SQLite Optimization with Phase Change Memory for Mobile Applications

Gihwan Oh† Sangchul Kim‡ Sang-Won Lee† Bongki Moon‡

Sungkyunkwan University
Suwon, 440-746, Korea
{wurikiji,swlee}@skku.edu

Seoul National University
Seoul, 151-744, Korea
{stdio,bkmoon}@snu.ac.kr

ABSTRACT

Given its pervasive use in smart mobile platforms, there is a compelling need to optimize the performance of sluggish SQLite databases. Popular mobile applications such as messenger, email and social network services rely on SQLite for their data management need. Those mobile applications tend to execute relatively short transactions in the autocommit mode for transactional consistency in databases. This often has adverse effect on the flash memory storage in mobile devices because the small random updates cause high write amplification and high write latency. In order to address this problem, we propose a new optimization strategy, called *per-page logging (PPL)*, for mobile data management, and have implemented the key functions in *SQLite/PPL*. The hardware component of *SQLite/PPL* includes phase change memory (PCM) with a byte-addressable, persistent memory abstraction. By capturing an update in a physiological log record and adding it to the PCM log sector, *SQLite/PPL* can replace a multitude of successive page writes made to the same logical page with much smaller log writes done to PCM much more efficiently. We have observed that *SQLite/PPL* would potentially improve the performance of mobile applications by an order of magnitude while supporting transactional atomicity and durability.

1. INTRODUCTION

With smart mobile devices becoming increasingly ubiquitous, mobile computing and data management are growing in importance and spectrum of their applications. Gartner identified *computing everywhere* as one of the top ten strategic technology trends for 2015 [12]. As more smart objects are created and become part of the *Internet of Things*, smartphones and tablets will carry out increasingly important and diverse functions such as analyzing information, interfacing with smart appliances that can tweet or post, paying for subscription services, and ordering products.

Even in the present time, mobile phones are used for more data than voice communication. The amount of data in text, e-mail messages, streaming video, music and other services on mobile devices surpassed the amount of voice data before the turn of the century. Worldwide mobile data traffic exceeded 200 petabytes per month at the end of 2010. Since then, all things data have continued to make dramatic surges with no sign of turning back. Mobile messaging was a USD 230 billion dollar business in 2013, and it is the most profitable segment of the mobile industry today. Worldwide mobile data traffic is expected to grow higher than 5,000 petabytes per month by the end of next year [28].

Given that the two dominant mobile platforms, Android and iOS, adopt SQLite as the standard database manager [2], it is not surprising that most popular mobile messengers as well as recent versions of Skype rely on SQLite to store text messages and history items in special database files. This clearly motivates the need of understanding the database workloads created by mobile messengers and other popular applications, and optimizing the performance of sluggish SQLite databases for the workloads [15, 16, 17, 20].

Another factor to consider is that most contemporary mobile devices use flash memory (*e.g.*, eMMC cards) as storage media to store data persistently. Flash memory does not allow any data to be updated in place, and all updates are carried out by writing the new content into a clean page at another location [3, 24]. Besides, as a block device, flash memory storage devices perform all I/O operations at the granularity of pages, whose size has been steadily increased by NAND flash vendors for higher sequential throughput. A page of 8KB size is now more common for NAND flash memory chips manufactured lately. With a larger page size, write amplification will become greater for small random writes, write latency will be elongated, and the life span of flash memory will be shortened. Consequently, the upward trend in flash page sizes may have significant and negative impact on the underlying databases and hence mobile applications relying on them for data management.

Another important observation can be made, coupled with the upward trend in flash page sizes, that motivates our work presented in this paper. Mobile applications including the popular messengers tend to update SQLite databases in the *autocommit* mode. This is partly because SQLite turns autocommit mode on by default. Autocommit ensures that individual statements will conform to the ACID properties of transactions, and consequently incurs per-statement transaction overhead, having negative impact on performance or resource utilization.

The negative effect of large pages and auto or frequent commits becomes more detrimental especially when mobile

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

applications produce a burst of small pieces of data. They will then be inserted into an SQLite database as a sequence of small records, each of which will in turn be causing SQLite to request one or two page write operations (depending on the journaling mode). There will be very high likelihood that the target of those page write operations is the same logical page, and the difference between two consecutive writes requested for the same logical page is very small.

Motivated by these observations, we propose a new optimization strategy for mobile data management, which would potentially improve the performance of mobile applications by an order of magnitude. The optimization strategy we propose in this paper has both hardware and software components. The hardware component includes phase change memory (PCM), which is among leading candidates for the next generation byte-addressable non-volatile memory. The hardware prototype called *unified memory system (UMS)* is equipped with a DIMM interface and enables random accesses to individual bytes like DRAM does. The software component takes advantage of the byte-addressability and minimizes redundant page writes while supporting transactional atomicity and durability for a flurry of small insertions created by mobile applications. The software component is implemented into the SQLite database manager so that small insertions or updates can be captured by SQLite as physiological log records. The physiological log records are written persistently to log sectors in PCM, each of which is associated with a database page in flash memory chips. We call this *per-page logging (PPL)* as opposed to the system-wide *sequential* logging carried out by conventional database servers for recovery purpose.

The per-page logging scheme is a reminiscence of *in-page logging (IPL)* we proposed for all-flash database systems [22]. Although the results from the preliminary evaluation were promising and demonstrated the potential benefit of IPL, it has never been implemented for any real database server due to the lack of fine grained logging capability of flash memory. The PPL method presented in this paper takes advantage of the byte-addressability and low read latency of PCM and overcomes the limitations of IPL.

We have implemented the PPL design into SQLite on the UMS platform. The main contributions of this work are summarized as follows.

- We have designed and implemented a new mobile database manager called *SQLite/PPL* on the UMS platform. To the best of our knowledge, this is the first transactional database manager that utilizes the differential write performance of PCM – fast for small writes but slow for large writes – to optimize transaction performance. *SQLite/PPL* does not use PCM as a caching or tiering device but as a logging device associated with individual data pages.
- *SQLite/PPL* optimizes data management for mobile applications by replacing redundant page writes with fine-grained log writes, which can be performed efficiently with byte-addressable PCM without the overhead of standard I/O stack. It also minimizes write amplification in flash memory devices.
- *SQLite/PPL* can avoid the costly database journaling of the vanilla SQLite without giving up the atomicity and durability of transactions. In addition, *SQLite/PPL* simplifies the commit and abort procedures since *SQLite/PPL* prevents any uncommitted changes from being propagated to data pages in flash memory.
- *SQLite/PPL* has been evaluated empirically with real traces obtained from popular mobile applications such as messengers and email. We have observed that *SQLite/PPL* can improve the performance of mobile applications by an order of magnitude.

The rest of the paper is organized as follows. Section 2 describes the hardware characteristics of UMS and the database workloads of mobile applications. In Section 3, we present the design and added features of *SQLite/PPL* and they are used for database operations. In Section 4, we evaluate the performance impact of *SQLite/PPL* for mobile applications with real-world traces and a publicly available mobile benchmark. Section 5 reviews the recent work on PCM from the standpoint of *SQLite/PPL* and compares the per-page logging with its precursor, in-page logging. Lastly, Section 6 summarizes the contributions of this paper.

