BoLT: Barrier-optimized LSM-Tree

Dongui Kim, Chanyeol Park, Sang-Won Lee, Beomseok Nam
College of Computing
SungKyunKwan University

ABSTRACT

Key-value stores such as LevelDB and RocksDB are widely used in various systems due to their high write performance. However, the background compaction operations inherent to the key-value stores are often to blame for write amplification and write stall. In particular, the SSTable size in the existing key-value stores introduces, upon compactions, a trade-off between the `fsync()` call frequency and the amount of amplified writes. Small SSTables require a larger number of `fsync()/fdatasync()` than large SSTables to maintain file consistency. On the contrary, large SSTables result in large overlaps and frequent rewrites of SSTables. In this paper, to reduce file consistency overhead without increasing key ranges of SSTables, we present a variant of LSM-tree, namely, BoLT (Barrier-optimized LSM-Tree), that minimizes the number of calls to `fsync()/fdatasync()` barriers while taking advantage of fine-grained SSTables. BoLT consists of four key elements: (i) *compaction file*, (ii) *logical SSTables*, (iii) *group compaction*, and (iv) *settled compaction*. We implement BoLT in LevelDB and HyperLevelDB and compare the performances against LevelDB, HyperLevelDB, RocksDB, and the state-of-the-art PebblesDB. Our experimental study shows that BoLT achieves significantly higher write throughputs than LevelDB and HyperLevelDB.

CCS CONCEPTS

• **Information systems → Storage management**.

KEYWORDS

Log-Structured Merge Tree; Key-Value Stores;

1 INTRODUCTION

Key-value stores have become key components in numerous NoSQL systems that process large volumes of data [1, 1–5, 13, 17, 23, 23]. To efficiently store, retrieve, and manage large amounts of data, various key-value stores, such as LevelDB [4] and RocksDB [6], have employed Log-Structured Merge (LSM) tree [29]

What makes LSM-tree stand out from other indexing structures is that LSM-tree buffers and batches multiple write operations. Thereby, it transforms random writes into sequential writes and benefits from a higher sequential write throughput of block device storage systems. However, to enforce sequential write access patterns, LSM-tree continuously performs merge-sort operations, referred to as *compaction*, in a background thread. Since the compaction keeps rewriting the same key-value pairs from one file to another file multiple times, it results in high write amplification and suffers from storage wear issues. Besides, the compaction thread often blocks writers from inserting new records into an in-memory buffer space called MemTable until a background thread finishes compaction [20, 25, 30]. The so-called *write stall* problem is known to hurt the write throughput of LSM-tree significantly [8, 12, 21, 24–28, 30].

In the past decade, numerous research has been conducted to improve the efficiency of LSM-tree [8–10, 12, 16, 19, 21, 22, 24–28, 30, 32, 37]. Some of these works made efforts to improve memory efficiency rather than compaction mechanism [8, 16, 24, 28, 37]. However, due to the unmatched high latency of block device storage, the improved performance of memory components in LSM-tree is often limited by the write stall problem.

To reduce the write amplification and write stalls due to compactions, PebblesDB [30] allows overlaps on a particular range of keyspace while trading the search performance. Wisckey [25] and HashKV [12] propose to store values in a separate space so that it can keep the LSM-tree size small and the compaction does not copy large values repeatedly. NVMKV [26] proposes a hardware-oriented approach that exploits FTL (Flash Translation Layer) and atomic multi-block write operations. LSM-trie [36], that uses a trie structure as an index, sacrifices range query performance to mitigate the write amplification problem. However, none of these previous research has investigated the data barrier overhead of the compaction process. Although the default write size between barriers in LevelDB is moderately configured (about 2MB on average), the frequent `fsync()/fdatasync()` calls

MIDDLEWARE, ACM/IFIP Middlware 2020,

https://doi.org/

will cause disk bandwidth under-utilized, thus making the key-value stores to perform sub-optimally [20].

In this work, we propose *BoLT* (Barrier-optimized Lsm-Tree) that decouples SSTables from physical files by introducing *logical SSTables*. With the logical SSTables, the compaction thread can store multiple SSTables in a single physical file, thus minimizing the number of calls to `fsync()/fdatasync()`. In addition, decoupling of SSTables from physical files in BoLT will increase the write size between barriers without modifying key ranges of SSTables. We note that the proposed optimizations in BoLT are complementary to most of the previous efforts.

The main contributions of this work are as follows:

- First, BoLT creates a single file per compaction, which we call a *compaction file*. In legacy LSM-trees, the size of SSTables is statically fixed in the range of 2 MB and 64 MB. When compaction merges multiple SSTables, the result is split into multiple new SSTables and the expensive `fsync()/fdatasync()` has to be called for each SSTable. Using the compaction file that we propose, each compaction creates only one compaction file, and it requires to call only two `fsync()/fdatasync()` calls - one for the compaction file and the other for MANIFEST file.
- Second, we propose to decouple SSTables from physical files by employing *logical SSTables*. Although the compaction file reduces the number of calls to `fsync()/fdatasync()`, the key range of each compaction file can become large and it may result in large overlaps between compaction files. Since such a large overlap significantly increases the overhead of subsequent compaction operations, we decouple SSTables from compaction files, so that we do not lose the benefit of fine-grained SSTables.
- Third, BoLT employs *group compaction* to avoid frequent compactions. Frequent compactions not only increase the data barrier overhead but also reduce the queue depth of storage devices. To mitigate this problem, we select multiple victim SSTables independent of their associated physical files and merge-sort them in single compaction, which helps further reduce the number of calls to `fsync()/fdatasync()`.
- Fourth, BoLT avoids rewriting logical SSTables across multiple levels, if possible. Since BoLT has the freedom of using fine-grained logical SSTables, SSTable size can be chosen to be small enough to avoid key range overlap. In this *settled compaction*, non-overlapping SSTables settle down in their compaction files without being copied to a new compaction file. Instead of rewriting such non-overlapping SSTables during compaction, we increase the level of non-overlapping logical SSTables in the MANIFEST file to reduce the I/O traffic.

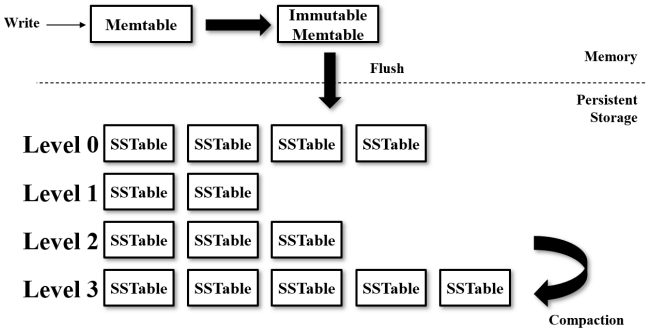


Figure 1: *LSM-Tree Architecture in LevelDB*

The rest of this paper is organized as follows. In Section 2, we present the background and motivation of BoLT. In Section 3, we present the design and implementation of BoLT and discuss how it decouples SSTables from physical files to reduce the data barrier overhead. In Section 4, we evaluate the performance of the proposed approaches against the state-of-the-art key-value stores. In Section 5, we discuss previous work that are relevant to ours. Finally, we conclude the paper in Section 6.

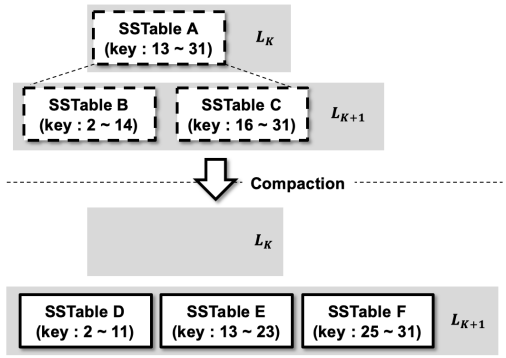
2 BACKGROUND AND MOTIVATION

2.1 Insertion into LSM-Trees

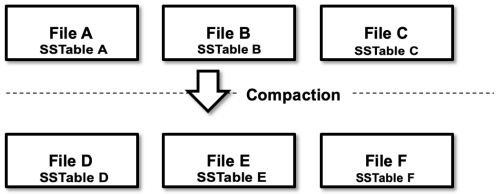
Figure 1 illustrates the architecture of LSM-tree implementation in LevelDB and RocksDB. LSM-tree-based key-value stores are designed for write-intensive workloads. To transform a burst of random writes into sequential writes, LSM-tree employs an in-memory buffer called *MemTable*. When the MemTable becomes full, the mini-batch of writes is sequentially flushed to disks in a sorted array file called *SSTable*. SSTables are organized into multiple levels, and the lowest level L_0 SSTables are gradually merge-sorted into upper-level L_k SSTables via a background *compaction* thread.

