Less is More: De-amplifying I/Os for Key-value Stores with a Log-assisted LSM-tree

Kecheng Huang*, Zhiping Jia*, Zhaoyan Shen*, Zili Shao†, and Feng Chen‡
*Shandong University, †The Chinese University of Hong Kong, and ‡Louisiana State University

***Abstract*— In recent years, Log-Structured Merge Tree (LSM-tree) based key-value stores, such as LevelDB and RocksDB, have been widely adopted in data center systems. Though optimized for high-speed write processing, the severe I/O amplification remains a critical constraint that hinders them from reaching their maximum performance potential. Unfortunately, this problem is deeply rooted in the fundamental design of the LSM-tree structure. A small number of frequently updated key-value items could quickly pollute the entire tree structure, causing repeated changes in the structure and quickly amplifying the amount of disk IOs across the levels in the tree. In this paper, we present a novel scheme, called *Log-assisted LSM-tree* (L2SM), to fundamentally address the long-existing I/O amplification problem. L2SM adopts a small-size, multi-level log structure to isolate selected key-value items that have a disruptive effect on the tree structure, accumulates and absorbs the repeated updates in a highly efficient manner, and removes obsolete and deleted key-value items at an early stage. We have prototyped the L2SM structure based on LevelDB. Our evaluation with the YCSB benchmark shows promising results by reducing the amount of disk IOs by up to 40.2%, increasing the throughput by up to 67.4%, and decreasing the average latency by up to 40.1%.**

***Index Terms*—Key-value store, LSM-tree, Bloom filter**

I. Introduction

In today's data centers, key-value (KV) data stores play a crucial role in providing high-speed data services for cloud applications. It is a great challenge to handle a massive amount of KV operations in a high-speed and efficient way. In the past decade, we have witnessed a wide-spread adoption of LSM-tree based KV stores, such as LevelDB [1] and RocksDB [2], as a popular solution in a real-world deployment [3]–[7].

LSM-tree based KV stores adopt a unique structure, which is particularly tailored for handling a large volume (billions or even more) of small KV data items. In LSM-tree, a basic storage unit, called *SSTable*, stores KV items sorted with their keys. The SSTables are organized in a multi-level structure. Each level maintains a sequence of SSTables with *non-overlapping* key ranges. A lower level contains several times more SSTables (wider) than its adjacent upper level, forming a tree-like structure. Incoming KV items are placed on the first level. If the number of SSTables at a level exceeds a preset threshold, selected SSTables are compacted into the lower level through merge sort operations. Upon a query operation, a binary search is performed, level by level (top-down), until finding the item or returning "not found".

The LSM-tree brings several important benefits. First, it converts small, random writes into sequential, append-only writes in large chunks, which optimizes I/O speed on disk. Second, it leverages the sorted data organization to remove the need for a complex indexing structure, greatly reducing the memory overhead. Third, it allows obsolete and deleted KVs to temporarily stay in the tree and remove them later via a compaction process, which eliminates small in-place writes.

All the above-said advantages make LSM-tree highly optimized for write-intensive workloads. However, several critical limitations inherent in this structure are fundamentally hindering LSM-tree based KV stores from achieving further performance improvement and must be addressed.

A. Critical Issues

As a fundamental design, the correct functioning of the LSM-tree structure relies on the *strict ordering* of KV items by their keys. However, enforcing such a strict order is very costly—each SSTable, and each level of SSTables need to be frequently reorganized, physically on disk. This process is highly inefficient due to the severe *I/O amplification* problem, which is caused by several intrinsic structural issues.

First, the data storage unit (SSTable) is much larger than the data operation unit (KV). To meet the requirement of strictly retaining the sorted and non-overlapping structure, LSM-tree frequently triggers merge sort operations at the granularity of SSTables. It means that even a slight change made by one KV update or insertion could incur a chain of operations to read, sort, and write a number of large SSTables, which raises severe I/O amplification problem.

Second, the locality information is lost and ignored in the data placement. Being sorted solely by the keys, "hot" data and "cold" data are mixed together within and across SSTables. A disturbing effect is that a small set of frequently updated KV data can repeatedly cause slight but heavy-cost changes. Due to the out-of-place writes, this small set of update-intensive data can quickly pollute many SSTables, spreading changes over the same level and involving both hot and cold data. This further magnifies the amplification effect.

Third, due to the tree-like structure, changes in the top-level are eventually rolled down to the bottom, level by level. Since the key range of an SSTable at a higher level often overlaps with a large number of SSTables at the lower level (multiple times), more SSTables may involve with merge sort operations, and the lower the level is, the more wide-spread such an effect would be. For this reason, the I/O amplification effect is further deepened and magnified, causing an "avalanche" across the LSM-tree levels, from top to bottom.

* Zhaoyan Shen is the corresponding author.

B. Optimizing LSM-tree Structure

The above-said problem, unfortunately, is deeply rooted in the fundamental design of the LSM-tree structure. As the KV data store scales up, these issues would become more prominent with a wider and deeper tree structure. A great challenge is—*how to retain the advantages of the LSM-tree based structure but avoid its negative effects?*

In this paper, we present a highly effective solution, called *Log-assisted LSM-tree* (L2SM), to fundamentally address the above-said issues. We aim to retain all the benefits brought by the basic LSM-tree structure but remove the detrimental I/O amplification effect. Our principal idea is to retain the stableness of LSM-tree to the maximum extent.

In essence, the source of all the previously mentioned problems is the frequent, disruptive changes to the tree structure, which forces the involved SSTables to be reorganized repeatedly. By removing, isolating, and delaying such changes, we can minimize their impact on the current status and stabilize the tree structure, which eliminates the root cause of I/O amplification from the beginning.

L2SM accomplishes this by extending the current LSM-tree with a separated log structure, called *SST-Log*. This log structure serves three important purposes. First, it is used as a buffer to isolate KV items that receive frequent updates. Hot KVs are separated from the LSM-tree, protecting the tree structure from being polluted repeatedly. Second, we identify and move "sparse" SSTables, which overlap with many SSTables in the lower level, out of the tree and give them a chance to be condensed in the log. Moreover, obsolete and deleted KVs are removed at an early stage, without being rolled down into lower levels, which avoids unnecessary I/Os and releases the occupied disk space. Third, the log also delays changes to the LSM-tree, which creates an opportunity to collapse multiple structure-impacting changes (e.g., overlapped updates, deletes) into a fewer number of operations.

All these optimization measures strive to achieve one goal—remove the operations that could potentially destabilize the LSM-tree structure as early as possible. In fact, due to the amplification effect, making such an optimization close to the source of the problem results in a super-linear reduction of I/O operations at greater effectiveness.

It is worth mentioning that our work is different from prior works in that L2SM focuses on addressing the structural problem of the original LSM-tree by proactively identifying and isolating the data that disrupts the tree structure in a separate log structure, which stabilizes the tree structure, removes unnecessary changes, and minimizes I/O amplification. Our contributions in this paper are summarized as follows:

- We introduce a novel Log-assisted LSM-tree to improve system performance and reduce disk IOs by enhancing LSM-tree with a small, dedicated log component.
- We design an auto-tuning Hotness Detecting Bitmap (HotMap) and an SSTable density estimation scheme to identify KV items that frequently cause small but heavy-cost changes to the LSM-tree.
- We propose Pseudo Compaction and Aggregated Compaction to condense the SSTables in the log and delay the changes to the LSM-tree.
- We have implemented a full-featured prototype based on Google's LevelDB [1] and performed a set of experiments to demonstrate the effectiveness of L2SM. We have also released the open-source code of L2SM [8].

The rest of the paper is organized as follows. Section II introduces background and motivation. Section III describes the design and implementation. Section IV gives the performance evaluation results. Section V describes the related work. The final section concludes this paper.

II. Background & Motivation

In this section, we introduce the background about LSM-tree based KV stores and give an example to explain the problem that motivates this paper.

A. LSM-tree based KV Store

A typical LSM-tree based KV store [9] consists of both in-memory and on-disk components. A *MemTable* (memory table) and an *ImmuTable* (immutable memory table) are maintained in main memory. A set of *SSTables* (sorted string table) store KV data persistently on disk and are logically organized in multiple levels. Except Level 0 (L0), the KV items of a level are sorted in order of the keys, meaning that the key ranges of SSTables on the same level are non-overlapping.

