2,156	5,365	2,124	7,076	1,972	19,606	4,069
PPL (UMS)	295	271	1,832	1,033	1,300	2,242	1,261	2,095	989	4,123	494	230
SSL-PCM (UMS)	60	35	270	597	354	572	597	884	558	1,586	525	414
SSL-Flash (PC)	3,374	1,107	2,004	1,465	3,148	1,672	3,046	3,198	3,691	3,724	9,663	2,559

Table 4: Transactional Latency and Checkpoint Performance: Vanilla SQLite vs. *SQLite/SSL*

Metric	AndroBench		Gmail		KakaoTalk		Facebook		Browser		Twitter	
	WAL	SSL	WAL	SSL	WAL	SSL	WAL	SSL	WAL	SSL	WAL	SSL
Transactional latency (msec)	37.0	0.04	17.3	1.5	18.1	0.5	27.8	5.9	24.0	3.4	90.8	5.3
# of checkpoints	21	1	12	5	16	4	16	12	15	11	35	17
Checkpoint latency (msec)	253.4	250.0	103.6	456.3	220.1	614.2	132.0	572.4	189.9	653.9	71.2	652.6
# of dirty pages / checkpoint	21.5	37	73.3	114.5	51.4	105.7	34.8	41.1	62.4	80.2	16.6	25.3

points are triggered when closing the database files, not during the normal operation. Please recall from the second column in Table 2 that those traces have several or more than tens database files. Meanwhile, in terms of average checkpoint latency (the fourth row in Table 4), *SQLite/SSL* takes longer than vanilla SQLite. The reason is quite obvious because, upon checkpoint, *SQLite/SSL* should double-write more pages whereas vanilla SQLite only needs to write fewer pages once from the WAL journal to the original database. However, across all workloads we tested, the average checkpoint time in *SQLite/SSL* is less than 1 second, which we believe is acceptable in mobile applications. As you see from the fifth row in Table 4, the average number of dirty pages to be written per checkpoint in *SQLite/SSL* is only 2x or less than that in vanilla SQLite, which is mainly due to the update locality in the mobile workloads. As will be discussed later, the checkpoint latency in both *SQLite/SSL* and vanilla SQLite will become intolerably high (*e.g.*, longer than 7 seconds) when workloads show no update locality.

5.3.2 Worst-case Performance

Until now, we have shown that *SQLite/SSL* can significantly outperform vanilla SQLite in real mobile workloads with strong update locality. In order to illustrate the limitations of *SQLite/SSL*, we carried out another experiment with a synthetic workload having no update locality. For this experiment, we created a `partsupply` table of the TPC-H benchmark using the dbgen tool, containing 60,000 tuples of 220 bytes each. In addition, we created two types of transactions accessing this table, `Random-A` and `Random-B`. In Random-A, each of 10,000 transactions autocommits after updating only one tuple of the table and consecutive transactions will access the data pages with no locality. Meanwhile, in `Random-B`, each of five transactions commits in batch mode after executing 2,000 update statements in sequence, each of which updates only one tuple of `partsupply` table with no locality. The reason why we chose 2,000 tuples as the unit of batch commit in Random-B workload is that each transaction is long enough to trigger a checkpoint upon every commit in either mode. We believe that Random-B is the worst-case workload from the perspective of *SQLite/SSL*.

For both random workloads, we measured the same set of update-related metrics presented in Table 1 and the result is presented in Table 5. From the table, we can observe that that both workloads show no update locality since the values of the last column are almost one, unlike the corresponding values of the real workloads shown in Table 1.

While running both *SQLite/SSL* and vanilla SQLite using those two random workloads, we also measured the same set of key performance metrics listed in Table 4. The result is presented in Table 6. In case of Random-A workload consisting of short transactions with no locality, *SQLite/SSL* still far outperforms vanilla SQLite in terms of both throughput

Table 5: Analysis of Update Patterns in Random Workloads: `Random-A` and `Random-B`

Trace	Total # of page writes (A)	Total # of distinct pages updated (B)	Total # of update TX (C)	Page writes / TX (D = A/C)	Avg. overwrites / page (E = A/B)
Random-A	11,092	9,953	10,000	1.1	1.1
Random-B	7,026	7,026	5	1405.2	1.0

Table 6: SQLite Performance for Random Workloads (Vanilla SQLite / *SQLite/SSL*)

Trace	I/O Time (sec)	Transaction latency (msec)	Checkpoint latency (sec)	# of checkpoints
Random-A	166.0 / 13.0	16.6 / 1.3	2.3 / 6.8	22 / 2
Random-B	45.5 / 38.7	9,092 / 7,743	7.5 / 8.0	5 / 5

and transactional latency. This is mainly because vanilla SQLite suffers from excessive WAL metadata writes and frequent `fsync` calls due to its force commit policy. In case of Random-B workload consisting of long transactions with no locality, *SQLite/SSL* and vanilla SQLite show almost the same performance in terms of transactional latency and throughput. This is because, upon every commit in either mode, almost 2,000 dirty pages have to be written to both WAL journal and original database. From this, we conclude that *SQLite/SSL* can perform no worse than vanilla SQLite even with the worst-case workload such as Random-B. With respect to the checkpoint latency, we observe that both *SQLite/SSL* and vanilla SQLite suffer from long checkpoint spikes for both random workloads. In particular, the checkpoint time of both SQLite versions can spike up to 8 seconds, which is hardly acceptable even in mobile application. But, we expect that these extreme random workloads would be rare in real mobile applications.

