2017 IEEE 37th International Conference on Distributed Computing Systems

LSbM-tree: Re-enabling Buffer Caching in Data Management for Mixed Reads and Writes

Dejun Teng*, Lei Guo†, Rubao Lee*, Feng Chen§, Siyuan Ma* Yanfeng Zhang‡ and Xiaodong Zhang*
*The Ohio State University
†Google Inc.
‡Northeastern University, China
§Louisiana State University

***Abstract*—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.**

In order 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 data sets. In LSbM, strong locality objects can be effectively kept in the buffer cache with minimum or without 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 effectiveness.

I. 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 paper, 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 [1] 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 [2], HBase [3], Cassandra [4], and Riak [5], and is the de facto model for write-intensive data processing.

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*.

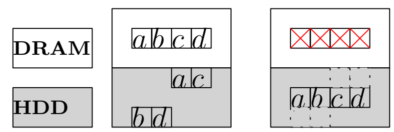


Fig. 1: An example of LSM-tree compaction induced cache invalidations

However, the impact to buffer cache was not considered in the original design of LSM-tree. 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], [4], [6]. 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 1, 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 contents of these objects remain unchanged, the cached

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.70

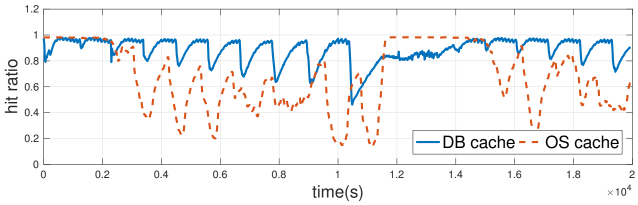


Fig. 2: Inabilities of buffer caches

data blocks of these 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 [7], [8].

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 VI. Figure 2 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 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.

A. 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 [4]. In general, the Key-Value store cache is an independent buffer in memory without any address indexing to the data source on disks [9]. 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, and the data will be retrieved from the buffer cache or the disk. This approach would reduce the amount of accesses to buffer cache or even bypass it. However, the key-value store would not be efficient for range queries and for accessing data with high spacial locality due to its nature of random stores and high efficiency for random accesses.

Dedicated compaction servers: This method is to use dedicated servers to conduct compactions, where the compacted data sets are kept in memory in these servers to be used to replace data sets in the buffer cache of users with an algorithm named *Incremental Warming up* algorithm [7]. This method attempts to reduce the LSM-tree induced misses by warming up the buffer cache in this way. The effectiveness of the dedicated server approach depends on the ability to identify specific compacted data sets with high locality. It is based on an assumption that newly compacted data sets 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 certain workloads, 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 [10]. Similar to LSM-tree, SM 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. 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. Firstly, data in each level are not fully sorted. As a result, the range query performance of SM is low. Secondly, 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.

B. Our solution

In this paper, 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 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 paper without notation is referred to the DB buffer cache).

The basic idea of LSbM 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 lower rate than the compaction rate. LSbM 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 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.

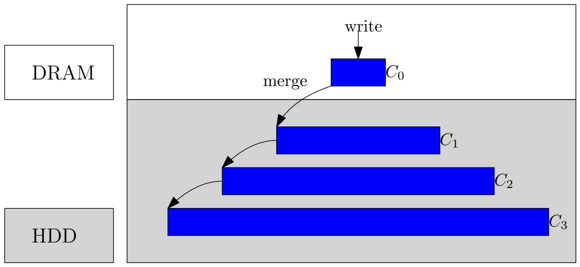


Fig. 3: Basic structure of an LSM-tree

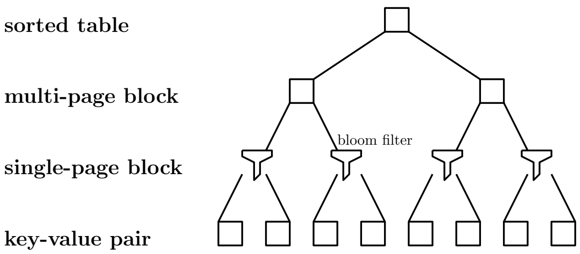


Fig. 4: The layered index structure of a sorted table

We have implemented a prototype of LSbM based on LevelDB [6], a widely used LSM-tree library developed by Google, and conducted extensive experiments with Yahoo! Cloud Service Benchmark (YCSB) [11]. We show that with a standard buffer cache and a hard disk storage, our LSbM implementation can achieve 2x performance improvement over LevelDB, and significantly outperforms other existing solutions.