All KV write operations are first served in the MemTable. If the MemTable is filled up, its KV items are sorted and converted to an ImmuTable, which can no longer be updated. Then, the ImmuTable is appended to L0 as a persistent SSTable. This process is called *Minor Compaction*. Thus, the key ranges of different SSTables in L0 may have overlaps. If the size of L0 exceeds a preset threshold, a heavy-weight *Major Compaction* process is triggered to compact all overlapping SSTables to L1. The major compaction process performs *Merge Sort* operations to merge the L0 SSTables with the SSTables in L1 that have overlapping key ranges. The sorted KV items are written back to L1 in newly formed SSTables. Similarly, if the size of L1 or other levels exceeds their size limit, the major compaction process is triggered to select one or multiple SSTables for compaction into the adjacent lower level. In this way, the KV updates are rolled down from the top to the bottom, level by level.

Query handling in LSM-tree KV stores is simple. To locate a KV item in the LSM-tree, we start searching from L0 to the last level. Except for L0, where we need to search all SSTables due to the possible key-range overlapping, for the other levels, only one SSTable needs to be searched due to the sorted structure. Thus, a KV query usually incurs several SSTable searches. To speed up this query process, each SSTable maintains a bloom filter to quickly determine whether a key is possibly in the table or not.

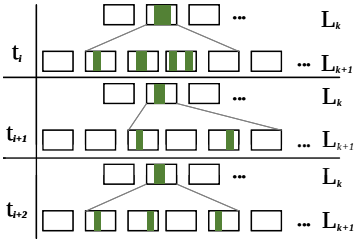

Fig. 1. An illustration of maintenance overhead in LSM-tree.

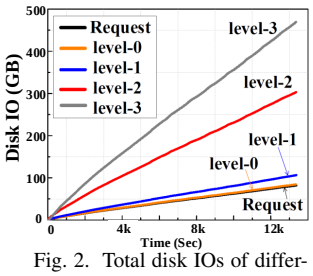

Fig. 2. Total disk IOs of different levels.

B. Motivation

The unique structure design of LSM-tree makes it highly efficient for managing a large amount of small KV items. However, in real workloads, a fast-pace incoming traffic of random, update-heavy KV requests can quickly pollute the LSM-tree structure, since the compaction process of LSM-tree has to maintain sorted, non-overlapping SSTables in each level, which spreads small, random and repeatedly updated entries down to lower levels. In LSM-tree, the deeper the level is, the heavier and more frequently the merge sort operations would be triggered. It forces the system to repeatedly undergo intensive, time-consuming maintenance operations, incurring huge I/O overheads.

We show a simple example in Figure 1 to illustrate the expensive maintenance cost. Assume two LSM-tree levels, $Level_k$ (denoted as L_k) and $Level_{k+1}$ (denoted as L_{k+1}). At time t_i, t_{i+1} and t_{i+2}, L_k is filled up and demands to compact one SSTable down to L_{k+1}. The selected SSTable overlaps with four SSTables at L_{k+1}. Thus, a merge sort operation involving five SSTables has to be performed each time. In total, there are 15 SSTables involved.

Specifically, due to the tree-like structure, the size of each level increases exponentially, from top to bottom. Compacting a victim SSTable at L_k would involve multiple times more SSTables (four SSTables in this example) at a lower level, L_{k+1}, causing an amplification effect. Even worse, due to the locality in real workloads, the three victim SSTables selected at L_k (each selected at time t_i, t_{i+1}, and t_{i+2}) are very likely to have overlaps (e.g., a few hot KV items could be repeatedly updated and pollute a range of KVs.). The amplification effect is further magnified over time.

To illustrate this effect, we have performed a preliminary test on LevelDB [1]. We randomly insert 80 million KV items to the data store. The size of each KV item is set as 1KB. The detailed experimental setup can be found in Section IV. Figure 2 shows the amount of disk IOs involved at each level along the time. We can see that, at each level, the disk IO amount increases with the arrivals of incoming requests, which is as expected. The amount of disk IOs of L0 increases at a rate nearly identical to the incoming requests. This is because the KV items buffered in memory are directly flushed into L0 as SSTables and the KV items of different SSTables in L0 are not sorted. Thus, there is almost no maintenance overhead in L0. However, as the level gets deeper, the growth rate of disk IO amount also increases. In other words, the lower the level is, the more disk IOs are involved, at an accelerating pace.

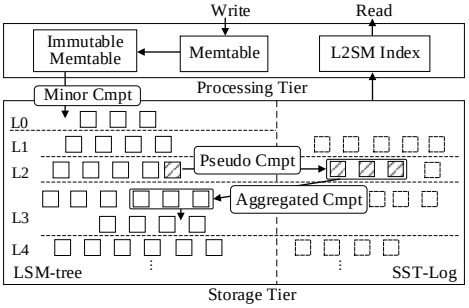

Fig. 3. Overview of L2SM architecture.

Thus, it is more severely impacted by the amplification effect. At the end of the test, the maintenance I/O overhead of L3 is up to 5 times as large as the amount of the incoming requests. This example vividly shows the I/O amplification over levels.

III. L2SM Design

We propose a highly efficient solution, called *Log-assisted LSM-tree* (L2SM), to fundamentally address the critical I/O amplification challenge. In this section, we will first give an overview of the proposed architecture, and then present the design details of each component.

A. Architecture Overview

A key design goal of L2SM is to retain the benefits of the LSM-tree structure, but remove the detrimental I/O amplification effect. L2SM accomplishes this by extending the core LSM-tree structure with a set of highly optimized mechanisms.

As illustrated in Figure 3, L2SM maintains KV data in both memory and disk storage. (1) **In-memory structures**: Similar to the classic LSM-tree, L2SM retains the two in-memory structures of LSM-tree, *MemTable* and *ImmuTable*, as a staging buffer to organize small, random KV items into large, sequential I/Os. (2) **On-disk structures**: Unlike the classic LSM-tree, data on the disk storage is divided into two parts, *LSM-tree* and *SST-Log*, respectively. The tree part functions similarly to the traditional LSM-tree. A critical change is the addition of the SST-Log part. SST-Log is a multi-level structure designed for achieving the key purpose—absorbing the operations that destabilize the LSM-tree structure. Except for L0 and the last level, each level of SST-Log aligns horizontally with the corresponding tree level, and also contains a list of SSTables. The purpose is to make the log absorb the most frequent and disruptive changes, protecting the tree in a stable status with minimal updates.

Data flow in the L2SM structure is as follows. (1) Incoming KV items are first packed into the MemTable and then converted to the ImmuTable. Next, the *Minor Compaction* (MC) process merges the ImmuTable into L0 as a persistent SSTable. (2) When the number of SSTables of any level exceeds the size limit, we monitor and identify the SSTables that could potentially impact the tree structure, such as those with a sparse or hot key range. A *Pseudo Compaction* (PC) process moves the selected SSTables into the same level's SST-Log, which is managed by a *Log Metadata Manager*. Note that PC does not incur any physical I/O but only updates the metadata structures. (3) If the size of an SST-Log level exceeds its limit,

an *Aggregated Compaction* (AC) process selects and evicts victim SSTables from the log and merges with the overlapped SSTables at the lower level of the tree. Hence, in L2SM, a KV item first moves horizontally from the tree to the log, and then moves back and vertically down to the tree, and so on. This process repeats and the log filters the disruptive updates out of the tree, level by level.

B. LSM-Tree and SST-Log

In L2SM, KV data are stored persistently in either the LSM-tree or the SST-Log. The two areas are *logically* separated by maintaining each level's log in a preset size proportional to the same level of the tree. The two service processes, PC and AC, are periodically activated to maintain the target sizes. In our prototype, the total size of all SST-Logs is set to no more than 10% of the LSM-tree. We will discuss the effect of SST-Log size in Section III-B2.

1) **LSM-tree***:* In L2SM, the LSM-tree part is designed to mostly maintain KV data that are rarely updated and have dense key coverage. The management in L2SM is similar to the traditional LSM-tree design the following differences.

First, in the original LSM-tree, the compaction process at each level merges SSTables into the next level, always in the top-down manner. In L2SM, the compaction process splits into two: the PC process moves SSTables horizontally from the tree to the log at the same level; the AC process merges SSTables from the log down into the lower level of the tree.

