A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads

DEJUN TENG, The Ohio State University
LEI GUO, Google Inc.
RUBAO LEE, The Ohio State University
FENG CHEN, Louisiana State University
YANFENG ZHANG, Northeastern University, China
SIYUAN MA and XIAODONG ZHANG, The Ohio State University

LSM-tree has been widely used in data management production systems for write-intensive workloads. However, as read and write workloads co-exist under LSM-tree, data accesses can experience long latency and low throughput due to the interferences to buffer caching from the compaction, a major and frequent operation in LSM-tree. After a compaction, the existing data blocks are reorganized and written to other locations on disks. As a result, the related data blocks that have been loaded in the buffer cache are invalidated since their referencing addresses are changed, causing serious performance degradations.

To re-enable high-speed buffer caching during intensive writes, we propose Log-Structured buffered-Merge tree (simplified as LSbM-tree) by adding a *compaction buffer* on disks to minimize the cache invalidations on buffer cache caused by compactions. The compaction buffer efficiently and adaptively maintains the frequently visited datasets. In LSbM, strong locality objects can be effectively kept in the buffer cache with minimum or no harmful invalidations. With the help of a small on-disk compaction buffer, LSbM achieves a high query performance by enabling effective buffer caching, while retaining all the merits of LSM-tree for write-intensive data processing and providing high bandwidth of disks for range queries. We have implemented LSbM based on LevelDB. We show that with a standard buffer cache and a hard disk, LSbM can achieve 2x performance improvement over LevelDB. We have also compared LSbM with other existing solutions to show its strong cache effectiveness.

CCS Concepts: • **Information systems** → **Record and buffer management**; **Key-value stores**; • **Software and its engineering** → **File systems management**;

Additional Key Words and Phrases: LSM-tree, compaction, buffer cache

An earlier version of this article was published in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS'17)*.
This work has been partially supported by the National Science Foundation under grants OCI-1147522, CNS-1162165, CCF-1453705, CCF-1513944, CCF-1629403, and CCF-1629291 and a grant from the Louisiana Board of Regents LEQSF(2014-17)-RD-A-01.
Authors' addresses: D. Teng, R. Lee, Y. Zhang, S. Ma, and X. Zhang, Computer Science and Engineering Department, The Ohio State University; emails: teng.102@osu.edu, liru@cse.ohio-state.edu, zhangyf@mail.neu.edu.cn, ma.588@osu.edu, zhang@cse.ohio-state.edu; L. Guo, Google; email: leguo@google.com; F. Chen, Department of Computer Science and Engineering, Louisiana State University; email: fchen@csc.lsu.edu.

https://doi.org/10.1145/3162615

ACM Reference format:
Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. 2018. A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads. *ACM Trans. Storage* 14, 2, Article 15 (April 2018), 26 pages.
https://doi.org/10.1145/3162615

1 INTRODUCTION

With the rise of cloud computing in enterprises and user-centric Internet services, the volume of data that are generated and accessed continues to increase at a high pace. There is an increasing need to access user data that are created and updated rapidly in real time. In this article, we aim to develop an efficient storage engine and its implementation to serve both intensive reads and writes. We present an effective and low-cost variant of LSM-tree to accomplish our goal.

LSM-tree [19] was originally designed for high throughput transaction systems. It writes data to disk sequentially and keeps them sorted in multiple levels by merge operations. LSM-tree can achieve a high write throughput and conduct fast range query processing on hard disk drives. For these merits, LSM-tree has been widely used in big data systems by industries, such as Bigtable [7], HBase [3], Cassandra [2], and Riak [5], and is the de facto model for write-intensive data processing.

As shown in Figure 1, LSM-tree writes data in a sorted order in the hierarchy of multiple levels, where the top level is in DRAM, and the rest of levels are on the disk. This structure allows the data writing and sorting in a batch mode at different levels concurrently. As an LSM-tree level is full, the sorted data entries in that level will be merged into the next level. This process is also called *compaction*.

However, the impact to buffer cache was not considered in the original design of LSM-tree. As shown in Figure 1, data objects *a*, *b*, and *c* are requested by user application. The indices of LSM-tree help locate the disk blocks containing those objects, and then those disk blocks are loaded into the buffer cache. Any future queries conducted on those disk blocks will be served by the buffer cache directly to avoid expensive disk accesses. Two types of buffer caches can be used to cache the frequently visited data in an LSM-tree storage environment: *OS buffer cache* and *DB buffer cache*. The cached data blocks in both OS buffer cache and DB buffer cache are directly indexed to the data source on the disk. However, the OS buffer cache is also temporarily used to cache the data blocks read for compactions, while the DB buffer cache is not. As OS buffer cache is used, all data requested by conducting both queries and compactions will be temporarily stored in it. As the capacity of OS buffer cache is limited, the data requested by queries can be continuously evicted by the data blocks loaded by compactions. Thus, compaction operations may cause capacity misses in OS buffer cache. To avoid these interferences, an LSM-tree based database is implemented with an application level DB buffer cache to serve queries only [2, 3, 7, 12]. However, cached data in DB buffer can also be invalidated by compactions.

The compaction operations frequently reorganize the data objects stored on the disk and change mapping indices of many data objects including the ones in the DB buffer cache. As a result, affected data objects in the DB buffer cache are invalidated, which are called *LSM-tree compaction induced cache invalidations*, causing a high miss rate on the DB buffer cache as a structural problem. As shown in an example in Figure 2(a), (b), (c), and (d) are frequently requested objects, which belong to two LSM levels stored on the disk. Originally, the disk blocks containing these objects are kept in the DB buffer cache. However, when the two levels are compacted into a single level by LSM-tree, the compacted data are written to a new location on the disk. Thus, even though the content of these objects remains unchanged, the cached data blocks of these

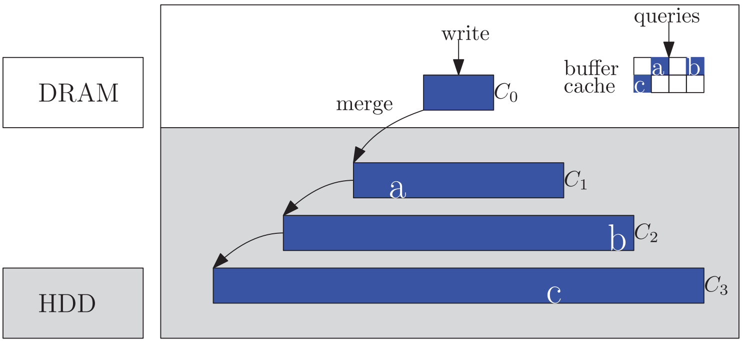


Fig. 1. Basic structure of an LSM-tree: (1) Data blocks in the buffer cache are directly indexed to the data source on the disk. (2) To make a fully sorted dataset in each level, compaction is timely conducted, which is also called "Eager" compaction.

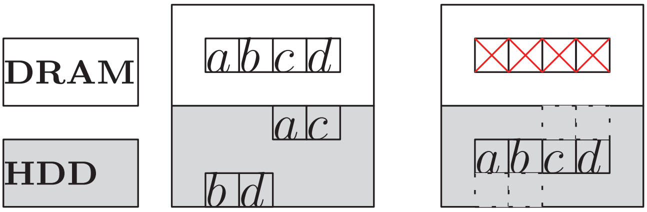


Fig. 2. An example of LSM-tree compaction induced cache invalidations.

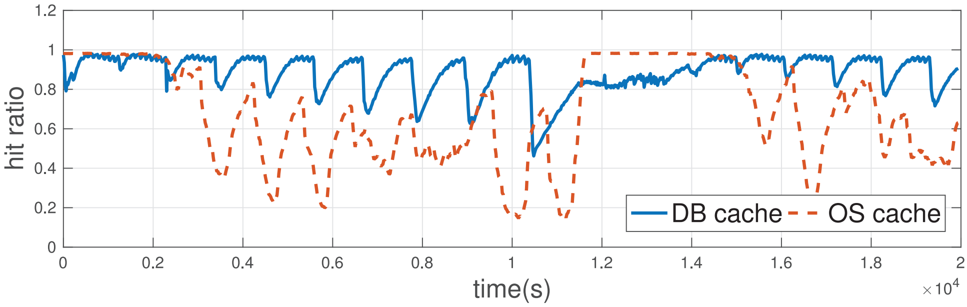


Fig. 3. Inabilities of buffer caches.

objects have to be invalidated, since the underlying disk data blocks have been moved. When those objects are requested again, the system has to load the new data blocks containing those objects from disk. With the changes of their referencing addresses, the access information of these objects is also lost. Even worse, since compaction writes are conducted in a batch mode, for workloads with high spatial locality, the corresponding DB buffer cache invalidations and data reloading occur in bursts, causing significant performance churns [1, 18].

We have conducted two tests to demonstrate the inabilities of the two buffer caches on LSM-tree. The read/write workloads of each test, and the experimental setup will be presented in detail in Section 6. Figure 3 shows hit ratios of the buffer caches (vertical bar) as time passes (horizontal bar) for workloads with both reads and writes. When only OS buffer cache is used (dashed line), the hit ratio goes up and down periodically. When compactions are conducted on the frequently visited

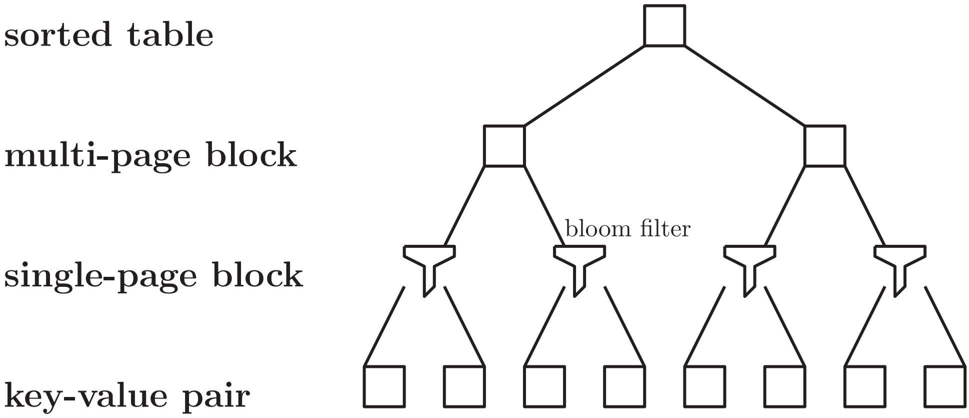


Fig. 4. The layered index structure of a sorted table.

data blocks, the data are pre-fetched into memory and hit ratio increases. However, those pre-fetched data are continuously evicted by the compactions conducted on the infrequently visited data blocks. When a DB buffer cache is used, the hit ratio also periodically goes up and down. The frequently visited data blocks that are cached in the DB buffer cache are invalidated due to reorganizing the data blocks on disks by compactions.

