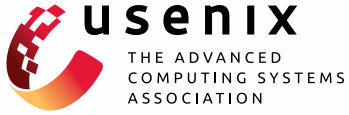


Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split

Wook-Hee Kim and Beomseok Nam, ***Ulsan National Institute of Science and Technology;***
Dongil Park and Youjip Won, ***Hanyang University***

https://www.usenix.org/conference/fast14/technical-sessions/presentation/kim-wook-hee

This paper is included in the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14).

February 17–20, 2014 • Santa Clara, CA USA

ISBN 978-1-931971-08-9

Open access to the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14) is sponsored by

Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split

Wook-Hee Kim[†], Beomseok Nam[†], Dongil Park[‡], Youjip Won[‡]

[†] *Ulsan National Institute of Science and Technology, Korea*
{*okie90,bsnam*}*@unist.ac.kr*
[‡] *Hanyang University, Korea*
{*idoitlpg,yjwon*}*@hanyang.ac.kr*

Abstract

Misaligned interaction between SQLite and EXT4 of the Android I/O stack yields excessive random writes. In this work, we developed multi-version B-tree with lazy split (LS-MVBT) to effectively address the Journaling of Journal anomaly in Android I/O. LS-MVBT is carefully crafted to minimize the write traffic caused by fsync() call of SQLite. The contribution of LS-MVBT consists of two key elements: (i) Multi-version B-tree effectively reduces "the number of fsync() calls" via weaving the crash recovery information within the database itself instead of maintaining a separate file, and (ii) it significantly reduces "the number of dirty pages to be synchronized in a single `fsync()` call" via optimizing the multi-version B-tree for Android I/O. The optimization of multi-version B-tree consists of three elements: lazy split, metadata embedding, and disabling sibling redistribution. We implemented LS-MVBT in Samsung Galaxy S4 with Android 4.3 Jelly Bean. The results are impressive. For SQLite, the LS-MVBT exhibits 70% (704 insertions/sec vs. 416 insertions/sec), and 1,220% performance improvement against WAL mode and TRUNCATE mode (704 insertions/sec vs. 55 insertions/sec), respectively.

1 Introduction

In the era of mobile computing, smartphones and smart devices generate more network traffic than PCs [1]. It has been reported that 80% of the smartphones sold in the third quarter of 2013 are Android smartphones [2]. Despite the rapid proliferation of Android smartphones, the I/O stack of Android platform leaves much to be desired as it fails to fully leverage the maximum performance from hardware resources. Kim et al. [3] reported that in an Android device, storage I/O performance indeed has significant impact on the overall system performance although it has been believed that the slow storage performance should be masked due to even slower network subsystem. The poor storage performance mainly comes from the discrepancies in interaction between SQLite and EXT4.

SQLite is a serverless database engine that is used extensively in Android applications to persistently manage the data. SQLite maintains crash recovery information for a transaction in a separate file which is log for write-ahead logging (or rollback journal). In an SQLite transaction, every update in the log or rollback journal and actual updates in the database table are separately committed to the storage device via `fsync()` calls. In TRUNCATE[1] mode, a single insert operation of 100 byte record entails 2 `fsync()` calls and eventually generates 9 write operations (36 KB) to the storage device. 100 byte of database insert amplifies to over 36 KB when it reaches the storage device[4]. The main cause of this unexpected behavior is that EXT4 filesystem journals the journaling activity of SQLite through heavy-weight `fsync()` calls. This is called the Journaling of Journal anomaly [4].

There are several ways to resolve the Journaling of Journal anomaly. One way is to tune the I/O stack in OS layer, such as eliminating unnecessary metadata flushes and storing journal blocks on a separate block device [5]. Another way is to integrate the recovery information into the database file itself so that the database can be restored without an external journal file. *Multi-Version B-Tree (MVBT)* proposed by Becker et al. [6] is an example of the latter. The excessive I/O operations also cause other problems such as shortening the lifetime of NAND eMMC since NAND flash cells can only be erased or written to a limited number of times before they fail.

In this work, we dedicate our efforts on resolving the Journaling of Journal anomaly from which the Android I/O stack suffers. Journaling of Journal anomaly is caused by two reasons: the number of `fsync()` calls in an SQLite transaction and the overhead of a single `fsync()` call in EXT4. In order to reduce the number

[1] one of the journal modes in SQLite

of `fsync()` calls as well as the overhead of a single `fsync()` call, we developed a variant of Multi-version B-tree, *LS-MVBT (Lazy Split Multi-Version B-Tree)*. The contributions of this work are summarized as follows.

- **LS-MVBT** We resolve the Journaling of Journal anomaly with multi-version B-tree that weaves transaction recovery information into the database file itself instead of using separate rollback journal file or WAL log file.
- **Lazy split** LS-MVBT reduces the number of dirty pages flushed to the storage device when a B-tree node overflows. Our proposed *lazy split* algorithm minimizes the number of modified B-tree nodes by combining a historical dead node with one of its new split nodes.
- **Buffer reservation** LS-MVBT further reduces the chances of dirtying an extra node by padding some buffer space in lazy split nodes. If a lazy split node is accessed again and additional data items need to be stored, they are stored in reserved buffer space instead of splitting it.
- **Metadata embedding** LS-MVBT reduces the I/O traffic by not flushing the database header page to the storage device. Instead, our proposed metadata embedding method moves the file change counter metadata from database header page into the last modified B-tree node which should be flushed anyway.
- **Disabling sibling redistribution** Sibling redistribution (migration of overflown data into left and right sibling nodes) has been widely used in database systems, but we show that it significantly increases the number of dirty nodes. LS-MVBT prevents sibling redistribution to improve write performance at the cost of slightly slowing search performance.
- **Lazy garbage collection** Version-based data structures require garbage collection for dead entries. LS-MVBT reclaims dead entries of a B-tree node only when the node needs to be modified by a current write transaction. This lazy garbage collection does not increase the amount of data to be flushed, since it only cleans up dirty nodes.

We implemented LS-MVBT in one of the most recent smartphone models, Galaxy-S4. Our extensive experimental study shows that LS-MVBT exhibits 70% performance improvement against WAL mode and 1,220% improvement against TRUNCATE mode in SQLite transactions. WAL mode may suffer from long recovery latency for replaying the log. LS-MVBT outperforms WAL mode not only in terms of transaction performance, e.g., insertion/sec, but also in terms of recovery time. Our experiment shows recovery time in LS-MVBT is up to 440% faster than that in WAL mode.

The rest of the paper is organized as follows: In section 2, we discuss other research efforts related to the Android I/O stack and database recovery modes including multi-version B-trees. In section 3, we present how multi-version B-tree (MVBT) resolves the Journaling of Journal anomaly. In section 4, we present our design of a variant of MVBT, LS-MVBT (Lazy Split Multi-Version B-tree). In section 5, we propose further optimizations including metadata embedding, disabling sibling redistribution, and lazy garbage collection that reduce the number of dirty pages. Section 6 provides the performance results and analysis. In section 7, we conclude the paper.