Second, the compaction process in the original LSM-tree selects SSTable for compaction based on the key-range order. In contrast, the PC and AC processes select SSTables based on their properties (hotness and density).

Third, the LSM-tree usually selects one SSTable for compaction into the lower level each time. The AC in L2SM usually selects multiple SSTables from the log for creating a denser structure and better I/O performance.

2) **SST-Log***:* SST-Log, a multi-level log structure, is maintained as an extension to the LSM-tree. SST-Log serves four main purposes: (1) It provides an isolated space to separate the "hot" (frequently updated) data, which repeatedly pollutes the structure, out of the tree. (2) It provides a buffer zone to identify and condense the "sparse" SSTables, which contain a few keys covering a wide range, before merging them into the tree. (3) It delays and mitigates disruptive operations, e.g., accumulating multiple updates into one. (4) It allows us to remove the obsolete and deleted data out of the tree early.

The basic structure of SST-Log is shown in Figure 3. Except for L0 and the last level, a linked list of SSTables, called a *log*, is maintained for each level. The log organizes the SSTables that are selected and moved from the same level of the tree in a *unidirectional* manner. That means, once an SSTable is moved from the tree into the log, it either stays in the log or is further merged by AC down into the lower level of the tree, but it never moves back to the same level of the tree. It guarantees that the most recent version can always be found if following the right order (e.g., $Tree_n$ *!* Log_n *!* $Tree_{n+1}$ *!* Log_{n+1}...).

Unlike at a tree level, the SSTables in a log may have overlapped key ranges. Multiple versions of the same KV items could co-exist simultaneously in a log. When looking for a KV item, all the SSTables whose key ranges cover the target key need to be searched. In contrast, in LSM-tree, the SSTables of a level are sorted by their keys and strictly non-overlapping (a key only appears in one SSTable). Allowing such overlapping is essential in L2SM, since the log is designed to absorb and accumulate multiple updates, which can be collapsed into one in the AC process. To accelerate the in-log search, we use Bloom Filters for quick filtering. More details will be discussed in Section III-D.

Determining the log size. SST-Log is a multi-level structure. The size (i.e., the number of SSTables) of each level's log has a limit. A naïve solution is to set the log sizes of different levels to the same, or the same percentage proportional to the tree size. Both solutions are sub-optimal for two reasons.

First, the log of each level serves as a buffer, which filters hot and sparse SSTables out of the tree. Due to the filtering effect, the lower the tree level is, the colder and the denser the SSTables are. It means that maintaining a large log may become unnecessary for lower levels. Second, due to the pyramid-like shape of the LSM-tree, the lower levels contain more SSTables than the upper levels, so maintaining the same size or percentage would be either too large or too small.

We design a scheme, called *Inverse Proportional Log Size*, to determine the log size of each level. It works as follows. The log size of a level is proportional to the size of the same level in the tree. From the top-level down, the log-to-tree ratio (proportion) of each level decreases, meaning that an upper level has a larger ratio while a lower level has a smaller ratio.

Assume ω is a preset percentage (e.g., 10%) of the entire SST-Log size to the LSM-tree size with h levels, and λ is the log-to-tree size ratio of the first level. For a given m (L0 size) and g (increasing rate of levels for the LSM-tree), we can calculate the value of λ to meet the requirement $\sum_{j=1}^{h-1} m \quad g^{j} \quad \lambda^{j} < (\sum_{i=0}^{h-1} m \quad g^{i}) \quad \omega$. Note that due to the pyramid-like shape of the tree structure, the decreasing ratio unnecessarily means that the log size decreases. For example, if a tree has 10 SSTables at L2, the log size of L2 is 5 SSTables (50%), and the tree has 20 SSTables at L3, the log size of L3 could still be 5 SSTables (25%). When the number of SSTables exceeds the log size limit, AC is activated to select and merge SSTables down to the next level.

C. Hotness and Density

To mitigate the disruptive interference to the LSM-tree structure, we need to identify the frequently updated and the sparse SSTables, and isolate them in the SST-Log part to stabilize the LSM-tree structure.

We use two metrics, *hotness* and *density*, to quantitatively measure the properties of an SSTable. *hotness* measures how frequently the KV items of an SSTable are updated. *density* measures how large the key range of an SSTable covers and impacts. The two metrics together describe the severity of

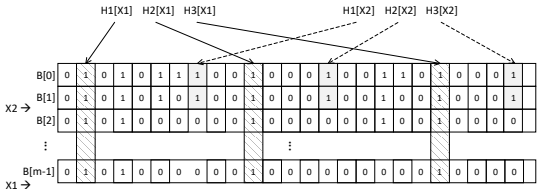

Fig. 4. An illustration of the HotMap scheme.

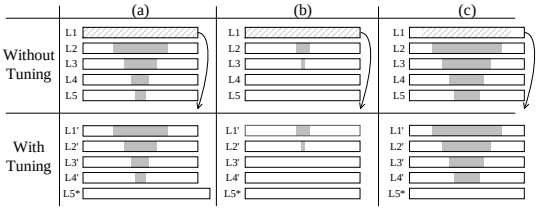

Fig. 5. An illustration of HotMap auto-tuning.

the potential disruptive interference that could be caused by holding the KV items in the LSM-tree. The former describes the temporal property (frequency), and the latter describes the spatial property (breadth).

1) **Hotness of SSTables***:* L2SM maintains a *Hotness Detecting Bitmap* (HotMap) to quantitatively measure the hotness of an SSTable. HotMap is a global, in-memory data structure consisting of multiple layers of *bloom filters* [10].

Bloom Filter. A bloom filter is an array of P bits, which are initialized to 0. A set of K hash functions (e.g., MurmurHash [11] with K seeds) are used to determine the corresponding K bits in the bit array. Upon an update to a key x, the k-th hash function $H_k(x)$ is computed and points to a bit in the array, which is set to 1. All K bits are set in such a way. Upon a query for a key, if all the K corresponding bits are found set, it indicates that the key has been updated.

Note that a bloom filter may have false positive but never have false negative. To control the false positive rate, a bloom filter of P bits is expected to record at most N unique keys (a.k.a. the *capacity* of the bloom filter) with a low false-positive rate. The capacity N is designated when the bloom filter is created. More details can be found in prior work [12].

Hotness Detecting Bitmap. We use multiple layers of bloom filters to record an abstract history of key updates. As shown in Figure 4, an M-layer HotMap is composed of M aligned bloom filters. The i-th update to a KV item sets the corresponding bits in the i-th bloom filter. Thus, a M-layer HotMap can record up to M updates for any given KV item. We do not further differentiate updates over M times. If m bloom filters indicate that a KV item is updated, we can determine that the KV item has been updated for no less than m times. By calculating the number of positive responses by HotMap layers, we can determine the relative hotness of a key. As shown in the figure, the number of updates of key X1 is m and the number of updates of key X2 is 2.

Hotness value calculation. An SSTable's hotness is determined by its contained keys. The hotness value of an SSTable is calculated as $\sum_{i=1}^{m} {}^{1}(x_i \quad 2^i)$, where x_i is the number of keys in the SSTable that are indicated positive in the i-th bloom filter (i.e., the number of keys being updated for i times). Note that an exponential function is adopted to assign different weights to different bloom filter layers, because we desire to identify SSTables that contain frequently updated hot keys, rather than many warm keys. In other words, the higher the layer is, the more vital it is.

Configuring HotMap. The effectiveness of the HotMap is determined by two important parameters, M and P. A critical challenge is how to determine a proper configuration for the two parameters, and how to retain their effectiveness and adapt to the workloads during runtime.

Configuring M. M, the number of HotMap Layers, determines how many updates we can use to differentiate hot and cold keys. A large M means a more accurate HotMap but also higher memory overhead. We use a simple method to set M. For a given workload with r requests to access n unique keys, the average number of requests being received by a key is $\tau = r/n$, and we use $M = \lceil r/n \rceil$ to set the number of layers. The rationale is simple—if a KV item has been updated more frequently than the average, it is considered as a hot item, and there is no need to further track its exact number of updates; otherwise, it is regarded as a cold item (below average). The τ is about 4.54 in a Skewed Zipfian distribution, and 2.32 in a Scrambled Zipfian distribution. Thus, in our prototype, we set M to 5 layers, which is more than sufficient.