In this article, we propose *Log-Structured buffered-Merge tree* (LSbM-tree or LSbM in short), an efficient and low-cost LSM-tree variant with a new *buffered merge* compaction method. LSbM-tree improves the overall query performance by retaining the locality in the DB buffer cache for both read and write intensive workloads (the term"buffer cache" mentioned in this article without notation refers to the DB buffer cache).

The roadmap of this article is as follows. Section 2 presents the background knowledge of LSM-tree, the merge algorithm of LSM-tree, some other solutions and how we are motivated to propose our solution. Section 3 introduces our LSbM-tree design. Sections 4 presents the management of the compaction buffer. Section 5 presents the techniques to query an LSbM-tree. We present the performance evaluation of LSbM-tree and comparisons with other LSM-tree variants in Section 6. Finally, we overview the related work in Section 7 and conclude this article in Section 8.

2 MOTIVATION

In this section, we analyze the internals of LSM-tree compactions and show how compactions generate frequent data movement on disk, which in turn invalidates the data in the buffer cache. We also briefly introduce some other solutions and our own solution in this section.

2.1 LSM-tree

In an LSM-tree, data are stored on disks in multiple levels with increasing sizes. Figure 1 shows the structure of an LSM-tree with three on-disk levels. The data in each of those levels are organized as one or multiple sorted structures for the purpose of efficient lookups, which are called *sorted tables*. Each sorted table is a B-tree-like directory structure and is optimized for sequential disk access [19]. As presented in Figure 4, a sorted table builds a layered index structure on a set of *Key-Value pairs*. Continuous Key-Value pairs are packed in a *single-page block*, which maps to one single disk page. For each single-page block, a *bloom filter* [6] is built to check whether a key is contained in this block. Bloom filter is a space-efficient data structure for a membership check. A negative result means the element is not present in the block. However, a positive result only means a high probability that the element is present in the block associated with a false positive rate. Multiple continuous single-page blocks are packed into one unit called *multi-page block*. All data in a multi-page block are sequentially stored on a continuous disk region for efficient sequential data accesses. In practice, a multi-page block is implemented as a regular *file* [3, 12], which maps to a continuous disk region. A sorted table contains only the metadata of a set of files to serve queries. For simplicity, we will use the term *file* to stand for multi-page block, and *block* for single-block in this article.

Following the notations in prior work [19], we call the in-memory write buffer (level 0) C_0, the first on-disk level C_1, the second on-disk level C_2, and so on. Denote the number of on-disk levels of an LSM-tree as k, and denote the maximum sizes of $C_0, C_1, \ldots, C_k$ as $S_0, S_1, \ldots, S_k$, respectively. We call such an LSM-tree as a k-level LSM-tree. To minimize the total amount of sequential I/O operations on disk, the size ratio r_i between C_i and C_{i-1} ($r_i = \frac{S_i}{S_{i-1}}, 1 \leq i \leq k$), should be a constant for all levels, denoted as r [19]. We call such an LSM-tree as a *balanced LSM-tree*. A small size ratio r corresponds to a large number of levels, k.

Newly arrived data objects are initially inserted and sorted in C_0 which is in memory and then merged into C_1. When C_1 is full, its data will be merged to C_2, and so on. Only sequential I/O operations are involved while merging operations are conducted. In this way, data are written to disk in a log fashion, and continuously merged to keep the sorted structure, which reflected by the name *Log-Structured Merge-tree*.

2.2 Compactions

A conventional LSM-tree maintains a fully sorted structure on each level for efficient random and sequential data accesses [19, 20]. Compactions have to be conducted frequently to keep all data in each level sorted.

Let us consider that new data objects are inserted into C_0 of an LSM-tree with a constant write throughput w_0. For simplicity, we assume each level has the same key range, where keys are evenly distributed in the range, and all inserted data objects are unique. Initially C_{i+1} is empty. After compacting r full-sized (S_i) sorted tables from C_i, the size of C_{i+1} increases to the limit size S_{i+1}. Then full-sized C_{i+1} is merged into C_{i+2} and becomes empty. We call such a process one *merge round*. During one merge round, one chunk of data in the first sorted table from C_i needs not to be merged with any chunks of data, since C_{i+1} is empty, and one chunk of data in the second sorted table from C_i needs to be merged with one chunk of data in C_{i+1}, and so on. Finally, one chunk of data in the r_{th} sorted table from C_i needs to be merged with $r - 1$ chunks of data in C_{i+1}. On average, each chunk of data in C_i needs to be merged with $\frac{(1+2+\ldots(r-1))}{r} = \frac{(r-1)}{2}$ chunks of data in C_{i+1}. The average number of I/O operations to compact one chunk of data down to next level is $1 + \frac{(r-1)}{2} = \frac{(r+1)}{2}$. With k on-disk levels, the total disk write rate is $\frac{(r+1)}{2}kw_0$.

Therefore, when a chunk of data is written from the write buffer to disk, $\frac{(r+1)}{2}k$ chunks of data on the disk will be updated accordingly. As a result, the corresponding data kept in the buffer cache, if any, have to be invalidated, causing cold misses for later accesses and performance churns as we have presented in Section 1.

2.3 Existing Solutions

To effectively use buffer caching with LSM-tree, researchers have proposed and implemented several methods. We briefly introduce three representative solutions and their limits as follows.

Key-Value store cache: This solution is to build a key-value store in DRAM on top of the LSM-tree [2]. As shown in Figure 5, the Key-Value store cache is an independent buffer in memory

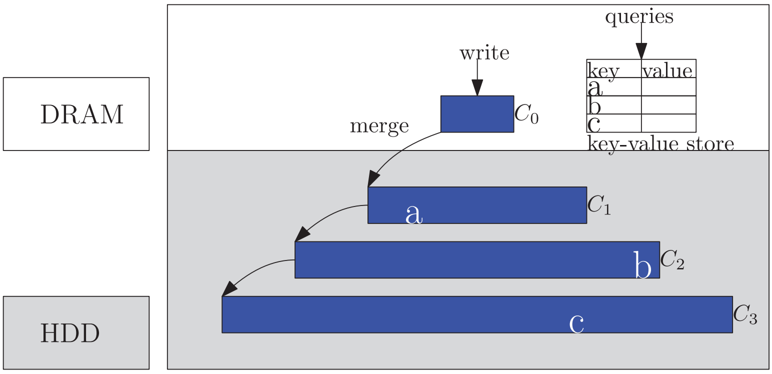


Fig. 5. Basic structure of an LSM-tree with a Key-Value store cache: (1) key-value store is an independent buffer cache in DRAM without any address index to the source on the disk. (2) “Eager” compactions are conducted to maintain a fully sorted LSM-tree.

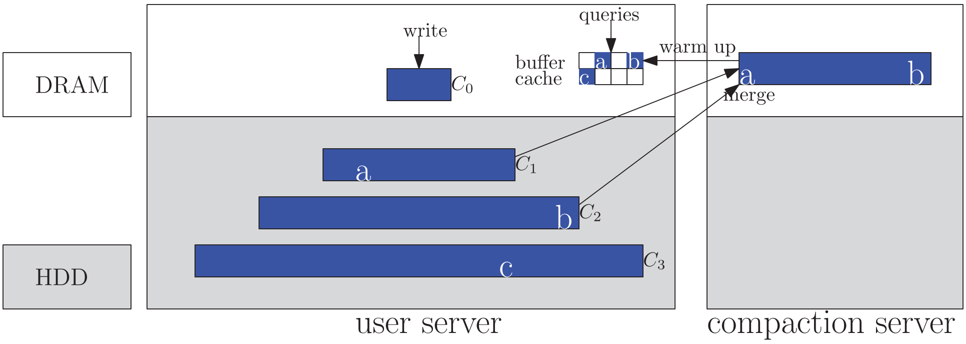


Fig. 6. Basic structure of an LSM-tree with a dedicated server to do compaction: (1) Recently compacted data blocks are preloaded to the buffer cache, assuming they have high locality. (2) “Eager” compactions are conducted to maintain a fully sorted LSM-tree.

without any address indexing to the data source on disks [25]. For a read access, it first checks the key-value store and hopes to have a fast access for a hit. Otherwise, it checks the LSM-tree, from where the data will be retrieved. This approach would reduce the amount of accesses to the LSM-tree or even bypass it. Since the in-memory key-value store is an independent buffer of the LSM-tree, it will not be interfered by the frequent reorganizations of the data in the LSM-tree. However, the key-value store would not be efficient for range queries and for accessing data with high spatial locality due to its nature of random stores and high efficiency for random accesses. Even though some Key-Value store data structures like skip-list or B+ tree support range queries, on-disk data still need to be loaded, since the in-memory Key-Value store cache may only contain partial dataset and it is independent on the on-disk data structure.

Dedicated compaction servers: This method is to use dedicated servers to conduct compactions, where the compacted datasets are kept in memory in these servers to be used to replace datasets in the buffer cache of users with an algorithm named *Incremental Warming up* algorithm [1]. This method attempts to reduce the LSM-tree induced misses by warming up the buffer cache in this way. As shown in Figure 6, all compactions are offloaded to a dedicated server, and after a compaction is conducted to merge C_1 and C_2 together, the newly generated blocks containing

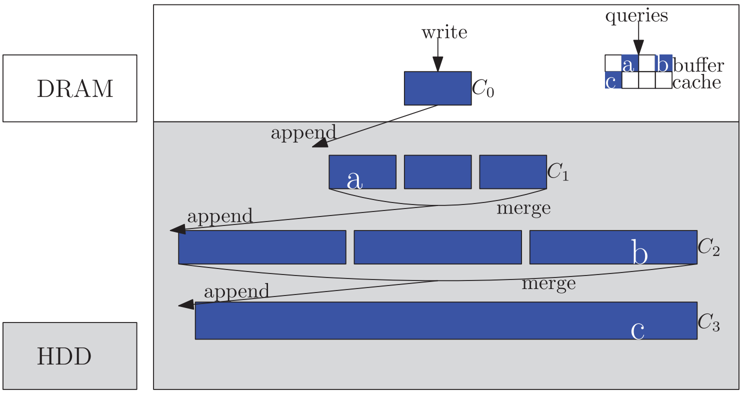


Fig. 7. Basic structure of an SM-tree: (1) Data blocks in the buffer cache are directly indexed to the data source on the disk. (2) Each level of LSM-tree is not fully sorted. This type of compaction is called “Lazy” compaction.

objects *a* and *b* then be used to replace the original data blocks in the buffer cache of the user server as a warming up process. The effectiveness of the dedicated server approach depends on the ability to identify specific compacted datasets with high locality. It is based on an assumption that newly compacted datasets would exhibit high locality of data accesses if they share common key ranges with data in the buffer cache. Having conducted experiments, we show that this assumption may not apply for workloads of whom not all reads lie in the hot range, thus, this approach may not be always effective.

