A Novel Multi-Stage Forest-Based Key-Value Store for Holistic Performance Improvement

Ziyi Lu, Qiang Cao, *Senior Member, IEEE*, Fei Mei, Hong Jiang, *Fellow, IEEE*, and Jingjun Li

Abstract—Key-value (KV) stores based on multi-stage structures are widely deployed to organize massive amounts of easily searchable user data. However, current KV storage systems inevitably sacrifice at least one of the performance objectives, such as write, read, space efficiency etc., for the optimization of others. To understand the root cause of and ultimately remove such performance disparities among the representative existing KV stores, we analyze their enabling mechanisms and classify them into two fundamental models of data structures facilitating KV operations, namely, the multi-stage tree (MS-tree), and the multi-stage forest (MS-forest). We build SifrDB, a KV store on a novel split forest structure, that achieves the lowest write amplification across all workload patterns and minimizes space reservation for the compaction. To mitigate the read amplification inherent in MS-forest, we introduce a bloom filer mechanism based on Sorted String Tables (SSTs). Furthermore, we also present a highly efficient parallel search approach that fully exploits the access parallelism of modern flash-based storage devices to substantially boost the read performance. Evaluation results show that under both micro and YCSB benchmarks, SifrDB outperforms its closest competitors, i.e., the popular MS-forest implementations, making it a highly desirable choice for the modern KV stores.

Index Terms—Key-value, multi-stage, LSM-tree, parallel search

1 Introduction

Key-value (KV) stores have become a fundamental component of modern storage systems supporting data-intensive applications. Most of KV stores are implemented based on multi-stage structures, such as LevelDB [1], RocksDB [2], Cassandra [3], BigTable [4], HBase [5], LSM-trie [6], PebblesDB [7], ForestDB [8], SlimDB [9], etc. Multi-stage structures aggregate small random writes in memory and write them to hierarchical storage levels as a set of sequential logs that are organized as Sorted String Tables (SSTs). The data in the lower stage are compacted into the higher stage with increasing capacity. To accelerate lookup, multi-stage structures maintain one or multiple logical indexing B-trees in each stage. Multi-stage structures are efficient for block devices including both HDDs and SSDs [10], [11], [12]. However, our in-depth empirical study reveals that existing implementations generally trade off at least one performance objective in favor of the optimization of others, resulting in large disparities in performances of writes, reads, and space efficiency.

To understand the root cause of and ultimately remove such performance disparities among the representative existing KV stores, we analyze their enabling mechanisms and identify two main structure models, namely, the multi-stage tree (MS-tree) structure that maintains one global logical index tree in each stage, as represented by LevelDB, and the multi-stage forest (MS-forest) structure that allows multiple logical index trees with overlapped key range in each stage, as typified by the size-tiered compaction in Cassandra (or Size-Tiered for brevity). In general, Size-Tiered has the advantage of high data ingest ratio but requires extra-large preserved space on compacting, while LevelDB is more efficient for reads and runtime space requirement.

With the knowledge and insight acquired from the theoretical and experimental analysis based on our proposed MS-tree/forest classification in Section 2, we build a KV store, called SifrDB, on top of a novel split forest structure[1] to address the existing problems from the perspectives of three important performance objectives, i.e., write, read, and space efficiency.

Specifically, SifrDB performs compaction by a method similar to that used in the stepped-merge [13] or the Size-Tiered [14] implementations that are popular in modern large-scale KV stores [3], [4], [5], to leverage the advantages of MS-forest for random writes. SifrDB splits the entire-stage KV pairs into a series of fix-sized Sorted String Tables (SSTs) with their own internal index, referred to as the *split storing* in this

• Z. Lu is with the Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, Ministry of Education, Wuhan National Laboratory for Optoelectronics, and the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: luziyi@hust.edu.cn.
• Q. Cao is with the Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, Ministry of Education, Wuhan National Laboratory for Optoelectronics, Wuhan 430074, China. E-mail: caoqiang@hust.edu.cn.
• F. Mei is with Huawei Technologies, Hangzhou 310028. E-mail: f_mei@hotmail.com.
• H. Jiang is with the University of Texas at Arlington, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.
• J. Li is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: jingjunli@hust.edu.cn.