2 Related Work

SQLite is a key component in the Android I/O stack which allows the applications to manage their data in a persistent manner [7]. In Android based smartphones, contrary to common perception, the major performance bottleneck is shown to be the storage device rather than the air-link [3], and the journaling activity is shown to be the dominant source of storage traffic [3, 4]. Lee et al. showed Android applications generate excessive amount of EXT4 journal I/O's, most of which are caused by SQLite [5]. The excessive I/O traffic is found to be caused by the misaligned interaction between the SQLite and EXT4 [4]. Jeong et al. improved the Android I/O stack by employing a set of optimizations, which include `fdatasync()` instead of `fsync()`, F2FS, external journaling, polling-based I/O, and WAL mode in SQLite instead of other journal modes. With these optimization methods, Jeong et. al achieved 300% improvement in SQLite performance without any hardware assistance [4].

Database recovery has been implemented in many different ways. While log-based recovery methods such as ARIES [8] are commonly used in many other server-based database management systems, rollback journal is used as the default atomic commit and rollback method in SQLite although WAL (Write-Ahead Logging) has become available since SQLite 3.7 [7].

In addition to the rollback journal and log-based recovery methods, many version-based atomic commit and rollback methods have been studied in the past. Version-based recovery methods integrate the recovery information into the database itself so that the database can be restored without an external journal file [6, 9, 10, 11]. Some examples include the *write-once balanced tree*

(WOBT) for indelible storage [9], version-based hashing method for accessing temporal data [12], and the *time-split B+-tree (TSBT)* [10] which is implemented in Microsoft SQL Server. Multi-version B+-tree (MVBT) proposed by Becker et al. [6] is designed to give a new unique version for each write operation. The version-based B-tree is proved to be asymptotically optimal in a sense that its time and space complexity are the same as those of the single-version B-tree. Becker's MVBT does not support multiple updates within a single transaction, but this drawback was overcome by *Transactional MVBT* which improved the MVBT by giving the same timestamp to the data items updated by the same transaction [13]. Our LS-MVBT is implemented based on the Transactional MVBT with several optimizations we propose in section 4.

The latest non-volatile semiconductor storage, such as NAND flash memory and STT-MRAM, sheds new light on the version-based atomic commit and rollback methods [14, 15]. Venkataraman et al. proposed a B-tree structure called CDDS (Consistent and Durable Data Structure) B-tree which is almost identical to MVBT except that it focuses on implementing multi-version information on non-volatile memory (NVRAM) [15]. For durability and consistency, CDDS uses a combination of `mfence` and `clflush` instructions to guarantee that memory writes are atomically flushed to NVRAM.

As write operations on flash memory systems have high latency, Li et al. developed FD-tree which is optimized for write operations on flash storage devices [16]. As the FD-tree needs a recovery scheme such as journaling or write-ahead-logging, the version-based recovery scheme can also be employed by FD-tree. If so, our proposed optimizations for multi-version B-tree can be employed on FD-tree as well.

Current database recovery schemes are based on the traditional two layers - volatile memory and non-volatile disks - but the advent of the NVRAM presents new challenges, i.e., write-ahead logging (WAL) causes some complications if the memory is non-volatile [17]. WAL recovery scheme is designed in a way that any update operation to a B-tree page has to be recorded in a permanent write-ahead-log file first while the dirty B-tree nodes stay in volatile memory. If a database node is also in permanent NVRAM, the logging is not "write-ahead". With NVRAM, the WAL scheme must be redesigned. An alternative solution is to use version-based recovery scheme for NVRAM as in CDDS B-tree. Lazy split, metadata embedding, and other optimizations that we propose in this work can be used to reduce the number of write operations even for CDDS B-tree.

3 Multi-Version B-tree

3.1 Journaling of Journal Anomaly in Android I/O

In the Android platform, `fsync()` call is triggered by the commit of an SQLite transaction. As the journaling activity of SQLite propagates expensive metadata update operations to the file systems, SQLite spends most of its insertion (or update) time on `fsync()` function call for journal and database files [4]. The issue of resolving Journaling of Journal anomaly boils down to two technical ingredients: (i) reducing the number of `fsync()` calls in an SQLite transaction and (ii) reducing the number of dirty pages which need to be synchronized to the storage in a single `fsync()` call. Both of these two constituents eventually aim at reducing the write traffic to the block device.

In rollback journal modes (DELETE, TRUNCATE, and PERSIST) of SQLite, a single transaction consists of two phases: database journaling and the database update. SQLite calls `fsync()` at the end of each phase to make the result of each phase persistent. In EXT4 with *ordered mode* journal, `fsync()` consists of two phases: (i) writing the updated data blocks to a file and (ii) committing the updated metadata for the respective file to the journal. Most database updates in a smartphone, e.g. inserting a schedule in the calendar, inserting a phone number in the address book, or writing a note in the Facebook timeline, are less than a few hundred bytes [5]. As a result, in the first phase of `fsync()`, the number of updated file blocks rarely goes beyond a single block (4 KB). In the second phase of `fsync()`, committing a journal transaction to the filesystem journal entails four or more `write` operations, including journal descriptor, group descriptor, block bitmap, inode table, and journal commit mark, to the storage. Each of these entries corresponds to a single filesystem block.

In an effort to reduce the number of `fsync()` calls in an SQLite transaction, we implemented version-based B-tree, *multi-version* B-tree by Becker et al. [6], which maintains update history within the B-tree itself instead of maintaining it in a separate rollback journal file (or log file). This saves SQLite one or more `fsync()` calls.

3.2 Multi-Version B-Tree

In multi-version B-tree (MVBT), each insert, delete, or update transaction increases "the most recent consistent version" in the header page of a B-tree. Each key-value pair stored in MVBT defines its own life span - $[version_{start}, version_{end})$. When a key-value pair is inserted with a new version v, the life span of the new key-value pair is set to $[v, \infty)$. When a key-value pair

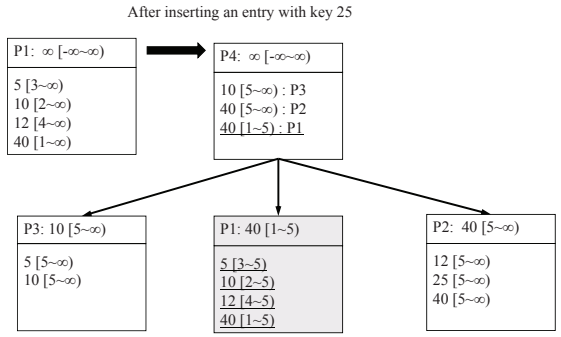


Figure 1: Multi-Version B-Tree split: After inserting an entry with key 25 into MVBT, three new nodes are created.

is deleted at version v, its life span is set to - $[v_{old}, v)$. Update transaction creates a new cell entry that has the transaction's version as its starting version $[v, \infty)$ and the old cell entry updates its valid end version to the previous consistent version number $[v_{old}, v)$. The key-value pair whose $version_{end}$ of life span is not ∞ is called a *dead* entry. The one with infinite life span is called a *live* entry. In multi-version B-trees, the search operation is trivial. A read transaction first examines the latest consistent version number and uses it to find valid entries in B-tree nodes, i.e., if a version of a read transaction is not within the life span of a key-value pair, the respective data is ignored by the read transaction.