Stepped-Merge algorithm (SM-tree in short): This algorithm is proposed to balance the tradeoff between compaction I/O and search cost for data warehouses [14]. Similarly to LSM-tree, SM-tree also organizes data into a multilevel structure of exponentially increasing sizes and compacts data with sequential I/Os. However, data objects in a level are not fully sorted and only be read out and sorted when they are moved to the next level. As shown in Figure 7, data blocks containing data objects *a*, *b*, and *c* are loaded in the buffer cache. When C_1 is full and the data in it are merged together with a merge sort. The merged data are then be appended to C_2 without merging with the data that already exist in C_2. As a result, the data block containing object *b* will not be invalidated. In fact, it will only be invalidated when C_2 becomes full and is sorted. Thus, the amount of compactions and the pace of cache invalidations can be reduced significantly. However, SM-tree may reduce the query performance in two ways. First, data in each level are not fully sorted, such that one range query conducted on one level may need multiple disk seeks. As a result, the range query performance of SM-tree is low. Second, for workloads with a large portion of repeated data, the entire database size can be unnecessarily large, since the obsolete data cannot be abandoned by compactions timely.

The goals of our work is to best utilize buffer cache for fast accesses of both random and range queries and to best utilize the merit of disk for long sequential accesses of range queries. The goals should be accomplished under the basic LSM-tree structure.

2.4 Our Solution

The basic idea of LSbM-tree is to add an on-disk compaction buffer to minimize frequent cache invalidations caused by compactions. The compaction buffer directly maps to the buffer cache and maintains the frequently visited data in the underlying LSM-tree but is updated at a much

Table 1. Parameters Explanation

variable	description
t_i	latency of checking the index of one sorted table
t_b	latency of checking the bloom filter of one block
t_m	latency of reading one block from buffer cache
t_s	latency of one disk seek
t_d	latency of reading one block from disk
f	fault rate of the bloom filter test
h	percentage of blocks read from buffer cache
n	number of sorted tables checked by one query
b	number of blocks read out by one range query

lower rate than the compaction rate. LSbM-tree directs queries to the compaction buffer for the frequently visited data that will be hit in the buffer cache and to the underlying LSM-tree for others including long-range queries. In short, using a small size of disk space as compaction buffer, LSbM-tree achieves a high and stable performance for queries by serving frequently data accesses with effective buffer caching, while retaining all the merits of LSM-tree for write-intensive workloads.

We have implemented a prototype of LSbM-tree based on LevelDB [12], a widely used LSM-tree library developed by Google, and conducted extensive experiments with Yahoo! Cloud Serving Benchmark (YCSB) [9]. We show that with a standard buffer cache and a hard disk storage, our LSbM-tree implementation can achieve 2x performance improvement over LevelDB and significantly outperforms other existing solutions.

3 SYSTEM DESIGN

The interference from compactions to buffer cache in LSM-tree is caused by the re-addressing of cached objects on the disk. Considering the root of this problem, we aim to control and manage the dynamics of the compactions of an LSM-tree. Ideally, the negative impact to buffer cache caused by compactions can be minimized at a very low cost.

Before introducing how LSbM-tree effectively keeps buffer cache from being invalidated by compactions, let us consider why LSM-tree needs to maintain a sorted structure on each level. In an LSM-tree, any queries will be conducted level by level. There may exist multiple versions of the same key at different levels, of which only the newest version is valid. Since the upper level contains more recent data, queries are conducted from upper levels to lower levels. The data in each level may belong to one or multiple sorted tables. The key ranges of those sorted tables may overlap with each other, and thus all of them need to be checked separately. For random data access, data object will be returned immediately once a matched key is found. Even though bloom filter can help avoid unnecessary block accesses, when the number of sorted tables in each level increases, the overhead of checking bloom filters and reading false blocks caused by false bloom filter tests becomes significant. For range query, all sorted tables will be searched and all data objects covered by the requested key range will be read out and merged together as the final result. Querying one level with multiple sorted tables may need multiple disk seeks. Therefore, the number of sorted tables in each level needs to be limited.

The latencies of one random access and one range query can be estimated by Formulas (1) and (2), and the parameters in those formulas are explained in Table 1. As shown in Formula (1), the latency of one random access comes from two types of operations: checking the indices and bloom filters to locate the data block containing the requested data object and reading the target data block. On average, one index check and one bloom filter check are needed to check one sorted table. Thus the latency of checking all sorted tables is $n \times (t_i + t_b)$. Among all the retrieved blocks, h, the hit ratio, of 1 data block is retrieved from the in-memory buffer cache, and the latency for this part is $h \times t_m$. The rest $1 - h$ of 1 data block are retrieved from disk, which needs one disk seek t_s and one disk block read t_d, and the latency for this part is $(1 - h) \times (t_s + t_d)$. Furthermore, a positive bloom filter test does not guarantee that the target object can be found in the data blocks whose bloom filter test is passed, $(n - 1) \times f$ unnecessary block accesses might be involved, and $1 + (n - 1) \times f$ blocks are read in total. As shown in Formula (2), the latency of one range query also comes from two types of operations: checking the indices to locate the blocks covered by the requested key range and reading those blocks from the buffer cache or the disk. For simplicity, we assume that exactly one disk seek is needed to retrieve the data in one sorted table covered by the requested key range, and all the requested blocks of one query can only be loaded from either the buffer cache or the disk. For range query, one index check is needed for each sorted table and bloom filter cannot be used to skip sorted tables, such that the latency of locating disk blocks in all sorted tables is $n \times t_i$. For the queries of which the requested blocks can be found in the buffer cache, the latency of loading all requested blocks is $h \times b \times t_m$. For the queries of which the requested blocks can only be retrieved from the disk, besides the latency to load b disk blocks from the disk, one disk seek is also needed for each checked sorted table. Thus the total latency is $(1 - h) \times (n \times t_s + b \times t_d)$,

$$Latency_{random} = n \times (t_i + t_b) + (h \times t_m + (1 - h) \times (t_s + t_d))(1 + (n - 1) \times f), \tag{1}$$

$$Latency_{range} = n \times t_i + (h \times b \times t_m + (1 - h) \times (n \times t_s + b \times t_d)). \tag{2}$$

In reality, especially for a hard disk drive, the latency of a disk seek, t_s, can be orders of magnitudes larger than the latencies of any other operations, such as the latencies of reading one block from disk (t_d) and memory (t_m), and the latencies of checking in-memory indices (t_i) and bloom filters (t_b). Thus the latencies of those operations can be negligible. Furthermore, when the requested range of one range query is relatively small, the latencies of loading b blocks from cache ($b \times t_m$) or disk ($b \times t_d$) can also be negligible. Therefore, when the values of $t_i, t_b, t_m, t_d, b \times t_m$, and $b \times t_d$ are all set to 0, the throughput, which is the reciprocal of the latency, of random accesses and range queries can be approximately calculated by Formulas (3) and (4),

$$Throughput_{random} \approx \frac{1}{(1 - h) \times t_s \times (1 + (n - 1) \times f)}, \tag{3}$$

$$Throughput_{range} \approx \frac{1}{(1 - h) \times n \times t_s}. \tag{4}$$

Formulas (3) and (4) show a tradeoff between a highly sorted structure and less sorted structure. Queries can benefit from a fully sorted structure of an LSM-tree level, i.e., a smaller n in Formulas (3) and (4). However, maintaining such a fully sorted structure needs to frequently conduct compactions, which may in turn invalidate the cached data objects, i.e., decrease the hit ratio, h, in Formulas (3) and (4). This tradeoff motivates us to find a way to maintain a fully sorted structure of the LSM-tree and, at the same time, keep the buffer cache from being invalidated by compactions. The basic idea of our design is to keep two data structures on disk. One data structure contains the whole dataset, and the data in each level are fully sorted as a conventional LSM-tree. We call it the *underlying LSM-tree.* Another data structure contains only the frequently visited data, and the data in it are not frequently updated by compactions. We call it the *compaction buffer.*

Figure 8 shows the structure of a three-level LSbM-tree. The left of the figure is the *underlying LSM-tree*, which consists of four levels C_0, C_1, C_2, and C_3. The middle box is the *buffer cache* in

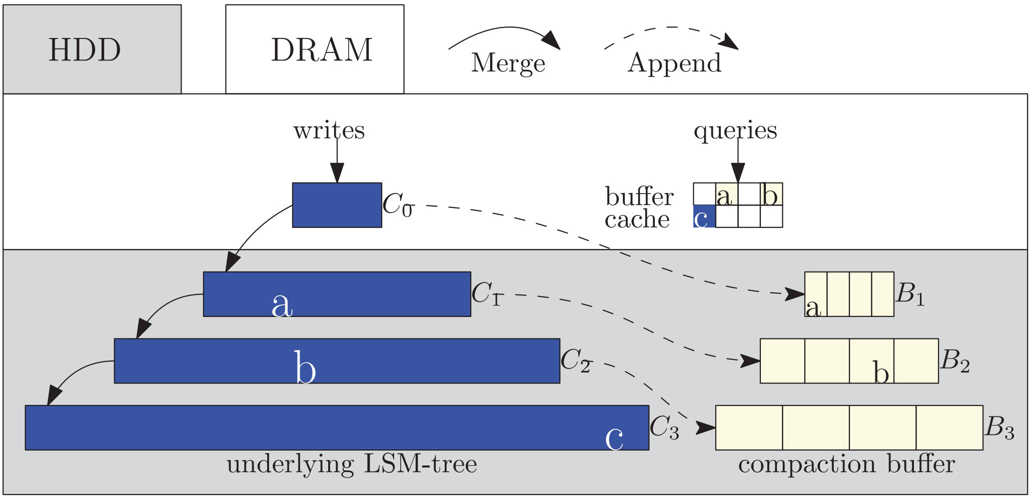


Fig. 8. The basic structure of an LSbM-tree.