Manuscript received 2 Apr. 2019; revised 15 Oct. 2019; accepted 17 Oct. 2019. Date of publication 29 Oct. 2019; date of current version 10 Jan. 2020. (Corresponding author: Ziyi Lu.) Recommended for acceptance by W. Yu Ph.D. Digital Object Identifier no. 10.1109/TPDS.2019.2950248

1. The name Sifr comes from letters in the words 'Split' and 'forest'.

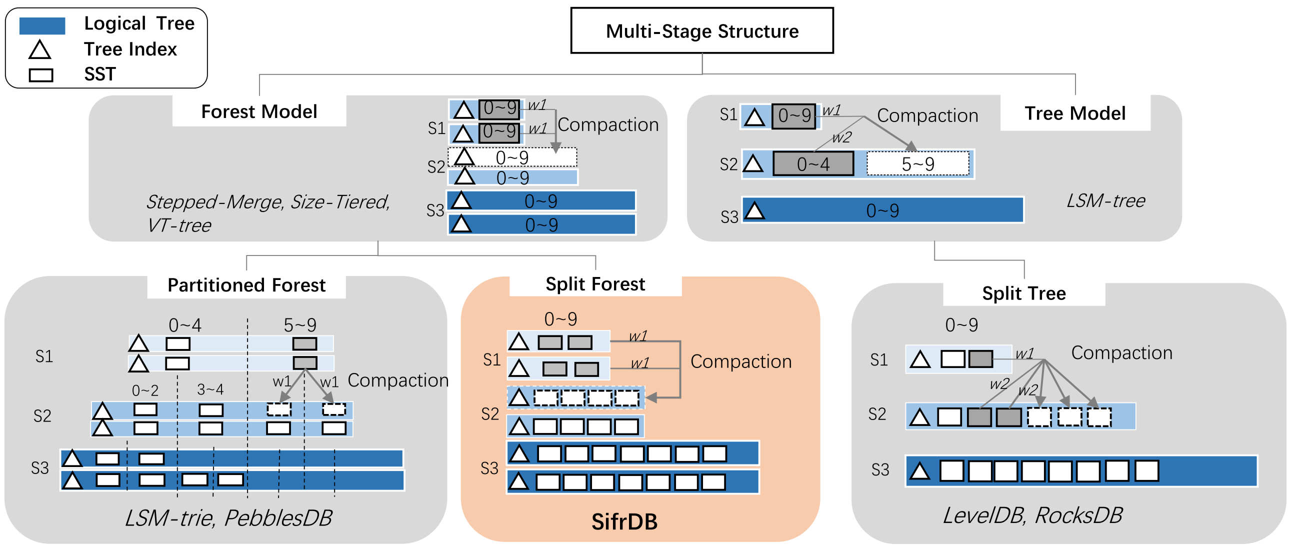


Fig. 1. A taxonomy of the popular multi-stage implementations under the MS-tree or MS-forest model. While rewriting in the MS-forest implementations only takes place across stages, it happens both across stages and within each stage in the MS-tree implementations (*w1* indicates data is written across the stages, and *w2* indicates data is written within a stage).

paper, so as to easily and efficiently detect key-range overlapping to enable sequential write optimization. Although the MS-tree implementations have adopted the split storing approach, to the best of our knowledge, SifrDB is the first that applies this approach to the MS-forest model.