2. BACKGROUND

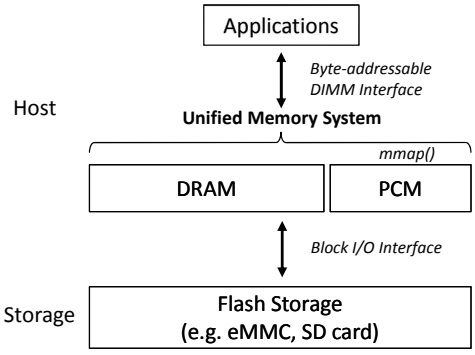


Figure 1: Unified Memory System (UMS)

2.1 PCM Unified with DRAM

Non-charge-based non-volatile memory technologies have been under active development and commercialization by leading industry manufacturers for quite some time [5, 14, 26]. Among those memory technologies, phase-change memory (PCM) is considered one of the promising candidates for the next generation byte-addressable non-volatile memory. It is a few years ago when a major semiconductor manufacturer started volume production of PCM and offered it for mobile devices in 1Gb packages [9]. Unlike DRAM and flash memory, PCM provides memory states without electric charges [14]. PCM devices use phase change material for a cell to remember a bit. The phase change material can exist in two different states, amorphous and crystalline, which can be used to represent zero and one. Switching between the two states can be done by application of heat at different temperature ranges for different durations [14].

PCM can be programmed in place without having to erase the previous state. Although PCM has a limited number of programming cycles due to repeated heat stress applied to the phase change material, it is considered to be more scalable and has greater write endurance than flash memory by a few orders of magnitude [14, 21]. Furthermore, in contrast to NAND type flash memory, PCM need not operate in

page mode and allows random accesses to individual bytes like DRAM does.

It is reported in the literature that the read and write latency of PCM are only an order of magnitude greater than those of DRAM [21]. As is shown in Table 1, however, contemporary PCM products do not deliver the promised performance as yet particularly for write operations. While PCM takes only about 408*ns* to read 4 bytes, it takes as long as 7.5 *us* to write 4 bytes [5, 25]. A similar disparity in read and write speeds has been observed in other PCM products as well [26]. A recent study confirms that some of the reported performance measurements are misleading especially for write latency [18]. The study also reports that while PCM read latency is about 16 times shorter than that of flash memory, PCM write latency is actually 3.5 times *longer* than that of flash memory.

Media	Access time		
	Read	Write	Erase
NAND Flash†	156 μs (4KB)	505 μs (4KB)	1.5 ms (512KB)
PCM‡	408 ns (4B)	7.5 μs (4B)	N/A

†Samsung K9F8G08U0M 16Gbits SLC NAND [29];
‡Samsung 58nm 1Gb PCM [5, 25]

Table 1: Access Speed: NAND Flash vs. PCM

The implication that can be derived from the current characteristics of PCM and flash memory is quite obvious. PCM is inferior to flash memory for page writes, while the opposite is true for fine-grained writes as well as page reads. Therefore, we argue that *PCM should be used as a byte-addressable memory rather than a block device.* We also argue that *caching or tiering is not the best use of PCM*, because either strategy would utilize PCM much like a block device between DRAM and flash memory SSDs or disk drives.

Figure 1 depicts a prototype development board that allows PCM to be accessed via DIMM interface alongside DRAM. With this board, an application can write a small amount of data (*e.g.*, a log record), much smaller than a page, persistently to PCM through the DIMM interface [25]. Since the write time of PCM is approximately proportional to the amount of data to transfer, force-writing a small record can be done very efficiently. Furthermore, it can avoid I/O stack overhead by writing data into PCM through the DIMM interface. The overheads in the IO stack can exceed the hardware access time for solid-state technologies, because it takes about 20,000 instructions to issue and complete a 4KB IO request under standard Linux [4].

In this paper, we call the prototype a unified memory system (UMS), as both DRAM and PCM are accessed in the same way through the DIMM interface. We are interested in the potential of PCM that would make updates durable in the database while avoiding the overhead of I/O stack as much as possible. This approach will be effective particularly when the amount of difference between successive changes is small.

2.2 Database Workloads for Mobile Apps

As is mentioned in Section 1, numerous mobile applications including popular messengers, Gmail, Facebook and Twitter rely on SQLite for data management. Those applications maintain database tables to store text and email messages sent to and from mobile devices over the Internet. Most of the messages are very small (rarely more than a hundred bytes). Once they are stored in the devices, they are seldom deleted or updated. Thus, the database workload from the mobile applications are mostly *small insertions*.

Individual tables in the database are associated with a surrogate key field, whose values are created automatically by SQLite. The surrogate key field identifies rows in the table uniquely and serves as the primary key. SQLite stores table records of key-value pairs (with surrogate keys being the keys) in the leaf nodes of a B^+-tree. One or more secondary indexes are commonly created for a table, which are also structured as a B^+-tree. Each entry of a secondary index consists of a key and a pointer, and the size of an entry is very small and mostly no larger than 30 bytes.

Write Amplification

Since many mobile applications utilize SQLite running in the autocommit mode, each record inserted into a table requires force-writing multiple pages including the one in the table and several pages in the secondary indexes. SQLite provides users with six different options for database journaling [2]. Among those, `Delete` and `WAL` journaling are most commonly used. If SQLite runs in the `Delete` (by default) or `WAL` journaling mode, each update incurs two physical page writes, one for database update and another for journaling [16].[1] That is, if a record is inserted into a table with k secondary indexes, a total of $2 \times (k + 1)$ pages will be written physically to storage.

The amount of write amplification caused by an individual insertion would be quite significant, particularly when the inserted record is very small. In our own traces, we have observed that a single message sent or received often incurs more than ten physical page writes. More importantly, the difference in content between two consecutive writes requested against the same logical page is very small, rarely more than 100 bytes. (See Figure 2 for the distribution of page differences of the traces from mobile applications.) The write amplification of the traces (measured by the amount of data written physically divided by the aggregate sum of messages in bytes) was more than 100. The negative impact will be aggravated as the page size increases in eMMC and SD cards commonly used for mobile devices. The same trend was also observed in the mobile benchmark.

Locality of Insertions

When a new record is inserted into a database table, a surrogate key automatically created by SQLite is stored together as part of the record. Since the surrogate keys are increasing monotonically, new records are inserted into the rightmost leaf node of the table (organized as a B^+-tree) in the append-only fashion. Therefore, spatial locality is very strong when successive insertions are made to a database table, because the records are always appended to the *same* leaf node of the table until the node becomes full.

[1] The `WAL` journaling of SQLite is different from the ARIES style write-ahead logging in that the former is a page level journaling while the latter is a physiological undo-redo update logging.