DRAM, which is used to cache data requested by conducting queries. The *compaction buffer* is in the right corner of the figure, which is on disk. Data in the compaction buffer are also stored as a multiple level structure. The data in each level consist a list of sorted tables. We call each list a *compaction buffer list*. We denote the compaction buffer list of level i as B_i, and the jth sorted table in B_i as B_i^j. B_i^0 contains the most recent data in B_i. Figure 8 shows a compaction buffer with three compaction buffer lists. $B_i (1 \leq i \leq 3)$ contains frequently visited datasets of level i but is updated in a very low rate to enable effective caching.

In level i of an LSbM-tree, read requests are dispatched to B_i for frequently visited data that are highly possible to be hit in the buffer cache and to C_i for others. As shown in Figure 8, objects a, b, and c are requested. Among them, a and b are read from the buffer cache, which are indexed in B_1 and B_2, respectively. Object c is loaded from C_3 of the underlying LSM-tree.

4 COMPACTION BUFFER

The compaction buffer maintains only the frequently visited data that are not frequently updated by compactions. Thus, the compaction buffer needs to be able to selectively keep only the frequently visited data in it and maintains the stability of them. In this section, we present our *buffered merge* algorithm that associates with LSM-tree to prepare data for the compaction buffer, and a *trim process* to keep only the frequently visited data in the compaction buffer. The disk space used by compaction buffer is limited and there's no additional I/O cost for the compaction buffer construction. Next, we will present the detailed operations to build the compaction buffer.

4.1 Buffered Merge

Figure 9 presents an illustration of how a buffered merge works. When C_i is full, all its data will be merged into C_{i+1} with a merge sort. At the same time, it will also be appended into B_{i+1} as B_{i+1}^0. Note that B_{i+1}^0 is built with the files of C_i which already exist on disk. Therefore, no additional I/O is involved. The data in B_{i+1}^0 which were formerly in C_i will not be updated by compactions. Furthermore, B_{i+1}^0 contains all the data in B_i but are fully sorted, thus it becomes a better candidate to serve queries compared to B_i. As a result, the sorted tables in B_i become unnecessary, which will be removed from the compaction buffer and then deleted from disk.

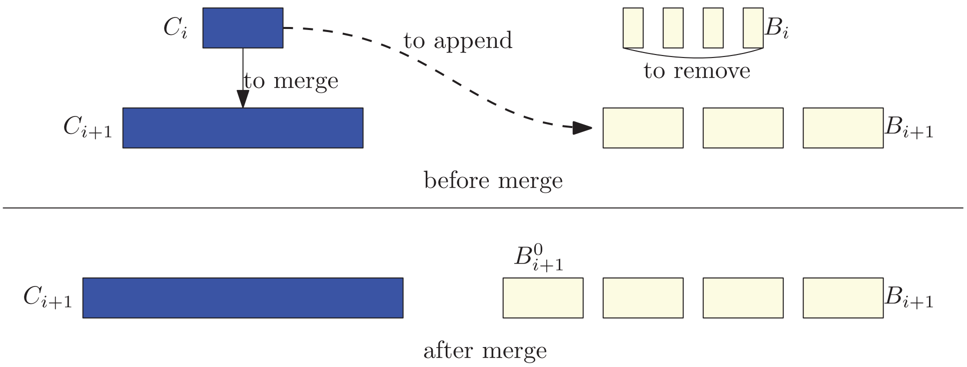


Fig. 9. An illustration of the buffered merge.

When a chunk of repeated data are written into level $i + 1$, the same amount of obsolete data in C_{i+1} are abandoned by compactions. However, the counterparts of those data in B_{i+1} also become obsolete but cannot be abandoned, since the data in a compaction buffer will not be compacted once appended. Those obsolete data take extra disk and memory space. Therefore, a compaction buffer list is not suitable for levels with repeated data inserted. In LSbM-tree, the existence of repeated data in level $i + 1$ can be easily detected by comparing the size of the data compacted into C_{i+1} and the size of C_{i+1} itself. If the size of C_{i+1} is smaller than the data compacted into it, then there must exist repeated data. When repeated data are detected, LSbM-tree freezes B_{i+1}. When C_{i+1} becomes full and is merged down to next level, B_{i+1} is unfrozen and continues serving as the compaction buffer list of C_{i+1}.

With the buffered merge, two data structures are created on level i: C_i and B_i. Data in C_i are fully sorted but are frequently updated by compactions, while data in B_i are not updated frequently but are not fully sorted. Frequently accessed data blocks in buffer cache are directly indexed in B_i, where infrequent update prevent data blocks in the buffer cache from being invalidated. In this way, the compaction buffer works as a buffer to hide the severe buffer cache invalidations caused by conducting compactions on the underlying LSM-tree. This leads to its name, the *compaction buffer*, and also the name of the merge algorithm, *buffered merge* algorithm.

When C_i is full and merged into C_{i+1}, data in B_i are deleted and the reads served by B_i are transferred to B^0_{i+1}. Note that B_i contains the frequently visited data and most of its data blocks are loaded in the buffer cache. A sudden deletion of B_i will cause a dramatic hit ratio drop and performance churn. To slow down the transfer process, we develop the buffered merge algorithm associated with a *gear scheduler merge algorithm* first introduced in bLSM-tree [20]. In bLSM-tree, data in level $i (0 \leq i < k)$ belong to two sorted tables, C_i and C'_i. When C_i is full, its data will be moved to C'_i and start to be merged into C_{i+1}. Meanwhile, C_i becomes empty and continues to receive data merged down from level $i - 1 (i > 0)$ or inserted by user applications (level 0). In each level, bLSM-tree defines two parameters, $in_{progress}$ and $out_{progress}$, to regulate the progresses of merging data into C_i and moving data out from C'_i. For simplicity, we simplify this regulation by fixing the total size of C_i and C'_i as a constant, which is the maximum size of level i, S_i. As the left side of Figure 10 shows, when a chunk of data are compacted into C_i, the same amount of data in C'_i are compacted into C_{i+1}. As a result, when C_i is full, C'_i must be empty, and the data in C_i can be moved to C'_i and start over. With this design, the compactions conducted on one level are driven by the compactions conducted on its upper level and eventually driven by the insertion operations conducted on the write buffer, C_0. As a consequence, data can be inserted into C_0 with a predictable latency [20].

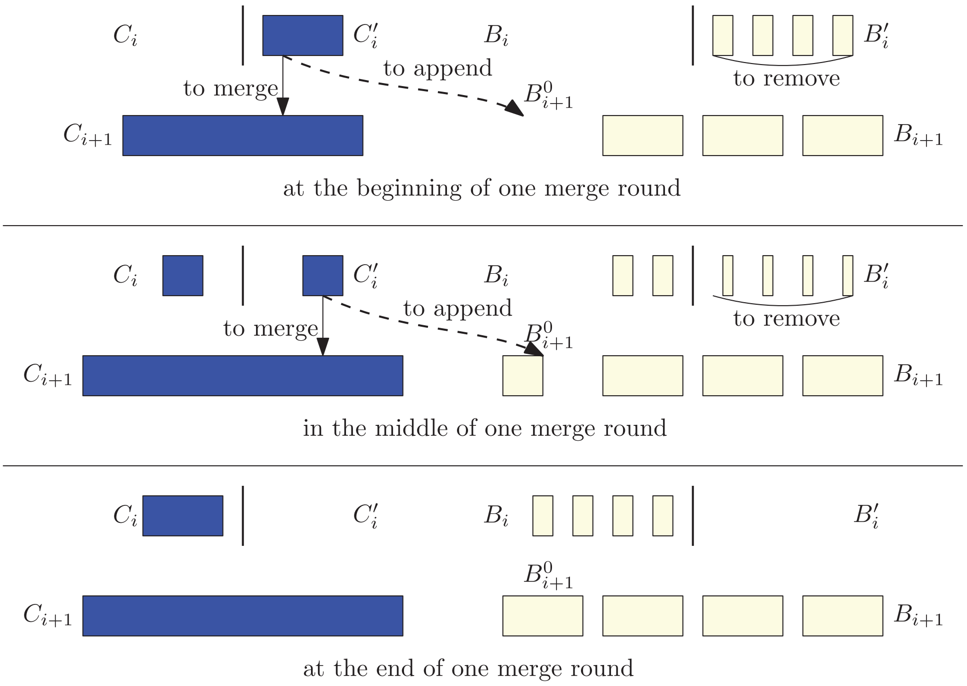

C_i
C'_i
B_i
B'_i
to merge
to append
B^0_{i+1}
to remove
C_{i+1}
B_{i+1}
at the beginning of one merge round
in the middle of one merge round
at the end of one merge round

Fig. 10. The detailed operations of the buffered merge.

We adapted this merge algorithm in our buffered merge operations. As shown in Figure 10, the ith$(0 \leq i < k)$ level of the compaction buffer is also divided into two parts: B_i and B'_i. Algorithm 1 describes the buffered merge algorithm based on bLSM-tree's merge algorithm. The additional operations of buffered merge compared to bLSM-tree's merge are highlighted with bold font. One compaction thread monitors the size of level 0 and data are inserted into C_0 by another thread. Once the total size of level 0 $(|C_0| + |C'_0|)$ exceeds the limited size of level 0, S_0, compactions are triggered (lines 2 and 3). Compactions are conducted level by level until the last level or a non-full-sized level is encountered (lines 5 and 6). For a specific level $i(0 \leq i < k)$, if C'_i is empty, data will be moved from C_i into C'_i. Meanwhile, data in B_i also be moved into B'_i (lines 8- and 9). Those data movements do not need any I/O operations but only index modifications. Note that B'_i only contains the frequently visited data of level i, thus the size of B'_i is a variable whose value is smaller than S_i. One parameter S'_i is used to record the size of B'_i when data are initially moved into it (line 10). An empty sorted table B^0_{i+1} is created in B_{i+1} to receive data moved from C'_i (line 11). After all those preparations, C'_i is guaranteed to be nonempty. One file is picked out from C'_i as f_a, and then f_a is merged into C_{i+1} with a merge sort (lines 12–16). The basic compaction units for different LSM-tree implementations are different. Here we inherit the implementation of LevelDB which compacts one file down to next level in each compaction. Instead of being deleted from disk, f_a then be appended into B^0_{i+1} (line 17). The removal of the data in B'_i keeps the same pace as the removal of the data in C'_i. In another word, $\frac{|B'_i|}{S'_i}$ equals to $\frac{|C'_i|}{S_i}$ during the entire merge round. To achieve this, B'_i removes the files with the smallest maximum keys after each compaction to remove the same portion of data as C'_i does in key order (lines 18–20). Different from the underlying LSM-tree, removing a file from the compaction buffer has a different meaning. A file removed from the compaction buffer will be marked as *removed*. All its indices except the minimum and maximum