Buffering new key-value records in a volatile MemTable can result in loss of data upon system failure. Therefore, when a record is inserted, updated, or deleted in the MemTable, it is also written in a log file for recovery purposes. Since writing a log entry per insertion slows down the insertion throughput, multiple key-value pairs can be written together via *group commit* at the cost of compromising failure atomicity.

If the size of a MemTable reaches a threshold, the MemTable changes its state to *immutable* and a new MemTable is created so that the new MemTable can serve incoming clients' requests while a background thread is flushing the old immutable MemTable to disks. When flushing an immutable MemTable to disks, a background compaction



(a) Logical View of LSM-tree



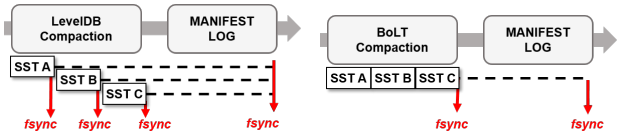
(b) Physical View of LSM-tree

Figure 2: *Compaction in LevelDB*

thread transforms it into SSTables. In most LSM-tree implementations, the MemTable is implemented as a SkipList, while an SSTable is a sorted array. Therefore, the compaction thread traverses a SkipList and transforms the SkipList into a sorted array and then flushes it as a file.

SSTables are organized in multiple levels, where each level consists of a set of SSTables. The combined size of SSTables at each level cannot exceed a predefined threshold, and the size limit grows exponentially per level. In LevelDB, the size limit of each level is multiplied by ten by default, i.e., 10 MB in level 1, 100 MB in level 2, and 1 GB in level 3, and so on. With such exponential growth of the size, LSM-tree limits its height in log scale. If the total size of SSTables at a certain level exceeds its limit, the compaction thread selects a victim SSTable from the overflowing level. Then, it merges the victim SSTable with other overlapping SSTables in the current and the next level.

In LevelDB, L_0 SSTables may overlap other L_0 SSTables in the same level 0. This is because a background compaction thread does not merge sort an immutable MemTable with existing L_0 SSTables, but flushes it from memory to disks. A rationale behind this design is that an immutable MemTable needs to be flushed to disks as fast as possible. Since a block device has a considerably higher latency than DRAM, a merge sorting with existing SSTables on disks will yield an unacceptably high latency. Since LSM-tree does not want to spend too much time on converting an immutable MemTable



(a) Data Barriers in LevelDB (b) Data Barriers in BoLT

Figure 3: *Data Barriers*

to an SSTable, LSM-tree defers the overhead of merge-sort to upper-level compaction, i.e., a compaction from level 0 to level 1 performs a merge sorting. Although LSM-tree allows only a few SSTables in level 0, compaction from level 0 to level 1 is the most expensive compared to higher levels due to large overlaps in level 0. In the worst case, the key range of a single L_0 SSTable may overlap all other L_0 SSTables and L_1 SSTables.

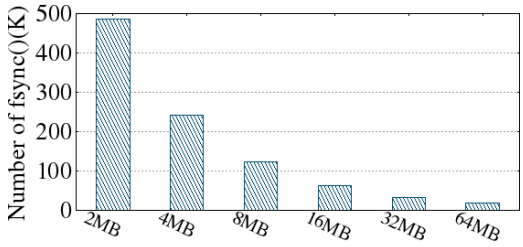
Figure 2 shows how LevelDB and RocksDB perform compaction. A background compaction thread selects a victim SSTable and merge-sorts it with overlapping SSTables in the next level. In the example, suppose we select a selected victim `SSTable A`, whose key range is (13–31). In the next level, we check which SSTables overlap the key range. Since the key ranges of `SSTable B` and `SSTable C` overlap, we merge-sort the key-values of three SSTables. After creating a large sorted array in DRAM, we partition the array, write new SSTables (`SSTable D`, `SSTable E` and `SSTable F` in the example). Since each SSTable is stored as a file, a compaction thread calls `fsync()/fdatasync()` for each file, as shown in Figure 3(a).

2.2 Write Amplification

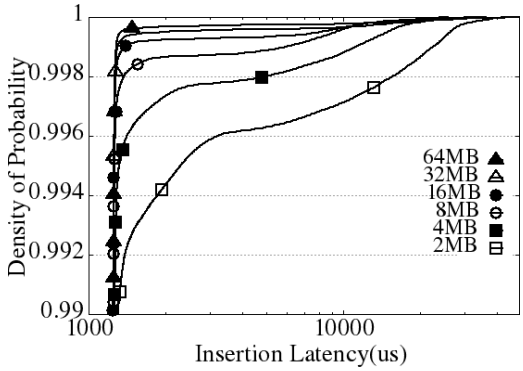
In flash memory and SSDs, it is known that the actual number of bytes written to the storage media is often much larger than the written data size because of its *erase* operation. This problem is referred to as the write amplification problem. Since LSM-tree stores data in a tiered data structure and a compaction thread copies key-value records from low-level to upper-level SSTables, the same key-value records are written multiple times, which aggravates the write amplification problem of SSDs. The high write amplification problem of LSM-tree not only hurts the write throughput but also causes the NAND wearing issues.

2.3 Write Stall

If compaction (disk I/O) cannot keep up with incoming write requests (memory copy), possibly due to a temporary write burst, the MemTable fails to buffer the requests. Then, subsequent write requests are blocked because of the MemTable size limit. In LevelDB, the foreground thread that handles users' write requests sleeps for 1 msec if there are more than



(a) Number of fsync() with various SSTable sizes



(b) Insertion tail latency with various SSTable sizes

Figure 4: Insertion performance with various SSTable sizes

8 SSTables in Level 0. This governor is called *L0SlowDown* trigger. If there are more than 12 SSTables in Level 0, the foreground thread is blocked, and it waits until the background compaction finishes and makes some space in level 0. This governor is called *L0Stop* trigger.

In recently proposed key-value stores such as HyperLevelDB and PebblesDB, such artificial blocking/slow down governors have been removed or weakened. By disabling or weakening artificial governors, HyperLevelDB and PebblesDB succeeded in improving the write throughput. However, they put more load on the system when a burst of write requests arrives in a batch, which may negatively affect the main application. Besides, with the removal of artificial governors, the number of SSTables in level 0 may increase infinitely due to the large latency gap between memory copy and storage writes. If there are too many SSTables in level 0, read queries will suffer because SSTables in level 0 overlap each other and read queries will have to access all of them.

2.4 Data Barrier Overhead

When SSTables are compacted, the output - a new sorted array of key-value records is created, partitioned, and stored as new SSTables files. To flush the updates, LSM-tree calls a data barrier (`fsync()/fdatasync()`) for each file to flush dirty

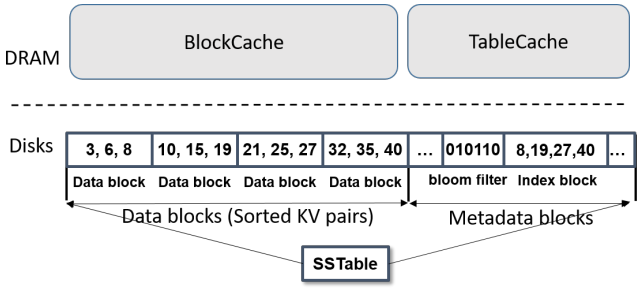


Figure 5: *SSTable and Metadata Caches in LevelDB*

pages from the file system cache. After flushing new SSTables, LSM-tree updates the MANIFEST file, which manages a list of valid SSTables and their key ranges. The MANIFEST file is a transactional log used to restore the previous consistent state of LevelDB. This file is necessary because Linux file systems do not consider the failure-atomic semantics of applications. I.e., when multiple disk pages are updated, the file system does not preserve the order of writing multiple dirty pages. To guarantee the failure-atomicity of each compaction, a background compaction thread flushes all dirty pages of new SSTables before it validates them by making changes to the MANIFEST file. That is, the MANIFEST file behaves as a commit mark for each compaction.