Configuring P. P, the bit array size, determines the effectiveness of a bloom filter. With a small bit array size (a small P), the bloom filter would suffer a high false-positive rate. Let ρ represent the ratio of hot keys (being updated more than τ times) to all the unique keys in a workload. Assume N unique keys in total and K hash functions used in the bloom filter, according to prior study on bloom filters [12], the bit array size should be set as $P = \frac{K \quad N}{\ln 2}$. In a Skewed Zipfian distribution, ρ is 6.5%, and 5% in Scrambled Zipfian distribution. According to the unique keys in our workloads, our prototype sets P to 4 million bits initially.

Auto-tuning HotMap. As time elapses, the HotMap will be gradually filled up, eventually losing the ability of reliably differentiating hot and cold keys. Thus an important task is to ensure the HotMap to be adaptive to the workloads and to be continuously effective during runtime.

To retain a low false-positive rate, we need to expand and update the HotMap periodically. A simple method is to keep track of the first (top) bloom filter, which records the oldest KV updates. If the first bloom filter is approaching to its capacity limit (a.k.a., having received N unique keys), we retire this (oldest) bloom filter by resetting all its bits to 0, enlarge its capacity by 10% and rotate it to the bottom layer; the second bloom filter, in turn, becomes the new top bloom filter. Thus, the HotMap size can be expanded.

This simple solution has two potential problems. First, the HotMap size may keep growing quickly. Second, adjacent layers may receive the same number of keys, thus cannot provide enough information for us to differentiate hot and cold keys. To handle these two issues, we propose an *Online Adaptive Auto-tuning* scheme to automatically adjust the M bloom filter arrays.

Several scenarios would trigger the tuning scheme to adjust the array size. As shown in Figure 5, if the first (top) bloom

filter is approaching to its capacity limit, which indicates that the HotMap is too small to reliably serve the purpose with a low false-positive rate, we retire this layer. When retiring this layer, we further check its following layer. If the size of the following layer is consumed more than 20%, we enlarge the size of the top bloom filter by 10%, reset all its bits to 0, and rotate it to the bottom layer, as shown in Figure 5(a); Otherwise, if the following layer is consumed less than 20%, to save space, we directly make the size of the top bloom filter the same as the current bottom layer, reset all its bits to 0, and rotate it to the bottom layer, as shown in Figure 5(b). The rationale is that if the second layer does not receive many keys, it is very likely that most keys are cold (updated only once) and the working set is not growing, thus the current HotMap size is sufficient.

Suppose the top bloom filter is large enough to contain all unique writes, if the unique keys accepted by any two adjacent layers are too close (e.g., the difference of accepted insertions between two layers is less than 10%, and each layer occupies more than 20% of the layer size), it means that the two adjacent layers are similar, which happens when the set of keys is repeatedly updated. Thus we may remove one layer to retain the effectiveness of the HotMap. In this case, we also retire the top filter layer by resetting and rotating it to the bottom layer, as shown in Figure 5(c). The size of the rotated layer equals the size of the current bottom layer.

The purpose of the above-said mechanism is to ensure that the multiple layers of the HotMap are able to adapt to workload changes, and also provide enough information for us to differentiate hot and cold keys. In Section IV, we run a set of workloads with various distributions to evaluate the effectiveness of this scheme.

Overhead. Maintaining and updating the HotMap would incur additional memory and computational cost. The memory overhead is relatively small, with $M \quad P$ bits in total. For typical workloads, M is 5 and P is 4 million, the overhead is roughly 2.5 MB. For different workloads, the HotMap ranges from 2.5 million to 40 million bytes.

The hash functions also incur extra computational overhead. Ideally, upon each key update, the HotMap should be updated, involving K hash-function calculations. To avoid the excessively high computational burden, we only update the HotMap when the KV items are compacted from L0 to L1. This is for two purposes. First, it avoids introducing a perceivable delay for each in-memory update in MemTable (the critical path). Since we only perform hash calculations when slow compaction I/Os happens, the delay is made asynchronous and negligible. Second, we should note that our purpose is to roughly differentiate the *relative* hotness, rather than accurately count the number of updates. Losing some accuracy would not incur significant differences, but brings significant speedup. Although we may miss some KV updates in memory, this optimization is acceptable because the updates happened in memory would not incur extra disk IOs anyway.

2) **Density of SSTables***:* Density is another important factor affecting the tree structure. A dense SSTable has a large number of KV items concentrated in one small range, which potentially overlaps with fewer SSTables of the lower level, and during the compaction, fewer SSTables would be involved in merge sort. Thus, it is desirable to isolate sparse SSTables in the log, leaving more dense SSTables in LSM-tree.

In L2SM, SSTables are of the same size (in most cases) and the KV items in an SSTable are sorted. We use the ratio of the number of KV items to the key range of an SSTable to indicate its density as follows.

An SSTable's key range is the difference between the first key and the last key within the SSTable. In real workloads, however, keys can be in different forms, such as a string of a fixed length (e.g., 16 bytes) or a random number. Thus, we cannot directly perform numeric subtraction between different keys. We simplify this procedure by converting the keys into a 128-bit binary value. For example, a string character is converted into its ASCII value. Then, we compare the two 128-bit binary values (the first and the last keys), bit by bit, to find the highest bit that differs in the two keys. Assuming the highest bit that differs is the i-th bit, the key range of this SSTable can be roughly estimated as 2^i. If the SSTable contains k KV items, its density is calculated as $k/2^i$. To simplify the calculation, we use the logarithm as the density value of the SSTable, which is $\lg(k/2^i) = \lg k \quad i$. In this work, we also use its inversion, $S = i \quad \lg k$, to describe the *sparseness* of an SSTable, as an alternative way to denote its density. For an SSTable, once it is created, its sparseness value S can be calculated. The larger the S is, the more lower-level SSTables would be involved in compaction.

As described above, the hotness and density (sparseness) values quantitatively represent the potential disruptive impact that maintaining an SSTable in LSM-tree could make. The pseudo compaction and aggregated compaction process leverage the two values to determine which SSTables should be held in the tree or isolated in the log.

D. Pseudo Compaction

At each level of the L2SM structure, Pseudo Compaction (PC) is responsible for extracting selected SSTables from the LSM-tree into the SST-Log. According to the property (hotness and density) of SSTables, PC can easily identify the "hot" and "sparse" SSTables, which would severely affect the stability of the tree structure, and isolate them in the log area, thus mitigating the I/O amplification effect.

In the traditional LSM-tree design, when the number of SSTables on a tree level exceeds a limit, the compaction process is activated to merge a number of SSTables into the lower level. In L2SM, PC is also triggered when a tree level is full, but the selected SSTables are moved *horizontally* into the same-level log without any change, involving no merge sort operations or any disk IOs.

Victim SSTables. To identify the best SSTables to move into the log, we use a *Combined Weight* to consider both hotness and density to quantitatively determine the SSTable's value of being held in SST-Log. Given an SSTable i, assume its hotness is H_i and sparseness is S_i, we use a weighted function

W_i = α H_i + (1 α) S_i to calculate a combined value, where α is a preset weight parameter (0.5 in default).

For hotness H, the larger the hotter; for sparseness S, the larger the sparser. To calculate the combined weight of an SSTable, both S and H need to be first normalized to the same scale, 0 1. The normalization is as follows.

When PC happens, we record the maximum hotness H_{max} and the minimum hotness H_{min} of all the under-checking SSTables. Thus, the normalized hotness of SSTable i can be expressed as $\frac{H_i}{H_{max} \quad H_{min}}$. Similarly, for sparseness, when PC happens, we record the maximum sparseness S_{max} and the minimum sparseness S_{min} of all the under-checking SSTables. S_i can be normalized as $\frac{S_i}{S_{max} \quad S_{min}}$. Therefore, the combined weight W_i for an SSTable can be calculated by W_i = α $\frac{H_i}{H_{max} \quad H_{min}}$ + (1 α) $\frac{S_i}{S_{max} \quad S_{min}}$.

When a tree level is filled up, PC moves the SSTables to the log, in the order of their combined weights from high to low. The SSTable with the highest combined weight is selected first and detached from the LSM-tree, and moved into the log structure. Note that this only involves metadata updates (several linked list operations) without physical data movement on disk; merge sort operations are not needed either. SSTables moved to the log are organized in a linked list. This process repeats until the number of SSTables is below the limit.