If a node overflows, the entries in the overflown node are distributed into two newly allocated nodes, which is referred to as "node split". An additional new node is then allocated as a parent node or an existing parent node is updated with the two newly created nodes. The life spans of the two new nodes are set to $[v, \infty)$. An overflown node becomes dead via setting the node's version range $[v_{old}, \infty)$ to $[v_{old}, v)$. In summary, a single node split creates at least four dirty nodes in version-based B-tree structures. (Please refer to [6] and [15] for more detailed discussions on the insertion and split algorithms of version-based B-tree.). In the commit phase of a transaction, SQLite writes dirty nodes in the B-tree using the `write()` system call and triggers `fsync()` to make the result of the `write()` persistent.

Figure 1 shows how an MVBT splits a node when it overflows. Suppose a B-tree node can hold at most four entries in the example. When a new entry with key 25 is inserted by a transaction whose version is 5, the node `P1` splits and a half of the live entries are copied to a new node, `P2`, and the other half of the live entries are copied to another new node, `P3`. The previous node `P1` now becomes a *dead node* and it becomes available only for the transactions whose versions are older (smaller) than 5. The two new nodes should be pointed by a parent node and the version range of the dead node should also be updated in the parent node. In the example, a new root node, `P4`, is created and the pointers to the three child nodes are stored.

The recovery in multi-version B-tree is simple and straightforward. Multi-version B-tree maintains the version numbers of currently outstanding transactions at the storage. In current SQLite, there can be at most one outstanding write transaction for a given B-tree [7]. In the recovery phase, the recovery module first reconstructs the multi-version B-tree in memory from the storage and determines the version number of aborted transaction. Then, it scans all the nodes and adjusts the life span of each cell entry to obliterate the effect of aborted transaction. The life span which ends at v, i.e., $[v_{old}, v)$, is revoked to $[v_{old}, \infty)$ and all cell entries which start at v are deleted.

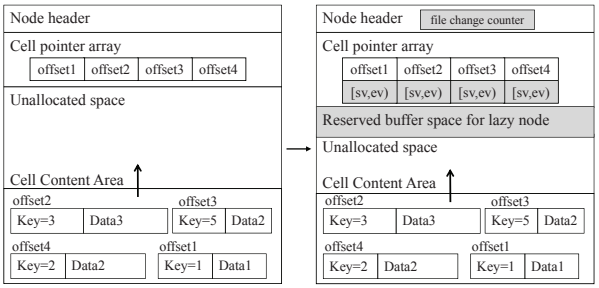
The recent eMMC controllers generate error correction code for 4 KB or 8 KB page, hence multi-version B-tree can rely on `fsync()` to atomically move from one consistent state to the next in the unit of page size. Even if the eMMC controller promises that only single sector writes are atomic and the B-tree node size is a multiple of the sector size, multi-version B-tree guarantees correct recovery as it creates a new key-value pair with new version information instead of overwriting previous key-value pairs. A multi-version B-tree node can be considered a combination of B-tree node and journal.

4 Lazy Split Multi-version B-Trees

MVBT successfully reduces the number of `fsync()` calls in an SQLite transaction as it eliminates the journaling activity of SQLite. Our next effort is dedicated to minimizing the overhead of a single `fsync()` call in MVBT. The essence of the optimization is to minimize the number of dirty nodes which are flushed to the disk as a result of a single SQLite transaction.

4.1 Multi-Version B-Tree Node in SQLite

We modified the B-tree node structure of SQLite and implemented a multi-version B-tree. Figure 2 shows the layout of an SQLite B-tree node which consists of two area: (i) *cell content area* that holds key-value pairs and (ii) *cell pointer array* which contains the array of pointers (offsets) each of which points to the actual key-value pair. Cell pointer array is sorted in key order. In the modified B-tree node structure, each key-value pair defines its own life span - $[version_{start}, version_{end})$, illustrated as $[sv, ev)$. The augmentation with start and end version number is universal across all the version-based B-tree structures [6, 15]. In our MVBT node design, we set



(a) Layout of SQLite B-Tree Node (b) Modified Layout of SQLite B-Tree Node

Figure 2: In modified Multi-Version B-Tree node, each key-value pair is tagged with its valid starting version and ending version.

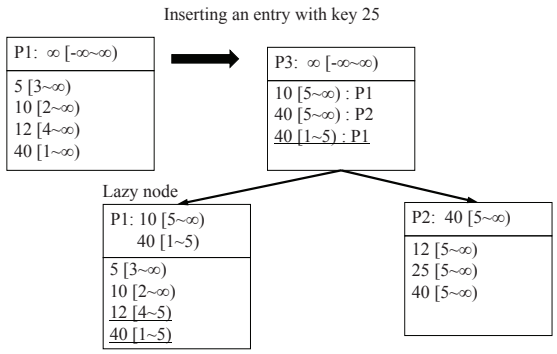


Figure 3: LS-MVBT: With the lazy split, an overflown node creates a single sibling node.

Algorithm 1
Lazy Split Algorithm

procedure
LazySplit$(n, parent, v)$

```
1: // n is an overflown B-tree node.
2: // parent is the parent node.
3: // v is the version of a current transaction.
4: newNode ← allocateNewBtreeNode()
5: Find the median key value k to split
6: for i ← 0, n.numCells − 1 do
7:     if k < n.cell[i].key ∧ v ≤ n.cell[i].endVersion then
8:         n.cell[i].endVersion ← v
9:         newNode.insert(n.cell[i])
10:        n.liveCells − −
11:    end if
12: end for
13: // Update the parent with the split key and version
14: maxLiveKey ← findMaxLiveKey(n, v)
15: parent.update(n, maxLiveKey, ∞)
16: maxDeadKey ← findMaxDeadKey(n, v)
17: parent.insert(n, maxDeadKey, v)
18: maxLiveKey2 ← findMaxLiveKey(newNode, v)
19: parent.insert(newNode, maxLiveKey2, ∞)
```

end procedure

aside a small fraction of bytes in the header of each node for *lazy split* and *metadata embedding* improvement.

4.2 Lazy Split

We develop an alternative split algorithm, *Lazy Split*, for MVBT that significantly reduces the number of dirty pages.

In MVBT, a single node split operation results in at least four dirty B-tree nodes as shown in Figure 1. The objective of maintaining a separate dead node in MVBT is to make garbage collection and recovery simple. On the other hand, creating a separate dead node yields an additional dirty page which needs to be flushed to disk. Unlike in other client/server databases, rollback operations do not occur frequently in SQLite, because SQLite allows only one process at a time to have write permission to a database file [7], and rollback operations of a version-based B-tree are already very simple. Therefore, we argue that benefit of creating a separate dead node in the legacy split algorithm of MVBT hardly offsets the additional performance overhead during `fsync()` that it induces.