In LevelDB, the default SSTable size is 2 MB. Therefore, the average write traffic between barriers is about 2 MB on average. However, it has been reported that key-value stores, including LevelDB and RocksDB, fail to leverage the high bandwidth of modern SSDs [20]. It is because a transaction processing system frequently calls `fsync()/fdatasync()` barriers, which block the system until the queue depth becomes 0, i.e., until the number of all pending I/O requests in its block device storage drops down to 0. Since the barriers considerably hurt the concurrency level of I/O requests, barriers should not be used unless it is inevitable. However, current key-value store implementations are designed to heavily rely on `fsync()/fdatasync()` calls, which result in significant under-utilization of the storage bandwidth.

In the experiments shown in Figure 4 we measure the number of `fsync()` calls and the insertion tail latency using YCSB `Load A` (50 GB) workload while varying the size of SSTables in stock LevelDB. As we increase the SSTable size, the number of `fsync()` calls decreases linearly and the insertion latency improves, taking advantage of the reduced number of `fsync()` calls and avoiding the write stall problem.

2.5 Searching LSM-Trees

For read operations, LSM-trees look up volatile MemTables and then SSTables from the lower level to the upper level, i.e.,

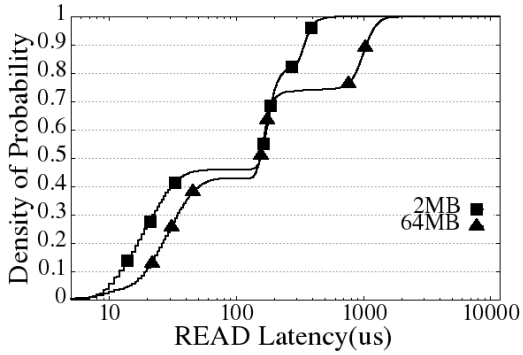


Figure 6: *Table Cache Eviction Overhead in RocksDB*

the recently stored records are searched first. LSM-tree relies on a two-level index for lookup operations. I.e., it stores a mapping table in the MANIFEST file that associates a partitioned key range with an SSTable file name. Using the mapping table in the MANIFEST file, a query filters out non-overlapping SSTables and reduces the number of SSTables to access in each level. In addition to the MANIFEST file, each SSTable has a bloom filter and index blocks to improve the search performance. Figure 5 shows an example of the SSTable format in LevelDB. Each SSTable contains a sequence of data blocks, where key-value records are stored sorted on the key, and metadata blocks that have an index and a bloom filter. The index block stores the largest key of each data block so that a binary search can locate a data block for a given query.

LevelDB and RocksDB have two in-memory caches - *TableCache* and *BlockCache*. BlockCache is used to cache recently accessed data blocks of SSTables. While the BlockCache stores key-value records, TableCache is used to cache index blocks and bloom filters. After identifying overlapping SSTables using the MANIFEST file, we check if the TableCache has metadata blocks of the SSTables. If they are not present in the cache, they are pulled from disks and cached in DRAM. With the bloom filter, a query determines the likelihood that its data is stored in each SSTable. If the bloom filter does not filter out an SSTable, it performs a binary search to determine which data block to access. The query then searches the BlockCache for the data block. If the data block is not found, we pull the block from disks, store it in the cache, and scan the block. If the requested record is found, the value is returned. If not, we look up the MANIFEST file for the next level and repeat the same steps.

2.6 Metadata Caching Overhead

Although we have shown that increasing the SSTable size helps improve the insertion performance, large SSTables may hurt the read performance due to the metadata caching overhead. To be specific, when the metadata for handling the given point query is missing from the memory cache, the metadata has to be fetched from the storage, the size of which is in proportion to the size of SSTables.

RocksDB is a fork of LevelDB optimized for a large number of CPU cores and faster storage devices. RocksDB uses larger SSTables to benefit from the high bandwidth of SSDs and reduce the number of files for very large databases. As such, the default SSTable size in RocksDB is 64 MB unlike LevelDB. However, we find large SSTables suffer from high *metadata caching overhead* due to large index blocks.

In particular, each query accesses a single 4 KB data block per SSTable. But, it has to access the entire index blocks of an SSTable if the index blocks are not present in the TableCache. Therefore, the cache miss penalty of TableCache is much higher than that of BlockCache. In LevelDB, the index block size for 2 MB SSTable is 30 KB. Note that the size of index blocks is proportion to that of SSTables. Thus, with 64 MB SSTables, the index block size of each SSTable becomes about 1 MB. I.e., each TableCache miss incurs 1 MB disk I/O.

In the experiments shown in Figure 6, we measure the point query latency with varying the SSTable size of RocksDB. We populated RocksDB with a 92 GB database and submitted 1 million point queries. In LevelDB and RocksDB, the TableCache size is determined by the number of SSTables, not bytes. For the experiments, we set the TableCache size (`max_open_files`) to 32,000.

Interestingly, even if 64 MB SSTables use 32 times larger cache space than 2 MB SSTables, Figure 6 shows that the tail latency of `64MB` is much higher than that of `2MB` because about 25% of queries suffer from a higher cache miss penalty. I.e., they need to read 32 times larger index blocks from disks and copy them to the TableCache.

If RocksDB uses smaller 2 MB SSTables with the same amount of memory for TableCache, i.e., we set the `max_open_files` to 32000, the metadata caching overhead is even further alleviated, and it significantly reduces the tail latency of read queries. However, as we discussed in Section 2.3, fine-grained SSTables increases the number of `fsync()/fdatasync()` calls and aggravates the write stall problem. That is, there is a trade-off between the write stall problem and the metadata caching overhead.

3 BOLT: DESIGN AND IMPLEMENTATION

In this section, we present BoLT (Barrier-optimized LSM-Tree). BoLT reengineers the LSM-tree implementation in LevelDB to decouple SSTables from files. By decoupling SSTables from files, BoLT reduces the data barrier overhead and mitigates the write stall problem. BoLT consists of four elements - i) *compaction file*, ii) *logical SSTable*, iii) *group compaction*, and iv) *settled compaction*.

3.1 Compaction File

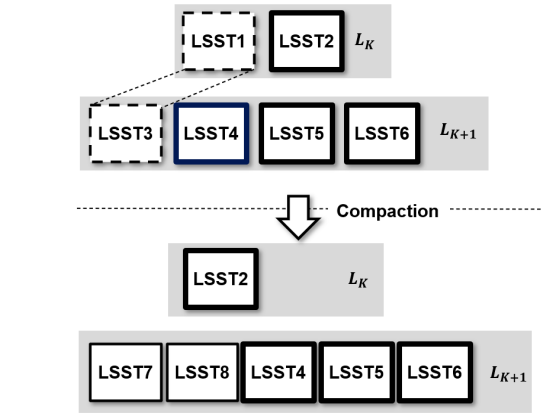
When a new file is created, `fsync()` must be called to flush dirty pages from the page cache to disks. To minimize the number of calls to `fsync()`, we need to reduce the number of files, i.e., SSTables. To reduce the number of files, compaction threads in BoLT do not store SSTables separately, but store them in a single physical file, which we call a *compaction file*. By creating a single file per compaction, BoLT minimizes the overhead of data barriers, as shown in Figure 3. Since each compaction merges a different number of SSTables depending on key range overlaps, the size of a compaction file is not static but determined by how many SSTables are merged by a compaction thread.