ALGORITHM 1: Buffered Merge

```
while true do
    if $|C_0| + |C'_0| < S_0$ then
        Continue;
    for each level $i$ from $0$ to $k - 1$ do
        if $|C_i| + |C'_i| < S_i$ then
            Break;
        if $|C'_i| = \emptyset$ then
            Move data from $C_i$ into $C'_i$;
            Move data from $B_i$ into $B'_i$;
            /* record the initial size of $B'_i$                              */
            $S'_i \leftarrow |B'_i|$;
            Create an empty sorted table in $B_{i+1}$ as $B^0_{i+1}$;
        $f_a \leftarrow$ pick one file from $C'_i$;
        /* merge the files with overlapped key ranges                         */
        $F_b \leftarrow$ all files in $C_{i+1}$ whose key range overlaps $[min(f_a), max(f_a)]$ ;
        $F_c \leftarrow$ merge $f_a$ and $F_b$ with a merge sort;
        /* install the result                                                 */
        Remove $f_a$ from $C'_i$;
        Replace $F_b$ with $F_c$ in $C_{i+1}$;
        Append $f_a$ to $B^0_{i+1}$;
        /* gradually remove the files in $B'_i$                               */
        while $\frac{|B'_i|}{S'_i} > \frac{|C'_i|}{S_i}$ do
            $f_d \leftarrow$ file in $B'_i$ with the smallest maximum key;
            Remove $f_d$ from $B'_i$;
```

keys will be removed from the memory, and all its data will be deleted from the disk. This is specifically designed for the query correctness, which will be discussed in Section 5.

For buffered merge, if all data appended are kept in the compaction buffer, the entire database size will be doubled. Further, as discussed in Section 3, conducting queries on infrequently visited data is more sensitive to the number of sorted tables being searched, since more disk seeks might be involved. Thus, those infrequently visited data should be removed from the compaction buffer. The queries conducted on them should be served by the underlying LSM-tree, which contains far less sorted tables in each level. In next section, we will explain how compaction buffer selectively keeps only the frequently visited data in it.

4.2 Trim the Compaction Buffer

During running time, the data in the compaction buffer continue to be evaluated to determine whether they are qualified to be kept in the compaction buffer. We call this process a *trim process*. The basic operation unit of the trim process is also a file. Algorithm 2 shows how LSbM-tree trims the compaction buffer to selectively keep only the files that contain frequently visited data in it.

The compaction buffer is trimmed level by level. For level $i (0 < i \leq k)$, B^0_i is newly appended to B_i and the frequently accessed data blocks in it are still actively being loaded into the buffer cache, and thus it should not be trimmed. Any other sorted tables need to be trimmed. For each file f in B^j_i,

ALGORITHM 2: Trim the Compaction Buffer

```
1 for each level $i (0 < i \le k)$ do
      /* selectively keep the files in $B_i$                                   */
2     for each sorted table $B_i^j (j > 0)$ do
3         for each file $f$ in $B_i^j$ do
4             $total \leftarrow$ number of blocks in $f$;
5             $cached \leftarrow$ number of cached blocks in $f$;
6             if $\frac{cached}{total} < threshold$ then
7                 Remove $f$ from $B_i^j$;
```

it is removed if it does not contain frequently visited data. In LSbM-tree, whether one file contains frequently visited data or not is measured by the percentage of its blocks cached in the buffer cache. The number of cached blocks for one file can be collected by an integer counter *cached*, and it will be increased once one of its blocks is loaded into buffer cache, and decreased once one of its blocks is evicted from buffer cache. Those operations are lightweight with little overhead. When the percentage of cached blocks is smaller than a *threshold*, this file will be removed from the compaction buffer (lines 4–7). The trim processes are conducted by an independent thread periodically. The interval time can be adjusted as an optimized parameter.

4.3 File Size

The size of the $file$ is a key factor for both the underlying LSM-tree and the compaction buffer. It defines the granularity of compactions and trim processes. As described in Algorithm 1, data are compacted down one file at a time. With a file size s, compacting S data from level i to level $i + 1$ needs up to $(r + 1)\frac{S}{s}$ input operations to load the data into the memory and another $(r + 1)\frac{S}{s}$ output operations to write the compacted data on to disk. A larger s brings a smaller number of I/O operations and higher compaction efficiency. Thus the file size of the underlying LSM-tree should not be too small. However, in the compaction buffer, a larger file size causes a lower precision of frequently visited data identification in the trim process. That is because the file with a larger key range has a higher possibility to contain both frequently and infrequently visited data. Furthermore, the key ranges of the files in the compaction buffer, which formerly belong to the upper level of the underlying LSM-tree, can be r times larger than the files in the underlying LSM-tree at the same level. As will be discussed in Section 5, removing one file from B_i may stop not only this file but also another $r - 1$ file in B_i whose key ranges overlap with the removed file from being used to serve queries. As a result, once one file is removed from B_i, queries conducted on up to r files can be redirected. Since all these files contain frequently visited data, a big performance drop may be caused. Thus the file size of the compaction buffer should not be too big.

The underlying LSM-tree and the compaction buffer requires different optimized file sizes, but all files of compaction buffer are created by the underlying LSM-tree. To solve this problem, we add an additional layer in the sorted table's index structure between the sorted table layer and file layer, *super-file*. Each super-file mapping to a fixed number of continuous files, and all these files stored in a continuous disk region. A super-file is the basic operation unit for the underlying LSM-tree while a file is the basic operation unit for the compaction buffer. With this design, even though all files of compaction buffer are created by the underlying LSM-tree, these two data structures can pick their own appropriate file sizes accordingly.

4.4 The Effectiveness of the Compaction Buffer

Our experiments in Section 6 will show that, the harmful buffer cache invalidations are minimized or even eliminated by the compaction buffer. Two reasons contribute to its effectiveness. First, instead of mapping to the LSM-tree, the buffer cache directly maps to the compaction buffer that is built by appending operations without changing the data addresses. Thus, the frequency of data updates in the compaction buffer is significantly reduced compared with that of the LSM-tree. Second, as the data in level C_i' of the LSM-tree is merged to its next level, compaction buffer list B_i' can start to transfer the reads on it to next level. The transferring is carefully and gradually done to minimize the chance of harmful cache invalidations.

The compaction buffer also adapts to a variety of workloads. For workloads with only intensive writes, no data will be loaded into the buffer cache and all appended data in the compaction buffer will be removed by the trim process. For workloads with only intensive reads, the compaction buffer is empty, since data can only be appended into the compaction buffer by conducting compactions. For workloads with both intensive reads and writes, loaded data in the buffer cache can be effectively kept by the compaction buffer while the underlying LSM-tree retains all the merits of a conventional LSM-tree.

4.5 Disk I/O and Storage Cost

Building the compaction buffer does not involve any additional I/O operations. As described in Section 4.1, all files used to build the compaction buffer already exist on disk. Only the indices of sorted tables are modified. Therefore, the I/O cost of building the compaction buffer is negligible.

With the trim process, all files left in the compaction buffer must contain a large portion of data blocks that are loaded in the buffer cache. Therefore, the size of the compaction buffer is determined by both of the buffer cache size and the read/write workloads. In general, the sizes of the buffer cache and the frequently visited data are relatively small. Thus, the additional disk space cost is low and acceptable which is proven by experiments in Section 6.5.

5 QUERY PROCESSING

With the buffered merge algorithm, the on-disk data of LSbM-tree consists of two data structures: the underlying LSM-tree that contains the entire dataset and the compaction buffer which contains only the frequently visited dataset. Those two data structures work together harmoniously and effectively to serve queries.

Algorithm 3 describes the steps of one random data access. Random data accesses are conducted level by level. While searching one level $i(0 < i \leq k)$ with a compaction buffer list, the indices and bloom filters of C_i will be checked first to validate whether this key belong to this level (lines 3–10). If the key is judged not belonging to C_i, then it is unnecessary to further check the sorted tables in B_i, since it is a subset of C_i. Otherwise, the sorted tables in B_i will be checked one by one (lines 12–23). Once a *removed* file whose key range covers the target key is encountered during this process, the operation of checking B_i is stopped immediately (lines 15 and 16). That is because the newest version of the target key may have been removed from B_i, and an obsolete version may be returned by mistake. If the target key is found in any one sorted table in B_i, then the target key and value will be returned (lines 17–23). Otherwise, the target block in C_i will be read out and serve the read (lines 24 and 25). If the target key is not found in any levels, then a not found sign is returned.

Even though each compaction buffer list contains multiple sorted tables, the indices and bloom filters of the underlying LSM-tree can help skip the levels that do not contain the target key. Thus, only the compaction buffer list of the level that may contain the target key needs to be checked.

ALGORITHM 3: Random Access

```
Function RandomAccess($ky$)
for each level $i$ from 0 to $k$ do
    $f \leftarrow$ file in $C_i$ where $ky \in [min(f), max(f)]$;
    if $f$ not found then
        Continue;
    $b \leftarrow$ block in $f$ where $ky \in [min(b), max(b)]$;
    if $b$ not found then
        Continue;
    if $ky$ not pass the Bloom Filter check of $b$ then
        Continue;
    /* check the compaction buffer first                    */
    for each sorted table $B_i^j$ from $B_i^0$ do
        $f' \leftarrow$ file in $B_i^j$ where $ky \in [min(f'), max(f')]$;
        if $f'$ not found then
            Continue;
        if $f'$ is marked as removed then
            Break;
        $b' \leftarrow$ block in $f'$ where $ky \in [min(b'), max(b')]$;
        if $b'$ not found then
            Continue;
        if $ky$ not pass the Bloom Filter check of $b'$ then
            Continue;
        if $ky$ is found in $b'$ then
            Return isFound;
    /* served by $C_i$                                      */
    if $ky$ is found in $b$ then
        Return isFound;
Return notFound;
```

The number of sorted tables each compaction buffer list has varies from 0 to r and $\frac{r}{2}$ on average. Further, the target key can be found in any of the $\frac{r}{2}$ sorted tables and then be returned. Therefore, in LSbM-tree, the number of additional sorted tables that need to be checked for each random access is only about $\frac{r}{4}$.