Algorithm 1 shows our *lazy split* algorithm that postpones marking an overflown node as dead, if possible. Instead of creating an extra dead node, lazy split algorithm combines a dead node with a live sibling node. I.e., the *lazy node* is a half dead node combined with one of the new split nodes. In the lazy split algorithm, the overflown node creates only one new sibling node. Once the median key value to split is determined, the key-value pairs whose keys are greater than the median value are copied to the new sibling node as live entries. In the overflown node, the end versions of the copied key-value pairs are changed from ∞ to the current transaction's version in order to mark them as dead entries. In the original MVBT, the key-value pairs whose keys are smaller than the median key value are copied to another new left sibling node, but lazy split algorithm does not create the left sibling node and does not change the end versions of the smaller half of the key-value pairs.

Figure 3 shows an example of lazy split. When key 25 is inserted into node `P1`, the greater half of the key-value pairs (key 12 and key 40) are moved to a new node, `P2`, and they are marked dead in `P1`. Instead of creating another new node and moving the smaller half of the key-value pairs to it, lazy split algorithm keeps them in the overflown node. The dead entries in the lazy node will be garbage collected by the next write transaction that modifies the lazy node. Note that the lazy node has two pointers pointing to it in its parent node: one for the dead entries and the other for the live entries. The same insert operation in the original MVBT will create a left sibling

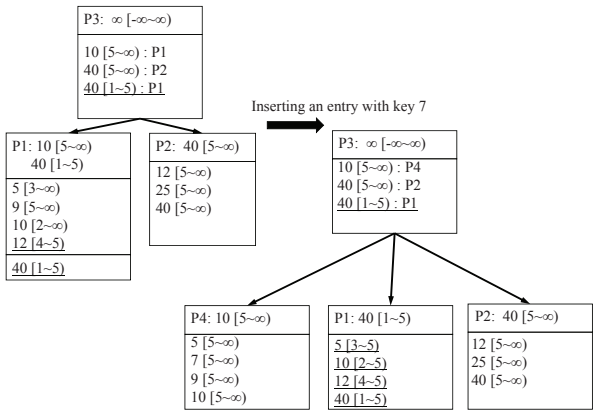


Figure 4: A new entry with key 9 is inserted into an overflown lazy node but its dead entries can not be deleted because transaction 5 is the current transaction and it may abort later. In this case, the reserved space can be used to hold the new entries and delay the node split again. But if the same transaction inserts an entry with key 7, the reserved space of the lazy node also overflows and we do not have any other option but to create a new left sibling node P4 and move the live entries ($5[5,\infty)$, $7[5,\infty)$, $9[5,\infty)$, and $10[5,\infty)$) to P4.

node, store the key 5 and key 10 in the left sibling node, and mark the two key-value entries dead in the historic dead node as shown in Figure 1. In the example, the valid version ranges of key 5 and key 10 are partitioned in the two nodes. This redundancy does not help anything especially when we consider the short lifespan of SQLite transactions. The dead entries are not needed by any subsequent write transactions and thus can be safely garbage collected in the next modification of the lazy node because a write transaction holds an exclusive lock for the database file. The legacy split algorithm of MVBT creates four dirty nodes but lazy split decreases the number of dirty nodes by one, creating only three dirty nodes.

4.3 Reserved Buffer Space for Lazy Split

The *lazy node* does not have any space left for additional data items to be inserted after the split. If an inserted key is greater than the median key value and is stored in a new node as in Figure 1, the lazy split succeeds. However, if a new inserted item needs to be stored in the lazy node, a new sibling node must be created as in the original MVBT split algorithm. In order to avoid splitting a lazy node, we reserve a certain amount of space in a LS-MVBT node to accommodate the inserted key in the lazy split node as shown in Figure 4.

To avoid cascade split, the size of the reserved buffer space should be sufficiently large to accommodate the newly inserted entries by a transaction. However, reserving too much space for buffer will make node utilization low and may entail more frequent node split creating larger amount of dirty pages. The size of the reserved buffer space needs to be carefully determined considering the workload characteristics. In smartphone applications, most write transactions do not insert more than one data item. Therefore, it is unlikely that an overflown node (lazy node) is accessed multiple times by a single write transaction.

In order to evaluate the effect of the reserved buffer space size, we ran experiments varying the sizes of reserved buffer space. Large reserved buffer space is only beneficial when a single transaction inserts a large number of entries into the same B-tree node. However, a large buffer space did not significantly reduce the number of dirty nodes in our experiments, but it hurt tree node utilization especially when the B-tree node size was small. In smartphone applications, it is very common that a transaction inserts just a single data item, hence we set the size of the buffer space just large enough to hold only one key-value item throughout the presented experiments in this paper. Even if reserved buffer space for one key-value item is used, a subsequent write transaction that finds the dead entries in the lazy node will reclaim the dead entries and create empty spaces.

Algorithm 2
Rollback Algorithm

procedure
Rollback(n, v)

```
1:  // n is a B-tree node
2:  // v is the version of aborted transaction
3:  for i ← 0, n.numCells − 1 do
4:      if n.cell[i].startVersion == v then
5:          remove n.cell[i]
6:          if n is an internal node then
7:              freeNode(n.child[i], v)
8:              continue
9:          end if
10:         deleteEntry(n.child[i])
11:     else if n.cell[i].endVersion == v then
12:         n.cell[i].endVersion ← ∞
13:         if n is an internal node then
14:             Delete a median key entry k that was used to split
                the lazy node.
15:         end if
16:     end if
17:     Rollback(n.child[i], v)
18: end for
end procedure
```

4.4 Rollback with Lazy Node

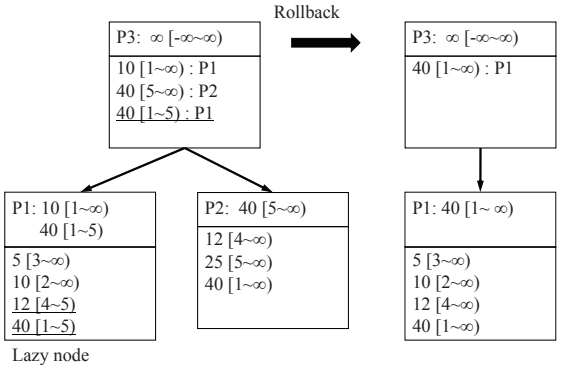


Figure 5: Rollback of transaction version 5 deletes node P2, reverts the end version of dead entries from 5 to ∞, and merges the entries in the parent node.

The rollback algorithm for the LS-MVBT is intuitive and simple. More importantly, as in the lazy split algorithm, the number of dirty nodes touched by the rollback algorithm of LS-MVBT is smaller than that of MVBT. Algorithm 2 shows the pseudo code of the LS-MVBT rollback algorithm. When a transaction aborts and rolls back, the LS-MVBT reverts its B-tree structures back to their previous states by reverting the end versions of the lazy nodes back to ∞ and deleting entries whose start versions are the aborted transaction's version. In the parent node, the lazy node has two entries: one for live entries and the other for dead entries. The parent entry of the live entries should be deleted from the parent node and the parent entry for the dead entries should be updated with its previous end version, ∞, to become active.

Figure 5 shows a rollback example. Note that node `P2` was created by a transaction whose version is 5, thus `P2` should be deleted. Since all the live entries in `P2` were copied from the lazy node `P1` by a transaction whose version is 5 and `P1` has historical entries, `P2` can be safely removed. The dead entries in `P1` should be reverted back to live entries by modifying the end versions. As the lazy node has two parent entries, the rollback process merges them and reverts back to the previous status by choosing the larger key value and by merging the valid version ranges.

