SFM: Mitigating Read/Write Amplification Problem of LSM-Tree-Based Key-Value Stores

HOYOUNG LEE1, MINHO LEE2, AND YOUNG IK EOM3
1Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea
2Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
3Department of Electrical and Computer Engineering, College of Computing and Informatics, Sungkyunkwan University, Suwon 16419, South Korea
Corresponding author: Young Ik Eom (yieom@skku.edu)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant through the Korea Government (MSIT) under Grant IITP-2015-0-00284 (SW Starlab Development of UX Platform Software for Supporting Concurrent Multi-Users on Large Displays), and in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT under Grant NRF-2015M3C4A7065696.

ABSTRACT Persistent key-value stores have been widely adopted as storage engines for modern IT infrastructures because they provide high performance with simple design principles. Moreover, many key-value stores commonly employ LSM-tree as their index structure due to its attractive features such as high write throughput and storage space efficiency. Unfortunately, LSM-tree has critical drawbacks in that it leads to write/read amplification problem. One of the prevalent solutions for remedying the write amplification problem is the tiering merge policy that reduces the number of rewrites by delaying merge operations. However, in spite of this advantage, the tiering merge policy may lead to a side-effect that induces high read amplification, increasing search/scan cost for upcoming read operations. In this paper, we concentrate on mitigating the high read amplification problem of the tiering merge policy, while maintaining its low write amplification. To achieve this, we propose a novel LSM-tree scheme, called Spatially Fragmented LSM-tree (SFM), which delays merge operations only for the non-read-intensive key-spaces. For this, SFM identifies the read intensity of each key-spaces by dynamically estimating their read/write hotness. We have implemented SFM based on PebblesDB and evaluated the performance benefits of our scheme under real-world workloads of Facebook. Experimental results clearly show that our scheme improves throughput by up to 1.67× compared with the conventional schemes while maintaining low write amplification, and also indicate that its latency is lowered by up to 41.41% on average by mitigating the read amplification problem of the existing schemes, by up to 43.68%.

INDEX TERMS Persistent key-value stores, LSM-tree, storage management, database management systems (DBMS), compaction, merge policy, read/write amplification problem.

I. INTRODUCTION

Recently, persistent key-value stores are widely adopted as a storage engine to efficiently handle large-scale data in modern IT infrastructures [1]–[11]. Key-value stores provide a simple design with several advantages such as high throughput, scalability, and flexibility. Meanwhile, many key-value stores including LevelDB and RocksDB generally employ Log-Structured Merge tree (LSM-tree) [12] as their index structure [3]–[6], [9], [13], [14]. With LSM-tree, key-value stores can achieve high performance because they buffer incoming write requests into main memory, and then subsequently flush them to the underlying persistent storage. Moreover, LSM-tree provides attractive features such as high storage space efficiency and simplicity of recovery by merging (i.e., compacting) the newly updated data with the previously stored data in an out-of-place update manner [14].

Internally, LSM-tree first absorbs incoming writes into the in-memory write buffer, called Memtable, which keeps key-value pairs in a sorted order. If the size of Memtable reaches a configured threshold, the buffered key-value pairs are flushed in the form of a file, called Sorted String Table (SSTable) file, into the underlying persistent storage. In the storage-side, these SSTable files go through a sequence of
levels that start from level 0. To efficiently manage the stored data, LSM-tree assigns key-value pairs written more recently to the lower level. If the capacity of a level is full, LSM-tree makes a free space by merging \textit{SSTable} files of the level with the ones of the next level and migrating the key-value pairs to the higher level. However, since merge operations continuously let the LSM-tree rewrite the existing \textit{SSTable} files, this merge policy, commonly referred to as the leveling merge policy, incurs high write amplification that limits the performance of key-value stores [14].

Previous researchers carried out diverse studies on LSM-tree to resolve performance degradation caused by high write amplification [15]–[20]. One of the state-of-the-art solutions for remedying the write amplification problem is the tiering merge policy which delays merge operations by stacking \textit{SSTable} files that have overlapped key-range in each level [13], [15], [16], [18], [19], [21]–[24]. The tiering merge policy reduces the number of rewrites for the \textit{SSTable} files by merging multiple \textit{SSTable} files at once. However, this approach may lead to a side-effect that read operations have to linearly search all the overlapped \textit{SSTable} files in each level until they find the corresponding key-value pair. By doing so, the tiering merge policy inevitably induces high read amplification that severely degrades the read performance of LSM-tree [14], [23].

In this paper, we propose a novel LSM-tree scheme, called Spatially Fragmented LSM-tree (SFM), which is for mitigating high read amplification of the tiering merge policy while maintaining low write amplification. To achieve this, SFM delays merge operations only for the key-spaces with low read intensity, not the entire key-space. In addition, our scheme selectively accumulates \textit{SSTable} files that belong to non-read-intensive key-spaces, by identifying the read/write hotness of key-ranges. We have implemented our scheme on top of PebblesDB [18] and evaluated its performance benefits, comparing it with the tiering based LSM-tree and leveling based LSM-tree. Experimental results clearly demonstrate that our scheme improves read/write throughput, by up to 1.67×, compared with the conventional schemes while maintaining low write amplification. Our scheme also improves its latency by up to 41.41% on average by mitigating the read amplification of existing schemes, by up to 43.68%. We verified these performance impacts of our scheme based on the read/write access pattern traced from the real-world application, called ZippyDB [11], [25], of Facebook.

The rest of this paper is organized as follows. In Section II, we describe the concept of LSM-tree and compares its representative merge policies: leveling merge policy and tiering merge policy. Then, in Section III, we present our motivation not only by analyzing the performance characteristics of the conventional schemes but also by revealing the read/write access pattern of real-world workloads. Based on these observations, we provide the overall design principles of our scheme in Section IV. In Section V, we evaluate the benefits of our scheme by comparing its performance with that of the conventional schemes. Moreover, we review the related works in Section VI, and discuss the limitations of our scheme and future work in Section VII. Finally, we conclude this paper in Section VIII.

II. BACKGROUND

A. OVERVIEW OF LSM-TREE

LSM-tree is one of the data structures that is commonly used for key-value stores. In the LSM-tree, a key is employed for identifying the key-value pair (i.e., \{key, value\}) [3]–[6], [13]. In the case of value, it contains the corresponding data object. By using the LSM-tree, clients can manage their own data as key-value pairs with the key-value interfaces such as \texttt{Get(key)}, \texttt{Put(key, value)}, and \texttt{Scan(key1, key2)} [5], [6]. Whenever write requests occur in the LSM-tree-based key-value stores, the incoming write requests are logged into the Commit log files that reside on the underlying persistent storage so that the key-value stores can ensure data consistency from system crashes or sudden power failures [13], [26]. Then, these write requests are temporarily absorbed into the in-memory write buffer called \texttt{Memtable} [27]. When the capacity of \texttt{Memtable} becomes full, the \texttt{Memtable} is converted to a read-only structure (i.e., \texttt{immutable Memtable}), and the key-value pairs contained in the \texttt{immutable Memtable} are flushed onto the underlying persistent storage. On the storage side, these flushed key-value pairs are managed differently, depending on the merge policy of the LSM-tree: leveling merge policy and tiering merge policy [14].