The roadmap of this paper is as follows. Section II presents the background knowledge of LSM-tree and its merge algorithm. Section III introduces our LSbM design. Sections IV presents the management of the compaction buffer. Section V presents the techniques to query LSbM. We present the performance evaluation of LSbM and comparisons with other LSM-tree variants in Section VI. Finally, we overview the related work in Section VII and conclude this paper in Section VIII.

II. Background

In this section, we analyze the internals of LSM-tree compactions, and show how compactions generate frequent data movement on disks which in turn invalidates the data in the buffer cache.

A. LSM-tree

In an LSM-tree, data are stored on disks in multiple levels of increasing sizes. Figure 3 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 [1]. 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* [12] 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* [6], [3], 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 paper.

Following the notations in [1], 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 $C_0, C_1, ..., C_k$ as $S_0, S_1, ..., S_k$, 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 [1]. 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*.

B. Compactions

A conventional LSM-tree maintains a fully sorted structure on each level for efficient random and sequential data accesses [1] [13]. 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 $\frac{(1+2+...(r-1))}{r} = \frac{(r-1)}{2}$ chunks of data in C_{i+1}. The average 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 I.

III. 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 this problem root, 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 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, thus all of them need 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.

There is a tradeoff between a highly sorted structure and less sorted structure. Queries can benefit from a fully sorted structure of an LSM-tree level, however, maintaining such a fully sorted structure needs to frequently conduct compactions, which may in turn invalidate the cached data objects. 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 data set 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 5 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 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

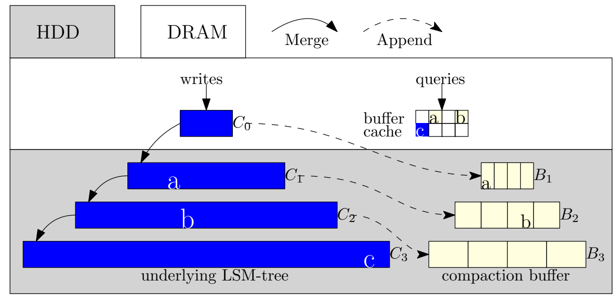


Fig. 5: The basic structure of a LSbM-tree

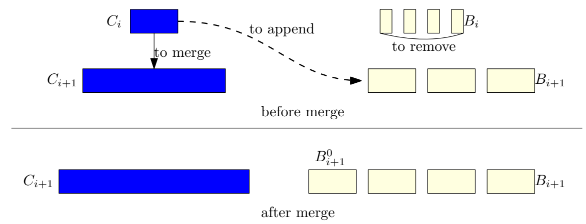


Fig. 6: An illustration of the buffered merge

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 j^{th} sorted table in B_i as B_i^j. B_i^0 contains the most recent data in B_i. Figure 5 shows a compaction buffer with three compaction buffer lists. $B_i (1 \leq i \leq 3)$ contains frequently visited data sets of level i, but is updated in a very low rate to enable effective caching.

In level i of LSbM, read requests are dispatched to B_i for frequently visited data which are highly possible to be hit in the buffer cache, and to C_i for others. As shown in Figure 5, 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.

IV. 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 maintain 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.

A. Buffered merge

Figure 6 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^0_{i+1}. Note that B^0_{i+1} is built with the files of C_i which already exist on disk. Therefore, no additional I/O is involved. The data in B^0_{i+1} which were formerly in C_i will

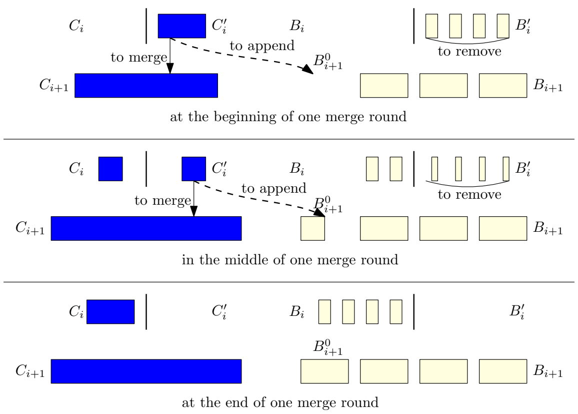


Fig. 7: The detailed operations of the buffered merge

not be updated by compactions. Furthermore, B^0_{i+1} 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.