The steps of conducting range queries are listed in Algorithm 4. For simplicity, we assume the requested key range can be covered by only one file in each sorted table. While searching level $i (0 < i \leq k)$ with a compaction buffer list, the index of C_i will be checked first. If the requested key range does not overlap any files in C_i, then continue searching other levels (lines 3–5). Otherwise, the sorted tables in B_i need to be checked first. While checking sorted tables in B_i, a file queue F is maintained to record all the files in B_i whose key ranges overlap with the requested key range (lines 8–14). Similarly to random data access, we stop checking B_i and clear the file queue F once a *removed* file is found overlapping the requested key range (lines 11–13), since B_i may contain incomplete requested data. Finally, if F is not empty after checking B_i, the requested data in the files of F will be read out. Otherwise, the requested data in the file f from C_i will be read out (lines 15–19).

ALGORITHM 4: Range Query

```
Function rangequery($rge$)
for each level $i$ from 0 to $k$ do
    $f \leftarrow$ file in $C_i$ where $rge \cap [min(f), max(f)] \neq \emptyset$;
    if $f$ not found then
        Continue;
    /* check the compaction buffer first */
    $F \leftarrow \emptyset$;
    for each sorted table $B_i^j$ from $B_i^0$ do
        $f' \leftarrow$ file in $B_i^j$ where $rge \cap [min(f'), max(f')] \neq \emptyset$;
        if $f'$ not found then
            Continue;
        if $f'$ is marked as removed then
            $F \leftarrow \emptyset$;
            Break;
        Append $f'$ in $F$;
    if $F = \emptyset$ then
        /* served by $C_i$ */
        Read out data objects from $f$ in range $rge$;
    else
        /* served by $B_i$ */
        for each file $f'$ in $F$ do
            Read out data objects from $f'$ in range $rge$;
```

Those two algorithms simplify the queries conducted on LSbM-tree by assuming there are only C_i and B_i in level $i (0 < i < k)$. However, one level of LSbM-tree may contain four components: C_i, C_i', B_i, and B_i'. For this case, the combination of C_i' and B_{i+1}^0 is treated as a whole, since data are moved from C_i' into B_{i+1}^0, and their key ranges complement each other. The data in B_i' are a subset of data in this combination. When conducting queries on level i, C_i and its compaction buffer list B_i will be checked first, and then the combination of C_i' and B_{i+1}^0, and its compaction buffer list B_i' are checked. Both follow the steps listed in Algorithms 3 and 4.

6 EXPERIMENTS

We compare the query performance results among LevelDB, SM-tree, bLSM-tree, bLSM-tree with Key-Value store cache, bLSM-tree with incremental warming up, and LSbM-tree with experiments in this section.

6.1 Experimental Setup

Our LSbM-tree implementation is built with LevelDB 1.15 [12]. It implements the buffered merge in the LevelDB code framework. We have also implemented the Stepped-Merge algorithm [10, 14], bLSM-tree, the incremental warming up method [1], and K-V store caching method used by Cassandra [2] as comparisons.

The hardware system is a machine running Linux kernel 4.4.0-64, which has two quad-core Intel E5354 processors, 8GB main memory. The maximum size of the DB buffer cache is set to 6GB. The rest memory space is shared by the indices of sorted tables, bloom filters, OS buffer cache, and the operating system. Two Seagate hard disk drives (Seagate Cheetah 15K.7, 450GB) are configured as RAID0 as the storage for LSM-tree. The HDD RAID is formatted with ext4 file system.

The size of level 0 is set to 100MB. With a size ratio $r = 10$, the maximum size S_i of level 1, 2, and 3 is 1GB, 10GB, and 100GB, respectively. The *file* size of tests conducted on LSbM-tree is set to 2MB, which is the default setting of LevelDB. We define a *super-file* contains r files. With $r = 10$, the size of a *super-file* for LSbM-tree is 20MB. For tests conducted on any other LSM-tree variants, the *file* size is set to 20MB. The size of a *block* is set to 4KB that is equal to the disk page size. The bloom filter is set to 15-bit per element. The key-value pair size is set to 1KB, which is the default value of *YCSB* and is a common case in industry environment [4, 9]. The interval of the trim processes is set to 30s, and a file can be kept in the compaction buffer only if 80% of its blocks are cached in the buffer cache.

6.2 Workloads

All writes are uniformly distributed on a dataset with 20GB unique data. With this write workload, all writes are updates on existing keys, which are randomly picked from the entire key space. Levels 1 and 2 will be full and merged down to next level after inserting 1GB and 10GB data, respectively. The maximum number of sorted tables that B_1 and B_2 have are 10, which is the size ratio r. On the other hand, all inserted data except the first 20GB data are repeated data for level 3. Those repeated data will be detected while conducting compactions on C_3. As a result, B_3 is frozen.

The read workloads in our experiments are based on *Yahoo! Cloud Serving Benchmark* (*YCSB*) [9], which provides a number of workload templates abstracted from real-world applications. We have built two workloads which are *RangeHot* and *Zipfian* workloads. RangeHot workload characterizes requests with strong spatial locality, i.e., a large portion of reads is concentrated in a hot range. In our test, 3GB continuous data range is set as the hot range, and 98% of the reads requests lie in this range. For Zipfian workload, the request frequency of tuples follows the Zipf law. In our tests, the Zipfian constant is set to 0.999. These workloads is generated at runtime with the **db_bench** utility provided in the LevelDB package.

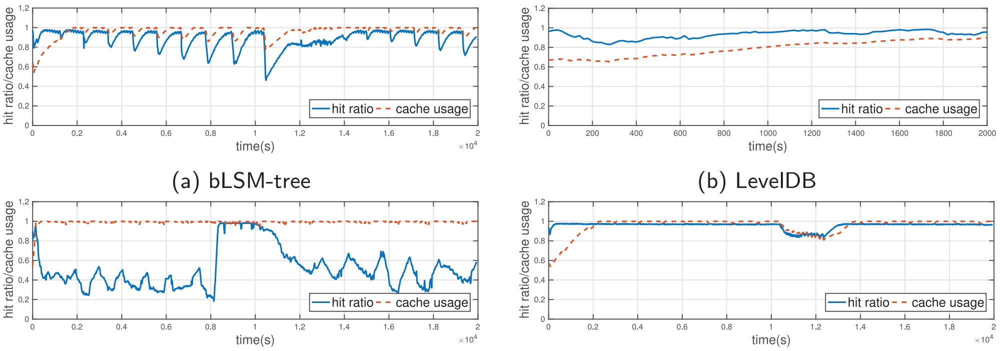
We conducted a series of experiments to evaluate the random read and range query performance of LSbM-tree and other LSM-tree variants under intensive writes with the workloads given above. The write throughput for all tests is 1,000 OPS (operations per second). All writes are sent via a single thread and any other random reads and range queries are sent by another eight threads. During these experiments, the DB buffer cache hit ratio, read throughput, database size and other statistics are measured online and stored to a log file on the disk. Each test is conducted for 20,000s.

6.3 Performance of Random Reads

6.3.1 RangeHot Workloads. We have evaluated LSbM-tree on its random read performance by RangeHot workload with intensive writes. We also conducted three tests on bLSM-tree, LevelDB, and bLSM-tree with incremental warming up for comparisons. Figure 11 shows the hit ratio changes (vertical bar) and DB buffer cache usage rate (dashed line) as time passes (horizontal bar) of all these tests.

bLSM-tree: As shown in Figure 11(a), the cached data in DB buffer cache are periodically invalidated by compactions in bLSM-tree and the hit ratio goes up and down. The big hit ratio drops periodically observed in every 1,000s are caused by the compactions conducted from C_1' to C_2. When $|C_2|$ increases from 0 to S_2, the invalidations become more and more severe.

LevelDB: LevelDB maintains only one sorted table at each level. Two merge operations exist on level $i (0 < i < k)$ simultaneously: merge from C_{i-1} to C_i and merge from C_i to C_{i+1} [12, 20]. The hit ratio also goes up and down following the similar pattern of the bLSM-tree (Figure 11(b)). However, the hot range in level 2 is updated every 2,000s instead of 1,000s as bLSM-tree does. This



(a) bLSM-tree

(b) LevelDB

(c) bLSM-tree with incremental warming up

(d) LSbM-tree

Fig. 11. Hit ratio changes of tests on RangeHot workloads.

phenomenon is caused by the *Non-uniformity* of key density which was discussed in detail in a previous work [16].

LSM-tree with incremental warming up: We simulated the incremental warming up algorithm on a single machine in the following way: Before the newly compacted blocks are flushed from memory, the blocks in the buffer cache that will be evicted in this compaction will be replaced with the newly generated blocks whose key ranges overlap with them [1]. This approach is based on an assumption that newly compacted blocks will also be frequently visited if they overlap with any blocks in the buffer cache. However, this assumption is true only if all overlapped ranges belong to the hot range. Let us assume that one key-value pair of level $i (0 \leq i < k)$ is loaded into the buffer cache by a read operation. The block containing that pair will be marked as *Hot* when it is being compacted down to the lower level. Since up to r blocks in level $i + 1$ share the same key range with that block, up to $r + 1$ newly generated blocks will be loaded into buffer cache after this compaction. Furthermore, those $r + 1$ blocks will cause the loading of another $(r + 1) * (r + 1)$ blocks when they are being compacted to level $i + 2$. Therefore, one read operation on level i will load as many as $(r + 1)^{k-i}$ blocks into buffer cache. If this read is conducted inside the hot range, then the incremental warming up can help pre-fetch hot data into buffer cache. Otherwise, the incremental warming up will load even more infrequently visited data blocks into buffer cache and continuously evict the frequently accessed data blocks. Figure 11(c) shows the change of the buffer cache hit ratio over time for test on bLSM-tree with incremental warming up. With 2% of reads lie out of the hot range, the hit ratio goes up and down periodically. When the compaction is conducted on the hot range, the hit ratio increases sharply because of a pre-fetching effect discussed above. However, those data blocks in the buffer cache then be gradually evicted by the infrequently accessed data blocks loaded by the incremental warming up process. This experiment shows that the incremental warming up method may not work for certain workloads.

LSbM-tree: As shown in Figure 11(d), the hit ratio of LSbM-tree keeps steady and high. Note that all data compacted to level 3 are repeated data, and thus the compaction buffer list of level 3 (B_3) is frozen. As a result, the buffer cache is still invalidated when data are compacted from C_2' to C_3. However, since the data in B_2' are gradually deleted as described in Section 4.1, part of the hot data is still kept in the buffer cache by B_2', and the invalidation issue is mitigated.