3.2 Logical SSTable

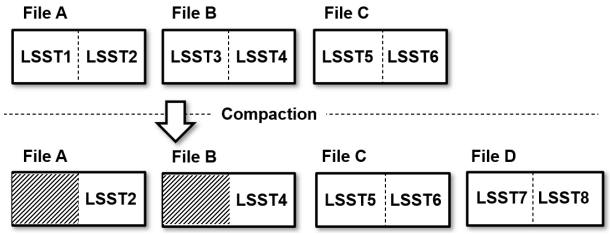
A drawback of a large compaction file is that it may aggravate the write amplification problem because the file size exponentially increases as a compaction thread keeps merging them in upper levels. That is, even if there is a small overlap in two compaction files, the entire key-value pairs from the two large compaction files need to be rewritten. Therefore, the compaction overhead and the write amplification increase. To mitigate the large key range overlap problem, BoLT partitions a large compaction file into small logical SSTables. SSTables do not need to be stored as an independent physical file. Even if we store multiple SSTables in the same compaction file, we can still perform lookup operations in the unit of logical SSTables if we know the their offsets in the file.

The MANIFEST file keeps track of the physical locations of logical SSTables. The MANIFEST file is log-structured, i.e., each compaction appends a set of deleted and created SSTables. In the stock LevelDB, the MANIFEST file stores a set of valid SSTables, their key ranges, and the file names for each level. To enable logical SSTables, the MANIFEST file in BoLT manages a set of compaction files and their logical SSTables along with the offsets. We note that the size of log entries is slightly larger than the legacy LevelDB because we store an offset of each SSTable, which is not necessary for legacy LevelDB. However, since the offset size is only 8-bytes, its performance effect and space overhead is negligible considering that SSTables are usually larger than a couple of megabytes.

In the example shown in Figure 7, BoLT stores two logical SSTables in compaction files - File A, B, and C. If a compaction thread selects `LSST1` as a compaction victim, it looks up the MANIFEST file to find out which SSTables in the next level overlap with it. Suppose `LSST3` overlap with `LSST1` in this example. The compaction thread i) reads key-value records from the two SSTables, ii) merge sorts them, iii) partitions



(a) Logical View of LSM-tree



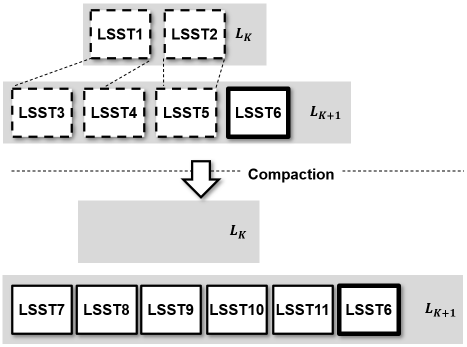
(b) Physical File Layout

Figure 7: *Barrier-optimized Compaction in BoLT*

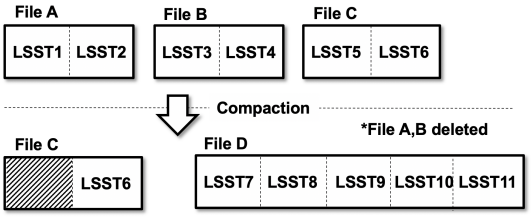
them into two logical SSTables (`LSST7` and `LSST8`), iv) and stores them in a new compaction file (`File D`).

After calling `fsync()` for the new compaction file (`File D`), we update the MANIFEST file to validate new logical SSTables (`LSST7`, and `LSST8`) and invalidate victim logical SSTables (`LSST1` and `LSST3`) atomically. Once the MANIFEST file is updated, subsequent queries will not access the invalidated logical SSTables. Therefore, we can safely delete the victim SSTables from physical files so that we can save disk space.

However, unlike stock LevelDB or RocksDB implementations, BoLT cannot simply unlink SSTable files since they are logical. In the example, `LSST1` is stored in `File A`, which has a valid SSTable `LSST2`. Instead of deleting or rewriting a compaction file, we punch a hole in the compaction file by calling `fallocate()` to reclaim unused disk blocks. We note that we do not call `fsync()/fdatasync()` when punching a hole because such a lazy metadata synchronization does not compromise the correctness nor the performance. We also note that the overhead of punching a hole in a compaction file is not significantly different from the overhead of deleting a physical SSTable file in stock LevelDB since both methods update the same amount of file system metadata.



(a) Logical View of LSM-tree



(b) Physical File Layout

Figure 8: *Group Compaction*

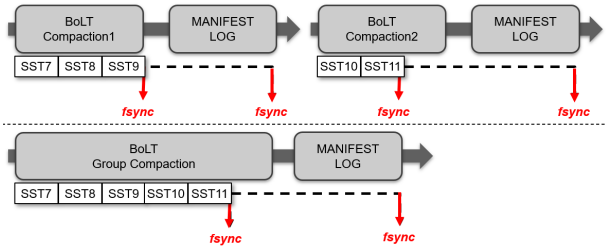
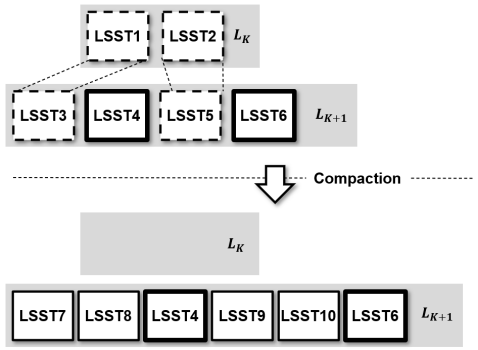


Figure 9: Data Barriers with Group Compaction

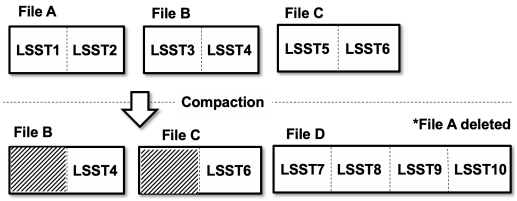
3.2.1 File Descriptor Cache. Since LevelDB and RocksDB manage a very large number of SSTable files, they avoid the overhead of filesystem metadata access by caching file descriptors of open SSTable files in TableCache. Since BoLT stores multiple logical SSTables in a compaction file, file descriptors are managed per compaction file rather than per SSTable. With large compaction files, the number of descriptors of open files in BoLT is much smaller than that of LevelDB and RocksDB. Hence, the filesystem metadata access overhead of BoLT is considerably reduced, as we will show in Section 4.

3.3 Group Compaction

Logical SSTables enable fine-grained compactions. Although fine-grained compactions help reduce the I/O traffic, compaction of small SSTables may result in frequent compactions



(a) Logical View of LSM-tree



(b) Physical View of LSM-tree

Figure 10: *Settled Compaction*

and eventually increase the I/O traffic and the data barrier overhead. Besides, the frequent small compactions fail to take advantage of sequential writes. To mitigate this problem, BoLT selects multiple victim logical SSTables and performs *group compaction*, i.e., it merge sorts multiple victim logical SSTables in single compaction.

Consider an example shown in Figure 8. When the size of level K (L_K) exceeds its limit, BoLT selects two victim logical SSTables instead of one. As such, a larger number of logical SSTables in the next level will participate in the compaction (`LSST1`~`LSST5` in the example). If we perform a compaction for `LSST1` and `LSST2` one by one, as shown in Figure 9, a compaction thread has to call `fsync()/fdatasync()` four times. However, if we perform a compaction for both `LSST1` and `LSST2` at the same time, the number of `fsync()/fdatasync()` decreases down to two.

Group compaction seems to be similar to the case when we use large SSTables as in RocksDB in that they take advantage of bulk writes by increasing SSTable sizes. However, our group compaction with logical SSTables is different from their approach in that group compaction does not increase the key ranges of SSTables. Thereby, the group compaction of logical SSTables in BoLT reduces the write amplification and the I/O traffic compared to RocksDB that uses large SSTables.