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, 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, there must exist repeated data. When repeated data are detected, LSbM 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 bLSM-tree's gear scheduler merge algorithm [13]. In bLSM, data in level $i(0 \le i < k)$ 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 defines two parameters $in_{progress}$ and $out_{progress}$ to regulated 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 7 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 consequences, data can be inserted into C_0 with a predictable latency [13].

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 */
      $\boldsymbol{S'_i \leftarrow |B'_i|}$;
      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**
      $\boldsymbol{f_d} \leftarrow$ **file in** $\boldsymbol{B'_i}$ **with the smallest maximum key**;
      Remove $\boldsymbol{f_d}$ **from** $\boldsymbol{B'_i}$;

We adapted this merge algorithm in our buffered merge operations. As shown in Figure 7, the $i^{th}(0 \le i < k)$ 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 (line 2-3). Compactions are conducted level by level until the last level or a non-full-sized level is encountered (line 5-6). 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 (line 8-9). 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 (line 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 (line 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 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 V.

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 III, conducting queries on infrequently visited data is more sensitive to the number of sorted tables 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.

B. Trim the compaction buffer

During running time, the data in the compaction buffer keep being 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 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, thus it should not be trimmed. Any other sorted tables need to be trimmed. For each file f in B^j_i, it is removed

Algorithm 2: trim the compaction buffer

1 **for** *each level* $i(0 < i \leq k)$ **do**
    `/* selectively keep the files in` B_i `*/`
2   **for** *each sorted table* $B^j_i (j > 0)$ **do**
3   **for** *each file* f *in* B^j_i **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^j_i;

if it does not contain frequently visited data. In LSbM, 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 light weight with little overhead. When the percentage of cached blocks is smaller than a *threshold*, this file will be removed from the compaction buffer (line 4-7). The trim processes are conducted by an independent thread periodically. The interval time can be adjusted as an optimized parameter.

C. 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 V, removing one file from B_i may stop not only this file, but also another $r-1$ files 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.

D. The effectiveness of the compaction buffer

Our experiments in Section VI 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 in order 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.

E. Disk I/O and storage cost

Building the compaction buffer does not involve any additional I/O operations. As described in Section IV-A, 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 neglectable.

With the trim process, all files left in the compaction buffer must contain a large portion of data blocks which 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 VI-E.

V. Query processing

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

Algorithm 3: random access

1 **Function** RandomAccess(ky)
2 **for** *each level* i *from* 0 *to* k **do**
3     $f \leftarrow$ file in C_i where $ky \in [min(f), max(f)]$;
4   **if** f *not found* **then**
5   Continue;
6     $b \leftarrow$ block in f where $ky \in [min(b), max(b)]$;
7   **if** b *not found* **then**
8   Continue;
9   **if** ky *not pass the Bloom Filter check of* b **then**
10   Continue;
    /* check the compaction buffer first */
11   **for** *each sorted table* B_i^j *from* B_i^0 **do**
12         $f' \leftarrow$ file in B_i^j where $ky \in [min(f'), max(f')]$;
13   **if** f' *not found* **then**
14   Continue;
15   **if** f' *is marked as removed* **then**
16   Break;
17         $b' \leftarrow$ block in f' where $ky \in [min(b'), max(b')]$;
18   **if** b' *not found* **then**
19   Continue;
20   **if** ky *not pass the Bloom Filter check of* b' **then**
21   Continue;
22   **if** ky *is found in* b' **then**
23   Return *isFound*;
    /* served by C_i */
24   **if** ky *is found in* b **then**
25   Return *isFound*;
26 Return *notFound*;

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 (line 3-10). If the key is judged not belong to C_i, 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 (line 12-23). Once a *removed* file whose key range cover the target key is encountered during this process, the operation of checking B_i is stopped immediately (line 15-16). 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, the target key and value will be returned (line 17-23). Otherwise, the target block in C_i will be read out and serve the read (line 24-25). If the target key is not found in any levels, 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. 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, the number of additional sorted tables that need to be checked for each random access is only about $\frac{r}{4}$.