Figure 12 compares the average buffer cache hit ratios and throughputs of bLSM-tree, LevelDB, bLSM-tree with incremental warming up, and LSbM-tree on RangeHot workload. It shows that LSbM-tree achieves a much higher buffer cache hit ratio and random read throughput than other LSM-tree variants on RangeHot workloads with intensive writes. As measured by Formula (3), a little increase of hit ratio (14%) brings a 1.98X throughput increase.

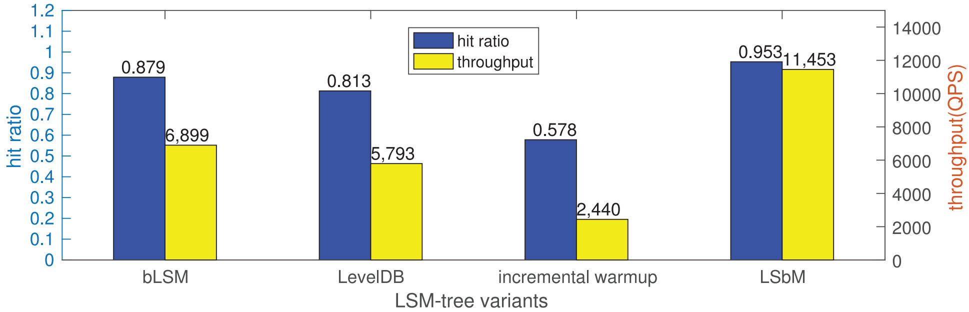
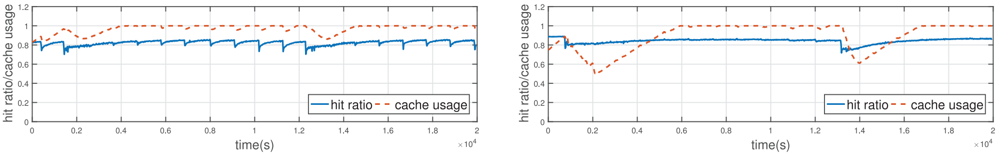


Fig. 12. Performance of tests on RangeHot workloads.



(a) bLSM-tree

(b) LSbM-tree

Fig. 13. Hit ratios of tests on Zipfian workloads.

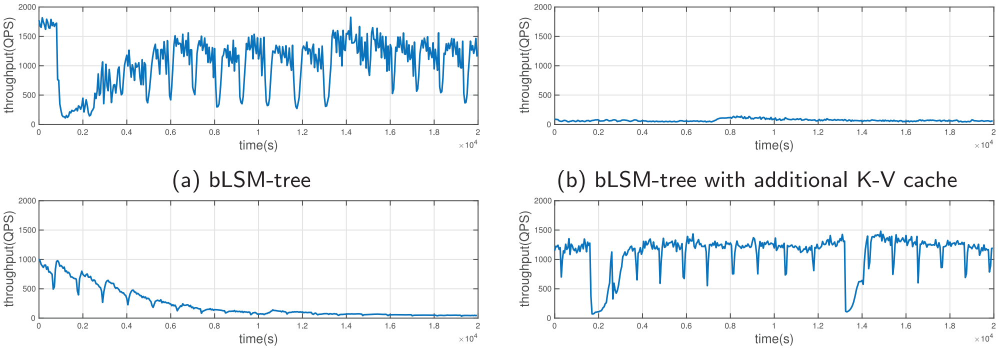
6.3.2 Zipfian Workload. We also evaluated the performance of bLSM-tree and LSbM-tree by the Zipfian workloads. Compared to the RangeHot workloads, the read distribution of Zipfian workloads highly skews on relatively smaller ranges, and most part of the entire data range are infrequently visited. As consequences, the buffer cache invalidation issue is less severe. As shown in Figure 13(a), the buffer cache is invalidated periodically. However, the drop scale is much smaller and the reload process is much shorter. In contrast, LSbM-tree can still gain a better performance by keeping the hot data in buffer cache from being invalidated. Figure 13(b) shows the changes of hit ratios overtime for tests on LSbM-tree. The hit ratios keep steady and high. The average throughput on Zipfian Workloads is 1,947 QPS (queries per second) for bLSM-tree and 2,048 QPS for LSbM-tree.

6.4 Performance of Range Queries

To evaluate the range query performance of LSbM-tree and other LSM-tree variants under intensive writes, we perform another set of tests with bLSM-tree, SM-tree, bLSM-tree with additional K-V store cache, and LSbM-tree. Each read operation follows the RangeHot workload but will read all the data lie in a 100KB range. The throughput changes over time for all tests are shown in Figure 14, and the overall throughputs of all tests are given in Figure 15.

bLSM-tree: Compared to random data accesses, the range queries conducted on bLSM-tree are influenced less by intensive writes (Figure 14(a)). That is because when the buffer cache is invalidated, the invalidated data can be loaded back to buffer cache by range queries more quickly than random data accesses, since sequential I/O is much faster on HDD than random I/O. The throughput is 1,066 QPS.

Key-Value store cache: In this test, a Key-Value store is built on top of the bLSM-tree. Among the 6GB cache spaces, 3GB is allocated to the Key-Value store cache, and the rest memory space is allocated to a DB buffer cache. As shown in Figure 14(b), the throughput of the test using an additional Key-Value store keeps extremely low, which is only 68 QPS on average. It is low because of two reasons. First, when an additional Key-Value store cache is used, the memory space for buffer



(a) bLSM-tree

(b) bLSM-tree with additional K-V cache

(c) SM-tree

(d) LSbM-tree

Fig. 14. Throughputs of tests on range queries.

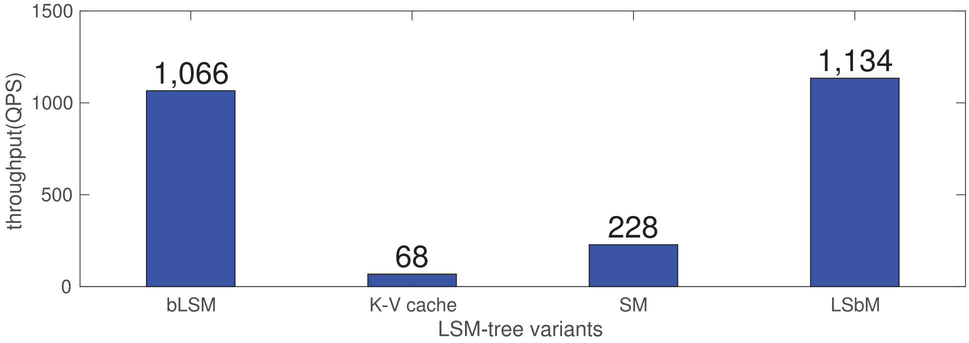
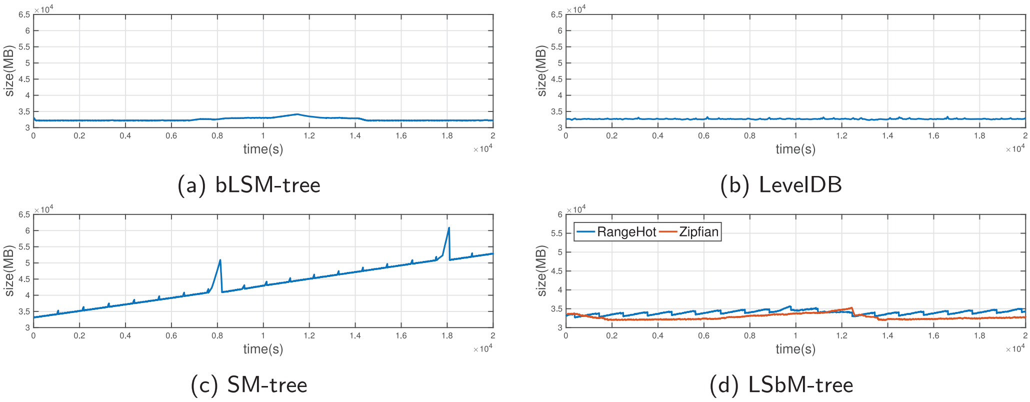


Fig. 15. Throughput comparisons of range queries.

cache is reduced, which causes capacity misses. Second, the data in the buffer cache will keep being invalidated by compactions, which causes *LSM-tree compaction induced cache invalidations*.

Stepped-Merge Method (SM-tree): We also implemented the Stepped-Merge Method. When the write buffer C_0 is full, instead of merging with C_1, it will be appended to C_1 as a sorted table of it. When $C_i (1 \leq i < k)$ is full, all sorted tables in C_i will be merged together and be appended to C_{i+1}. As a result, each level of SM-tree contains 0 to r sorted tables. The range query throughput of SM-tree is only 228 QPS. It is low because of two reasons. First, when range queries must be served by disk, conducting range queries on one level with multiple sorted tables may need multiple disk seeks. Second, with the given write workloads, obsolete data will be piled in level 3 and loaded into buffer cache by range queries, which reduces the effective buffer cache capacity. As shown in Figure 14(c), in the first 10,000s, the total number of sorted tables searched for each query increases and the throughput decreases. At around 10,000s, level 2 becomes full and all its sorted tables are compacted together as a sorted table of level 3. The total number of sorted tables then decreases, and the range query throughput increases a little bit. However, due to the shrinking of effective buffer cache capacity caused by the obsolete data, the throughput becomes extremely low. Thus, to achieve high performance range queries, the compacted structure of the underlying LSM-tree must be retained.

LSbM-tree: By contrast, LSbM-tree achieves the best performance. The buffer cache invalidation issue is further mitigated compared to bLSM-tree and the throughput is 1,134 QPS



(a) bLSM-tree

(b) LevelDB

(c) SM-tree

(d) LSbM-tree

Fig. 16. Database size changes of tests on different LSM-tree variants.

(Figure 14(d)). The sorted structure on underlying LSM-tree can support on disk range queries efficiently; meanwhile, the compaction buffer can serve fast data accesses by effectively keeping frequently visited data in the buffer cache.

6.5 Database Size

Among all the tests conducted above, LevelDB, bLSM-tree, SM-tree, and LSbM-tree have different on-disk data structures, and thus the database size of those LSM-tree variants may differ from each other. While conducting tests, the realtime database sizes are recorded, and the changes of those sizes over time can be found in Figure 16.