B. LEVELING MERGE POLICY

In the LSM-tree based on leveling merge policy, the key-value pairs residing on the persistent storage are classified into a sequence of levels starting from level 0 (i.e., \(L_0\) to \(L_{\text{max}}\)), in which the recently written key-value pairs are indexed to the lower level [12], [14]. Each level consists of multiple \textit{SSTable} files that store key-value pairs in a sorted fashion [14]. In a level, each \textit{SSTable} file has its own key-range that does not overlap with the others. If the capacity of the level becomes full, some of the \textit{SSTable} files in the level are migrated into the higher level with a merge operation. Whenever the merge operation is triggered, it not only selects the victim \textit{SSTable} files but also merges them with the \textit{SSTable} files on the next level.

Figure 1 depicts the merge operation of leveling merge policy in the LSM-tree. Under the leveling merge policy, the victim \textit{SSTable} file (i.e., \(50-70\)) is first selected from the level \(L_k\). Then, it is merged with the overlapped ones (i.e., \(52-60, 61-69\)) on the next level \((L_{k+1})\). During the merge operation, invalid key-value pairs of these \textit{SSTable} files are discarded, and the rest of key-value pairs are integrated into the key-value sequence in sorted order. This key-value sequence is then split up based on the amount of key-value pairs, and new \textit{SSTable} files (i.e., \(50-60, 61-70\)) are created. Finally, these new \textit{SSTable} files are assigned to \(L_{k+1}\).
In the meantime, the leveling merge policy induces a high write amplification problem because it rewrites not only victim SSTable files but also overlapped SSTable files on the next level. Since the key-value store based on LSM-tree is continually filled up by key-value pairs over its lifetime, the capacity of lower levels becomes full most of the time. Therefore, the merge operation cascades from $L_0$ to $L_{\text{max}}$, and SSTable files of each level are constantly rewritten. As a result, these rewrites for SSTable files easily saturates I/O bandwidth provided by the persistent storage, and thus incoming read/write operations from clients are frequently blocked [28]–[30]. Moreover, these rewrites increase the maintenance cost of the data centers by shortening the lifespan of the underlying persistent storage such as SSDs [17], [18], [20].

C. TIERING MERGE POLICY

In contrast with the leveling merge policy, tiering merge policy allows key-ranges of SSTable files to be overlapped in each level [14], [18]. Under the tiering merge policy, each level of LSM-tree is first partitioned into disjoint key-spaces and multiple SSTable files are accumulated in the stack until the number of SSTable files reaches the configured threshold $T$. Once the number of the SSTable files in the stack exceeds the threshold, the SSTable files of the stack have to be migrated into the next higher level with the merge operation. For this, the merge operation based on the tiering merge policy compacts all SSTable files in the victim stack and reorganizes them to new SSTable files, and then attaches these new SSTable files into the stacks of the next level.

Figure 2 illustrates the merge operation of the tiering merge policy in the LSM-tree. In this example, we assume that the threshold of the stack is $2$ ($T = 2$). Under the tiering merge policy, the merge operation first reads all SSTable files (i.e., $54-63, 57-70, 50-65$) of the victim stack on the level $L_k$ into main memory, and removes invalid key-value pairs. Then, it reorganizes the rest of the key-value pairs to new SSTable files (i.e., $50-60, 61-70$) based on the key-ranges of the stacks in the next level ($L_{k+1}$). These new SSTable files are finally added into the stacks of $L_{k+1}$ according to their key-ranges. In this way, the merge operations under tiering merge policy migrate SSTable files into the next level without rewriting the SSTable files of the next level. Moreover, they are not triggered immediately even if the key-ranges of the SSTable files are overlapped in the level. For these reasons, the merge frequency decreases and multiple SSTable files are merged at once. Based on these design principles, the tiering merge policy achieves a lower write amplification than the leveling ones by up to $2.4$–$3.0$ [18].

Unfortunately, the tiering merge policy has a trade-off problem that increases read amplification while reducing write amplification [14], [23]. This is because read operations under the tiering merge policy have to linearly search all SSTable files accumulated in the stack. Assume the threshold of the stack is configured to $4$ ($T = 4$). Then, SSTable files with overlapped key-ranges may be accumulated up to 4 tables on each level. Hence, the read operation should look up the SSTable files up to 4 times per level in order to find the target key-value pair. If the target key-value pair exists in the $L_{\text{max}}$, the read operation should repeat these inefficient look-up operations from $L_0$ to $L_{\text{max}}$. In this case, the read amplification of the tiering merge policy may increase up to $4 \times L_{\text{max}}$ times. Furthermore, the read amplification problem becomes worse as the threshold of the stack becomes higher [14], [23]. Consequently, this growth of the read amplification increases the search/scan cost of the LSM-tree and degrades the read performance of the key-value stores.

III. MOTIVATION

In this section, we study the behavior of conventional merge policies including leveling merge policy and tiering merge policy. To achieve this, we first show the performance characteristics of the conventional merge policies under the real-world workload. Then, we analyze the characteristics of the conventional merge policies based on the access patterns of the real-world workloads. We finally discuss the inefficiency of the conventional merge policies.

A. THE PERFORMANCE OF THE CONVENTIONAL MERGE POLICIES

To understand the characteristics of the conventional merge policies, including leveling merge and tiering merge policies,
we ran the ZippyDB workload [11], [25], [31] traced from a distributed key-value store of Facebook on LevelDB and PebblesDB (See Section V-A for more details of the experimental setup). In our experiments, we evaluated the performance of the leveling merge and tiering merge policies by varying the threshold $T$ from 2 to 8. Figure 3 exhibits that the tiering merge policy has lower throughput by up to 55.45% than the leveling one. This is because the tiering merge policy incurs high read amplification. In the case of tiering merge policy, incoming read operations have to linearly search SSTable files in that the key-ranges of SSTable files may be overlapped in the stack. Thus, the tiering merge policy has a higher search/scan cost than that of leveling merge policy. As shown in Figure 3, LSM-tree exhibits lower read throughput under the tiering merge policy as the threshold $T$ becomes higher.

On the other hand, Table 1 shows that the tiering merge policy has lower Write Amplification Factor (WAF) by up to 63.35% than the leveling merge policy. The leveling merge policy always merges SSTable files that have an overlapped key-range whenever a merge operation is triggered, whereas the tiering merge policy accumulates the overlapped SSTable files in the level and merges multiple SSTable files in a batch. Accordingly, the tiering merge policy reduces the number of rewrites of SSTable files by delaying the merge operations until the number of SSTable files in the stack reaches the configured threshold. Consequently, the write amplification of LSM-tree decreases with the lower frequency of merge operations under the tiering merge policy. As the threshold $T$ increases, the tiering merge policy further reduces write amplification in that more SSTable files can be merged at once.