5 Optimizing LS-MVBT for Android I/O

5.1 Lazy Garbage Collection

In multi-version B-trees, garbage collection mechanism is needed as dead entries must be garbage-collected to create empty spaces and to decrease the size of the trees. While a periodic garbage collector that sweeps the entire B-tree is commonly used in version-based B-trees [18, 15], we implemented *lazy garbage collection* scheme in SQLite in order to avoid making extra B-tree nodes dirty and to reduce the overhead of `fsync()`.

When a B-tree node needs to be modified, lazy garbage collection scheme checks if the node contains any dead entries whose versions are not needed by an active transaction. If so, the dead entries can be safely deleted. The dead entries in a B-tree node will be reclaimed only when a new live entry is modified or is added to the node. Since the node will become dirty anyway by the live entry, our lazy garbage collection does not increase the number of dirty nodes at all.

5.2 Metadata Embedding

In SQLite, the first page of a database file (header page) is used to store metadata about the database such as B-tree node size, list of free pages, file change counter, etc. The file change counter in header page is used for concurrency control in SQLite.[2] When multiple processes are accessing a database file concurrently, each process can detect if other processes have changed the database file by monitoring the file change counter. However, this concurrency control design of SQLite induces significant overhead on I/O traffic since the header page must be flushed just to update 4 bytes of file change counter for every write transaction. This results in a large performance gap between WAL mode and the other journal modes in SQLite (DELETE, TRUNCATE, and PERSIST) since WAL mode does not use the file change counter.

In this work, we devised a method called "Metadata Embedding" to reduce the overhead of flushing database header page. In metadata embedding, we maintain the database header page at the RAM disk so that the most recent consistent and valid version ("file change counter") in the database header page is shared by transactions and the database header page is exempt from being flushed to the storage in every `fsync()` call. Since the RAM disk is volatile, the file change counter in the RAM disk can be lost. Therefore, in metadata embedding, we let the most recent file change counter be flushed along with the last modified B-tree node. When a transaction starts, it reads the database header page at the RAM disk to access the file change counter. When a write transaction modifies the database table, it increases the file change counter and flushes it to the database header page at the RAM disk and to the last modified B-tree node. Since the last modified B-tree node has to be flushed to the storage anyway, metadata embedding makes the modified file change counter persistent without extra overhead.

[2]The race condition is handled by file system lock (`fcntl()`) in SQLite.

When a system recovers, the entire multi-version B-tree has to be scanned by a recovery process. Therefore, it is not a problem to find the largest valid consistent version number in the database and use it to rollback some changes made to the database file. If other parts of the header page are changed, we flush the header page as normal. Note that other parts of the header page are modified much less frequently than the file change counter.

5.3 Disabling Sibling Redistribution

Another optimization method used in LS-MVBT to reduce the I/O traffic is disabling redistribution of data entries between sibling nodes. If a B-tree node overflows in SQLite (and in many other server-based database engines), it redistributes its data entries to left and right sibling nodes. This is to avoid node split which requires allocation of additional nodes and changes in the tree organization. This redistribution modifies four nodes - two sibling nodes, the overflown node, and its parent node. In general, it is well known in the database community that sibling redistribution improves the node utilization, keeps the tree height short, and makes search operation faster, but we observed that it significantly hurt the write performance in the Android I/O stack.

In flash memory, time to write a page (page program latency) is 10 times longer than the time to read a page (read latency)[19] and subsequently, from SQLite's point of view, database updates, e.g., insert, update, and delete, take much longer than database search. Furthermore, search operations in smartphones are not as dominant as in client/server enterprise databases. Given these facts, we devise an approach opposite to the conventional wisdom: we disable sibling redistribution. In LS-MVBT, if a node overflows, we do not attempt to redistribute the entries in the overflown node to its siblings. Instead, LS-MVBT immediately triggers a lazy split operation.

6 Evaluation

We implemented the lazy split multi-version B-tree in SQLite 3.7.12. In this section, we evaluate and analyze the performance of the LS-MVBT compared to other traditional journal modes and WAL mode. Our testbed is *Samsung Galaxy-S4* that runs Android OS 4.3 (Jelly Bean) on Exynos 5 Octa Core 5410 1.6GHz CPU, 2GB DDR2 memory, and 16GB eMMC flash memory formatted with EXT4 file systems.

Many latest smartphones, including Samsung Galaxy S4, adjust the CPU frequency in order to save the power consumption. We fixed the frequency to the maximum 1.6 GHz so as to reduce the standard deviation of the experiments.

The evaluation section flows as follows. First, we examine the performance of SQLite transaction (`insert`) under three different SQLite modes: LS-MVBT, WAL mode, which is the default in Jelly Bean, and TRUNCATE mode, which is the default mode in Ice Cream Sandwich. Second, we take a detailed look at the block I/O behavior of SQLite transaction for LS-MVBT and WAL. Third, we observe how the versioning nature of LS-MVBT affects the search performance via examining the SQLite performance under varying mixture of `search` and `insert/delete` transactions. Fourth, we examine the recovery overhead of LS-MVBT and WAL. The final segment of the evaluation section is dedicated to quantifying the performance gain of each of the optimization techniques proposed in this paper, which are lazy split, metadata embedding, and disabling sibling redistribution, in an itemized as well as in an aggregate manner.

6.1 Workload Characteristics

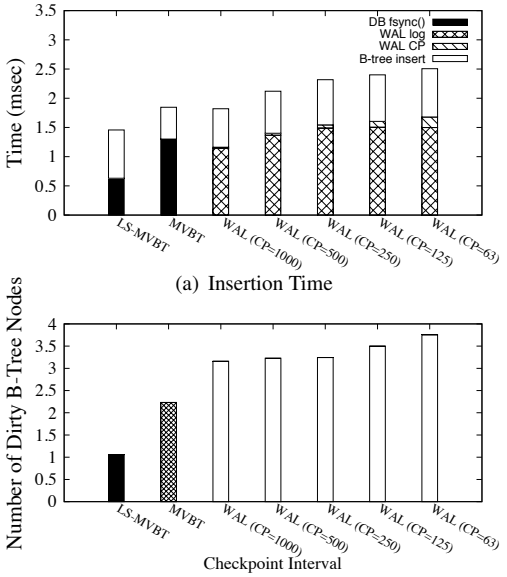
To accurately capture the workload characteristics of the smartphone apps, we extracted the database information from Gmail, Facebook, and Dolphin web browser apps in a testbed smartphone. Out of 136 tables in the device, the largest table contains about 4,500 records, and only 15 tables have more than 1,000 records. It is very common for smartphone apps to have such small number of records in a single database table unlike enterprise server/client databases. As most tables have less than thousands of records, we focused on evaluating the performance of LS-MVBT with rather small database tables. As for the reserved buffer space of LS-MVBT, we fix it to one cell for all the presented experiments.