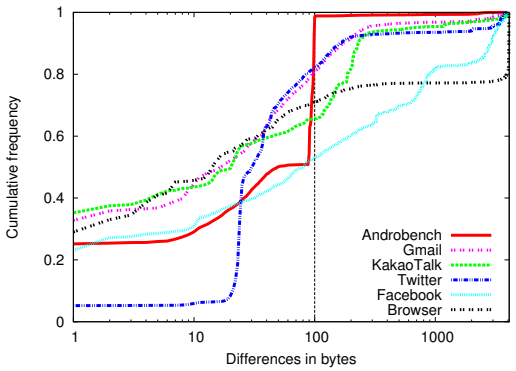


Figure 2: Differences between page writes

In the case of a secondary index, spatial locality may or may not exist depending on the attribute which the secondary index is created for. For example, if a messenger user exchanges messages for a while with someone or a group of friends on a certain topic, then the secondary indexes created for the `person` and `topic` fields will receive a sequence of index entries to insert with the same index keys. Thus, spatial locality will be strongly present in the secondary indexes while the user exchanges messages. But, of course, it may not be the case for other secondary indexes.

We analyzed one of our traces to demonstrate the locality of insertions visually in the address-time space. Figure 3 shows only a small segment of the trace for the clarity of presentation. It clearly shows that the same logical pages were overwritten many times consecutively, and the trend was prevalent in both tables and indexes.

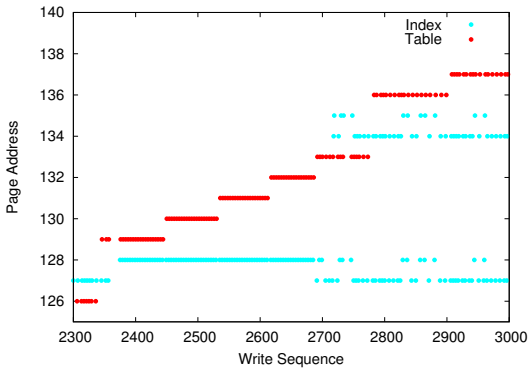


Figure 3: Locality in writes (KakaoTalk)

3. DESIGN OF *SQLite/PPL*

We have designed and implemented a new mobile database manager called *SQLite/PPL* on the UMS platform. *SQLite/PPL* is a variant of SQLite augmented with several new features for the realization of the *per-page logging* strategy. This section presents the overall architecture of *SQLite/PPL*, shown in Figure 4, and provide the detailed description of modules and data structures (in gray color) added to the vanilla SQLite.

3.1 Design Overview

SQLite is a software library that implements a serverless, transactional SQL database engine [2]. Mobile applications are linked to the SQLite library to utilize its database management functions. SQLite stores databases as separate files in flash memory. Concurrent transactions can access the same database but only one transaction can update the database at any moment in time, because the entire database file is locked by the writer transaction until it finishes. The *pager* module is in charge of managing the DRAM buffer pool and the B^+*-tree* module processes select and update statements. These components of SQLite remain unchanged in the *SQLite/PPL*.

SQLite/PPL differs from the vanilla SQLite in the way transactional atomicity and durability are supported. SQLite relies on the costly journaling, usually in either `Delete` or `WAL` mode, to ensure that both the before and after images of an updated page are saved until the updating transaction commits or aborts. Journaling is also commonly used for a multi-page write, which is not guaranteed to be atomic on most storage media, disk or flash memory. This is the major cause of sluggish performance of SQLite, since every page update requires two physical page writes [16].

On the other hand, *SQLite/PPL* relies on per-page logging instead of journaling as well as the non-volatility of PCM for the transactional support. Instead of writing a page twice – in the database and its journal – for each update, *SQLite/PPL* captures the change in a physiological log record and adds it to the PCM sector of the page being updated. The page itself remains unchanged in the flash memory until its log records are merged to it later. Therefore, *SQLite/PPL* can replace a multitude of successive page writes against the same logical page by writing potentially much smaller log records into the PCM sector much more quickly. Furthermore, *SQLite/PPL* can avoid redundant page writes by turning off journaling without giving up the atomicity and durability of transactions.

However, leaving log records in the PCM sector of each page alone is not sufficient for transactional atomicity and durability. The log records in the PCM sector represent changes made by a transaction whose state changes over time from `active` to either `committed` or `aborted`. When a transaction commits, all of its log records become permanent and can be merged (or applied) to the data page in flash memory immediately or lazily. When a transaction aborts, all the log records become invalidated and can be discarded immediately or simply ignored. While a transaction is still active, all of its log records must be kept in the PCM sector at least until the transaction commits or aborts. That way it will be feasible to determine the visibility of the log records (or their changes) to other concurrent transactions depending on their *end mark*.[2]

In order to keep track of transaction states, *SQLite/PPL* maintains a single *transaction log*, which is shared by all transactions accessing the same database. The transaction log stores only three types of records: `transaction_begin`, `commit`, or `abort`. Since the transaction log stores no update

[2] The end mark is the location of the last valid commit record in the `WAL` journal. It is set on the first read operation of a transaction and does not change for the duration of the transaction. Thus, it ensures that a read transaction only sees the database content (*i.e.*, a snapshot) as it existed at a single point in time [2].

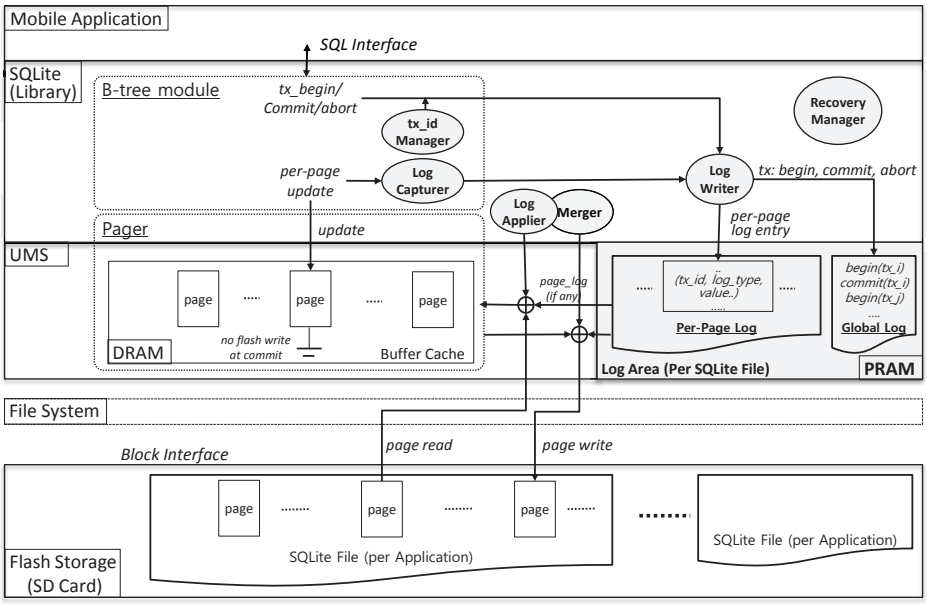


Figure 4: *SQLite/PPL* on UMS: Architecture

log records, it is expected to be kept small and subject to very light I/O activities.

The key data structures we have added to *SQLite/PPL*, namely, *per-page log* and *transaction log* are stored in the PCM area of UMS, while data pages are stored in flash memory. Thus, data pages are accessed via the standard I/O stack, but the per-page log and transaction log are accessed by `mmap` system calls through the DIMM interface, hence avoiding the overhead of I/O stack.