Miscellaneous issues. Compared with the LSM-tree levels, a major difference in the SST-Log is that the SSTables in one log level are not sorted and the key ranges of different SSTables may have overlaps. This has two effects.

First, we cannot simply use a binary search to only access one SSTable to search for a target key. We need to search all the SSTables whose key ranges covering the target key, involving multiple disk IOs. To accelerate the query process, we maintain in-memory bloom filters for SSTables in the log. Upon a query, we first find SSTables having the related key ranges. Then we use the bloom filter to quickly locate the SSTables that may contain the target key, and finally perform the in-SSTable lookup.

Second, we may have multiple versions of the same KV item at one level. Since the SSTables in the log also maintain the related metadata that can indicate their freshness, L2SM always begins the search from the newest SSTable that possibly contains the target key. Once the target key is found, we stop the search process and return the item without further searching in the other SSTables. This minimizes the involved I/Os and also guarantees the correctness.

E. Aggregated Compaction

Aggregated Compaction (AC) is responsible for reclaiming the log space. It attempts to retain the most structure-impactful SSTables in the log, and return the cold and dense SSTables back to the lower level of the tree. When AC happens, we need to particularly consider the following issues.

Maintaining the query correctness. A key may have multiple versions of value data existing in the log. It is desirable to accumulate and collapse these multiple versions into one before merging into the lower level. However, this process may not be able to be completed at one time (in order to avoid the high overhead). Thus, we must ensure that the same-key data are evicted/merged in a strict chronological order. In other words, the lower-level tree should never contain data newer than the upper-level log.

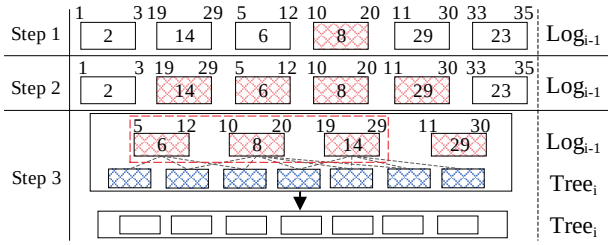

Fig. 6. An example for aggregated compaction.

Considering both density and hotness. AC needs to consider both hotness and density of an SSTable to determine whether to continue holding it in the log or not. An extremely sparse or extremely hot SSTable should continue to remain in the log. We need to have a balanced way to integrate both considerations when choosing SSTables for eviction.

Controlling the involved I/Os. When merging an SSTable from the log back to the tree, multiple SSTables in the lower-level tree could be involved (due to the spanning structure). A sparse SSTable may incur a merge sort operation involving a large number of SSTables. To control the cost of such I/Os, we need to estimate the number of involved SSTables and ensure the incurred I/Os under a reasonable level.

Removing deleted and obsolete data. Deleted and obsolete data could also appear in the log. Such data should be removed during AC, rather than drained down to the lower level, thus mitigating the unnecessary I/Os at an early stage.

Compaction process. When the log exceeds its size limit, AC is triggered to return the cold and dense SSTables back to the tree, while keeping the hot and sparse SSTables in the log. To determine the SSTables for eviction, we calculate the combined weight of hotness and density W of all the SSTables in the log (see Section III-D).

The whole process is as follows: (1) Find the coldest-densest "seed" SSTable that has the smallest W and use this "seed" to recursively find all the SSTables in the log that have key range overlapping with it. Sort all the SSTables based on their version order. (2) Based on the SSTables from Step 1, starting from the oldest SSTable, put it into the victim *Compaction Set* (CS), find the lower-level SSTables in the tree that have overlapping key ranges with SSTables in CS and put them into the victim *Involved Set* (IS). (3) Repeat Step 2 until all the SSTables found in Step 1 are placed in the CS or the ratio of SSTables in the IS and CS is larger than a predefined value (configured as an empirical value 10 to control IO amplification caused by AC). (4) Finally, start real merge sort. Starting from the oldest one first, collapse the SSTables and remove all deleted and obsolete KV items. Then the keys are merged with the overlapping SSTables on the lower tree level, and finally we insert the generated new SSTables into the lower tree level, which completes the AC process.

Figure 6 provides an example to illustrate how AC works. The number in a block indicates its chronological order (the

smaller the older). Suppose there are six SSTables in log and SSTable 8 with key range "10-20" is the coldest-densest seed. There are three SSTables (6, 14, and 29) overlapping with it. According to the working process of AC, the victim CS includes three SSTables (14, 6, and 8) in the first batch. Although 29 also overlaps with the seed, it is set aside first, since it exceeds the IO limit. Finally, the three SSTables (6, 8, and 14) are compacted with the SSTables that have key range overlaps in the lower tree level.

The aforementioned design brings several benefits. First, we can control the involved SSTables in each compaction process, avoiding the high I/O impact. Second, the old-version data are always drained downstream to the next level before the new version, which guarantees the correctness. Third, multiple overlapping SSTables in the log are merged first. The deleted and obsolete data are removed first, which reduces the amount of KVs before merging with the lower-level SSTables. Finally, both hot and sparse SSTables are safely maintained in the log, ensuring the efficacy of the log structure.

IV. Performance Evaluation

In this section, we first introduce our L2SM prototype system and then present and analyze the experimental results.

A. Experimental Setup

We have prototyped L2SM based on Google's LevelDB [1] by adding about 2,000 lines of code. The added code is mainly in `db_impl.cc` and `version_set.cc`, which are for compaction management and version control, respectively.

To improve the read performance, L2SM maintains additional bloom filters in memory. The original LevelDB maintains a bloom filter on disk for each SSTable, which is loaded into memory when needed. For a fair comparison, in this work, we implemented a version of LevelDB that also uses in-memory bloom filters. We denote the stock LevelDB as "OriLevelDB", and the enhanced LevelDB as "LevelDB". For each implementation, we configure the SSTable size to 5 MB. The capacity growth factor of adjacent levels is 10. Other parameters are configured using the default values of the original LevelDB.

Our experiments are conducted on a workstation, which features an Intel i7-8700 3.2GHz processor, 32GB memory, and a 500GB SSD. For the software, we use Ubuntu 18.04 LTS with Linux Kernel 4.15 and Ext4 file system. For the benchmark, we have extended the standard `db_bench` tool with the Yahoo! Cloud Serving Benchmark (YCSB) suite [13], in which the workload generator is wrapped as a class named `generator`. Workloads with three different types of distributions, namely *Skewed Latest Zipfian*, *Scrambled Zipfian*, and *Random*, are tested for the evaluation. Each distribution provides a set of workloads with different combinations of KV operations. These distributions can be accessed through API functions, `sk_zip`, `scr_zip` and `normal_ran`.

B. Overall Performance

We first compare the overall performance of the proposed L2SM and LevelDB in terms of throughput (Thousand Operations per Second, a.k.a. KOPS) and average latency. In this set of experiments, we first randomly load 50 million KV items and then issue 50 million mixed KV read/write requests with different distributions to the KV store. The size of KV items varies from 256B to 1KB.

Figure 7(a) presents the evaluation results of workloads with the Skewed Latest Zipfian distribution. The horizontal axis represents the Read:Write ratios (from 0:1 to 9:1) of each workload. The left and right vertical axes show the throughput and latency, respectively.

As shown, L2SM outperforms LevelDB in both throughput and latency across the board. When the Read:Write ratio is 0:1 (write-only), L2SM achieves a throughput of 29.3 KOPS, which is 67.4% higher than that of LevelDB (17.5 KOPS). The throughput gain is mainly due to the collaboration of the proposed PC and the AC. During the working process, PC selects KV items that seriously pollute the LSM-tree and AC moves others to the lower tree level. Thus, the maintenance overhead is significantly mitigated. With the increment of read requests, the performance gain of L2SM over LevelDB decreases. In particular, when the Read:Write ratio of the workloads are 1:9, 3:7, 5:5, 7:3 and 9:1, the relative performance improvement decreases to 59.5%, 41%, 32.5%, 28.4% and 8.7%, respectively. This is as expected, since the optimization strategies mainly focus on writing process. Actually, for read operations, in addition to searching the LSM-tree, L2SM also needs to search the SST-Log. This extra search process would introduce overhead as analyzed later in Section IV-D. Meanwhile, we should note that L2SM still maintains a higher throughput than LevelDB even for very read-intensive workloads. This also indicates the effectiveness of our proposed optimizations like the in-memory bloom filters.