As data are written in the database, compactions are conducted to merge data from upper levels to lower levels and eventually stay at the last level. Note that with the given write workloads, all inserted data to level 3, the last level, are repeated data. In bLSM-tree and LevelDB, when a chunk of data are compacted into level 3, the same size of obsolete data are deleted during compaction. Thus the database sizes do not change a lot. However, due to the lazy compaction method of SM-tree, data will be piled in level 3, and the obsolete data will not be deleted until level 3 is full. As a result, the database size keeps increasing. Even worse, when one level is full, additional spaces are needed to merge the entire level together. As shown in Figure 16(c), the database size of SM-tree bursts periodically. The small bursts observed every 1,000s are caused by compacting sorted tables in level 1, and the big bursts observed every 10,000s are caused by compacting sorted tables in level 2.

The database size of LSbM-tree is slightly greater than the bLSM-tree and LevelDB, since the compaction buffer takes additional space. However, the size of the compaction buffer is limited due to the trim process. Different from other LSM-tree variants, the database size of LSbM-tree is also read workload dependent. Figure 16(d) shows the database size changes of LSbM-tree on RangeHot and Zipfian workloads. For Zipfian workloads, the reads are highly skewed in a smaller hot range. Thus, most of the data are removed from the compaction buffer during the trim process and the compaction buffer is relatively smaller.

The average database sizes of bLSM-tree, LevelDB, SM-tree, and LSbM-tree with different read workloads can be found in Figure 17. It proves that with the given read and write workloads, the additional space cost of LSbM-tree is very low, which is about 4% more space than that of bLSM-tree and LevelDB. In contrast, the amount of repeated data due to lazy compaction in SM-tree add as high as about 50% more disk space.

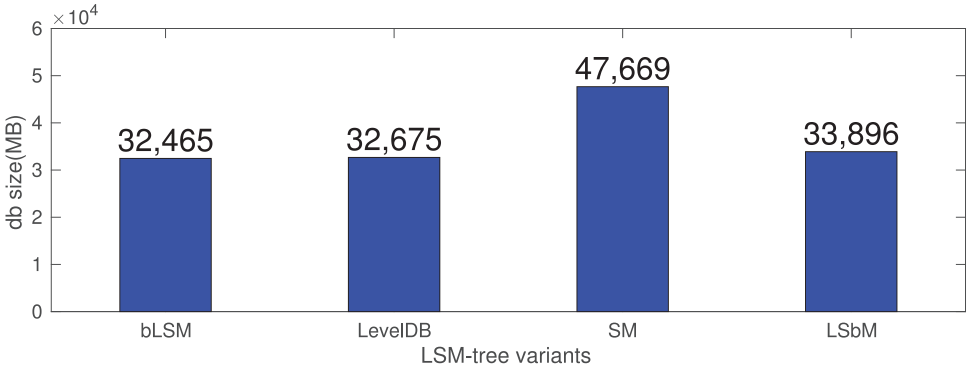


Fig. 17. Database sizes of different LSM-tree variants.

7 OTHER RELATED WORK

The key design issues of LSbM-tree have been introduced in our previous work [22]. In this article, we further discuss the read/write latency model of LSM-tree, and we compare LSbM-tree with other solutions on more workloads.

To reduce the compaction cost, some production systems only compact data partially in runtime and run a full compaction during system idle time. In HBase , the former is called minor compaction, while the latter is called major compaction [3]. However, disabling major compaction during runtime mainly reduces the compaction of old data. These old data refer to the last level data in a standard LSM-tree, which are less frequently accessed than new data. Thus, just like SM-tree, this approach cannot avoid the interference from compactions to buffer caching. In practice, HBase still suffers low read performance during intensive writes [1].

RocksDB compiles and utilizes several techniques to best utilize the efficiency of flash-based storage systems like SSD. It adapts the stepped merge algorithm as its universal compaction method and uses Key-Value cache as an option for certain workloads [10]. However, both Key-Value cache and stepped merge methods have their limits as discussed in Section 1. VT-tree is an extension of LSM-tree to avoid unnecessary merges for presorted data [21]. LSM-trie reduces the write amplification of an LSM-tree with an SM-tree-like merge algorithm and optimizes the random data access performance on the multiple sorted tables structure in each level [24]. An FD-tree is proposed for data indexing on SSDs without random writes, which has a similar idea to LSM-tree and is enhanced with fractional cascading technique for a low memory usage [15]. However, a lookup needs multiple disk accesses if the object is not found in the first level. LOCS exploits the internal parallelism among flash channels to improve SSD-based LSM-trees [8, 23]. cLSM is an LSM-tree variant supporting scalable concurrency with multi-core processors [11]. WiscKey is proposed to separate the storage of key and value, which significantly eliminates the write amplification issue of LSM-tree compactions on SSD [17]. It exploits the internal parallelism of SSD and multi-thread to achieve high range query performance even the values of continuous keys are discretely stored on disk. In contrast, our LSbM-tree is a low cost general solution to solve the fundamental problem of data caching in LSM-tree.

8 CONCLUSIONS

LSM-tree is effective for write-intensive workloads. However, as both read- and write-intensive workloads co-exist, the compaction-induced buffer cache invalidations can periodically degrade cache performance by missing data blocks with high locality. Existing methods to address this

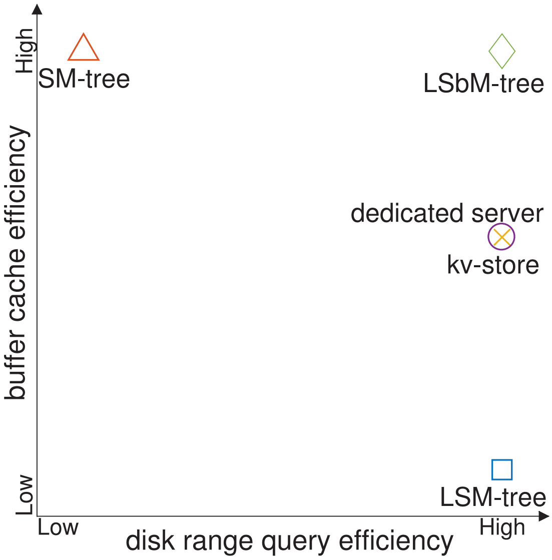


Fig. 18. LSbM-tree compares with other LSM-tree variants.

issue, such as a KV-store cache to bypass the buffer cache, SM-tree by reducing the compaction frequency, and others, are effective for certain workloads. There are two limits in these methods. First, the optimization has been focused on certain workload patterns at the cost of scarifying performance of other patterns. For example, SM-tree reduces the amount of compaction-induced cache invalidations by a lazy compaction, but the non-fully-sorted LSM-tree is less efficient for range queries. KV-store is very effective to frequently random data accesses by building an independent cache in DRAM, but it is not effective to range queries. The dedicated compaction server method is effective as the frequently accessed data are also recently compacted data in LSM-tree by its warming up process, but this assumption may not always true in practice. Second, KV-store and the dedicated compaction server methods require additional memory resources.

In this article, we propose LSbM-tree, a low-cost solution by adding a small compaction buffer on disk, which plays three major roles to fundamentally address the invalidation issue and other issues of existing methods and to retain all the merits of LSM-tree and other methods. First, the compaction buffer is selectively and adaptively built and managed, where the data blocks are consistently kept with the ones in the buffer cache. Thus, the LSM-tree-induced invalidations are minimized and eliminated. Second, LSbM-tree best utilizes both buffer cache and disks for both random accesses and range queries. Finally, the compaction buffer is only built when it is necessary without computing overhead. Under either read only or write only workloads, the compaction buffer is adaptively shrunk and disappears eventually. Thus, the additional resource is only related to a small disk space.

Figure 18 presents a comprehensive comparison among the five representative LSM-tree based methods by the efficiency of memory cache (y-axis, hit ratios) and by the efficiency of range queries in disks (x-axis, data access throughput). LSM-tree does well for range queries in disks but cause compaction-induced cache invalidations to lower the cache performance, while SM-tree has the opposite performance results. Both KV-store and the Dedicated server methods maintain a fully sorted LSM-tree structure, thus they retain the high efficiency of range queries on disks. However, their cache performances are high for certain workloads and suboptimal for others, thus comprehensively, the cache performance is degraded at a certain level. In contrast, LSbM-tree is the best performer for both disk range query efficiency and for memory cache efficiency.

In this article, we have made a strong case for LSbM-tree to best utilize buffer cache and disks for various workloads with both intensive reads and writes.

ACKNOWLEDGMENTS

We thank the anonymous referees for the valuable comments. We have also received constructive feedbacks from related open source communities, which is helpful for us to identify and address several critical issues from real-world systems. An early version of the article draft was posted as a technical report in arXiv [13].

REFERENCES

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

[2] Apache. 2017. Cassandra. Retrieved April 6, 2017 from http://cassandra.apache.org/.

[3] Apache. 2017. HBASE. Retrieved April 6, 2017 from https://hbase.apache.org/.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[5] Basho. 2017. Riak. Retrieved April 6, 2017from http://basho.com/riak.

[6] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet Math.* 1, 4 (2004), 485–509.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4.

[8] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11).* IEEE, 266–277.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing.* ACM, 143–154.

[10] Facebook. 2017. RocksDB. Retrieved April 6, 2017 from http://rocksdb.org/.

[11] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the 10th European Conference on Computer Systems.* ACM, 32.

[12] Google. 2017. LevelDB. Retrieved April 6, 2017 from http://code.google.com/p/leveldb.

[13] Lei Guo, Dejun Teng, Rubao Lee, Feng Chen, Siyuan Ma, and Xiaodong Zhang. 2016. Re-enabling high-speed caching for LSM-trees. arXiv preprint arXiv:1606.02015 (2016).

[14] H. V. Jagadish, P. P. S. Narayan, Sridhar Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *VLDB*, Vol. 97. Citeseer, 16–25.

[15] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1-2 (2010), 1195–1206.

[16] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards accurate and fast evaluation of multi-stage log-structured designs. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16).* USENIX Association, 149–166.

[17] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16).* 133–148.

[18] Ryan Mcguire. 2014. Compaction Improvements in Cassandra 2.1. April 6, 2014 from http://www.datastax.com/dev/blog/compaction-improvements-in-cassandra-21.

[19] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.

[20] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM, 217–228.

[21] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Presented as part of the Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13).* 17–30.

[22] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE, 68–79.

[23] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 16.

[24] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. 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 Association, 71–82.

[25] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for GPUS to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.* 8, 11 (2015), 1226–1237.

Received April 2017; revised November 2017; accepted November 2017