3.2 Added Functions

SQLite/PPL is augmented with six functional modules (`tid manager`, `log capturer`, `log writer`, `log merger`, `log roller` and `recovery manager`) to manage the data structures and provide transaction support. The overall architecture and the functional modules are illustrated in Figure 4. The functional modules be described below in more detail.

Tid manager The vanilla SQLite does not generate transaction identifiers. It relies on database journaling for transactional atomicity and durability without using transaction ids. This is possible because at most one writer transaction is allowed to access the same database at any moment in time, and a reader transaction can always avoid uncommitted changes (*e.g.*, by checking the end mark to determine its snapshot of database in the case of `WAL` journaling).

SQLite/PPL also allows at most one writer transaction to access the same database. However, *SQLite/PPL* scatters update log records across multiple log sectors in PCM. Moreover, a log sector may store update log records created by different (not concurrent) transactions simultaneously, because committed updates (or their log records) may not be merged to their corresponding data pages immediately. This warrants the use of transaction ids in order to distinguish committed log records from uncommitted ones. The `tid manager` generates a new tid for each writer transaction. A reader transaction is not assigned a tid, but instead uses the tid of the most recently committed writer transaction to determine its snapshot of database. All transactions are considered a reader until they make the first write request.

Log capturer The vanilla SQLite does not generate update log records. This is because, again, it relies on database journaling for transaction support, which is carried out by making dual page writes for each logical page update.

SQLite/PPL has the `log capturer` inside the B$^+$-tree module of SQLite. The B$^+$-tree module processes an update SQL statement by realizing the change in a database table (stored as a B$^+$-tree) as well as zero or more secondary B$^+$-tree indexes. All those page-level changes are captured in physiological log records (*e.g.*, a new entry inserted to a leaf node) and passed to the `log writer`. Since the B$^+$-tree nodes of the table and secondary indexes adopt the slotted page organization, the log records are created based on the slot number of an entry and the before and after images of the entry. The transaction id of the writer transaction is included in all the update log records of the transaction.

TxID	Log Type	Length	Offset	CRC	After Image

Figure 5: Log Format in *SQLite/PPL*

The format of a log record is shown in Figure 5. **TxID** stores the id of a writer transaction. **LogType** stores the type of an operation, that is, one of `insert`, `delete` or `update` in a *per-page log* record, or one of `begin_transaction`, `commit` and `abort` in a transaction log record. **Offset** and **length** point to the location in the page where a change is made. Note that the page id is not necessary because an update log record always belongs to a certain per-page log sector, which is associated with a data page. **After-image** stores a new value and is used for redoing a change. The log records of *SQLite/PPL* do not contain **Before-image**. This is because *SQLite/PPL* never propagates uncommitted changes to database and hence no undo recovery is necessary. Note also that the LSN (log sequence number) field is

not necessary, because *SQLite/PPL* disallows two or more concurrent writer transactions to access the same database.

We add the **CRC** field to the log format to ensure the atomicity of writing a long log record. The current prototype of UMS guarantees the atomicity of a PCM write no larger than 32 bytes. The four byte CRC field is used to check the validity of a log record longer than 32 bytes.

Log writer This module is responsible for writing *per-page* log records persistently in PCM. Per-page log sectors (4KB each by default in the current implementation) are created on an on-demand basis. A log sector is assigned to a data page, when the data page gets updated for the first time. Then, the current and all subsequent log records belonging to the data page are written sequentially to the log sector until it becomes full. When it does, overflow log records are written to a common pool of log sectors instead of adding another per-page log sector. We have made this design choice to better utilize the PCM storage. Note that the data page remains unchanged until the update log records are merged to it.

The `log writer` is also responsible for writing *transaction* log records persistently. Since a transaction log is shared by all transactions accessing the same database, a single chunk of PCM is allocated separately for each database.

Log merger A log merge event is triggered when a transaction commits. However, update log records are applied to the corresponding data pages only if the log sector the log records belong to is full. After all the update log records in a full log sector are applied, the log sector is emptied and released back to the pool of free log sectors. The log records in a non-full log sector are not merged but kept in the log sector until they are merged by a subsequent merge event. This is why update log records created by different writer transactions may exist in the same log sector simultaneously.

An alternative approach is to apply all committed updates eagerly at once. This approach, however, may increase the volume of page writes to flash memory and lower transaction throughput. The current implementation of *SQLite/PPL* adopts a lazy merge policy to maximize the utilization of PCM and minimize the aggregate volume of page writes made to flash memory.

It is rare for mobile applications executing relatively short transactions but a log merge event can also be triggered when *SQLite/PPL* runs out of free per-page log sectors in the PCM storage. In such a rare event, a victim log sector will be chosen and released back to the pool of free sectors after merging all the log records. The current implementation of *SQLite/PPL* selects as a victim a log sector that stored a new update log record least recently.

This merge procedure guarantees that only committed changes are merged to data pages because *SQLite/PPL* allows only one writer transaction at any moment in time. Consequently, uncommitted changes can always be rolled back because they are not merged until committed. Note that a log merge event has no effect on the data page frames in the DRAM buffer pool, as they are always up to date.

Log roller A data page is fetched from flash memory on a page fault. There is nothing further to be done if the log sector of the data page is empty or there is no committed log record. If it is not empty, all the committed changes must be applied or rolled forward to compute an up-to-date copy of the data page. The `log roller` determines whether an update log record is committed or not by referencing the transaction log. Fetching update log records from PCM to roll them forward can be done efficiently, as the read latency of PCM is at least an order of magnitude lower than that of flash memory.

Recovery manager *SQLite/PPL* can detect an abnormal termination by checking the transaction log. If the last log record is `begin_transaction`, *SQLite/PPL* must have crashed before restart. When it restarts, the `recovery manager` extracts the transaction id from the `begin_transaction` record and removes all the uncommitted update log record created by the transaction. This will be sufficient to roll back all uncommitted changes, because *SQLite/PPL* executes at most one writer transaction at any time and it never merges uncommitted changes to the data pages in flash memory.

3.3 Database Operations in *SQLite/PPL*

With the added functions described in Section 3.2, *SQLite/PPL* performs basic database operations such as `read`, `write`, `commit` and `abort` differently from the vanilla SQLite. This section describes how those operations are performed by *SQLite/PPL* using the added modules.

3.3.1 Write

The B$^+$-tree module of SQLite processes an update SQL statement by inserting a new entry to or, deleting or updating an existing one from the leaf nodes of a table and its secondary indexes. While the vanilla SQLite relies on the *steal* and *force* buffer management policies to propagate the changes to database, *SQLite/PPL* never lets dirty pages flushed to database before commit time. Instead, *SQLite/PPL* lets the `log capturer` produce physiological log records of the changes and passes them to the `log writer` so that they can be written to the per-page log sectors in PCM immediately and persistently. This ensures that update log records are written to per-page log sectors in the temporal order of their events.