5.3.3 Recovery Performance

To evaluate the recovery performance of *SQLite/SSL* and vanilla SQLite, we turned off the power of the UMS board while running each of the six traces. In the case of vanilla SQLite, the UMS board was turned off when the WAL journal file was almost filled. Similarly, in the case of *SQLite/SSL*, the board was turned off when the *SLA* file of 750KB size was almost filled. Therefore, the recovery time will correspond to the worst case in each mode. For each mode, we measured the time taken to restart the SQLite database and took the average of restart times from three separate runs. Table 7 shows the average recovery times of both modes on the UMS board.

Table 7: Recovery Time (in seconds)

Trace	vanilla SQLite	*SQLite/SSL*
AndroBench	0.1	0.34
Gmail	0.1	0.22
KakaoTalk	0.1	0.8
Facebook	0.1	0.35
Browser	0.1	0.63
Twitter	0.1	0.26

The recovery time in vanilla SQLite was about 0.1 seconds (the second column in Table 7) consistently irrespective of the traces. This is because the recovery process in the mode consists of reading all the pages in the WAL file and creating the WAL index in the DRAM. In contrast, the recovery time in *SQLite/SSL* was varying depending on the traces. As explained in Section 4.4, the recovery process in *SQLite/SSL* consists of reading SQL logs from *SLA*, re-executing them in sequence, and carrying out *SSL-checkpoint* and *WAL-checkpoint*. Thus, the CPU time taken in re-executing SQL statements would not be marginal. In a separate experiment using the commodity PC with higher CPU performance, we observed the recovery time in *SQLite/SSL* was slightly reduced consistently across all the traces we tested.

Even though *SQLite/SSL* takes longer than vanilla SQLite in terms of recovery time, its recovery time was less than one second across all traces tested. Considering that the recovery time in Table 7 is the worst-case one and also the huge performance benefit for normal operations, we believe that its recovery time would be acceptable in practice.

6. CONCLUSION

In this paper, we presented the design and implementation of *SQLite/SSL*, a type of logical logging scheme, for mobile applications. For the durability of committing transactions, it force-writes only all update SQL statements from each transaction while vanilla SQLite force-writes all pages updated by each transaction in their entireties redundantly. Our main contributions are in three folds. First, we made an important observation about the characteristic of transactional workload in SQLite-based mobile applications: *short transactions with high update locality.* Second, based on this observation, we showed that the concept of logical logging is, though not new at all, a perfect fit for modern SQLite-based mobile applications. In addition, we showed how the WAL mode in vanilla SQLite can be used as transaction-consistent checkpoint mechanism. Third, we demonstrated that the logical logging can realize its full potential by using a real PCM board with DIMM interface as its log device.

Acknowledgements. This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the "SW Starlab" (IITP-2015-0-00314) supervised by the IITP(Institute for Information & communications Technology Promotion. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696). We would like to thank the anonymous PVLDB reviewers for valuable comments which helped to improve the paper.

7. REFERENCES

[1] AndroBench (SQLite Benchmark). http://www.androbench.org/wiki/AndroBench, 2017.

[2] Atomic Commit In SQLite. http://www.sqlite.org/\atomiccommit.html, 2017.

[3] Well-Known Users of SQLite. http://www.sqlite.org/\famous.html, 2017.

[4] Write-Ahead Logging. http://www.sqlite.org/\wal.html, 2017.

[5] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, 2015.

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

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, 2010.

[8] H. Chung et al. A 58nm 1.8V 1Gb PRAM with 6.4MB/s program BW. In *Proceedings of Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 500–502, 2011.

[9] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, 2013.

[10] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High Performance Database Logging Using Storage Class Memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pages 1221–1231, 2011.

[11] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Survery*, 13(2):223–242, June 1981.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, 1983.

[14] HP Enterprise. HPE 8GB NVDIMM Single Rank x4 DDR4-2133 Module. https://www.hpe.com/h20195/v2/getpdf.aspx/c04939369.pdf?ver=3, Feb 2017.

[15] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 309–320, 2013.

[16] Joel Hruska. How do SSDs work? http://www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work, Feb 2016.

[17] 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*, SIGMOD '13, pages 97–108. ACM, 2013.

[18] O. Kennedy, J. A. Ajay, G. Challen, and L. Ziarek. Pocket Data: The Need for TPC-MOBILE. In *TPCTC*, 2015.

[19] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering ApproachesRecovery. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, pages 33–45, 2014.

[20] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, 2016.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, 2009.

[22] T. Lee, D. Kim, H. Park, S. Yoo, and S. Lee. FPGA-based Prototyping Systems for Emerging Memory Technologies(Invited Paper). In *Rapid System Prototyping (RSP) '14*, pages 115–120, 2014.

[23] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar 1977.

[24] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking Main Memory OLTP Recovery. In *IEEE 30th International Conference on Data Engineering (ICDE 2014)*, pages 604–615, 2014.

[25] Numonyx. Omneo P8P 128-Mbit Parallel Phase Change Memory. Data Sheet 316144-06, Apr 2010.

[26] G. Oh, S. Kim, S.-W. Lee, and B. Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. *PVLDB*, 8(12):1454–1465, 2015.

[27] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 343–354, 2016.

[28] E. Shein. Chatbots Take Off. http://cacm.acm.org/news/207002-chatbots-take-off/, September 2016.

[29] D. Q. Tuan, S. Cheon, and Y. Won. On the IO Characteristics of the SQLite Transactions. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 214–224, 2016.