Algorithm 4: range query

1 **Function** rangequery(rge)
2 **for** *each level* i *from* 0 *to* k **do**
3   $f \leftarrow$ file in C_i where $rge \cap [min(f), max(f)] \neq \emptyset$;
4   **if** f *not found* **then**
5   Continue;
  `/* check the compaction buffer first */`
6   $F \leftarrow \emptyset$;
7   **for** *each sorted table* B_i^j *from* B_i^0 **do**
8     $f' \leftarrow$ file in B_i^j where $rge \cap [min(f'), max(f')] \neq \emptyset$;
9   **if** f' *not found* **then**
10   Continue;
11   **if** f' *is marked as removed* **then**
12       $F \leftarrow \emptyset$;
13   Break;
14   Append f' in F;
15   **if** $F = \emptyset$ **then**
    `/* served by` C_i `*/`
16   Read out data objects from f in range rge;
17   **else**
    `/* served by` B_i `*/`
18   **for** *each file* f' *in* F **do**
19   Read out data objects from f' in range rge;

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, continue searching other levels (line 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 (line 8-14). Similar 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 (line 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 (line 15-19).

Those two algorithms simplify the queries conducted on LSbM by assuming there are only C_i and B_i in level $i(0 < i < k)$. However, one level of LSbM 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.

VI. 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.

A. Experimental setup

Our LSbM-tree implementation is built with LevelDB 1.15 [6]. 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 [7], and K-V store caching method used by Cassandra [4] as comparisons.

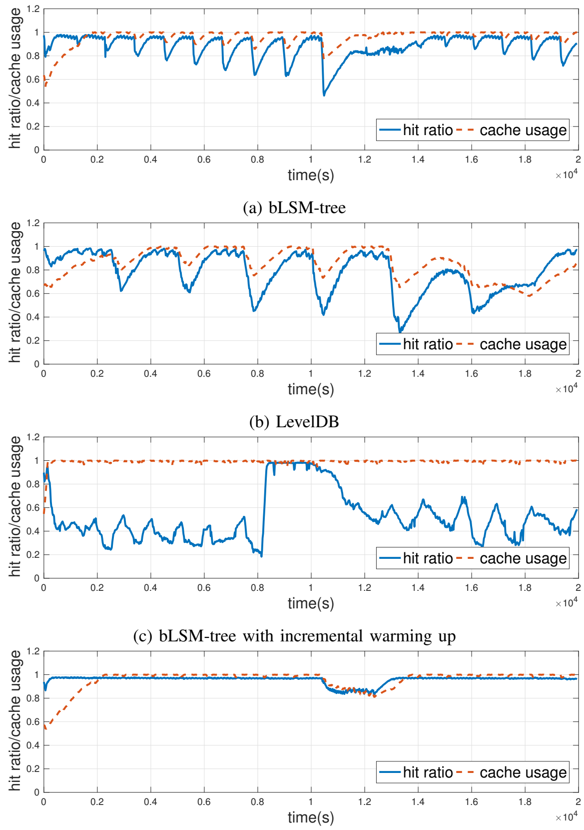
The hardware system is a machine running Linux kernel 4.4.0-64, which has two quad-core Intel E5354 processors, 8 GB main memory. The maximum size of the DB buffer cache is set to 6GB. The rest memory space is shared by the indices of sorted tables, bloom filters, OS buffer cache, and the operating system. 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 100 MB. 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 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* is 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 1 KB which is the default value of ***YCSB*** and is a common case in industry environment [11] [15]. The interval of two trim processes is set to 30 seconds, and a file can be kept in the compaction buffer only if 80% of its blocks are cached in the buffer cache.

B. Workloads

All writes are uniformly distributed on a data set with 20GB unique data. With this write workload, 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***) [11], which provides a number of workload templates abstracted from real-world applications. We have built the *RangeHot* workload, which 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. This workload is generated at run time with the **db_bench** utility provided in the LevelDB package.



(a) bLSM-tree

(b) LevelDB

(c) bLSM-tree with incremental warming up

(d) LSbM-tree

Fig. 8: Hit ratio changes of tests on RangeHot workloads

We conducted a series of experiments to evaluate the random read and range query performance of LSbM and other LSM-tree variants under intensive writes. 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 8 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 20,000 seconds.