In effect, *SQLite/PPL* abolishes the database journaling and the steal/force buffer management of the vanilla SQLite, and adopts write-ahead logging *per-page* with the *no-steal/no-force* buffer management policy. Note that this buffer management policy of *SQLite/PPL* imposes no additional constraint on the buffer pool management, because dirty pages can still be evicted from the buffer pool and simply discarded when they are evicted. Even commit procedure does not have to flush dirty pages in the buffer pool, because those changes must have been written persistently in the per-page log sectors already.

3.3.2 Read

On a page hit, a read operation can just return the page frame found in the buffer pool, because the page in the buffer pool is always up to date. On a page fault, a read operation needs to fetch a data page from flash memory, which may or may not have a per-page log sector associated with it. If not, the page fetched from flash memory will be returned with no further action. If there is a log sector associated with the data page, then the `log roller` applies all the committed changes in the log sector to the data page before it is returned. The `log roller` can determine which log record is committed by referencing the transaction log, which stores the `transaction_begin`, `commit`, and `abort` log records of

transactions. This guarantees that all transactions in *SQLite/PPL* run in the same level of *serializable* isolation as the transactions running in the vanilla SQLite [2].

3.3.3 Commit

Unlike the vanilla SQLite with the force policy, *SQLite/PPL* adopts a very simple procedure for a commit operation. At commit time, the vanilla SQLite flushes all dirty pages from the buffer pool to either the database and rollback journal files (in the `Delete` journal mode) or the `WAL` journal file (in the `WAL` journal mode). The commit time overhead is significant, 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.

In contrast, *SQLite/PPL* does just one thing, adding a `commit` record to the transaction log persistently and atomically, which is guaranteed by the PCM of UMS. This simple procedure is sufficient, because all the updates made by the committing transaction have been written to the per-page log sectors already and they do not need to be redone.

3.3.4 Abort

When a transaction aborts, SQLite drops the dirty pages updated by the transaction from the buffer pool. The same is done by *SQLite/PPL* too. Since SQLite adopts the steal buffer management policy, some of the changes made by the aborting transaction may have been written to database or journal files. Those changes are rolled back by copying the before images from the `Delete` journal to database or by dropping the after images stored in the `WAL` journal.

SQLite/PPL keeps all uncommitted changes as log records in the PCM area, and thus rolling back the changes can be done much more efficiently. All that is necessary is to drop the uncommitted log records from the per-page log sectors, and this can be done either eagerly or lazily. The uncommitted log records can be removed *eagerly* from the log sectors at abort time. Alternatively, they can be left in the log sectors until they are removed *lazily* by a subsequent log merge event. The current implementation of *SQLite/PPL* adopt the eager method for better utilization of PCM storage. Just one more thing to do is adding an `abort` record to the transaction log.

3.4 Recovery

The vanilla SQLite has its own ways to detect a failure when it restarts. When it runs in the `Delete` journal mode, the existence of a hot journal indicates that SQLite crashed previously before restart. A hot journal is the one that needs to be rolled back in order to restore the consistency of its database. When vanilla SQLite runs in the `WAL` journal mode, the absence of a commit record in the `WAL` journal indicates a crash.

SQLite/PPL detects a failure much the similar way to the vanilla SQLite running in the `WAL` journal mode. If the last log record in the transaction log is not either `commit` or `abort`, then *SQLite/PPL* must have crashed previously before restart. Since at most one writer transaction runs against the same database at any time, recovering from a crash is not different from aborting a transaction at all. All the uncommitted update log records are removed from the per-page log sectors and an `abort` log record is added to the transaction log.

4. EVALUATION

In this section, we present the results of empirical evaluation of *SQLite/PPL* and analyze the impact on the performance of mobile applications. For the evaluation, we collected real traces from five popular mobile applications, all of which uses SQLite for data management, in addition to a publicly available mobile benchmark program. We tested the workloads with the *SQLite/PPL* running on the UMS board. For comparison, we also tested the same workloads with the vanilla SQLite on the UMS board in the `Delete` and `WAL` journal modes.

4.1 Experimental setup

All the experiments were conducted with the UMS development board described in Figure 1. The UMS board 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 3.8.4.1, and the size of database page is set to 4KB to match the page size of the underlying file system. The current implementation of *SQLite/PPL* is based on the same version of SQLite. To evaluate the effect of flash storage performance, we used two different types of flash SD cards denoted by `SD Card1` and `SD Card2`. Table 2 compares the random and sequential performance of the SD cards. Table 2 also includes the performance measurements of our PCM for comparison.

Storage Media	Random IOPS (4KB Page)		Sequential IO (MB/sec)	
	Read	Write	Read	Write
SD Card1†	2158.3	516.3	45.5	5.1
SD Card2‡	2528.3	410.3	23.4	12.0
PCM	6250.1	1219.6	25.0	4.9

†SDSDB-008G, ‡MB-MSBGA

Table 2: IO Performance of Flash SD Cards

4.2 Workloads from Mobile Applications

We used real traces from five representative mobile applications: `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. We modified the source code of SQLite to capture all the transactions and their SQL statements. We also added a publicly available benchmark called `AndroBench`, which aims at measuring the storage performance of Android devices. To provide better insight into the observations, the characteristics of the traces and benchmark are described below from the database processing standpoint.

KakaoTalk is the dominant mobile messenger application in Korea. It is functionally similar to other messengers such as Whatsapp, Viber and iMessenger. SQLite stores text messages in `kakaotalk.db`. For pictures, SQLite stores the compressed thumbnails of pictures as a blob and the file paths of pictures. The pictures themselves are stored in a separate folder. SQLite runs in the autocommit mode when the mobile device is

Trace	Total # of TXs	Total # of batch TXs	SQLs / batch TX	% of join queries	% of update queries	page writes / TX	Size of a log (bytes)	# of logs per TX	DB size (MB)
KakaoTalk	4,342	432	10.55	0.1	75.6	1.80	383.66	2.68	0.45
Twitter	2,022	17	26.35	1.2	76.3	1.53	424.41	1.25	6.08
Browser	1,522	1,493	2.99	17.0	58.4	3.44	159.86	5.00	2.51
Facebook	1,281	262	7.87	0.6	65.1	2.90	274.89	4.44	1.95
Gmail	984	806	12.93	8.9	68.3	6.14	171.26	30.03	0.74
AndroBench	3,081	2	1.5	0	100	1.38	470.66	0.98	0.19

Table 3: Analysis of Mobile Application Traces

connected to the network. When the device is disconnected from the network, messages are stored in the server and sent to the device later in a batch (enclosed by `begin_transaction` and `commit`/`abort`) when it is reconnected to the network.

Gmail includes common operations such as saving new messages in the inbox, reading from and searching for keywords in the inbox. The Gmail application 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. The read-write ratio was about 3 to 7 with more writes than reads. In the Gmail trace, most of the SQLite transactions are processed in the batch mode.

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, and most of the SQLite transactions process text messages in the autocommit mode except when the mobile device is disconnected from the network.

Facebook was obtained from a Facebook application that reads news feed, sends messages and uploads photo files. A total of 11 files were created by the Facebook application, but `fb.db` was accessed most frequently by many SQL statements. Similarly to Gmail, this trace includes a large number of insert statements, because Facebook uses SQLite to store most of the information on the screen in a database. One thing that distinguishes it from the Gmail trace is that Facebook stores many small thumbnail images in a SQLite database as blobs and the number of updates per transaction tends to be high. The read-write ratio was, like in the case of the Gmail trace, about 3 to 7.