3.4 Settled Compaction

With fine-grained partitioning, the probability of having overlapping SSTables becomes lower than coarse-grained partitioning. Therefore, the group compaction may find overlapping logical SSTables are scattered in compaction files. For example, suppose a compaction thread selects LSST1 and LSST2 as victim SSTables as shown in Figure 10. Since LSST1 overlaps LSST3 and LSST2 overlaps LSST5, they are merge sorted and stored in a new compaction file (File D). Since LSST4 does not overlap LSST1 nor LSST2, LSST4 stays in its original compaction file (File C). Although we do not move LSST4 to a new compaction file, LSST4 can be located between new logical SSTables, i.e., LSST8 and LSST9 in the logical view of LSM-tree. This is because the MANIFEST file determines the logical view of LSM-tree. In particular, BoLT guarantees correct search results only if the MANIFEST file ensures that read transactions access logical SSTables in sorted order. I.e., the logical view of LSM-tree is independent of the physical layout of logical SSTables in compaction files.

With the group compaction and logical SSTables, BoLT allows us to choose any SSTables as victims even if they are not contiguous. Therefore, to further optimize the compaction performance, BoLT employs *settled compaction*, which is inspired by the *stitching* in VT-tree [33]. When the compaction thread has to select N logical SSTables in a certain level K to reduce the size of level K, the settled compaction selects N logical SSTables with minimal overlaps as victims. In the best case, the settled compaction may find non-overlapping logical SSTables. Then, we select those as victims and update the MANIFEST file to promote them to the next level. I.e., non-overlapping logical SSTables do not need to be copied to a new compaction file. If the selected victim logical SSTables are scattered across compaction files, the settled compaction i) merge sorts them with overlapping logical SSTables in the next level, ii) creates a new compaction file, iii) punch holes in their old compaction files, and iv) updates the MANIFEST file to validate the new logical SSTables.

4 EXPERIMENTS

4.1 Experimental Setup

We overhauled the LSM-tree implementations of LevelDB (v1.20) and HyperLevelDB, and replaced them with BoLT [1]. We evaluate the performance of BoLT and HyperBoLT against the stock LevelDB (v1.20), RocksDB (v.6.7.3), and the state-of-the-art PebblesDB [30].

PebblesDB is an extension of HyperLevelDB that shares the same goal with our work in that it employs *guards* to fragment a large sorted run into smaller chunks to mitigate the write amplification problem. I.e., PebblesDB avoids expensive compactions and improves the write performance by allowing overlapping SSTables at the same level. We note that the RocksDB code base is very different from LevelDB, and it employs various ad-hoc optimizations such as multi-threaded compaction, tiered compaction, disabled seek-compaction, and multi-threaded read. Since these optimizations are independent of BoLT designs, we can replace the LSM-tree implementation of RocksDB with BoLT to improve its performance. We leave the application of BoLT in RocksDB as our future work.

We run experiments on a workstation with two Intel Xeon E5-2620v4 CPUs (2.10GHz), 32GB of DRAM, and a Samsung 860 EVO 500G SSD. We set the memory size of the testbed machine to 8 GB using the kernel boot parameter mem. We reduce the memory size in consideration of the effect of file system page cache. The size of daily workloads has been reported to be about 500 GB in a production key-value store system [10], i.e., the volume grows 0.5 TB per day. For realistic performance evaluation with our relatively small workload size, which is about 1/10 the size of the data center workload, we reduce DRAM size proportionally, as was done in [10, 30].

As was done in previous studies [10, 30], we turned off compression optimizations in all key-value stores for ease of analysis. And we applied bloom filters to all key-value stores with 10 bloom bits, 1% of false-positive rate, as is commonly used in industry [15]. We set MemTable size to 64 MB for all key-value stores. As for the SSTable size, we use the default size for each key-value stores unless stated otherwise, i.e., 2 MB for LevelDB, 64 MB for RocksDB, and various sized SSTables for PebblesDB as suggested by [30]. As for BoLT, we set the logical SSTable size to 1 MB.

We use YCSB [14] workloads and four client threads for all experiments. YCSB consists of six realistic workloads. In each of Load A and Load E workload, denoted as LA and LE, we submit 50 million write requests with 23-byte keys and 1 KB values to fill the database with 50 GB of data. For the other workloads, we submit 10 million read/write queries. As recommended in [14], we submit YCSB workloads in the order of LA, A, B, C, F, D, delete database, LE, and E.

4.2 Quantification of BoLT Designs

To quantify the performance effect of each design in BoLT, we implemented four variants of BoLT in LevelDB and HyperLevelDB. +LS denotes the performance of BoLT when we use 1 MB logical SSTables and compaction files. +GC denotes the performance of BoLT when we enable the group compaction in addition to logical SSTables and compaction files. +STL denotes the performance of BoLT when we use the settled compaction along with the group compaction, logical

[1] Our implementation is available at https://github.com/DICL/BoLT.

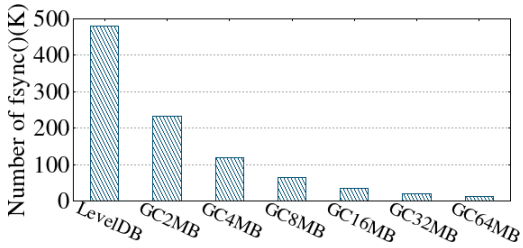


Figure 11: Number of fsync() calls

SStables, and compaction files. +FC denotes the performance with all optimizations enabled, including the file descriptor cache.

4.2.1 *Group Compaction Size:*

In the first set of experiments shown in Figure 11, we measure the number of `fsync()` calls using the write-only LA workload while varying the size of group compaction. The total number of `fsync()/fdatasync()` called by stock LevelDB is about twice higher than BoLT (GC2MB), which selects two 1 MB logical SSTables per compaction. Although LevelDB selects one 2 MB SStable and BoLT selects two 1 MB logical SSTables, i.e., the same number of key-value records as victims, BoLT reduces the number of `fsync()` calls since it calls `fsync()` only once for compaction file, whereas LevelDB calls `fsync()` for each 2 MB. As we increase the group compaction size, the number of `fsync()` calls decreases linearly and the write throughput improves. For the rest of the experiments, we set the group compaction size to 64 MB since it shows the best performance in the experiments.

4.2.2 *LevelDB Results:*

In the write-only LA and LE workloads, shown in Figure 12(a), +LS shows a similar performance with the stock LevelDB. Although we reduce the number of `fsync()` calls per compaction by decoupling SSTables from physical files, small compactions are triggered more frequently than the stock LevelDB. Therefore, the total number of calls to `fsync()` is not reduced, which hurts the write throughput. By enabling the group compaction, +GC reduces the number of `fsync()` calls and shows a 2.5x higher write throughput than the stock LevelDB for LA and LE. When we enable the settled compaction, +STL shows a higher write throughput because it does not rewrite non-overlapping logical SSTables and reduces the write amplification. The right small graph in Figure 12 shows the total number of bytes written by each configuration. +STL reduces the total disk I/O by 9.53% compared to the case when we disable it. +FC shows the performance of BoLT with the file descriptor cache. Although the file descriptor cache is a trivial optimization, it allows BoLT to avoid frequent file system metadata access, and its performance effect is as significant as other optimizations.