### B. SPATIAL LOCALITY: READ/WRITE ACCESS PATTERN OBSERVED IN REAL-WORLD WORKLOADS

To analyze the performance of the LSM-tree more specifically, we also profiled the access pattern of ZippyDB workload with the tracing/analyzing tools of RocksDB [11], [31]. Figure 4 shows the key-value pairs accessed by read/write operations while running the workload. In Figure 4, Figure 4a shows the access pattern of the read operations, and Figure 4b shows the access pattern of the write operations. As shown in both Figure 4a and Figure 4b, ZippyDB workload has a high spatial locality, as observed in many applications using key-value stores [11]. By these spatial locality characteristics, the skewness of the workload is established based on the key-ranges. Moreover, the read/write hotness of some key-spaces may be clearly defined to read-hot key-spaces or write-cold key-spaces. Unfortunately, delaying merge operations on the write-cold key-spaces is not helpful in reducing write amplification because few write operations occur in the range. Furthermore, SSTable files stacked in read-heavy key-spaces induce high read amplification by increasing the number of SSTable files to be read. In a nutshell, applying the tiering merge policy to the whole key-spaces may significantly deteriorate the read amplification problem of the LSM-tree.

### IV. DESIGN

Our observations ask the following question: **How to resolve the read amplification problem induced by the tiering merge policy while maintaining its low write amplification?** One simple solution is to selectively employ the tiering merge policy and the leveling merge policy according to the read/write access pattern on the key-spaces. To achieve this, we introduce a novel LSM-tree design, called Spatially Fragmented LSM-tree (SFM), which adopts the tiering merge policy only for the key-spaces with low read intensity. In the SFM, its merge operation tracks read/write operations for the partitioned key-spaces in each level and identify the read intensity of each key-space. If a key-space is considered to be non-read-intensive, the merge operation in SFM accumulates SSTable files in the stack up to the configured threshold. Otherwise, it limits the number of SSTable files in the stack to 1.
In this section, we first provide the overall design principle of SFM in Section IV-A. Then, in Section IV-B, we describe how our scheme estimates the read/write hotness of key-spaces and identifies read intensity for them. Moreover, we present the metadata management policy of SFM and its hotness inheritance policy in Section IV-C, and Section IV-D, respectively. In Section IV-E, we finally explain how SFM initializes the hotness information of key-spaces.

A. SPATIALLY FRAGMENTED LSM-TREE (SFM)

In SFM, each level is first partitioned into disjoint key-spaces by random key boundaries, similarly to Fragmented LSM-tree (FLSM) [18] which is one of the representative LSM-tree adopting tiering merge policy. At this step, the key-range of each key-space is probabilistically determined based on the distribution of the inserted keys, with the same principle as that of FLSM. On these fragmented key-spaces, our scheme classifies the read intensity of each key-space at the granularity of the stack in that the read/write amplification increases or decreases according to the number of SSTable files inside the stack. By doing so, our scheme can identify the read intensity of each key-space by tracking the read/write hotness for its stack (See Section IV-B for more details of our read intensity identification policy).

With the partitioned key-spaces, the read/write operations must access the stack before they access the target SSTable file. Accordingly, the read intensity of each key-space can be distinguished by the read/write hotness for each stack. Moreover, since the number of key boundaries increases as the capacity of the level is expanded, the key-ranges of the stacks can be divided more fine-grained in the higher level in SFM. Thus, our scheme can accurately track the read/write hotness of the key-spaces even if many levels are configured in the LSM-tree. Using the identified read intensity of each stack, our scheme adaptively controls the capacity for each individual stack. Specifically, if the key-range of the stack is considered to be read-intensive, the number of SSTable files in that stack is limited to 1. Otherwise, the number of SSTable files in the stack is allowed to grow up to the configured threshold.

If the number of SSTable files in the stack exceeds its threshold, SFM triggers a merge operation to migrate SSTable files of the stack into the next level. Figure 5 depicts an example of the merge operation of SFM. During the merge operation, our scheme first reads all SSTable files from the victim stack (i.e., 51-70 and 41-68) into the memory, and merges them according to the key-value sequence (i.e., 41-70), by sorting their key-value pairs. This key-value sequence is then sliced into new SSTable files (i.e., 41-55 and 56-70) by the key boundaries of the next level. Finally, the new SSTable files are flushed onto the storage, and added to the stacks in the next level. Likewise, the new SSTable files in the next level can be accumulated or not, according to the capacity of each stack. In this way, SFM can reduce read amplification of the tiering merge policy by limiting the number of SSTable files to 1 for the read-intensive key-spaces. These stacks decreases the search/scan cost of the tiering merge policy. At the same time, by stacking multiple SSTable files to non-read-intensive key-spaces, our scheme can achieve low write amplification as in the conventional tiering merge policy.

B. READ INTENSITY IDENTIFICATION

To identify the read intensity of the key-spaces, SFM counts the number of the reads/writes for each stack. In our scheme, since read operations always linearly read all SSTable files in

![FIGURE 5. Overview of SFM merge policy.](image-url)
LSM-trees [5], [6], [14], [18].

stacks and SSTable information, the read/write operations access the desired sstableAddr addresses of them (i.e., files included in the stack (i.e., SSTable Metadata management in our scheme. As shown in Figure 6, each stack. Figure 6 illustrates overview of the metadata management.

At the last step of the merge operation, the read intensity of the key-spaces is finally identified by estimating their read/write hotness. In order to estimate read/write hotness of the key-spaces, our scheme compares the read/write access count of each stack with the average read/write access count to capture the dynamically varying access patterns of the workloads. If the access count of the stack is greater than the average access count of all the stacks in the level, the stack is considered to be a hot one, and otherwise, it is considered to be a cold one. By estimating the hotness of each stack, our scheme finally classifies the key-spaces into the following two categories: the read-intensive key-spaces and the non-read-intensive ones. In these two categories, the read-hot and write-cold key-spaces are defined as read-intensive key-spaces, and the other key-spaces are defined as non-read-intensive ones.

C. METADATA MANAGEMENT

SFM manages the whole stacks of each level by employing a data structure, called StackMetadata, which describes each stack. Figure 6 illustrates overview of the metadata management in our scheme. As shown in Figure 6, StackMetadata stores information on the stack, including the key-range of the stack (i.e., keyRange), the number of SSTable files included in the stack (i.e., sstableNum), and addresses of them (i.e., sstableAddr). By referring to these information, the read/write operations access the desired stacks and SSTable files in each level similarly to other LSM-trees [5], [6], [14], [18].