Browser was obtained while the Android web browser read online newspapers, surfed several portal sites 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. Since the URLs of all visited web pages are stored, the history table receives many update statements. In addition, cookie data are frequently inserted and deleted when web pages are accessed. Thus, the cookie table also received a large number of update statements. Among the six database files the browser creates, `browser2.db` was the dominant target of most SQL statements as the main database file. Another interesting thing about this trace is that it includes quite a large number of join queries. The read-write ratio was about 4 to 6.

AndroBench is a write-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].

Table 3 summarizes the characteristics of the traces. The second, third and fourth columns of the table show the distribution of transactions and SQL statements executed in the batch mode (enclosed by `begin_transaction` and `commit`/`abort`) and autocommit mode. The fifth and sixth columns show the query distribution by type. The seventh column shows the average number of logical page writes requested by a committing transaction. The eighth and ninth columns show the average length of a log record and the average number of log records per transaction, respectively. Lastly, the tenth column shows the size of a database created by each trace. It is noteworthy that the Gmail trace contains more batch transactions, a considerably higher number of SQL statements per transaction, and twice more pages written per transaction than the other traces. This is because multiple email messages are downloaded together when the application is started, and each downloaded message requires updating more than one tables and several secondary indexes for labels.

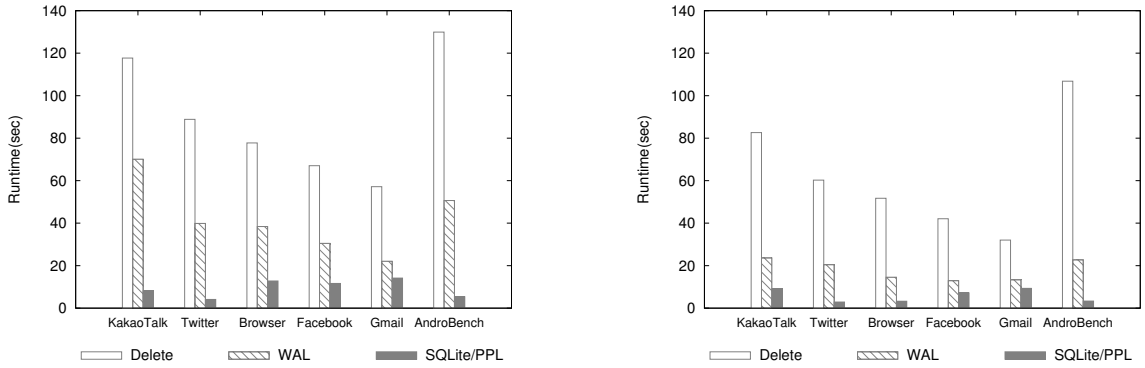
4.3 Performance Analysis

We measured the performance of *SQLite/PPL* running on the UMS by replaying the traces and by running the benchmark program. To analyze it comparatively with the vanilla SQLite, we also measured the performance of vanilla SQLite running on the UMS in the `Delete` and `WAL` modes.

4.3.1 Baseline Performance

Figure 6 shows the elapsed times taken by vanilla SQLite and *SQLite/PPL* to process each workload completely. As is shown clearly in the figure, *SQLite/PPL* processed transactions much faster than the vanilla SQLite running in `WAL` and `Delete` journal modes, by up to 8.25 and 16.54 times, respectively, for the five traces. The speed gap was wider for the mobile benchmark, namely, 9.47 times and 24.27 times faster than the vanilla SQLite running in `WAL` and `Delete` journal modes, respectively.

The considerable gain in performance was direct reflection of reductions in the number of write operations summarized in Table 4. *SQLite/PPL* performs a page write only when



(a) SDcard1 (b) SDcard2

Figure 6: SQLite Performance: `Delete` vs. `WAL` vs. *SQLite/PPL*

Mode	KakaoTalk		Twitter		Browser		Facebook		Gmail		AndroBench	
	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS	SQLite	OS
Delete	15,411	44,995	6,120	40,381	9,171	28,178	6,941	23,719	11,528	13,287	20,776	32,088
WAL	9,644	10,119	3,424	12,447	6,824	9,973	4,569	9,812	8,249	3,447	4,846	12,020
SQLite/PPL	1,300	2,242	494	230	989	4,123	1,261	2,095	1,832	1,033	295	271

Table 4: I/O Count (# of Physical Pages Written in Flash Storage)

the update log records are merged to it. Since a multitude of updates can be collected in the PCM log sector and merged to the data page at once, the number of page writes can be reduced considerably. On the other hand, the number of writes performed by the vanilla SQLite was much higher than that done by *SQLite/PPL*, because, with the force policy, the vanilla SQLite force-writes all the updated pages when a transaction commits.

The least performance gain was observed in the Gmail trace. This is because, as is shown in Table 3, the Gmail trace generated the most intensive logging activities in all aspects including the total volume of log records and the average volume of log records per transaction. This led to the increased level of PCM log write activities, frequent log merge events, and hence the highest number of page writes to flash memory per transaction, which becomes dominant in processing time.

The force-write operations by the vanilla SQLite at commit time increase the number of page writes not only in the database files but also in the journal files. This increases the frequency of file metadata updates and hence the number of page writes carried out by the file system.

The disparity in performance of the vanilla SQLite running in the two journal modes has been reported in the literature [16]. The vanilla SQLite was slower when it ran in the `Delete` mode, because it had to create and delete a rollback journal file for each transaction, update the file metadata more frequently, perform write barriers more frequently, and write pages more randomly.

We evaluated the effect of flash storage performance by running the same traces on two different types of flash SD cards, `SD Card1` (shown in Figure 6(a)) and `SD Card2` (shown in Figure 6(b)). As is shown in Table 2, the two types of SD cards exhibit somewhat different random-to-sequential performance ratios. This affected the performance measurements for both the vanilla SQLite and *SQLite/PPL*, but the effect was not substantial.

Table 4 presents a drill-down analysis of the I/O activities for the traces. For each trace, we counted separately the number of page writes requested by the vanilla SQLite or *SQLite/PPL*, and the number of metadata page writes requested by the file system. As expected, *SQLite/PPL* wrote a far smaller number of data pages to flash memory than the vanilla SQLite. The reduced number of pages written to database files also reduced significantly the number of metadata page writes carried out by the file system.

One notable point in Table 4 is that the number of metadata page writes was a few times larger than the number of data page writes requested by the vanilla SQLite in all cases except for the Gmail trace in the `WAL` journal mode. While no more than four data pages are updated by each transaction in all traces except Gmail, the file system still need to write several metadata pages to maintain the consistency against the changes made to database files. Even with *SQLite/PPL*, more metadata pages were written than data pages in three traces, KakaoTalk, Facebook and Web, but *SQLite/PPL* reduced the absolute number of metadata page writes considerably.