The average latencies of L2SM and LevelDB with different workloads show a similar trend. When there are only write requests in the workload (Read:Write ratio is 0:1), the average request latency of L2SM is 34.11 μs, which is 40.1% lower than that of LevelDB. As the ratio of read request increases, the relative latency improvement decreases. For the workloads with Read:Write ratio being 1:9, 3:7, 5:5, 7:3, and 9:1, the latency improvements of L2SM over LevelDB are 37.3%, 29.1%, 24.6%, 22.1%, and 8%, respectively.

We further evaluate L2SM with the workloads of the Scrambled Zipfian and the Random distributions. The results are shown in Figures 7(b) and (c).The similar trend can be observed. L2SM achieves higher improvement for both throughput and latency with write-intensive workloads. For workloads of the Scrambled Zipfian distribution, the highest performance gain of L2SM over LevelDB are 46.3% and 31.3% for throughput and latency, respectively. For workloads of the Random distribution, the highest performance gains of L2SM over LevelDB are 29.6% and 22.9% for throughput and latency, respectively. Note that the performance improvement

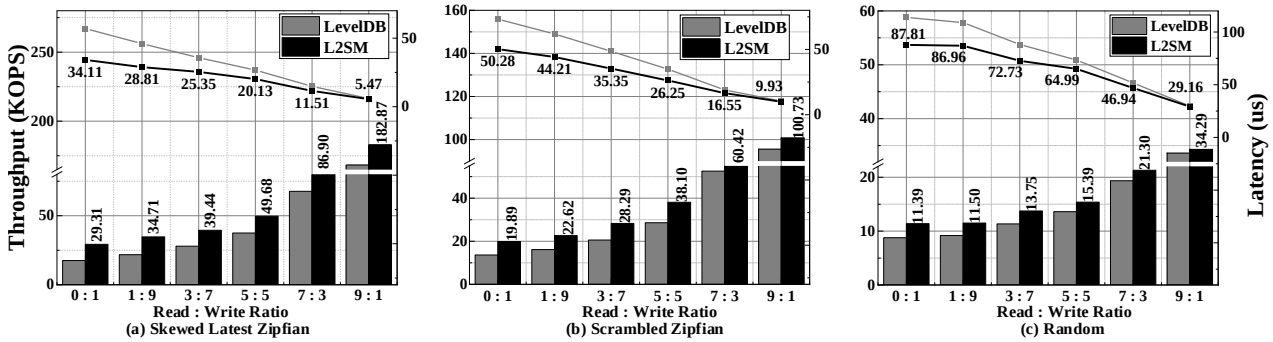

Fig. 7. Throughput and latency for workloads with different Read:Write ratios.

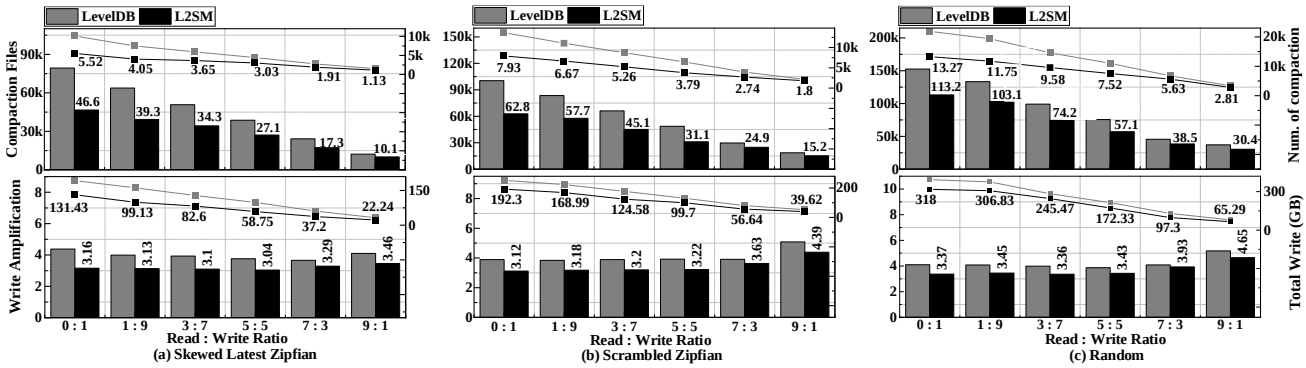

Fig. 8. Occurrences of compaction, involved files, write amplification and total writes for workloads with different Read:Write ratios.

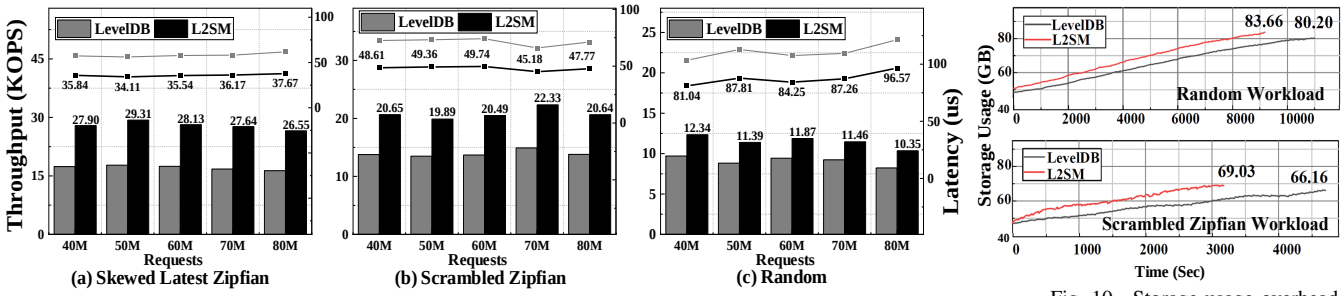

Fig. 9. Performance for workloads with different numbers of requests.

Fig. 10. Storage usage overhead.

is the lowest for the workloads with the Random distribution. This is because the Random distribution workloads have the weakest access locality and receive the least benefits.

C. Compaction Effect

The write amplifications (WAs) of these two platforms under different workloads are shown in Figure 8. We can observe that for different workloads, the WAs of LevelDB range from 3.19 to 5.18. However, the WAs of the proposed L2SM design is much lower, ranging from 3.04 to 4.65. For the write-only workload (the Read:Write ratio is 0:1) of the Skewed Latest distribution, L2SM achieves the highest WA improvement, which is 27.8%. With a read-intensive workload (the Read:Write ratio is 9:1) of the Random distribution, L2SM has the lowest WA improvement, which is 17.8%. Based on the result, we can conclude that the proposed L2SM design effectively reduces the KV store maintenance overhead.

Figure 8 also gives the occurrences of compaction and involved files of L2SM and LevelDB with different workloads. In L2SM, by isolating those update-intensive and sparse items in the SST-Log, the occurrences of compaction are reduced remarkably. With the write-only workload (Read:Write = 0:1) of the Skewed Latest Zipfian distribution, L2SM triggers 5,523 compactions, which is 45.4% lower than that of LevelDB. The SSTables involved in these compaction operations also decrease from 79,382 to 46,654, which is 41.2% lower. When the ratio of read requests increases to 90%, L2SM can still reduce the occurrences of compaction and the involved SSTables by 25.6% and 17.6%, respectively. For the mixed workloads of the Scrambled Zipfian distribution, L2SM also reduces the occurrences of compaction by 16.8%–42.1%, and the number of involved SSTables by 17.9%–37.5%, compared with LevelDB. Even for the Random workload, which carries fewer hot keys with a weak locality, L2SM still reduces the number of compaction operations by 16.7%–39.4%, and involved SSTables by 18.2%–25.8%, compared to LevelDB.