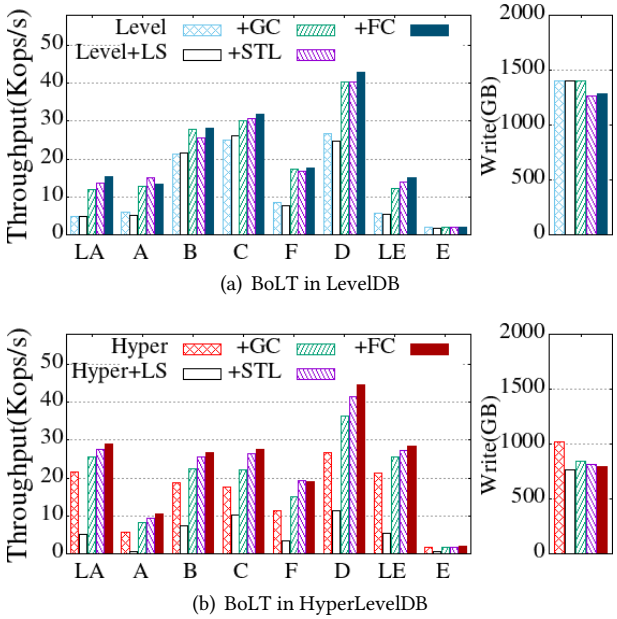
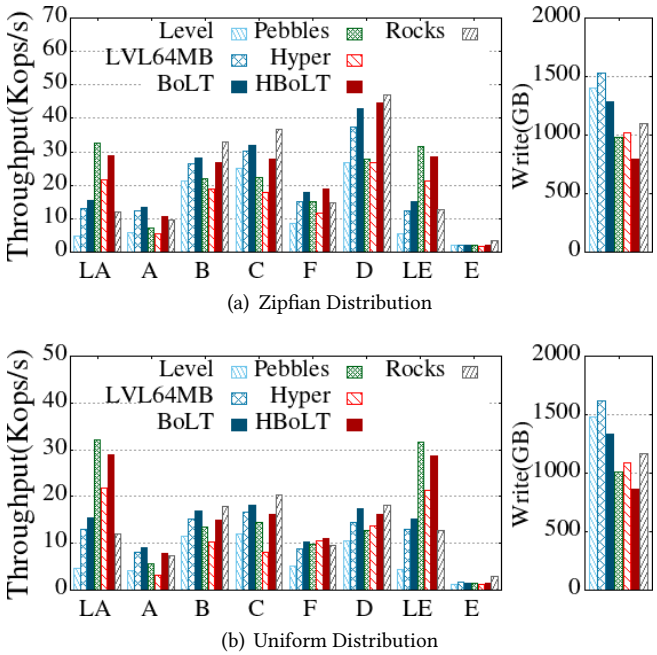


Figure 12: Quantifying the Benefits of BoLT Designs

While BoLT is designed primarily to improve the compaction performance, Figure 12(a) shows that BoLT improves the read performance as well. In particular, workload B is a read dominant 95% workload, and workload C is read-only workload. In those read-intensive workloads, BoLT outperforms the stock LevelDB for the following reasons. First, compaction reduces the number of overlapping SSTables across multiple levels. Since BoLT performs more compactions in the preceding write workload LA, read transactions access a smaller number of SSTables than the stock LevelDB. Second, LevelDB performs compactions even for read-only workloads. I.e., compaction is triggered if an SSTable is accessed more than a predefined number of times. LevelDB employs such *seek compaction* because compaction helps improve the read performance. Since BoLT improves the compaction performance, the read throughput also improves due to its low compaction overhead. Third, BoLT employs fine-grained logical SSTables, which considerably reduces the cache pollution and improves the read performance. While the legacy compaction pollutes the TableCache with large SSTables, as we described in Section 2.6, BoLT suffers from less cache pollution. This effect is most notable in workload D, where 95% of queries access recently inserted records. By enabling the group compaction and settled compaction, BoLT achieves a 51% higher throughput than the stock LevelDB.

4.2.3 *HyperLevelDB Results:*

We also evaluated the performance effect of BoLT in HyperLevelDB, as shown in Figure 12(b). Most of the results



(a) Zipfian Distribution

(b) Uniform Distribution

Figure 13: YCSB Throughputs **(Level: LevelDB, Hyper: HyperLevelDB, Pebbles: PebblesDB, Rocks: RocksDB, HBoLT: HyperBoLT)**

are similar to the results from LevelDB experiments except the performance of +LS. For most workloads, +LS shows the worst throughput. This is because SSTable size in HyperLevelDB is much larger than LevelDB, i.e., it ranges in between 16 MB ~ 64 MB. Therefore, HyperLevelDB triggers compactions less frequently than LevelDB. As a result, the write throughput of HyperLevelDB is about 4x higher than that of LevelDB in LA and LE workloads. In contrast, BoLT without the group compaction (+LS) triggers compaction per 1 MB logical SSTable. Although the fine-grained logical SSTables help reduce the number of overlapping SSTables and save disk IO by 25%, the increased number of calls to fsync() slows down compaction and hurts the write throughput.

It is also noteworthy that the throughput of +LS in HyperLevelDB is even lower than that of +LS in LevelDB for read-intensive workloads (workload B, C, D). This is because HyperLeveDB disables L0Stop governor, which blocks foreground threads if the number of SSTables in level 0 exceeds a threshold value. Without L0Stop governor, the number of overlapping SSTables in level 0 can be very large, which eventually hurts the read performance. Since +LS suffers from a large number of fsync() calls, it has a significantly larger number of logical SSTables in level 0. As a result, +LS shows the worst read performance.

However, if we enable the group compaction, the number of calls to fsync() decreases considerably. Therefore, +GC consistently outperforms HyperLevelDB for all workloads. When we enable all optimizations, BoLT shows up to 33% higher throughputs for write-intensive workloads and up to 56% higher throughput for read-intensive workloads (workload C) than HyperLevelDB.

4.3 Comparative Performance

4.3.1 Throughput:

In the experiments shown in Figure 13, we compare the performance of BoLT and HyperBoLT against the stock LevelDB, HyperLevelDB, RocksDB, and the state-of-the-art PebblesDB. For the experiments, we insert 50 million key-value records in each of LA and LE workloads. The key size of each record is 23 bytes and the value size is set to 1 KB for Figure 13(a) and 13(b).

We note that the SSTable size is one of the most significant differences between LevelDB, HyperLevelDB, and RocksDB. To evaluate the performance effect of SSTable size in LevelDB, we show the performance of LevelDB with 64 MB SSTables, denoted as LVL64MB. For both BoLT and HyperBoLT, we set the logical SSTable size to 1 MB.

With the increased SSTable size, LVL64MB shows 2.75x higher write thoughput than LevelDB in LA workload. Although LVL64MB aggravates the write amplification by 9% compared to the case when we use 2 MB SSTables, it benefits from less frequent compactions. While LVL64MB takes advantage of large SSTables, BoLT benefits from small logical SSTables, and it shows a 17% higher write throughput than LVL64MB in LA workload. This is because BoLT effectively decreases the total number of bytes written to disks by 16% and also reduces the number of fsync() calls. Compared to LevelDB, BoLT shows a 3.24x higher write throughput.

The write throughput of PebblesDB is even 2x higher than that of BoLT. This is because PebblesDB is a fork of HyperLevelDB and also because PebblesDB does not perform compactions even if there are overlapping SSTables at the same level. By avoiding compactions, PebblesDB improves the write throughput but sacrifices the search performance because a larger number of overlapping SSTables result in more SSTable accesses for read queries. We note that BoLT consistently outperforms PebblesDB in all other workloads, including write-intensive (50% write 50% read) workload A, except write-only LA and LE workloads.

While stock LevelDB uses a single mutex lock for all SSTables, HyperLevelDB allows multiple writer threads when inserting records. With the improved parallelism, HyperLevelDB shows up to a 65.9% higher write throughput than LVL64MB in workload LA. However, HyperLevelDB is outperformed by PebblesDB since HyperLevelDB does not allow overlapping SSTables except level 0 and perform compactions eagerly. It is noteworthy that the read throughput

of HyperLevelDB is also lower than that of PebblesDB. This is because PebblesDB uses much larger cache space than LevelDB and HyperLevelDB. While the SSTable size of HyperLevelDB ranges in between 16 MB and 256 MB, that of PebblesDB ranges in between 64 MB and 512 MB. Since the TableCache size in LevelDB and its variants is determined by the number of SSTables, not by the number of bytes, key-value stores benefit from a larger TableCache if they use larger SSTables. I.e., compared to 2 MB SSTables in LevelDB, 512 MB SSTables in PebblesDB use 256x larger TableCache space in the default configuration. Besides, PebblesDB requires a significant amount of memory space to keep bloom filters for all SSTables. While other LevelDB variants incurs disk I/O to access bloom filters of SSTables, PebblesDB does not access disks to read bloom filters. Due to those differences, PebblesDB often crashes if memory is insufficient. Taking advantage of larger cache space and a higher cache hit ratio, PebblesDB outperforms HyperLevelDB in read-intensive workloads.