In our scheme, StackMetadata additionally store not only hotness information of the stack (i.e., readCount, writeCount), but also currentThreshold that indicates the threshold for each stack. By doing so, the number of the read/write accesses for the stack is counted without additional metadata accesses or updates. Moreover, SFM uses these StackMetadata in order to determine the stacks to be merged. As described in Section IV-A, currentThreshold of the newly created stack is set to the configured threshold $T$. Based on these design principle, our scheme insert StackMetadata whose sstableNum exceeds its threshold currentThreshold into the next merge queue. Whenever merge operation is triggered, threshold currentThreshold of each stack is defined to 1 or $T$ according to the identified read intensity. Likewise, these StackMetadata objects are inserted into the next merge queue once their sstableNum exceeds the estimated threshold currentThreshold. In this way, SFM manages the stacks of the entire level with StackMetadata.

D. HOTNESS INHERITANCE

Whenever new stacks are created, initializing the read/write access counts of the StackMetadata may increase the read/write amplification. This is because it is difficult to properly identify the read/write hotness of the newly created stacks until read/write operations are sufficiently collected in their key-ranges. If the threshold of read-intensive stacks is configured to $T$, multiple SSTable files may be accumulated in the newly created stacks, and read operations have to search all SSTable files in the stacks. On the other hand, once the threshold of non-read-intensive stacks is set to 1, the merge operations may be frequently triggered due to the limited capacity of the stacks.

In order to address this problem, we designed a hotness inheritance technique. We implemented this inheritance technique based on the design of SFM, in which the key-range of each stack is split into fine-grained key-ranges in the next higher level. With this design principle, the hotness information (i.e., readCount, writeCount) of StackMetadata are inherited from a level to the next level during the merge operation. Figure 7 illustrates our hotness inheritance technique. Whenever new stacks are created in the next level ($L_{k+1}$), they receive hotness information related to their key-ranges from the parent stacks on the level $L_k$. By doing so, new stacks start with the previously collected hotness information even if they are newly created. After the merge operation, the read/write hotness of these new stacks are estimated independently from the parent stacks. As a result, this technique improves the accuracy of our hotness estimation, and thus the read/write amplification of SFM remains settled.

E. INITIALIZATION OF HOTNESS INFORMATION

Since old access counts may degrade the accuracy of the hotness estimation, SFM has to re-initialize read/write access counts of the expired stacks. Similarly to the initialization policy of ElasticBF [32], our scheme logs the total number of read/write operations into the global variable currentCount.
and into the local variable stackCount of the StackMetadata, whenever the read/write operations occur. As the stackCount of each stack is updated only when its key-range includes the key-value pair of the read/write operation, our scheme can estimate the logical time elapsed since the last access to the stack by calculating elapsedCount (currentCount - stackCount) of each stack. Whenever the merge operation occurs, our scheme initializes the access counts of all the expired stacks whose elapsedCount exceeds lifeCount, which is a configurable parameter.

The variable lifeCount has the limit value for elapsedCount, and the access counts of the expired stacks are handled as the old access counts in SFM. Therefore, it is important to appropriately set lifeCount because the accuracy of hotness estimation can be aggravated depending on the value of lifeCount. When the lifeCount is set too large, old and latest access counts may be reflected together in hotness estimation and the read intensity of key-spaces may not be identified properly. On the contrary, the estimated hotnesses can be changed frequently with too small value of lifeCount. When the read/write intensities of key-spaces may not be evaluated accurately in this case also. In order to mitigate these problems, we empirically set lifeCount to 50K, which has been demonstrated to be most adequate in our experiments.

V. EVALUATION
A. EXPERIMENTAL SETUP
1) SYSTEM ENVIRONMENT
To verify the performance impacts of our scheme, we have implemented SFM on top of PebblesDB [18] and performed experiments on the system equipped with Intel Core i7-6700 CPU, and 8GB DRAM. The Ubuntu 16.04 LTS with Linux 4.15 kernel is used on our system. Especially, the entire data used for our experiments have been handled on the RAID0 array constructed with two Intel NVMe SSD 750 Series 400GB (total 800GB) devices where the RAID0 array has been formatted to the Ext4 [33] file system with writeback mode.

2) BENCHMARKING TOOLS
For experiments, we compared our scheme with LevelDB [5] as a leveling-based key-value store (LSM scheme) and PebblesDB as a tiering-based key-value store (FLSM scheme). To generate access patterns of real-world applications, we used the state-of-the-art benchmark, called MixGraph [11], [31], which is imported from the db_bench [34] benchmark of RocksDB. We migrated the MixGraph benchmark and the related libraries onto LevelDB, PebblesDB, and SFM. For the migrated MixGraph benchmarks, we excluded some parameters adjusting I/O intervals (i.g., sine_a, sine_b) [31] in order to accurately evaluate the performance of our scheme. We also eliminated some parameters diversifying the size of value objects (i.g., value_k, value_sigma) [31] because they may induce unpredictable side-effects. Instead, we fixed the size of all value objects to 1KB.

3) WORKLOADS
Three workloads are used for our experiment, including read-intensive, write-intensive, and read-write-mixed ones, where ZippyDB is used for the three types of workloads. The read-intensive workload is constructed by configuring read/write ratios of ZippyDB workload to 85% reads, 14% writes, and 1% scans. Note that ZippyDB workload exhibits high skewness in that the hottest 1% keys take up about 50% of total accesses [11]. To evaluate the performance of our scheme in terms of write performance, we set the write-intensive workload by synthetically configuring read/write ratios of ZippyDB workload to 14% reads, 85% writes, and 1% scans. Moreover, we additionally set the read-write-mixed workload in order to evaluate our scheme from various performance perspectives. The read-write-mixed workload is constructed with 50% reads, 49% writes, and 1% scans. For all experiments, we first randomly filled the 128GB dataset to the underlying persistent storage before running each workload. On the pre-filled dataset, 8 threads read/write/scan 64GB key-value pairs (8× larger than the capacity of the main memory) per each workload, in which each key-value pair consists of 48 bytes key and 1KB value.

4) MEASUREMENTS
For the three schemes, LSM, FLSM, and SFM, we measured not only the throughput and average latency but also the read/write amplification, varying the threshold $T$ from 2 to 8. In the case of throughput and average latency, they are measured by db_bench benchmark. On the other hand, we evaluated read/write amplification of the three key-value stores by using iostat [35] monitoring tool. Furthermore, we additionally compared space amplification of the three key-value stores by monitoring the available capacity of the underlying persistent storage.

5) KEY-VALUE STORE CONFIGURATION
We concentrated not only on avoiding various side-effects caused by the default (untuned) configuration but also on
equally setting the configurations of the three key-value stores. For all experiments, the capacities of the write buffers for the three key-value stores are set to 64MB. In the meantime, the capacities of caches for the reads are set to 256MB. To confirm the impact of our scheme, we disabled both compression and Commit log in all key-value stores. Moreover, all SSTable files are allocated with Bloom filters [36] with 10 bits_per_key. In the case of lifeCount variable for SFM, it was set to 50K, as described in Section IV-E. The other configurations of all key-value stores are set to the default values in all experiments.