6.2 Analysis of `insert` Performance

In evaluating the SQLite transaction performance, we focus on `insert` since `insert`, `update`, and `delete` generate similar amount of I/O traffic and show similar performances.

For the first set of experiments, we initialize a table with 2,000 records and submit 1,000 transactions, each of which inserts and deletes a random key value pair[3]. In WAL mode, checkpoint interval directly affects the transaction performance as well as recovery latency: with longer checkpoint interval, the transaction performance improves but the recovery latency gets longer. In SQLite, the default checkpoint interval is when 1,000 pages become dirty. The default interval can be changed by a pragma statement or a compile-time option. Checkpoint also occurs when *.db file is closed. If an app opens

[3]The performance of sequential key insertion/deletion is not very different from the presented results.



(a) Insertion Time

(b) Number of Dirty B-Tree Nodes per Transaction

Figure 6: Insertion Performance of LS-MVBT, MVBT, and WAL with Varying Checkpointing Interval (Avg. of 5 runs)

and closes a database file often, WAL mode will perform checkpointing operations frequently. For the comprehensiveness of the study, we vary the checkpoint intervals to 63, 125, 250, 500 and 1,000 pages. We first examine the time for a single `insert` transaction. For a fair comparison, the average insertion time in WAL mode includes the amortized average checkpointing overhead.

Figure 6(a) illustrates the result. Insertion time of MVBT and LS-MVBT consists of two elements: (i) the time to manipulate the database which is essentially an operation of updating the page content in memory, *B-tree insert*, and (ii) the time to `fsync()` the dirty pages, *DB fsync()*. Insertion time of WAL mode consists of three elements: (i) the time to manipulate the database, *B-tree insert*, (ii) the time to commit the log to storage, *WAL log*, and (iii) the time for checkpointing, *WAL CP*.

The average insertion time of LS-MVBT (1.4 ms) is up to 78% faster than that of WAL mode (2.0∼2.5 ms), but the insertion time of the original MVBT is no better than that of WAL mode. Throughout the various checkpointing intervals, LS-MVBT consistently outperforms WAL mode (even without including the checkpointing overhead). There is another important benefit of using LS-MVBT. In WAL mode, according to our measurement, the average elapsed time for each checkpoint is 7.6∼9.2 msec which is ×3 the average `insert` latency. Therefore, in WAL mode, the transactions that trigger checkpointing suffer from sudden increases in the latency. LS-MVBT outperforms WAL in terms of average query response time as well as in terms of the worst case bound.

We examine the number of dirty B-tree nodes per `insert` in MVBT, LS-MVBT, and WAL mode (Figure 6(b)). The number of dirty B-tree nodes in LS-MVBT is significantly lower than WAL mode. For an `insert`, LS-MVBT makes just one B-tree node dirty on average while WAL mode generates three or more dirty B-tree nodes. In WAL mode, not all dirty B-tree nodes are flushed to storage, but `fsync()` is called for log file commit, and the dirty nodes are flushed by the next checkpointing.

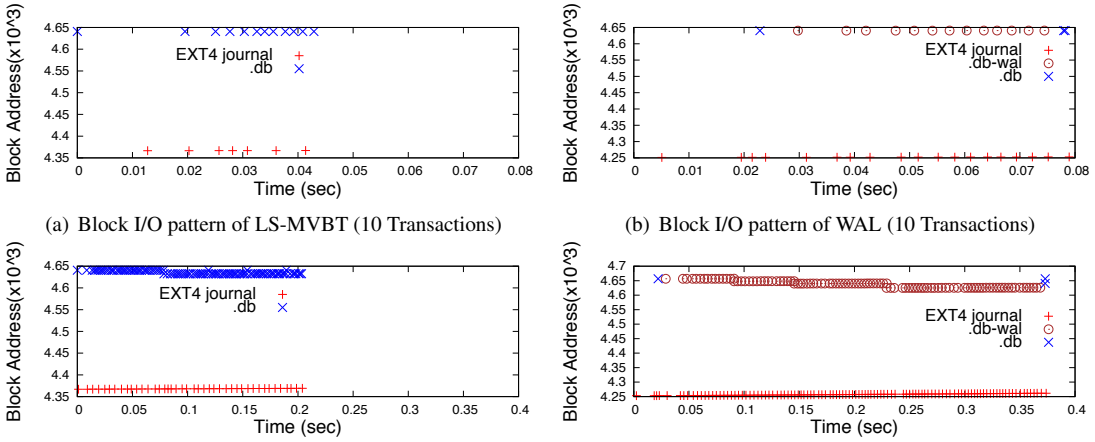
An interesting observation from Figure 6 is that the insertion performance gap between LS-MVBT and WAL is significant (40%) even when the checkpointing interval is set to 1,000 pages. When the checkpoint interval is 63 pages, the average transaction response time of WAL (2.5 msec) is 78% higher than that of LS-MVBT.

6.3 Analysis of Block I/O Behavior

For more detailed understanding, we delve into the block I/O behaviors of SQLite transactions in LS-MVBT and WAL mode. Figure 7 shows block I/O traces of an insert operation in LS-MVBT and WAL mode. Let us first examine the detailed block I/Os in LS-MVBT. When an `fsync()` is called, the updated database file contents are written to the disk. Then, the updated metadata for the file is committed to EXT4 journal. For a single insert transaction, one 4 KB block is written to the disk for file update. Three 4 KB blocks are written to EXT4 journal, which correspond to journal descriptor header, metadata, and journal commit mark. In WAL mode, 8 KB blocks are written to the disk for log file update. Eight 4 KB blocks are written to EXT4 journal. If checkpointing occurs, there will be more accesses to a block device.

Figure 7(a) and 7(b) show the number of accesses to a block device when 10 insert transactions are submitted. Interestingly, the total number of block device accesses for 10 insert transactions in WAL mode is 84% higher than that in LS-MVBT. However, with 100 insert transactions, the number of block device accesses in WAL mode is only 46% higher than that in LS-MVBT as shown in Figure 7(c) and 7(d). In LS-MVBT, the number of block device accesses increases linearly with the increased number of insertions whereas WAL mode accesses block devices less frequently when the size of batch insert transaction is larger.

Since WAL mode writes more data than LS-MVBT per each block device access, we measure the amount of I/O traffic caused in every 10 msec. Figure 8 shows the block access I/O traffic for LS-MVBT and WAL mode. For the experiment we submit 1,000 insert transactions and measure how many blocks are accessed per every



(a) Block I/O pattern of LS-MVBT (10 Transactions)

(b) Block I/O pattern of WAL (10 Transactions)

(c) Block I/O pattern of LS-MVBT (100 Transactions)

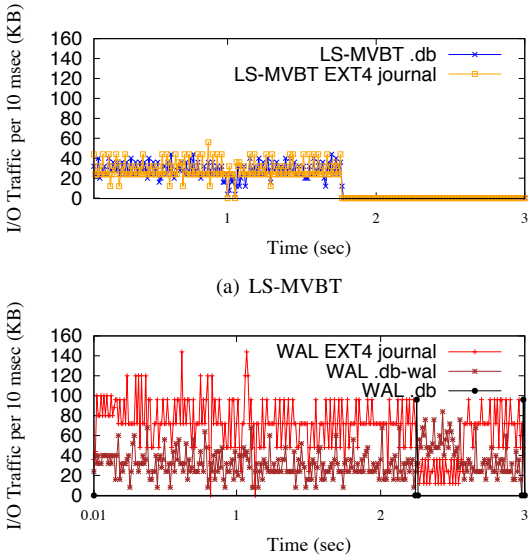
(d) Block I/O pattern of WAL (100 Transactions)

Figure 7: Block Trace of Insert SQLite Operation: LS-MVBT vs WAL

10 milliseconds. The block access I/O traffic per 10 milliseconds for LS-MVBT fluctuates between 24 KB to 40 KB, and the EXT4 journal blocks are accessed about 24∼44 KB per 10 milliseconds. In WAL mode, the database file blocks are accessed only three times: when the database file is opened, when checkpointing occurs in 2.25 seconds, and when the database file is closed.