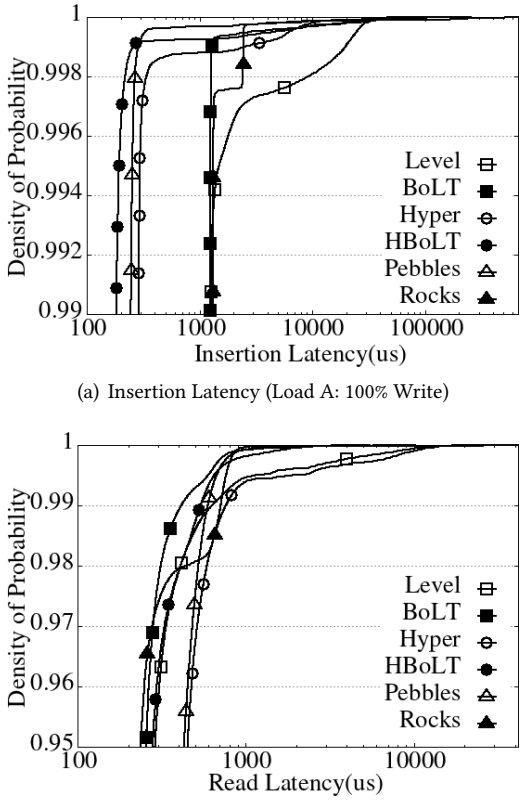
Although PebblesDB outperforms HyperLevelDB, HyperBoLT shows even higher read throughputs than PebblesDB because i) HyperBoLT does not allow overlapping SSTables at the same level, ii) the compaction performance of HyperBoLT is comparable to PebblesDB, and iii) HyperBoLT suffers less from cache pollution taking advantage of fine-grained logical SSTables.

The RocksDB codebase is very different from LevelDB and HyperLevelDB. While HyperLevelDB has superior write performance, RocksDB is known to have good read performance because RocksDB employs a highly concurrent thread synchronization mechanism that allows multiple read transactions to benefit from parallelism [18, 30]. With the optimized concurrency control, RocksDB outperforms HyperBoLT in terms of read throughput. We note that RocksDB's concurrency optimization is complementary to BoLT designs.

4.3.2 *Tail Latency:*

Figure 14 shows the 99th percentile and 95th percentile tail latency of workload `A` and `C` shown in Figure 13(a). BoLT shows insertion latencies lower than LevelDB and comparable to RocksDB up to the 99.5th pecentile because it reduces the compaction overhead. The tail latency of LevelDB, BoLT, and RocksDB is around 1 msec because they use `L0SlowDown` governor, which makes foreground write threads sleep for 1 msec if the number of SSTables in level 0 exceeds a threshold value.

Although BoLT outperforms LevelDB and RocksDB, HyperLevelDB shows a superior insertion tail latency due to the synchronization optimization of HyperLevelDB. HyperLevelDB also uses `L0SlowDown` governor, but it does not trigger it for 99.9% of insertions in this experiments due to its synchronization optimization.



(a) Insertion Latency (Load A: 100% Write)

(b) Read Latency (Workload C: 100% Read)

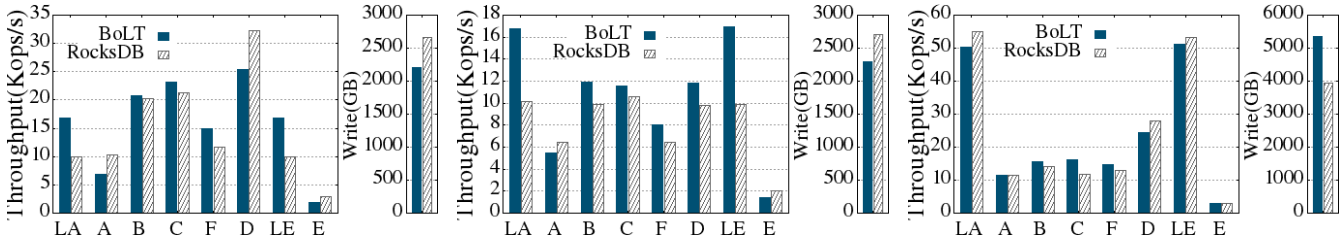
Figure 14: Tail Latency of Writes and Reads

Compared to HyperLevelDB and its variant PebblesDB, HyperBoLT shows a lower insertion tail latency up to the 99.85th percentile. However, it is noteworthy that PebblesDB outperforms HyperBoLT in the 99.9th percentile insertion tail latency because PebblesDB allows overlapping SSTables, which reduces the compaction overhead.

In terms of read tail latency shown in Figure 14(b), RocksDB, BoLT, HyperBoLT, and LevelDB show comparable performance up to the 97th percentile. However, we note that the 98th percentile read latency of RocksDB suddenly increases due to the cache pollution caused by large SSTables.

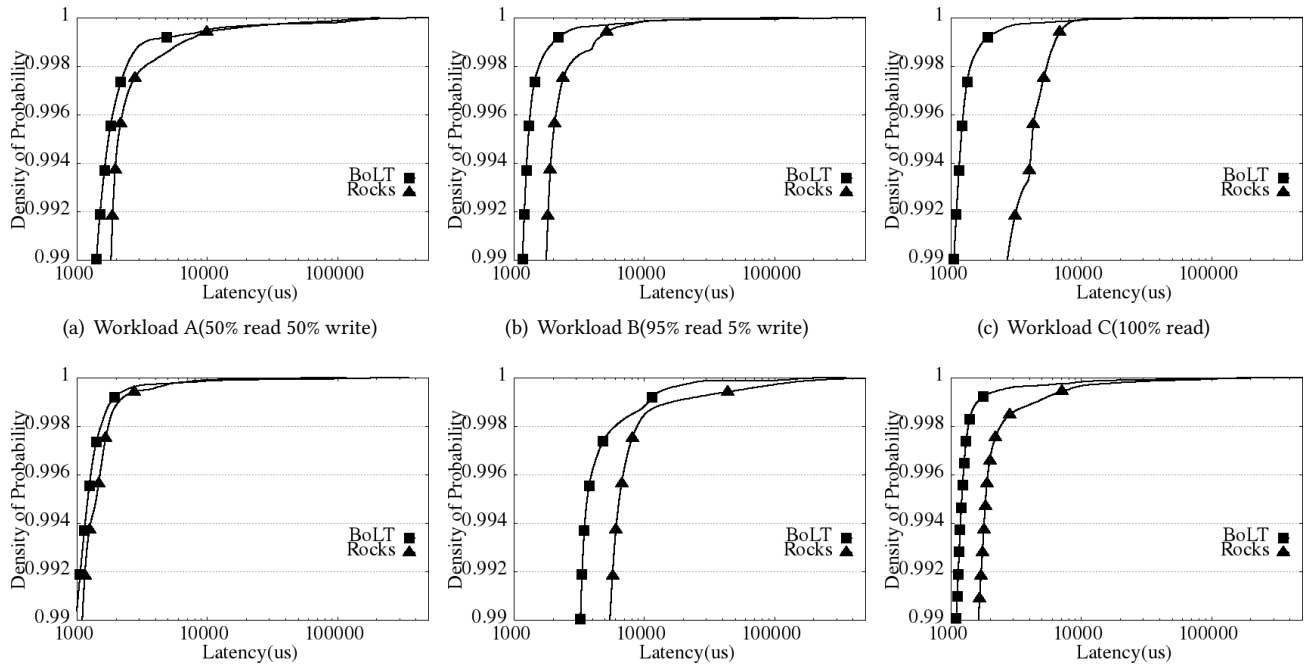
4.3.3 *TableCache Effect: Large vs. Small SSTables.*

As discussed above, efficient management of scarce memory space is desired when key-value stores manage terabytes of data. To evaluate the performance of key-value stores in an environment where memory is not sufficient for a database, we doubled the database size to 100 GB. In the experiments shown in Figure 15, we insert 1 billion 100-byte records, or 100 million 1 KB records into key-value stores in `LA` and `LE` workloads. With the increased database size, HyperLevelDB and its variants, i.e., PebblesDB, and HyperBoLT run out of



(a) 100 million 1 KB key-value records (Zipfian) (b) 100 million 1 KB key-value records (Uniform) (c) 1 billion 100 byte key-value records (Zipfian)

Figure 15: Throughput Comparison: BoLT vs RocksDB



(a) Workload A(50% read 50% write) (b) Workload B(95% read 5% write) (c) Workload C(100% read)

(d) Workload D(95% latest read 5% write) (e) Workload E(95% scan 5% write) (f) Workload F(50% read-modify-write 50% read)

Figure 16: Tail Latency Comparison: BoLT vs RocksDB

memory, and they become unresponsive. Therefore, in this large experiments, we compare BoLT only against RocksDB. While we used the default settings for each key-value store in the previous experiments, we modified the parameters of BoLT for fair comparisons in this experiment. In particular, we set the TableCache size of BoLT equal to that of RocksDB, and we set `L0SlowDown` and `L0Stop` to the default values of RocksDB, 20 and 36, respectively. We also set the maximum size of level 1 to the default value of RocksDB, i.e., 256 MB.