Trace	SQLite `Delete`	SQLite `WAL`	*SQLite/-PPL*
KakaoTalk	41.2	13.1	7.5
Twitter	11.5	3.9	0.3
Browser	19.6	6.5	1.8
Facebook	13.3	6.1	2.6
Gmail	3.7	1.8	1.6
AndroBench	66.5	16.3	4.0

Table 5: Average Latency (in millisec)

Some users of mobile applications may be concerned with the latency of an individual operation as much as the over-

all throughput. We measured the processing time of individual queries executed by the vanilla SQLite and *SQLite/PPL* with `SD Card2`. Table 5 summaries the average time taken to process individual queries of each trace. The difference in the average latencies was proportional to that in the total processing times of each trace. Note that all the tests in our experiments were done by a single thread.

4.3.2 Effect of Log Sector Size

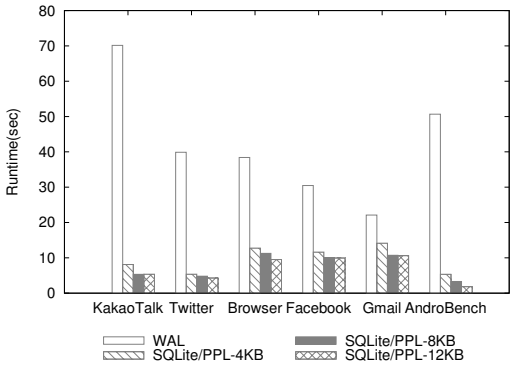


Figure 7: Effect of Log Sector Size

As is described in Section 3.2, a log merge event is triggered by a committing transaction but not all log sectors are subject to a log merge operation. A log sector is merged only if it contains a committed update and is full. Thus, with respect to write optimization, it will be beneficial to make a log sector larger and delay merge operations, as a merge operation costs a page write. However, a read operation may take longer to roll forward more log records to create an up-to-date data page.

To evaluate the effect of a log sector size, we measured the elapsed time of *SQLite/PPL* with different log sector sizes, 4KB, 8KB and 12KB. We did not include the `Delete` journal mode case of vanilla SQLite for the clarity of presentation in Figure 7. A clear trend was that the performance of *SQLite/PPL* improved as the size of a log sector increased from 4KB to 12KB consistently across all the traces and benchmark we tested. With no trade-offs observed between small and large log sectors, this result indicates that write optimization has a larger effect on performance than read optimization and that 8KB appears to be large enough to obtain the best performance except for the `AndroBench`.

4.3.3 Read Performance

Trace	Total pages read	Pages rolled forward	Elapsed time†	Time to roll forward†
KakaoTalk	66	11	28.4	2.1
Twitter	26	10	17.9	2.8
Browser	48	13	55.7	1.7
Facebook	316	113	130.4	34.5
Gmail	63	17	25.3	4.2

†Times measured in millisec

Table 6: Read Performance of *SQLite/PPL*

As is explained above, for a page read operation, *SQLite/PPL* has to roll forward the changes stored in the log records if there is a log sector associated with the page to read. In order to understand the read overhead of *SQLite/PPL*, we extracted all `select` statements from the traces and replayed them with *SQLite/PPL*, immediately after running the whole traces, so that updates are left with data pages in their log sectors. Table 6 shows the total elapsed times taken to process the statements and the time taken to roll forward log records. The time for rolling forward was within 26% of the total elapsed time of the read-only traces, and it was negligible compared with the total elapsed time for read-write traces.

4.3.4 Effect of All in PCM

In order to assess the performance gain by *SQLite/PPL* more accurately, we ran *SQLite/PPL* and the vanilla SQLite with the entire databases (as well as journal files for the vanilla SQLite) stored in PCM. In Figure 8, `Delete-on-PCM`, `WAL-on-PCM`, and `SQLite/PPL-on-PCM` denote the runtime measurements of the vanilla SQLite in `Delete` and `WAL` journal modes, and *SQLite/PPL*, respectively. With the databases stored in PCM, *SQLite/PPL* outperformed the vanilla SQLite considerably and consistently for all the traces and benchmark. This result matches the performance trend observed above in this section.

For comparison, Figure 8 also includes the runtime measurements of *SQLite/PPL*, denoted by `SQLite/PPL`, executed in the standard configuration. Even with the databases stored in flash memory, *SQLite/PPL* outperformed the vanilla SQLite accessing the databases stored in PCM for `KakaoTalk`, `Twitter` and `AndroBench`, and performed comparably for `Browser`, `Facebook` and `Gmail`.

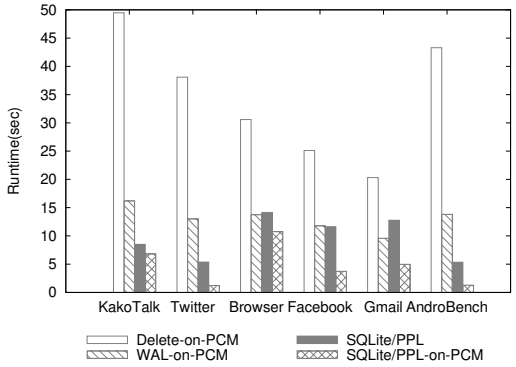


Figure 8: Effect of All in PCM

5. RELATED WORK

Our work on *SQLite/PPL* is unique in that the hardware and software components work in harmony to accelerate the performance of a transactional database system. To the best of our knowledge, *SQLite/PPL* is the first that adopts phase change memory (PCM) with a memory abstraction and minimizes write latency by storing the changes in PCM on a per-page basis. In this section, we briefly survey the recent work on PCM from the standpoint of *SQLite/PPL*, and compares the *per-page logging* with its precursor, *in-page logging*.

5.1 Recent Study on PCM

Since the late 2000s, the phase change memory (PCM) has been actively studied in the operating system and architecture communities [7, 8, 21, 31]. While it is considered a promising technology for the next generation byte-addressable non-volatile memory, some of the benefits from PCM are overpromised. For example, a recent study confirms that, for a page write, PCM could deliver longer latency than flash memory when it is accessed through the standard I/O stack [18]. For this reason, we argue that caching or tiering is not the best use of PCM.

Based on recent studies [7, 31], it appears that the overhead of I/O stack can be best avoided by exposing PCM as a persistent memory abstraction and providing direct access to it. *SQLite/PPL* presented in this paper is similar in spirit to them in that it accesses PCM at byte level through the DIMM interface on the UMS board and avoids the I/O stack overhead. The difference is of course that *SQLite/PPL* uses PCM as a persistent but transient memory and stores the majority of data in flash memory.

PCM has also been studied to accelerate database systems. Fang *et al.* proposed to use PCM as a write-ahead logging device in a transactional database system [10]. However, we argue against it, because the sequential page writes of PCM is not even better than that of a commodity flash memory SSD, especially when PCM is accessed through the I/O stack. There are other studies that have considered PCM for in-page logging [19, 30] and transaction-wise logging [11], but they do not support transactional atomicity and durability in the presence of failure.

A hybrid storage engine called SOFORT proposes a fast recovery mechanism that takes advantage of non-volatility of storage class memory [27]. SOFORT builds on an underlying assumption that the storage class memory takes 700ns to write a page. Non-volatile memory has also been studied to reduce logging delay in a transactional database system, which can adopt it as either a DIMM device or an I/O device [6, 13, 32]. While those bodies of research are similar to our work in that non-volatile memory plays a critical role, *SQLite/PPL* attempts a different problem, namely, buffering writes in PCM.