B. EXPERIMENTAL RESULTS

1) PERFORMANCE UNDER READ-INTENSIVE WORKLOAD

In this section, we compare the throughput and latency of the three schemes, LSM, FLSM, and SFM along with the read/write amplification induced by them. Figure 8 exhibits the performance of the three schemes under the read-intensive workload as described in Section V-A3. In Figure 8, we have normalized the throughput and latency of the three schemes by using LSM as the baseline.

Figure 8a shows the throughput of the three schemes. As shown in Figure 8a, the throughput of SFM is larger, by up to 1.25–1.49\times, than that of FLSM. It is because SFM does not accumulate SSTable files for read-intensive key-spaces in contrast with the FLSM. Under the read-intensive workload, as the threshold $T$ increases, the throughput of SFM decreases in that the number of accumulated SSTable files increases. Meanwhile, the throughput of SFM is lower, by up to 8.19–29.25%, than that of LSM. This is because the tiering merge policy intrinsically induces the read amplification. In the case of SFM, once read operations are generated onto the write-hot key-spaces, they must search the stacked SSTable files in each level. Unfortunately, these key-spaces are widespread under the read-intensive workload because large amount of read operations accesses the overall key-spaces. As a result, SFM cannot exceed the throughput of LSM under the read-intensive workload. However, SFM outperforms LSM in terms of write amplification. As shown in Figure 8b, SFM maintains low write amplification similarly to FLSM because it also delays the merge operations by stacking the SSTable files. In the case of LSM, it has a high write amplification compared with SFM in that it unconditionally triggers merge operations whenever the SSTable files are migrated from the upper-level and the overlapped SSTable files exist.

Meanwhile, the average latency of the three schemes is shown in Figure 8c. As presented in Figure 8c, the average latency of SFM is smaller, by up to 17.78–26.65%, than that of FLSM. This is because the read intensities of the key-spaces are accurately identified under our hotness estimation. With our hotness estimation, leveling merge policy is applied to read-intensive key-spaces, whereas tiering merge policy is applied to non-read-intensive key-spaces. Based on this design principle, the read-intensive key-spaces are allowed to assign one overlapped SSTable file on each level, and thus read operations can find target key-ranges by searching only one SSTable file in each level for each read operation. In contrast with SFM, FLSM accumulates SSTable files regardless of their read/write hotness, and this naive merge policy may cause high read amplification. However, the average latency of SFM increases with a higher threshold $T$,
similarly to FLSM. Even if SFM properly evaluates the read intensities of key-spaces, the read amplification of the tiering merge policy inevitably increases with the number of the accumulated \textit{SSTable} files. Figure 8d shows the read amplification of the three schemes. In Figure 8d, the read amplification of both FLSM and LSM increases as the number of the stacked \textit{SSTable} files increases up to \( T \). Nonetheless, SFM mitigates the read amplification of FLSM, by up to 29.35–32.60%, due to its efficient hybrid merge policy. These experimental results demonstrate that SFM remedies the read amplification problem of FLSM with its hotness estimation.

2) PERFORMANCE UNDER WRITE-INTENSIVE WORKLOAD

Figure 9 exhibits the performance of the three schemes under the write-intensive workload. In the case of Figure 9a, it represents the normalized throughput of the three schemes. With the write-intensive workload, the throughput of SFM is higher, by up to 1.07–1.92 \( \times \), than that of LSM, as shown in Figure 9a. This is because SFM reduces the number of rewrites for \textit{SSTable} files by delaying the merge operations and accumulating the \textit{SSTable} files only for the non-read-intensive key-spaces. Furthermore, the throughput of SFM is also higher, by up to 1.14–1.52 \( \times \), than that of FLSM. Although FLSM presents low merge frequency, it still has significant search/scan overhead caused by the tiering merge policy. Due to such search/scan overhead, the throughput of FLSM degrades under the write-intensive workload. In the meantime, SFM exhibits high throughput by efficiently mitigating the read amplification problem compared with FLSM. Figure 9b shows that SFM has lower write amplification, by up to 46.77–61.52%, than LSM. These experimental results clearly demonstrate that SFM improves read throughput with the negligible overhead of write amplification.

Figure 9c plots the average latency of the three schemes. Under write-intensive workload, the average latency of SFM is the lowest among the three schemes. Specifically, the average latency of SFM is lower than that of LSM, by up to 36.01–50.45%. With the leveling merge policy that assigns only one overlapped \textit{SSTable} file on each level, the merge operations of LSM constantly cascades from the lowest level (\( L_0 \)) to the highest level (\( L_{\text{max}} \)). For that reason, the merge frequency of LSM significantly increases as the amount of key-value pairs in the key-value store grows. Hence, incoming read/write requests are frequently blocked because bulk merge operations saturate I/O bandwidth provided by the underlying persistent storage. To alleviate this problem, SFM not only identifies the read/write hotness of each key-spaces, but also lazily performs merge operations for the non-read-intensive key-spaces. Besides, SFM accurately traces the read/write hotness of each key-space with its hotness inheritance technique. Consequently, the read/write operations under SFM can better exploit I/O bandwidth provided by the underlying persistent storage. In the case of FLSM, its average latency is higher, by up to 1.20–1.71 \( \times \), compared to SFM. This is because the read amplification considerably increases on some read-intensive key-spaces, even if stacking \textit{SSTable} files reduces write amplification. Especially, this read amplification problem is more aggravated when \( T = 8 \). Figure 9d presents the read amplification of the three schemes. As presented in Figure 9d, SFM alleviates the read amplification of FLSM, by up to 24.61–43.68%. From these experimental results, we can
verify that SFM accurately identifies the read intensity of key-spaces.

3) PERFORMANCE UNDER READ-WRITE-MIXED WORKLOAD
Figure 10 shows the performance of the three schemes with the read-write-mixed workload. Under read-write-mixed workload, the throughput of LSM is higher than that of FLSM, by up to 1.30–1.67×, as shown in Figure 10a. It is because the overall search/scan cost of FLSM deteriorates by the stacked SSTable files. By comparison, the throughput of SFM is higher, by up to 1.13–1.22×, than that of FLSM. Since the read intensity of the key-spaces are dynamically identified by SFM, read operations can find target key-value pairs by searching only one SSTable file per each level. At the same time, merge operations can be delayed only under the non-read-intensive key-spaces, and SFM alleviates the read amplification problem of FLSM. Nevertheless, SFM does gain the performance advantages when $T = 2$. This is because a similar amount of the read/write operations access onto the majority of key-spaces under the read-write-mixed workload. On these key-spaces, SFM does not improve the read throughput since it also has to ensure write amplification as low as that of FLSM. With Figure 10b, we confirmed that SFM improves the write amplification problem compared with LSM: the write amplification of LSM is higher than that of SFM, by up to 1.56–2.06×. Moreover, the write amplification of SFM shows negligible difference (less than 5%) compared to that of FLSM.