C. Performance of random reads

We have evaluated LSbM 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 8 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 8a, the cached data in DB buffer cache are periodically invalidated by compactions in bLSM and the hit ratio goes up and down. The big hit ratio drops periodically observed in every 1,000 seconds 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} [6] [13]. The hit ratio also goes up and down following the similar pattern of the bLSM-tree (Figure 8b). However, the hot range in level 2 is updated every 2,000 seconds instead of 1,000 seconds as bLSM-tree does. This phenomenon is caused by the *Non-uniformity* of key density which was discussed in detail in a precious 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 [7]. 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, 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 8c 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 8d, the hit ratio of LSbM keeps steady and high. Note that all data compacted to level 3 are repeated data, 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 IV-A, part of the hot data is still kept in the buffer cache by B_2' and the invalidation issue is mitigated.

Figure 9 compares the average buffer cache hit ratios and throughputs of bLSM, LevelDB, bLSM with incremental warming up, and LSbM on RangeHot workload. It shows that LSbM achieves a much higher buffer cache hit ratio and random read throughput than other LSM-tree variants on

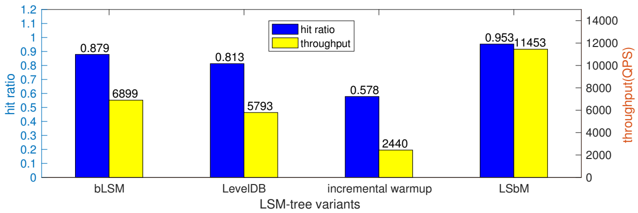
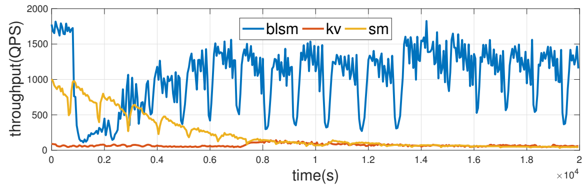
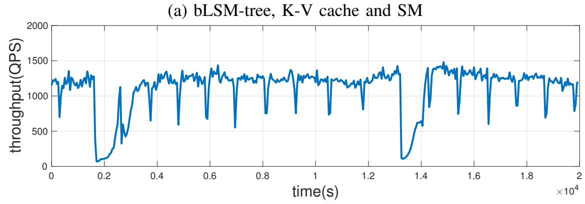


Fig. 9: Performance of tests on RangeHot workloads



(a) bLSM-tree, K-V cache and SM



(b) LSbM-tree

Fig. 10: Throughput changes of tests on range queries

RangeHot workloads with intensive writes.

D. Performance of range queries

In order to evaluate the range query performance of LSbM 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. 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 10, and the overall throughputs of all tests are given in Figure 11.

bLSM-tree: Compared to random data accesses, the range queries conducted on bLSM-tree are influenced less by intensive writes. 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. The throughput of the test using an additional Key-Value

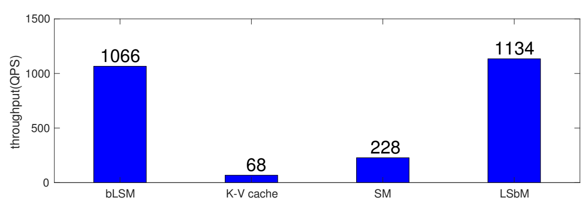


Fig. 11: Throughput comparisons of range queries

store is only 68 QPS. It is low because of two reasons. Firstly, when an additional Key-Value store cache is used, the memory space for buffer cache is reduced, which causes capacity misses. Secondly, 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. Firstly, when range queries must be served by disk, conducting range queries on one level with multiple sorted tables may need multiple disk seeks. Secondly, with the given write workloads, obsolete data will be piled in level 3 and loaded into buffer cache by range queries, which reduce the effective buffer cache capacity. As shown in Figure 10a, in the first 10,000 seconds, the total number of sorted tables searched for each query increases and the throughput decreases. At around 10,000 seconds, 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 shrunk 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 achieves the best performance. The buffer cache invalidation issue is further mitigated compared to bLSM-tree and the throughput is 1,134 QPS (Figure 10b). 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.

E. Database size

Among all the tests conducted above, LevelDB, bLSM-tree, SM-tree and LSbM-tree have different on-disk data structures, 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 12.