5.2 In-Page vs. Per-Page Logging

The *per-page logging (PPL)* proposed in this paper is similar in spirit to *in-page logging (IPL)* in that updates are captured in physiological log records, stored closely with corresponding data pages, and applied to the data pages later in a merge event. It was a while ago when IPL was proposed for enterprise database servers to get over the limited performance of flash memory for random writes [22]. Since then, however, IPL has never been implemented for any real database server for a few critical technical reasons.

IPL assumed all-flash devices as underlying storage media. This requires that both data pages and log sectors be stored in flash memory and the only way to co-locate them is to store both data pages and their log records in the same block of flash memory. Therefore, merging update log records to their pages is performed at the granularity of a flash memory block, which is much larger than a flash memory page, and a merge event is triggered by a flash memory block becoming full not by a committing transaction. This coarse-grained merge procedure requires more sophisticated commit procedures such as *three phase commit* and *redundant logging* [23].

Since flash memory is not byte-addressable, IPL cannot write log records individually and immediately to log sectors in flash memory. Instead, log records need to be collected in memory and grouped by data pages they belong to. They are then written to flash memory when an in-memory log sector becomes full or its page frame is evicted from the buffer pool. This requires in-memory log sectors to be maintained for all dirty pages in the buffer pool and increases the overhead of buffer management.

These restrictions has a critical and negative impact on the performance of IPL, especially in comparison with PPL with no such restriction. Unlike PPL that can write a log record individually and immediately to PCM when it is created, IPL has to do it by performing a page write no matter how full the log sector is of log records. If dirty pages are often evicted from the buffer pool after undertaking only a few updates, then IPL will end up with performing too many page writes only to write a small number of log records in flash memory. Obviously, this will lead to increased write amplification and update latency, and consequently, reduced throughput of a database server. Unfortunately, this will only get aggravated with the current upward trend in flash memory page sizes. Storing log sectors in PCM has been considered to get over the limitation of flash memory [19], but it has never been implemented into a transactional database server to the best of our knowledge.

PPL takes advantage of the byte-accessibility and low read latency of PCM in the UMS board and overcomes the limitations of IPL. PPL need not maintain the in-memory log sectors any longer. Update log records can be written to the PCM log sectors immediately without increasing the average latency of log write operations, because the write time of PCM is approximately proportional to the amount of data to transfer. PPL handles read operations that need to roll forward changes efficiently by reading update log records quickly from the PCM log sectors.

6. CONCLUSIONS

Among many promising non-volatile memory technologies under active development, the phase change memory (PCM) is expected to arrive in market earlier than others. PCM is absolutely faster than flash memory for a small update but it could deliver longer latency than flash memory for a page write, when it was accessed through the standard I/O stack. For this reason, we argue that caching or tiering is not the best use of PCM and a persistent memory abstraction with DIMM interface is the best way to avoiding the latency of I/O stack.

In this paper, we present the design and implementation of *SQLite/PPL*, which is augmented with several new features for the realization of the *per-page logging* strategy. *SQLite/-PPL* adopts phase change memory with a persistent memory abstraction and minimizes write latency by capturing an update in a physiological log record and adding it to the PCM log sector. In effect, *SQLite/PPL* can replace a multitude of successive page writes made against the same logical page with much smaller log writes done to PCM significantly more efficiently.

We have evaluated the performance gain by *SQLite/PPL* with real traces obtained from popular mobile applications as well as a publicly available mobile benchmark. *SQLite/-PPL* outperformed the vanilla SQLite significantly and consistently across all the traces and benchmark.

Acknowledgments

This research was supported in part by Institute for Information & communications Technology Promotion(IITP) (R0126-15-1088), Samsung Electronics, and Korea Institute of Science and Technology Information (RFP-2015-19).

7. REFERENCES

[1] AndroBench. `http://www.androbench.org/`.

[2] SQLite. `http://www.sqlite.org/`.

[3] A. Ban. Flash file system, April 4 1995. US Patent 5,404,485.

[4] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *MICRO '43*, pages 385–395, 2010.

[5] Hoeju 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.

[6] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *SOSP '13*, pages 197–212, 2013.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS '11*, pages 105–118, 2011.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *SOSP '09*, pages 133–146, 2009.

[9] Duglas Perry. Micron Has Phase Change Memory in Volume Production. http://www.tomshardware.com/-news/micron-pcm-phase-change-memory,16330.html, July 2012.

[10] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High Performance Database Logging Using Storage Class Memory. In *ICDE '11*, pages 1221–1231, 2011.

[11] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. PCMLogging: Reducing Transaction Logging Overhead with PCM. In *CIKM '11*, pages 2401–2404, 2011.

[12] Gartner, Inc. Top 10 Strategic Technology Trends for 2015. http://www.gartner.com/newsroom/id/2867917, October 2014.

[13] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of VLDB Endowment*, 8(4):389–400, December 2014.

[14] ITRS. International Technology Roadmap for Semiconductors. `http://efutures.ac.uk/sites/-default/files/ITRS_2009.pdf`, 2009.

[15] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *USENIX Annual Technical Conference '13*, pages 309–320, 2013.

[16] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. X-FTL: Transactional FTL for SQLite Databases. In *SIGMOD '13*, pages 97–108, 2013.

[17] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *USENIX FAST '12*, pages 17–29, 2012.

[18] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering ApproachesRecovery. In *USENIX FAST '14*, pages 33–45, 2014.

[19] Kangnyeon Kim, Sang-Won Lee, Bongki Moon, Chanik Park, and Joo-Young Hwang. IPL-P: In-Page Logging with PCRAM(demo paper). *Proceedings of the VLDB Endowment*, 4(12):1363–1366, 2011.

[20] 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 *USENIX FAST '14*, pages 273–285, 2014.

[21] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory As a Scalable Dram Alternative. In *ISCA '09*, pages 2–13, 2009.

[22] Sang-Won Lee and Bongki Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In *SIGMOD '07*, pages 55–66, 2007.

[23] Sang-Won Lee and Bongki Moon. Transactional In-Page Logging for Multiversion and Read Consistency and Recovery. In *ICDE '11*, pages 876–887, 2011.

[24] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[25] Taemin Lee, Dongki Kim, Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. FPGA-based Prototyping Systems for Emerging Memory Technologies(Invited Paper). In *Rapid System Prototyping (RSP) '14*, pages 115–120, 2014.

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

[27] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *DaMoN '14*, pages 8:1–8:7, 2014.

[28] Portio Research. Mobile Data Usage Trends 2011-2015. http://www.portioresearch.com/, 2011.

[29] Samsung Electronics. 2G x 8 Bit NAND Flash Memory (K9F8G08U0M). Data sheet, 2007.

[30] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement. In *HPCA-16*, pages 1–12, 2010.

[31] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11*, pages 91–104, 2011.

[32] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proceedigns of VLDB Endowment*, 7(10):865–876, June 2014.