We further measure the total number of disk IOs of the two designs with different workloads. For the write-only workload, the size of the incoming KV items is about 25 GB. For the workloads of the Random distribution, the total disk IO amount of LevelDB is nearly 398 GB, which is 16.76 times larger than the original input. For L2SM, in contrast, the total disk IO amount is about 318 GB, which is 20.1% lower than that of LevelDB. The workloads of the Random distribution

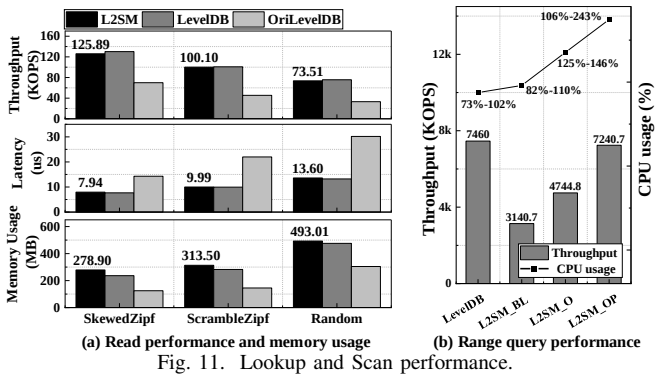

Fig. 11. Lookup and Scan performance.

suffer the highest total disk IO. This is because the random requests have weak data locality, which would incur higher maintenance overhead and achieve less benefit. Meanwhile, L2SM still reduces the most total disk IO amount with the workload of the Skewed Latest Zipfian distribution, which is 40.2% lower than that of LevelDB (219.8GB).

D. Read Limitation

Read. Figure 11(a) depicts the read performance and the relevant memory usage of different workloads between LevelDB and L2SM. Specifically, L2SM provides roughly the same throughput and latency as LevelDB. The throughput of L2SM is inferior to LevelDB by 0.55%–2.82%. The same situation occurs in latency, where L2SM is 0.65%–3.40% slower than LevelDB. As for memory usage, L2SM needs 7.5% to 11.3% more memory space than LevelDB to maintain the bloom filters for SSTables in the log. Compared with the "OriLevelDB" (with on-disk bloom filters), both L2SM and LevelDB show great enhancement on latency and throughput ranging from 44.5%–54.9% and 86.2%–128.3%, respectively. The cost is that more memory space is needed to maintain their bloom filters (61%–123%, compared to OriLevelDB).

Range Query. To evaluate the performance of range query, we use the YCSB benchmark to first issue 50 million KV pairs to populate the database, and then perform 10 million range query requests. The size of these KV pairs varies from 256B to 1KB. The results are shown in Figure 11(b). *L2SM_BL* denotes the results of L2SM without any optimization for range queries; *L2SM_O* denotes the design that organizes the SSTables in each SST-Log in an ordered manner; *L2SM_OP* denotes the design that also uses the parallelized search method to perform range queries with two threads. As shown, compared with LevelDB, L2SM without any optimization suffers 57.9% throughput reduction for range queries. Organizing the SSTable in a sorted manner accelerates the process, which alleviates the performance degradation to 36.4%, compared with LevelDB. Parallelizing the search operations further improves the throughput and almost completely hides the performance loss (only 2.9%). Since more threads are involved, the CPU consumption of *L2SM_OP* is higher than the stock L2SM.

E. Scalability

To evaluate the scalability of L2SM, we measure the L2SM performance with an increasing number of requests. Same as the above experiments, 50 million KV items are loaded into L2SM first. Figure 9 shows the performance in terms of throughput and latency. When the number of requests increases from 40 million to 80 million, L2SM shows no obvious performance degradation compared with LevelDB. The relative throughput improvement over LevelDB is retained at 60.4%–65.2% for the Skewed Latest Zipfian, 47.4%–50.1% for the Scrambled Zipfian, and 24.2%–29.1% for the Random distribution. The latency also shows a stable improvement, 37.5%—39.1% for the Skewed Latest Zipfian, 31.5%–33% for the Scrambled Zipfian, and 20.2%–22.2% for the Random distributions. As for total I/O amount, L2SM saves disk IO at the rate of 41.1%–43% for the Skewed Latest Zipfian, 30%–32.1% for the Scrambled Zipfian, and 21.8%–24.1% in the Random. These results show that L2SM scales well for handling a large amount of KV requests.

F. Comparison with RocksDB and PebblesDB

We further compare the performance of L2SM with two representative, the state-of-the-art LSM-tree based KV stores, RocksDB [2] and PebblesDB [14]. When comparing with PebblesDB, we increase the log-to-tree ratio from 10% to 50%. Note that the space overhead of PebblesDB over LevelDB is about 200%. The other parameters of L2SM are the same as in Section IV-A. We use the workloads described in Section IV-B to evaluate the performance of the three KV stores.

Figure 12 shows the system latency, throughput, total write, and disk usage comparisons of L2SM with RocksDB and PebblesDB under workloads with the Skewed Zipfian, Scrambled Zipfian, Random, and Uniform distributions. As shown, L2SM outperforms RocksDB across the board. For latency, the improvement of L2SM over RocksDB ranges from 34.9% to 61.1%; for throughput, the improvement is 55.6% 159.5%; for disk IO, L2SM reduces disk writes by up to 69.8%.

Similar trends can be observed when comparing L2SM with PebblesDB. Although L2SM uses much less extra disk space, L2SM outperforms PebblesDB in all the workloads except the Uniform distribution. With the Skewed Zipfian workload, L2SM receives the highest performance gain, 17.9% and 14.4% in terms of throughput and latency, respectively. With the Random workload, L2SM outperforms PebblesDB by 9.9% and 7.3% for throughput and latency, respectively. The overall disk IO saving of L2SM over PebblesDB ranges from 15% to 26.5% in the three workloads.

We have also evaluated the tail latencies for the Skewed Zipfian workload running with the three KV stores. L2SM's 99th percentile tail latency is 0.03% and 0.18% lower than PebblesDB and RocksDB, respectively, which means that L2SM's tail latency remains at a low level with substantially improved throughput.

Our test with the Uniform distribution simulates an append-mostly workload, which has more than 60% of KVs never being updated and 30% being updated only once in a uniform, random manner. Even for such a challenging workload, L2SM incurs minimal overhead. L2SM shows only 1.4%, 2.5%, and 1.7% performance loss in terms of throughput, latency, and

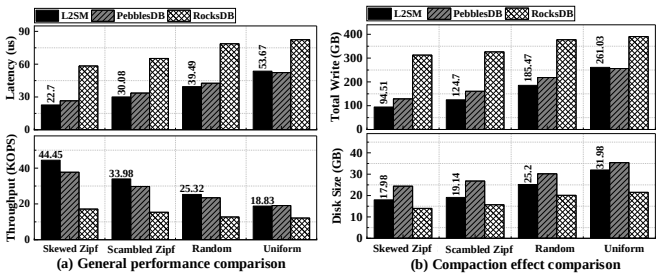

Fig. 12. Performance and I/O comparison with PebblesDB and RocksDB.

disk IO, respectively, compared to PebblesDB. Compared to RocksDB, L2SM outperforms it in all workloads.

In terms of disk space consumption, both L2SM and PebblesDB need extra space to maintain the SST-Log structure and the Fragmented-LSM-tree structure, respectively. As shown in Figure 12(b), compared to PebblesDB, L2SM saves up to 26.3% disk space. Compared to RocksDB, PebblesDB consumes 50.2% to 74.3% more disk space, while L2SM needs only 28.4% to 48.7% extra space.

G. Overhead Analysis

Storage overhead. For storage space, the log at each level of L2SM structure demands more disk space. As described in Section III-B, when determining the log size of each layer, we have a preset space threshold of SST-Log, which is less than 10% of the original LSM-tree. Thus, the extra storage space is less than 10%. We have run different workloads with the Random and Zipfian distributions and recorded the occupied storage of LevelDB and L2SM. Figure 10 shows the storage status of these implementations along the execution process. As expected, for both workloads, the storage requirement of L2SM is larger than LevelDB. For workload of the Scrambled Zipfian distribution, the storage space overhead of L2SM ranges from 4.3% to 9.2%. For the Random distribution, the storage space overhead ranges from 4.2% to 8.7%.

Memory overhead. The memory overhead of L2SM is mainly due to the in-memory bloom filters for SSTables and the HotMap. L2SM maintains in-memory bloom filters to improve read performance. Figure 11(a) provides the memory consumption of these three implementations under different workloads. As shown in the figure, the original LSM-tree (OriLevelDB) requires the least memory, and L2SM requires slightly more memory than LevelDB. Under different workloads, the memory overhead ranges from 3.2% to 11.3%, compared to LevelDB. The overhead mainly comes from the bloom filters for SSTables in the log and the in-memory HotMap maintained by L2SM.