Figure 10c exhibits the average latency of the three schemes under the read-write-mixed workload. Even though a similar amount of the read/write operations access several key-spaces, SFM shows lower average latency, by up to 6.19–17.45%, compared with that of FLSM. It is because SFM not only dynamically estimates the read/write hotness of key-spaces but also periodically initializes their old access counts. Under SFM, the read/write counts of several key-spaces whose elapsedCount exceeds lifeCount may be classified as old access counts. Moreover, these old access counts may be initialized. Due to this hotness re-initialization, SFM can precisely identify the current hotness of key-spaces, although read/write operations indiscriminately access several key-spaces. Despite these efforts, the average latency of SFM is still higher than that of LSM, by up to 1.20–1.74×. This is because SFM also focuses on achieving low write amplification similarly to that of FLSM. For this, SFM applies the tiering merge policy to several key-spaces whose read/write hotness is measured similarly. In the meantime, Figure 10d shows the read amplification of the three schemes. From Figure 10d, we can confirm that SFM mitigates read amplification of FLSM, by up to 11.46–18.93%. In addition, SFM decreases read amplification more and more with higher value of $T$. As described above, it is because SFM is designed to dynamically trace the read/write hotness of key-spaces. These experimental results further prove that SFM mitigates not only the write amplification problem of LSM but also the read amplification problem of FLSM, even on the read-write-mixed workload.

4) SPACE AMPLIFICATION
To confirm the performance impacts of our scheme in more detail, we also compare space amplification of the three schemes, LSM, FLSM, and SFM. For this, we measured
The available capacity of the underlying persistent storage, varying the threshold $T$ of FLSM and LSM from 2 to 8. Note that the threshold of LSM is always fixed to 1 because it is designed based on the leveling merge policy. Moreover, the write-intensive workload was employed in our experiments for space amplification. Space amplification of the three schemes is represented in Figure 11. As shown in Figure 11, space amplification of LSM is lowest among the three schemes. It is because LSM limits the number of the overlapped SSTable files to 1 in each level by adopting the leveling merge policy. Meanwhile, the space amplification of SFM is lower, by up to 11.83–37.16%, than that of FLSM when $T = 2$. Since SFM partially applies the leveling merge policy only for read-intensive key-spaces, the number of SSTable files in several key-spaces is limited to 1. Based on this hybrid merge policy, SFM efficiently alleviates the space amplification problem of FLSM. For the same reason, this gap between SFM and FLSM widens, by up to 15.36–40.13%, when $T = 4$. In the case of FLSM, the storage space can be required by up to the configured threshold $T$. However, as shown in the experimental results, space amplification of FLSM does not reach the configured threshold $T$. This is because ZippyDB workload has spatial locality. Due to this characteristic of workload, FLSM only utilizes the storage space corresponding to write-hot key-spaces in that the majority of write operations are dominated only for some key-spaces. Consequently, space amplification of FLSM with $T = 8$ becomes higher, by up to 1.18–1.39×, than that of SFM. These experimental results clearly show that SFM has advantages not only of the read/write amplification but also of the space amplification.

VI. RELATED WORK

A. LSM-TREES BASED ON TIERING MERGE POLICY

In both industry and academia, many researchers and developers carried out diverse studies on tiering merge policy to alleviate the write amplification problem of LSM-tree. In the case of industry, non-relational databases such as RocksDB [21] at Facebook, Cassandra [13], and AsterixDB [22] at Apache have been supporting tiering merge policy. In academia, some studies, including Write Buffer (WB) tree [15], LWC-tree [16], dCompaction [19], PebblesDB [18], and Partial Tiering [24] have introduced tiering merge policy for LSM-tree. Representatively, PebblesDB is built on top of LSM-tree based on tiering merge policy, called the fragmented log-structured merge tree (FLSM) [18]. In FLSM, the whole key-space of each level is partitioned into fine-grained key-ranges. These key-ranges are described as a data structure like a Skip-list, called Guard. FLSM delays the merge operations by attaching the overlapped SSTable files to these Guards. As a result, PebblesDB remedies write amplification problem of LSM-tree by reducing the merge frequency. However, this approach leads to a side-effect that increases read amplification of LSM-tree. In order to mitigate these trade-off problems between leveling merge policy and tiering merge policy, SFM adopts a hybrid merge policy that selectively applies leveling merge policy or tiering merge policy according to the read intensity of each key-space.

B. OPTIMIZATION FOR SPECIAL WORKLOADS ON LSM-TREES

Various studies, including LHAM [37], LSM-trie [38], SlimDB [39], TRIAD [20], Mathieu et al. [40], Cao et al. [11], EvenDB [41], and SplitKV [42], have analyzed real-world workloads and optimized the LSM-tree based on the characteristics of real-world workloads. Especially, TRIAD [20] discovers that the hot key-value pairs on the persistent storage are more frequently rewritten under skewed workloads. To decrease these rewrites, it first classifies the key-value pairs on the write buffer (i.e., Memtable) to hot or cold ones, and then flushes only cold key-value pairs to the underlying persistent storage. By doing so, hot key-value pairs are updated only in the main memory without writing to the persistent storage, reducing unnecessary rewrites. As a result, this approach decreases the write amplification of the LSM-tree by reducing both the amount of key-value pairs to be merged and the frequency of the merge operations. Despite these advantages, LSM-trees adopting this approach may be high read amplification because it is designed only for mitigating write amplification problems. Moreover, this approach is not optimized from the perspective of spatial locality since it only considers the skewness among key-value pairs.

Meanwhile, spatial locality among the key-spaces is discovered by Cao et al. [11]. They traced and analyzed the workloads from the key-value stores of Facebook, including UDB (key-value stores to manage social graph data), ZippyDB (a distributed key-value store), and UP2X (a distributed key-value store for AI/ML systems) [11]. Consequently, they revealed the detailed characteristics of the real-world workloads, such as spatial locality, temporal locality, and size variation of keys and values. With these analyses, they argued that conventional LSM-tree-based key-value
stores should be reconsidered for spatial locality because the existing benchmarks including YCSB [10] have determined the access popularity on the basis of key units, not on the basis of the key-range units [11]. To reflect the characteristics closer to the real-world workloads, they proposed a novel benchmark called MixGraph [31], which is normalized based on the detailed characteristics of the real-world workloads. Moreover, they provide an opportunity to customize the key-value stores to real-world workloads by providing some tools not only to trace/analyze the workloads but also to replay the traced workloads. These tools are embedded in db_bench benchmark of RocksDB [31].