When the checkpointing occurs at 2.25 seconds, the I/O traffic for WAL log file increases by approximately 20 KB, from 40 KB to 60 KB, but it decreases to 40 KB when the checkpointing finishes at 2.6 seconds. In WAL mode, the number of accesses to the EXT4 journal blocks is consistently higher than any other block access types, which explains why WAL mode shows poor insertion performance. We are currently investigating what causes this high number of EXT4 journal accesses in WAL mode.

In summary, LS-MVBT accesses 9.9 MB (5 MB EXT4 journal blocks and 4.9 MB database file blocks) in just 1.8 seconds, while WAL accesses 31 MB blocks (20.7 MB EXT4 journal blocks, 9.764 MB WAL log blocks, and only 0.9 MB database file blocks) in 3 seconds.



(a) LS-MVBT

(b) WAL

Figure 8: I/O Traffic at Block Device Driver Level (1,000 insertions)

6.4 Search Overhead

LS-MVBT makes the insert/update/delete queries faster at the cost of slow search performance. In LS-MVBT, node access has to check its children's version information in addition to the key range. Moreover, LS-MVBT does not perform sibling redistribution which results in poor node utilization. Lee et al. [5] reported that write operations are dominant in smartphone applications, and the SQL traces that we extracted from our testbed device confirm this. However, the search and the write ratio can depend on individual user's smartphone usage pattern, hence we examine the effectiveness of LS-MVBT with varying the ratio of search and write transactions. We initialize a database table with 1,000 records, and submit a total of 1,000 transactions with varying ratios between the number of `insert/delete` and `search` transactions. Each `insert/delete` transaction inserts and deletes a random data from the database table, and the `search` transaction searches a random data from the table. For notational simplicity, we term `insert/delete` as `write`.

Figure 9 illustrates the result. We examine the throughput under three different SQLite implementations: LS-MVBT, WAL mode, and TRUNCATE mode.

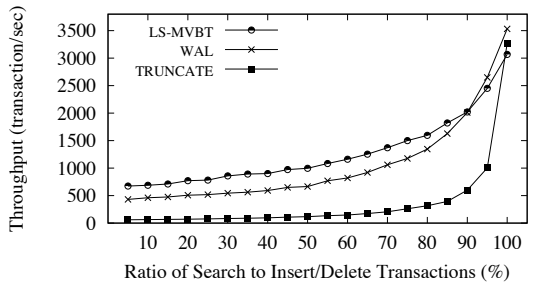


Figure 9: Mixed Workload (Search:Insert) Performance (Avg. of 5 runs)

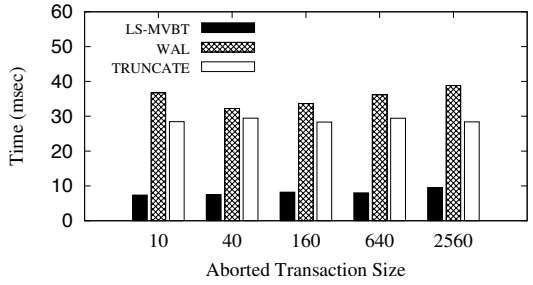
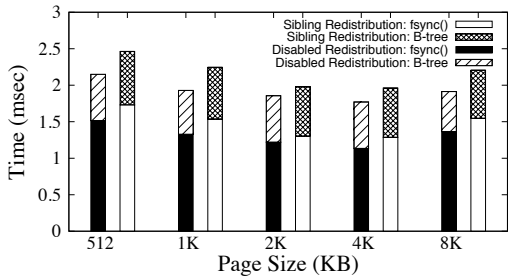
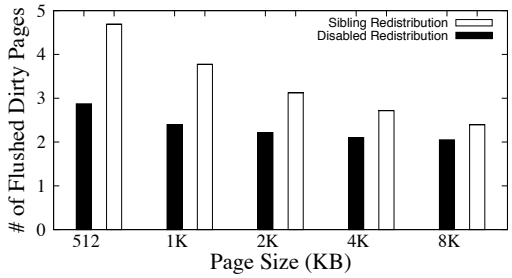


Figure 10: Recovery Time with Varying Size of Aborted Transaction



(a) Insertion Time (With vs. Without Redistribution)



(b) Number of Dirty B-tree Nodes (With vs. Without Redistribution)

Figure 11: *The average elapsed time and the number of flushed dirty nodes per insertion. (Average of 1,000 insertions): Rebalancing data entries hurts write performance when a node splits.*

As we increase the ratio of search transactions, the overall throughput increases because a search operation is much faster than a write operation. As long as at least 7% of the transactions are writes, LS-MVBT outperforms both WAL and TRUNCATE modes. In LS-MVBT, the performance gain on write operations far outweighs the performance penalty on search operations. This is mainly due to asymmetry in latencies of writing and reading a page in NAND flash memory: writing a page may take up to 9 times longer than reading a page [19].

6.5 Recovery Overhead

Recovery latency is one of the key elements that govern the effectiveness of a crash recovery scheme. While WAL mode exhibits superior SQLite performance against the other three journal modes, i.e., DELETE, TRUNCATE, and PERSIST, it suffers from longer recovery latency. This is because in WAL mode, the log records in the WAL file need to be replayed to reconstruct the database. In this section, we examine the recovery latencies of TRUNCATE, WAL, and LS-MVBT under varying number of outstanding (or aborted equivalently) insert statements in an aborted transaction at the time of crash: 10, 40, 160, 640, and 2560.

Figure 10 illustrates the recovery latencies of LS-MVBT, WAL, and TRUNCATE. When the aborted transaction inserts less than 10 records, WAL mode recovery takes about 4∼5 times longer than LS-MVBT. As the transaction size grows from 10 insertions to 2,560 insertions, WAL recovery mode suffers from a larger number of write I/Os and its recovery time increases by 20%. LS-MVBT recovery mode also increases by 28% but from much shorter recovery time. TRUNCATE mode recovery time slightly increases, by only 3%, but its recovery time is already 3.9 times longer than LS-MVBT when the transaction size is just 10. LS-MVBT needs to read the entire B-tree nodes for recovery but it only updates the nodes that should rollback to a consistent version.