V. Related Work

In recent years, many optimizations for LSM-tree based key value stores have been proposed [14]–[21].

Handling write amplification. PebblesDB [14] builds a key-value store using a fragmented log-structured merge tree to combine the design ideas from skip lists and LSM-tree. It relaxes the requirement of maintaining non-overlapping key ranges at each level and introduces guards to avoid rewriting data in the same level, which reduces compaction cost. However, the coarse-grained data structure of PebblesDB incurs severe storage space overhead as evaluated in the experiments. LWC-store [15] uses a partitioning method to vertically group entries and defines lightweight compaction by only merging metadata to decrease write amplification. LSM-trie [16] conducts an LSM-based and prefix-style hash index for managing massive small key-value pairs. It proposes a partitioned tiering method to reduce write amplification. dCompaction [17] defines virtual SSTable and virtual merge to delay the required compaction for lowering the overall compaction overhead. Unlike these schemes, L2SM tries to retain the tree structure and only uses a small SST-Log as an extension to store selected KV items.

Key & Value separation. WiscKey [22] reduces compaction IOs by separating the keys from the values and only manages the keys and metadata in the LSM-tree, through which the cost of write operations can be greatly decreased. Based on that, HashKV [23] further optimizes the value management using a hash-based structure to improve the read performance. These KV separation-based schemes could bring heavy garbage collection burden and the space efficiency may also be influenced by the value space management.

Hot & Cold separation. TRIAD [24] allows cold entries to enter the LSM-tree by holding hot entries in memory. However, the small Memtable size refrains the scope of hotness detection and its efficacy. Anti-caching [25] maintains an in-memory LRU chain to separate hot and cold records. Hot records are added to the tail of the chain and cold records are evicted to disks. Siberia [26] logs the access timestamps of all records and analyzes the log offline to predict the hot records. Then hot data is kept in memory and cold data is moved to disk. Funke et al. [27] proposes a hardware-assisted monitoring component, which uses the CPU's Memory-Management Unit to separate hot and cold tuples in HyPer. Cold tuples are further compressed and stored in virtual memory pages. These hot/cold data separation techniques work on individual data records and are used for the purpose of improving read/write performance. In contrast, L2SM uses a mechanism, called HotMap, to separate hot and cold SSTables and aims to minimize I/O amplification.

Performance optimization. Many prior optimization solutions [28]–[31] aim to directly enhance the performance of LSM-tree based data stores. Monkey [30] identifies the important tuning knobs and environmental parameters that determine the worst-case performance and further models the worst-case lookup and update costs. Its goal is to provide maximum throughput under uniformly random workloads, and for other workloads, it tries to achieve maximum lower-bound throughput. L2SM, in comparison, aims to solve the write amplification issues caused by the structural problem rooted in the LSM-tree itself. By delaying the deletion of invalid data components, LSbM-tree [32] improves the hit ratio of LSM-tree data in the OS page cache. ElasticBF [33] adopts a dynamic bloom filter adjustment policy to tune the false positive rate based on the hotness and access frequency of keys. VT-Tree [34] modifies the merge operation using a stitching

method to avoid page rewriting caused by independent segments whose range is not overlapped with other segments. NoveLSM [35] is a customized implementation of LSM-trees on NVM. Its write operations achieve a steady speed using NVM-based memory. LOCS [36] optimizes LSM-tree performance for SSDs through Open-Channel SSDs. SSTables in LOCS can be accessed in parallel by rearranging SSTables with different ranges into different channels. Unlike these prior works, L2SM aims to maintain the stability of the LSM-tree structure by identifying and isolating the KV items that disrupt the tree structure in a separate log structure. Many techniques mentioned above are orthogonal to our work and they can complement each other in optimizing LSM-tree based key-value stores.

VI. Conclusion

In this paper, we present a novel LSM-tree based structure, called L2SM, to address the I/O amplification problem. L2SM achieves its design goal by introducing a special SST-Log structure to isolate hot and sparse SSTables from the LSM-tree, a Pseudo Compaction and an Aggregated Compaction process to absorb high-cost updates in the log. We have built a prototype based on LevelDB. Our experimental results show that L2SM can achieve significant improvement.

Acknowledgements

We thank the anonymous reviewers for their constructive comments and feedback. The work described in this paper is partially supported by the grants from the National Science Foundation for Young Scientists of China (Grant No.61902218), and the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15273616, GRF 15206617, GRF 15224918).

References

[1] LevelDB. https://github.com/google/leveldb.
[2] RocksDB. https://github.com/facebook/rocksdb.
[3] Cassandra. https://github.com/apache/cassandra.
[4] HBase. https://github.com/apache/hbase.
[5] MongoDB. https://github.com/mongodb/mongo.
[6] C. Luo and M. J. Carey, "Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 531–543, 2019.
[7] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage Management in AsterixDB," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 841–852, 2014.
[8] L2SM Store. https://github.com/ericaloha/L2SM.
[9] A. Petrov, "Database Internals: A Deep Dive into How Distributed Data Systems Work," O'Reilly Media, Inc, 2019.
[10] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM (CACM)*, vol. 13, no. 7, pp. 422–426, 1970.
[11] MurmurHash. https://sites.google.com/site/murmurhash/.
[12] A. Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.
[13] YCSB. https://github.com/brianfrankcooper/YCSB.
[14] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.
[15] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores," in *IEEE Mass Storage Systems and Technologies (MSST)*, 2017.
[16] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items," in *USENIX Annual Technical Conference (ATC)*, 2015.
[17] F. Pan, Y. Yue, and J. Xiong, "dCompaction: Speeding up Compaction of the LSM-Tree via Delayed Compaction," *Journal of Computer Science and Technology (JCST)*, vol. 32, no. 1, pp. 41–54, 2017.
[18] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter," in *ACM Symposium on Cloud Computing (SoCC)*, 2018.
[19] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "An Efficient Memory-Mapped Key-Value Store for Flash Storage," in *ACM Symposium on Cloud Computing (SoCC)*, 2018.
[20] L. Wu, W. Lin, X. Xiao, and Y. Xu, "LSII: An Indexing Structure for Exact Real-time Search on Microblogs," in *IEEE International Conference on Data Engineering (ICDE)*, 2013.
[21] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "FloDB: Unlocking Memory in Persistent Key-Value Stores," in *ACM European Conference on Computer Systems (EuroSys)*, 2017.
[22] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating Keys from Values in SSD-conscious Storage," in *USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[23] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "HashKV: Enabling Efficient Updates in KV Storage via Hashing," in *USENIX Annual Technical Conference (ATC)*, 2018.
[24] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores," in *USENIX Annual Technical Conference (ATC)*, 2017.
[25] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-Caching: A New Approach to Database Management System Architecture," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1942–1953, 2013.
[26] J. J. Levandoski, P. Larson, and R. Stoica, "Identifying Hot and Cold Data in Main-Memory Databases," in *IEEE International Conference on Data Engineering (ICDE)*, 2013.
[27] F. Funke, A. Kemper, and T. Neumann, "Compacting Transactional Data in Hybrid OLTP & OLAP Databases," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1424–1435, 2012.
[28] F. Mei, Q. Cao, H. Jiang, and L. Tian, "LSM-Tree Managed Storage for Large-Scale Key-Value Store," in *ACM Symposium on Cloud Computing (SoCC)*, 2017.
[29] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs," in *USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[30] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal Navigable Key-Value Store," in *ACM Conference on Management of Data (SIGMOD)*, 2017.
[31] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data," *Proceedings of the VLDB Endowment*, vol. 10, no. 13, pp. 2037–2048, 2017.
[32] D. Teng, L. Guo, R. Lee, F. Chen, S. Ma, Y. Zhang, and X. Zhang, "LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.
[33] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores," in *USENIX Annual Technical Conference (ATC)*, 2019.
[34] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building Workload-Independent Storage With VT-trees," in *USENIX Conference on File and Storage Technologies (FAST)*, 2013.
[35] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redesigning LSMs for Nonvolatile Memory with NoveLSM," in *USENIX Annual Technical Conference (ATC)*, 2018.
[36] 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 *ACM European Conference on Computer Systems (EuroSys)*, 2014.