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 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

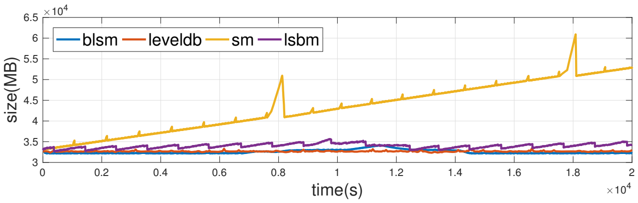


Fig. 12: Database size changes of tests on different LSM-tree variants

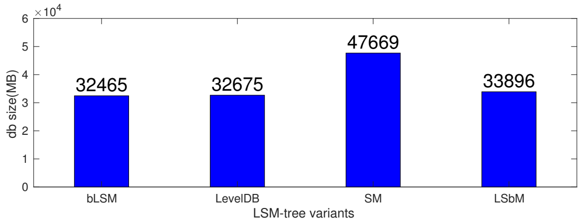


Fig. 13: Database Size of different LSM-tree variants

entire level together. As shown in Figure 12, the database size of SM bursts periodically. The small bursts observed every 1,000 seconds are caused by compacting sorted tables in level 1, and the big bursts observed every 10,000 seconds are caused by compacting sorted tables in level 2. The database size of LSbM is slightly greater than the bLSM and LevelDB, since the compaction buffer takes additional space. However, the size of the compaction buffer is limited due to the trim process.

The average database sizes of bLSM-tree, LevelDB, SM-tree, LSbM-tree can be found in Figure 13. It proves that with the given read and write workloads, the additional space cost of LSbM is very low, which is about 4% more space than that of bLSM and LevelDB. In contrast, the amount of repeated data due to lazy compaction in SM add as high as about 50% more disk space.

VII. Other related work

In order to reduce the compaction cost, some production systems only compact data partially in run time, 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 run time 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, this approach cannot avoid the interference from compactions to buffer caching. In practice, HBase still suffers low read performance during intensive writes [7].

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 [14]. However, both Key-Value cache and stepped merge methods have their limits as discussed in Section I. VT-tree is an extension of LSM-tree to avoid unnecessary merges for presorted data [17]. 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 [18]. 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 [19]. 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 [20] [21]. cLSM is an LSM-tree variant supporting scalable concurrency with multi-core processors [22]. WiscKey is proposed to separate the storage of key and value, which significantly eliminates the write amplification issue of LSM-tree compactions on SSD [23]. 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 is a low cost general solution to solve the fundamental problem of data caching in LSM-tree.

VIII. Conclusion

LSM-tree has been effectively designed and implemented for write-intensive workloads. However, when both read- and write-intensive workloads co-exist, the LSM-tree compaction induced buffer cache invalidations periodically drop cache performance for accessing high locality data. Existing methods to address the issue, such as a KV-store cache to bypass the buffer cache, SM by reducing the compaction frequency, and others, are partially effective, leaving several other performance issues, such as ineffective range queries and remaining LSM-tree induced invalidations. We propose LSbM 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 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 workload, the compaction buffer is adaptively shrunk and disappears eventually. Thus, the disk capacity overhead is minimized. In this paper, we have made a strong case for LSbM to best utilize buffer cache and disks for various workloads with both intensive reads and writes.

IX. Acknowledgement

We thank for the encouraging and critical comments from the anonymous referees. 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 paper draft was posted as a technical report in arXiv [24]. 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.

REFERENCES

[1] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[3] "Hbase," http://hbase.apache.org/.

[4] "Cassandra," http://cassandra.apache.org/.

[5] "Riak," http://basho.com/riak.

[6] "Leveldb," http://code.google.com/p/leveldb.

[7] M. Y. Ahmad and B. Kemme, "Compaction management in distributed key-value datastores," *Proceedings of the VLDB Endowment*, vol. 8, no. 8, pp. 850–861, 2015.

[8] "Rocksdb," http://rocksdb.org/.

[9] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[10] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*, vol. 97. Citeseer, 1997, pp. 16–25.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[12] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[13] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 217–228.

[14] "Datastax," http://www.datastax.com/dev/blog/compaction-improvements-in-cassandra-21.

[15] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[16] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 149–166. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lim

[17] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 17–30.

[18] X. Wu, Y. Xu, Z. Shao, and S. 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 Association, 2015, pp. 71–82.

[19] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1195–1206, 2010.

[20] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 16.

[21] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 266–277.

[22] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 32.

[23] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 133–148.

[24] L. Guo, D. Teng, R. Lee, F. Chen, S. Ma, and X. Zhang, "Re-enabling high-speed caching for lsm-trees," *arXiv preprint arXiv:1606.02015*, 2016.