C. AUTO-TUNING FOR LSM-TREES

Some research, including Monkey [28], Dostoevsky [23], Thonangi and Yang [43], ElasticBF [32], and Mutant [30], have focused on developing the auto-tuning technique for LSM-tree. In the case of Monkey [28], it adjusts the number of Bloom filters for SSTable files differently on each level in order to resolve the trade-off problem between the read/write performance and the main memory utilization. Specifically, it reduces false-positive rates of LSM-tree by allocating more Bloom filters to SSTable files of the lower level. With this approach, the look-up cost of the LSM-tree can be decreased without additional memory allocation. Nevertheless, it does not consider the hotness among the key-spaces inside the same level. SFM further distinguishes the hotness of each key-spaces in each level and selectively adjusts to merge policies according to the hotness of each key-space. In this sense, Monkey can be orthogonally applied to our scheme.

Similarly to Monkey, Dostoevsky [23] extends the principle that most look-up/search operations occur in the highest level ($L_{max}$). In order to alleviate the trade-off problem between reads and writes, it introduces a novel merge policy, called a lazy leveling policy, that selectively applies the merging level policy or the tiering merge policy to each level. Under the lazy leveling policy, write amplification of LSM-tree can be reduced because it adopts the tiering merge policy only to the lower levels excluding $L_{max}$. In other words, this approach applies the merging level policy only to $L_{max}$. By doing so, the lazy leveling policy efficiently mitigates the trade-off problem of LSM-tree by reducing not only the write amplification but also the read amplification. The lazy leveling policy can co-exist with SFM because our scheme classifies the read/write hotness of key-spaces within the same level.

Meanwhile, ElasticBF [32] dynamically allocates Bloom filters to SSTable files based on their hotness within the same level. For it, this approach estimates the access frequency of each SSTable file and dynamically adjusts the amount of Bloom filters for SSTable files at a fine granularity. As a result, this approach effectively reduces the overall false-positive rate in the given main memory budget. This approach can be partially applied to SFM. Since both SFM and ElasticBF identify the hotness of the key-spaces, only the fine-grained Bloom filters management approach of ElasticBF can be added to SFM. Once applying this approach to SFM, more Bloom filters may be assigned to SSTable files in the read-intensive key-spaces. In the meantime, SSTable files corresponding to the non-read-intensive key-spaces may be constructed with a relatively small amount of Bloom filters. In consequence, the read performance of SFM may be further improved by adopting ElasticBF, without allocating the additional memory space for Bloom filters.

VII. DISCUSSION

A. LIMITATIONS

The SFM can alleviate read amplification problem of the conventional tiering-based LSM-trees while maintaining low write amplification. To achieve this, SFM first estimates the read intensity of each fragmented key-space by tracking its read/write hotness. Then, SFM applies the leveling merge policy to the read-intensive key-spaces in order to mitigate the read amplification problem. Consequently, the tiering merge policy is applied only to non-read-intensive key-spaces under SFM. Unfortunately, this approach is not without limitations.

Since SFM identifies read intensities of key-spaces based on their read/write hotness, it may not provide benefits to some key-spaces in which the frequency of read operations is similar to that of write operations. As described above, this is because SFM applies the tiering merge policy to such read-write-mixed key-spaces in order to ensure low write amplification. Thus, SFM cannot reduce the read amplification for such key-spaces, compared with that of the tiering merge policy.

B. FUTURE WORK

Several optimizations can be suggested to alleviate the limitations of not only the tiering merge policy but also the SFM. One of the opportunities to improve their limitations is to adaptively tune the threshold $T$ of each key-space according to its read/write hotness. In case of the tiering merge policy, write amplification can be further reduced when the threshold $T$ is configured to higher value. Meanwhile, configuring threshold $T$ to higher value leads to the side-effect of increasing read amplification. As with the tiering merge policy, SFM cannot be completely free from this trade-off problem between the read and write amplification because it delays the merge operations by stacking SSTable files up to the statically configured threshold $T$ for non-read-intensive key-spaces. Hence, dynamically capturing the read/write intensity of each key-space and adaptively tuning its threshold may further improve the intrinsic trade-off problem of the SFM.

Under this adaptive approach, the highly write-intensive key-spaces may be assigned relatively high threshold $T$ to reduce write amplification. Likewise, the threshold $T$ of some key-spaces may be relatively lower according to its read/write hotness. In the extreme case, threshold $T$ of the highly read-intensive key-spaces may have minimum value. However, this adaptive approach has to be designed with more specific hotness estimation policy in that it requires more subdivided criteria for identifying the read/write intensity
of key-spaces. We leave exploiting this adaptive approach for future work. We expect that this adaptive approach will further develop the tiering-based LSM-trees, including SFM.

VIII. CONCLUSION

We focused on mitigating the read amplification problem of LSM-tree while maintaining low write amplification in the key-value stores. To achieve this, we introduced a novel LSM-tree scheme that delays merge operations only for non-read-intensive key-spaces by identifying the read intensity of each key-space. We further proposed the efficient hotness estimation technique with the hotness initialization/initialization technique in order to accurately extract non-read-intensive key-spaces. Experimental results clearly show that our scheme not only improves throughput by up to 1.67× but also reduces write amplification by up to 61.52%, compared to the conventional schemes. Moreover, we demonstrate that our scheme decreases the average latency by up to 41.41%, compared with the conventional schemes, by alleviating their read amplification problem. Especially, we confirmed these performance benefits of our scheme based on the read/write access pattern traced from the real-world application.

REFERENCES

[1] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” 2015, arXiv:1512.01274. [Online]. Available: https://arxiv.org/abs/1512.01274.

[2] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, “Atlas: Baidu’s key-value storage system for cloud data,” in Proc. 31st Symp. Mass Storage Syst. Technol. (MSST), May 2015, pp. 14–18.

[3] G. DeCandia, D. Hastorun, M. Jambpani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” ACM SIGOPS Oper. Syst. Rev., vol. 41, no. 6, pp. 205–220, 2007.

[4] 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 Trans. Comput. Syst., vol. 26, no. 2, pp. 1–26, 2008.

[5] S. Ghemawat and J. Dean. (2021). Leveldb is a Fast Key-Value Storage Library Written at Google That Provides an Ordered Mapping From Strings to Key Strings. Values. Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/google/leveldb

[6] Facebook. (2021). Rocksdb: A Persistent Key-Value Store for Fast Storage Environments. RocksDB. Accessed: May 11, 2021. [Online]. Available: https://rocksdb.org/

[7] T. G. Armstrong, V. Ponnukanti, D. Borthakur, and M. Callaghan, “LinkBench: A database benchmark based on the Facebook social graph,” in Proc. Int. Conf. Manage. Data (SIGMOD), 2013, pp. 1185–1196.