In write-only `Load A` and `Load E` workloads shown in Figure 15(a) and 15(b), where we insert 100 million 1 KB records, the write throughput of BoLT is up to 58% higher than that of RocksDB. It is noteworthy that Figure 15(c) shows the write throughput of RocksDB is better when we insert small records. This is because RocksDB's SSTable structure is more efficient than LevelDB's SSTable structure. In particular, when we store 100-byte key-value records, LevelDB variants, i.e., BoLT, HyperLevelDB, HyperBoLT, and PebblesDB require 223 bytes for each record whereas RocksDB requires only 141 bytes. I.e., LevelDB variants need 58% more disk space. However, if we store 1 KB key-value records, LevelDB variants require 1.138 KB, and RocksDB requires 1.057 KB for each record. I.e., the write amplification difference decreases to only 7%. Due to the SSTable format difference, RocksDB performs much fewer compactions than LevelDB variants when we insert small records. Figure 15(c) shows that the total number of written bytes in RocksDB is even smaller than that of BoLT. We note that the key elements of BoLT

are independent of SSTable format. If we integrate BoLT into RocksDB, we believe the performance of RocksDB will be further improved as if BoLT improves the write throughput of LevelDB.

Although the SSTable file format of BoLT is not as efficient as RocksDB, the read throughput of BoLT is higher or comparable to that of RocksDB for most workloads. This is because small logical SSTables of BoLT effectively reduces the metadata caching overhead discussed in Section 2.6. That is, the fine-grained logical SSTables in BoLT help leverage the insufficient TableCache more efficiently than RocksDB in two ways. First, smaller logical SStables pollute the cache less, thus yielding higher hit ratio. Second, upon cache miss, the penalty of reading the missed index block is less with smaller SSTables. In RocksDB, the index block size is 1MB while it is 30 KB in BoLT.

For workload `E`, BoLT is consistently outperformed by RocksDB because workload `E` is a range query workload and RocksDB benefits from larger SSTables. As Zipfian workload `D`, RocksDB benefits from its highly optimized caching mechanisms and the locality (95% latest read).

Figure 16 shows the tail latency CDF for the experiments shown in Figure 15. For all workloads, RocksDB shows higher tail latencies than BoLT although RocksDB uses a highly concurrent thread synchronization mechanism while BoLT use a single global lock for both reads and writes as in LevelDB. Again, this is mainly because of the overhead of reading large index blocks upon TableCache misses.

5 RELATED WORK

In the past decade, numerous studies have been investigated to reduce the overhead of compaction algorithms of LSM-trees [7, 9–12, 25, 30, 31, 34]. WiscKey [25] and HashKV [12] reduce the compaction overhead by storing values separately from keys. TRIAD [9] performs compactions only when the overlap of SSTables are sufficiently large. SlimDB [31] and PebblesDB [30] reduce the frequency of compactions by allowing overlapping SSTables on each level. SILK [10] proposes an I/O scheduler for LSM-based key-value stores to reduce the interference between client writes, flushes and compactions, thus avoiding latency spikes. bLSM [32] employs a "spring and gear" scheduler that throttles merge operations to ensure that compactions at each level make steady progress and they complete at the same time.

BoLT is different from these previous works in that BoLT points out the file system barrier overhead as the main cause of compaction overhead. To reduce the barrier overhead, BoLT redesigns the SSTable file format to enable logical SSTables. Instead of lowering the barrier overhead in the key-value store layer, we may reduce the barrier overhead in the file system layer. BarrierFS [35] is an order-preserving file system for modern flash storage. In legacy file systems for HDD, the write ordering can be enforced only by expensive `fsync()/fdatasync()` system calls. I.e., to enforce the write order, all the dirty blocks must be flushed although the durability is not required. To avoid such unnecessary flushes, BarrierFS introduced new system calls - `fbarrier()` and `fdatabarrier()`, to separate the ordering guarantee from the durability guarantee. With this separation, legacy key-value stores, including LevelDB and RocksDB, can reduce the number of calls to expensive `fsync()/fdatasync()` calls. Consider the example shown in Figure 3(a), where single compaction calls `fsync()` four times. In the example, only the last one is needed for the durability guarantee, and the third one is necessary only for the ordering guarantee. I.e., the commit mark (MANIFEST) needs to be written after data (SSTables) are written. With BarrierFS, we can reduce the number of `fsync()` calls as much as BoLT does. But BoLT not only reduces the number of `fsync()` calls but it also reduces the amount of written data using fine-grained logical SSTables and the settled compaction.

6 CONCLUSION

In this work, we present BoLT (Barrier-optimized LSM-Tree) that minimizes the data barrier overhead to resolve the write stall problem of LSM-trees. The root cause of the write stall problem is a significant difference in memory and block device storage latency. I.e., background compaction threads that perform disk I/O cannot keep up with foreground memory writes. To help background threads to keep up with faster foreground memory writes, BoLT effectively reduces the number of `fsync()` calls. By doing so, BoLT makes compaction threads run faster and reduces the number of SSTables accumulated in level 0. In our performance study, we show that BoLT improves the write throughput of LevelDB by 3.24x and that of HyperLevelDB by 1.44x.

ACKNOWLEDGMENT

We would like to give our special thanks to our shepherd Dr. Annette Bieniusa and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the R&D program of National Research Foundation of Korea (NRF) (grant No.2018R1A2B3006681, No.2016M3C4A7952587, and No.2018R1A2B2005502), Institute of Information & communications Technology Planning & Evaluation (IITP) (grant No. 2018-0-00549 and 2015-0-00314), NST (grant B551179-12-04-00), and ETRI R&D program (grant 19ZS1220) funded by Ministry of Science and ICT, Korea. The corresponding author is Beomseok Nam.

REFERENCES

[1] HBase. https://hbase.apache.org/.

[2] HyperLevelDB Performance Benchmarks. http://hyperdex.org/performance/leveldb/.

[3] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[4] Leveldb. https://github.com/google/leveldb.

[5] Oracle Berkeley DB. https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[6] RocksDB. https://rocksdb.org/.

[7] Muhammad Yousuf Ahmad and Bettina Kemme. Compaction management in distributed key-value datastores. *Proc. VLDB Endow.*, 8(8):850–861, April 2015.

[8] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache storage for the next billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 123–136, 2009.

[9] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2017.

[10] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 753–766, July 2019.

[11] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.

[12] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Optimal bloom filters and adaptive merging for lsm-trees. *ACM Trans. Database Syst.*, 43(4), December 2018.

[16] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36. ACM, 2011.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[18] Paul Dix. Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB. https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/.

[19] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, page 32. ACM, 2015.

[20] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pages 127–144. ACM, 2017.

[21] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of the 17th Usenix Conference on File and Storage Technologies (FAST)*. USENIX Association, 2019.

[22] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2018.

[23] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[24] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13. ACM, 2011.

[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[26] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2015.

[27] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[28] Suman Nath and Aman Kansal. FlashDB: Dynamic self-tuning database for nand flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 410–419. ACM, 2007.

[29] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[31] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow.*, 10(13):2037–2048, September 2017.

[32] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012.

[33] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[34] X. Sun, J. Yu, Z. Zhou, and C. J. Xue. Fpga-based compaction engine for accelerating lsm-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1261–1272, 2020.

[35] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 211–226, Oakland, CA, February 2018. USENIX Association.

[36] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2015.

[37] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST)*, 2005.