6.6 Performance Effect of Optimizations

In order to quantify the performance effect of the optimizations made on MVBT, we first examine the effect of sibling redistribution in SQLite B-tree implementation by enabling and disabling the sibling redistribution. We use the average insertion time and the average number of dirty B-tree nodes for each insertion as performance metrics in Figure 11. We insert 1,000 records of 128 bytes into an empty table, and vary the node sizes of B-tree in SQLite from 512 bytes to 8 KB.

Figure 11(a) shows the average insertion time when sibling redistribution is *enabled* and *disabled*. When sib-

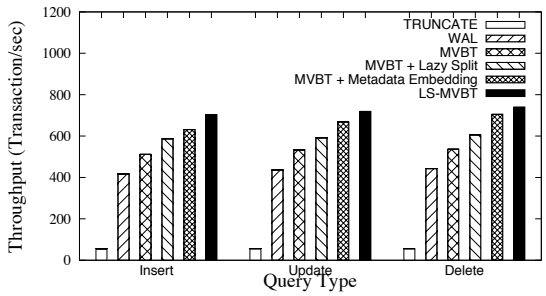


Figure 12: Performance Improvement Quantification (Avg. of 5 runs)

ling redistribution is disabled, insertion time decreases as much as 20%. In the original B-tree, 70% of the insertion time is spent on `fsync()` and most of the improvement comes from the reduction in `fsync()` overhead. Figure 11(b) shows the average number of dirty B-tree nodes per a single `insert` transaction. With 1 KB node size, the number of dirty pages in an `insert` is reduced from 3.7 pages to 2.4 pages if sibling redistribution is disabled. Since metadata embedding can save another dirty page, with disabled sibling redistribution and metadata embedding, the average number of dirty B-tree nodes per a single insertion transaction can drop down to fewer than 2 nodes, i.e., approximately 50% of disk page flush can be saved.

With a larger node size, the number of dirty B-tree nodes decreases because node overflow occurs less often. However, we observe that the elapsed `fsync()` time grows with larger node sizes (4 KB and 8 KB) since the size of nodes that need to be flushed increases, and also the time spent in B-tree insertion code increases because more computation is required for larger tree entries. After examining the effect of B-tree node size on insert performance (Figure 11), we determine that 4 KB node size yields the best performance. In all experiments in this study, B-tree node size is set to 4 KB. [4]

6.7 Putting Everything Together

It is time to put everything together and examine real world implications. In Figure 12, we compare the performance of the multi-version B-trees with different combinations of the optimizations for three different types of SQL queries. The performances are measured in terms of transaction throughput (number of transactions/sec). *MVBT* denotes the multi-version B-Tree with disabled sibling redistribution. *MVBT + Metadata Embedding* denotes the multi-version B-tree with metadata embedding optimization and disabled sibling redistribution. *MVBT + Lazy Split* is the multi-version B-tree with lazy split algorithm and disabled sibling redistribution. Finally, *LS-MVBT* denotes the multi-version B-tree with metadata embedding, lazy split algorithm, and disabled sibling redistribution. All three schemes employ lazy garbage collection and use one reserved buffer cell for lazy split. We compare these variants of multi-version B-trees with TRUNCATE journal mode and WAL mode.

TRUNCATE mode yields the worst performance (60 ins/sec), which is well aligned with previously reported results [4]. Via merely changing the SQLite journal mode to WAL, we increase the query processing throughput (insertions/sec) to 416 ins/sec. Via weaving the crash recovery information into the B-tree, which eliminates the need for a separate journal (or log) file, and via disabling sibling redistribution, we achieve 20% performance gain against WAL mode. Via augmenting metadata embedding in MVBT, we achieve 50% performance gain against WAL mode.

Combining all the optimizations we propose together, (metadata embedding, lazy split, and disabling sibling redistribution), we are able to achieve 70% performance gain in an existing smartphone without any hardware assistance.

7 Conclusion

In this work, we show that lazy split multi-version B-tree (LS-MVBT) can resolve the Journaling of Journal anomaly by avoiding expensive external rollback journal I/O. LS-MVBT minimizes the number of dirty pages and reduces the Android I/O traffic via *lazy split*, *reserved buffer space*, *metadata embedding*, *disabling sibling redistribution*, and *lazy garbage collection* schemes.

The optimizations we propose exploit the unique characteristics of Android I/O subsystem: (i) write is much slower than read in the Flash based storage, (ii) dominant fraction of storage accesses are write, and (iii) there are no concurrent write accesses to database.

By reducing the underlying I/O traffic of SQLite, the lazy split multi-version B-trees (LS-MVBT) consistently outperforms TRUNCATE rollback journal mode and WAL mode in terms of write transaction throughput.

One future direction of this work is to improve LS-MVBT in order to support multiple concurrent write transactions. With the presented versioning scheme, modifications to B-tree nodes should be made in commit order. As multicore chipsets are widely used in recent smartphones, the need for concurrent write transactions would increase and multi-version B-tree should be improved to fully support concurrent write transactions.

[4] With 4 KB of node size, an internal tree page of SQLite can hold at most 292 key-child cells when the key is integer type while the maximum number of entries in leaf node is dependent on the record size.

Acknowledgement

We would like to thank our shepherd Raju Rangaswami and the anonymous reviewers for their insight and suggestions on early drafts of this paper. This research was supported by MKE/KEIT (No.10041608, Embedded System Software for New Memory based Smart Devices).

References

[1] M. Meeker, "KPCB Internet trends year-end update, Kleiner Perkins Caufield & Byers," Dec 2012.

[2] "Market share analysis: Mobile phones, worldwide, 3q13," *http://www.gartner.com/document/2622821*.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[4] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proceedings of the USENIX Annual Technical Conference (ATC 2013)*, 2013.

[5] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the 12th International Conference on Embedded Software (EMSOFT 2012)*, 2012.

[6] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *VLDB Journal*, vol. 5, no. 4, pp. 264–275, Dec. 1996.

[7] "Sqlite," *http://www.sqlite.org/*.

[8] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, 1992.

[9] M. C. Easton, "Key-sequence data sets on indelible storage," *IBM Journal of Research and Development*, vol. 30, no. 3, pp. 230–241, May 1986.

[10] D. Lomet and B. Saltzberg, "Access methods for multiversion data," in *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1989.

[11] P. J. Varman and R. M. Verma, "An efficient multiversion access structure," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 391–409, 1997.

[12] G. Kollios and V. Tsotras, "Hashing methods for temporal data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 902–919, 2002.

[13] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen, "Transactions on the multiversion B+-tree," in *the 12th International Conference on Extending Database Technology (EDBT '09)*, 2009.

[14] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proceedings of the 2nd USENIX conference on File and Storage Technologies (FAST)*, 2003, pp. 43–58.

[15] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[16] Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, 2009.

[17] G. Graefe, "A survey of B-tree logging and recovery techniques," *ACM Transactions on Database Systems*, vol. 37, no. 1, Feb. 2012.

[18] B. Sowell, W. Golab, and M. A. Shah, "Minuet: A scalable distributed multiversion b-tree," in *Proceedings of the VLDB Endowment, Vol. 5, No. 9*, 2012.

[19] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspension," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012.