[8] Facebook. (2018). Cassandra on RocksDB at Instagram. Facebook for Developers. Accessed: May 11, 2021. [Online]. Available: https://developers.facebook.com/videos/8-2018/cassandra-on-rocksdb-at-instagram

[9] J. Jin, J. Bae, C. Chung, Arvind, and S. Lee, “Pikc: High-speed in-memory key-value store with bounded tails,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2020, pp. 173–187.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in Proc. 1st ACM Symp. Cloud Comput. (SoCC), 2010, pp. 143–154.

[11] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, “Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook,” in Proc. 18th USENIX Conf. File Storage Tech. (FAST), 2020, pp. 209–223.

[12] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” Acta Inform., vol. 33, no. 4, pp. 351–385, 1996.

[13] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” ACM SIGOPS Oper. Syst. Rev., vol. 44, no. 2, pp. 35–40, Apr. 2010.

[14] C. Luo and M. J. Carey, “LSM-based storage techniques: A survey,” VLDB J., vol. 29, no. 1, pp. 393–418, Jan. 2020.

[15] H. Amur, D. G. Andersen, M. Kaminsky, and K. Schwan, “Design of a write-optimized data store,” CERCs, Brookfield, CO, USA, Tech. Rep. 193, 2013, pp. 1–12.

[16] T. Xiao, P. Huang, H. Yuan, A. Arora, K. Gupta, and P. Konka, “TRIAD: Creating synergies between memory, disk and log in structured key-value stores,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2017, pp. 363–375.

[17] P. Dillingner. (2020). Universal Compaction. RocksDB Wiki Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Universal-Compaction

[18] S. Alsulseaie, Y. Altowairi, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheellangi, K. Faraza, and E. Gabrielova, “AsterixDB: A scalable, open source BDMS,” Proc. VLDB Endowment, vol. 7, no. 14, pp. 1905–1916, Oct. 2014.

[19] N. Dayan and S. Idrees, “Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging,” in Proc. Int. Conf. Manage. Data, May 2018, pp. 505–520.

[20] H. Lee, M. Lee, and Y. I. Eom, “Partial tiering: A hybrid merge policy for log structured key-value stores,” in Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp), Jan. 2021, pp. 20–23.

[21] A. Tai, A. Kryczka, S. O. Kanajarla, C. Petersen, M. Antonov, M. Walij, K. Jamieson, M. J. Freedman, and A. Cidon, “Live recovery of bit corruptions in datacenter storage systems,” 2014, arXiv:1805.02790. [Online]. Available: https://arxiv.org/abs/1805.02790

[22] Y. Wu. (2020). Write Ahead Log File Format. RocksDB Wiki Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-File-Format

[23] R. K. Madhulal. (2021). Memtable. RocksDB Wiki Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/facebook/rocksdb/wiki/MemTable

[24] N. Dayan. M. Athanasouiss, and S. Idrees, “Monkey: Optimal navigable key-value store,” in Proc. ACM Int. Conf. Manage. Data, May 2017, pp. 79–94.

[25] O. Balmou, F. Dini, W. Zwaneepoel, K. Gupta, R. Chandhiramooorth, and D. Didona, “SILK: Preventing latency spikes in log-structured merge key-value stores,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2019, pp. 753–766.

[26] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdardsson, Y. Vignfus, and A. Gavrilovska, “Mutant: Balancing storage cost and latency in Ism-tree data stores,” in Proc. ACM Symp. Cloud Comput. (SoCC), 2018, pp. 152–173.

[27] Z. Cao. (2020). RocksDB Trace, Replay, Analyze, and Workload Generation. RocksDB Wiki Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/facebook/rocksdb/wiki/RocksDB-Trace-Replay-Analyze#trace-and-Workload-Generation

[28] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, “Elasticfch: Elastic Bloom filter with hotness awareness for boosting read performance in large key-value stores,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2019, pp. 739–752.

[29] M. Kerrisk. (2021). ext4(5). Linux Manual Page. Accessed: May 11, 2021. [Online]. Available: https://man7.org/linux/man-pages/man5/ext4.5.html

[30] P. Dillingner. (2021). Benchmarking Tools. RocksDB Wiki Github. Accessed: May 11, 2021. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[31] M. Kerrisk. (2021). iostat(1). Linux Manual Page. Accessed: May 11, 2021. [Online]. Available: https://man7.org/linux/man-pages/man1/iostat.1.html
[36] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The bloomier filter: An efficient data structure for static support lookup tables,” in Proc. 15th Annu. ACM-SIAM Symp. Discret. Algorithms (SODA), 2004, pp. 30–39.

[37] P. Muth, P. O’Neil, A. Pick, and G. Weikum, “The LHAM log-structured history data access method,” VLDB J. Int. J. Very Large Data Bases, vol. 8, nos. 3–4, pp. 199–221, Feb. 2000.

[38] 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 Proc. USENIX Annu. Tech. Conf. (ATC), 2015, pp. 71–82.

[39] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, “SlimDB: A space-efficient key-value storage engine for semi-sorted data,” Proc. VLDB Endowment, vol. 10, no. 13, pp. 2037–2048, Sep. 2017.

[40] C. Mathieu, C. Staelin, N. E. Young, and A. Yousefi, “Bigtable merge compaction,” 2015, arXiv:1407.3008. [Online]. Available: https://arxiv.org/abs/1407.3008

[41] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovic, and R. Shahout, “EvenDB: Optimizing key-value storage for spatial locality,” in Proc. 15th Eur. Conf. Comput. Syst., Apr. 2020, pp. 1–16.

[42] S. Han, D. Jiang, and J. Xiong, “Splitkv: Splitting IO paths for different sized key-value items with advanced storage devices,” in Proc. 12th USENIX Workshop Hot Top. Storage File Syst. (HotStorage), 2020, pp. 1–8.

[43] R. Thonangi and J. Yang, “On log-structured merge for solid-state drives,” in Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE), Apr. 2017, pp. 683–694.

HOYOUNG LEE received the B.S. degree in computer engineering from Kwangwoon University, South Korea, in 2019. He is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Sungkyunkwan University. His research interests include database management systems, file and storage systems, and operating systems.

MINHO LEE received the B.S. degree in electronic and electrical engineering and the M.S. and Ph.D. degrees in electrical and computer engineering from Sungkyunkwan University, South Korea, in 2013, 2015, and 2021, respectively. He is currently a Postdoctoral Researcher with the Department of Electrical and Computer Engineering, Sungkyunkwan University. His research interests include virtualization, file and storage systems, and operating systems.

YOUNG IK EOM received the B.S., M.S., and Ph.D. degrees in computer science and statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. From 2000 to 2001, he was a Visiting Scholar with the Department of Information and Computer Science, University of California at Irvine, Irvine, USA. He was a President of the Korean Institute of Information Scientists and Engineers, in 2018. His research interests include virtualization, operating systems, file and storage systems, cloud systems, and UI/UX systems. Since 2021, he has been serving as the Vice Chair for the IEEE CTSoc Technical Committee of MDT. 