Furthermore, to harness the advantages of the MS-forest while avoiding its disadvantages. SifrDB employs several optimization techniques. First, An SST-level bloom filter mechanism is designed to reduce the cost of searching SSTs. Second, we design a novel parallel-search mechanism for SifrDB to fully exploit the access parallelism of SSDs. Finally, an early-cleaning technique is proposed to ensure that the runtime space requirement for compaction is kept at a minimal level, which solves a serious problem suffered by the Size-Tiered [15]

In summary, this paper makes the following contributions:

- We present MS-tree/forest classification for existing KV implementations based on multi-stage structure, and then analyze their structural advantages and disadvantages experimentally.
- We propose a new multi-stage structure, SifrDB, to combine the advantages of MS-tree/forest structures. We also design several techniques, as early-cleaning, SST-level bloom filters, and parallel-search mechanism, to avoid negative effects of the forest structure.
- We implement SifrDB based on LevelDB. Evaluation results show that SifrDB outperforms the MS-forest implementations (i.e., Size-Tiered and PebblesDB) consistently in both microbenchmarks and YCSB, while achieving 11× higher throughput than the MS-tree implementations (i.e., LevelDB and RocksDB) in random writes. In a data store with low memory provision, which has become a trend in the cloud store [16], [17], SifrDB exhibits the best read performance among all the implementations.

The rest of the paper is organized as follows. We introduce the background and MS-tree/forest classification in Section 2, then describe the motivation in Section 3. We further present and evaluate SifrDB in Section 4 and Section 5, respectively. In Section 6 we discuss related work and conclude in Section 7.

2 Background

2.1 MS-Tree/Forest Classification

Multi-stage structures batch random writes in memory and sequentially write them to storage organized as hierarchy of stages with exponential growth capacity. Each stage comprises one or multiple Sorted String Tables (SSTs) that contain a sorted sequence of key-value pairs and a B-tree-like storing structure, which is introduced by BigTable [4] and is a simple but efficient mechanism for block storage devices. An SST is always stored as an immutable file and usually consists of two parts: the body composed of sorted KV strings and the tail containing the index data built on top of the sorted key-value strings. Multi-stage structures further maintain one or multiple logical indexing trees upon SSTs in each stage to locate requested keys. A logical index tree and its indexing SSTs are collectively referred to as a logical *tree* in the rest of the paper unless specially noted otherwise. The key ranges of the SSTs within the same tree do not overlap.

Next, we introduce the MS-tree/forest classification that reveals the essential properties of the existing popular KV store implementations. It is these properties that help anchor our proposed research.

Fig. 1 illustrates a taxonomy of popular multi-stage implementations under the MS-tree/forest classification. The MS-tree model means there exists only one tree in each stage. The logical index tree within it is used to precisely position a query key to a candidate SST. In contrast, in each stage of the MS-forest model there exist multiple independent trees with overlapped key-range SSTs. It means multiple SSTs in a stage could involve a target key.

MS-Tree Model. MS-tree model originally is introduced as the log-structured merge-tree [18]. For the MS-tree model, in each stage only one sorted index tree is allowed and a compaction on a stage S_i merge-sorts the tree in S_i with the tree in S_{i+1}, and writes the resulted new tree to S_{i+1}. In other words, data in the MS-tree model is re-written not only across the stages (*w1* in Fig. 1), but also within a stage (*w2* in Fig. 1). For example, in the MS-tree model demonstrated in Fig. 1, when S_1 is full, the compaction process merge-sorts the tree in S_1 and the tree in S_2 to produce a new tree that is written to S_2, and the two trees that participate the merge are deleted. After the compaction, S_1 is emptied and S_2 becomes larger.

MS-Forest Model. For the MS-forest model, multiple trees with overlapped key-range are allowed in each stage (in Fig. 1 each stage allows two trees), and a compaction on a stage S_i merge-sorts the multiple trees in S_i to a new tree that is directly written to S_{i+1}. Therefore, data in the MS-forest model is re-written only across the stages (*w1* in Fig. 1).

The fundamental virtue of the MS-forest model is that it incurs much lower write amplification than the MS-tree model. However, the latter is more efficient for reads, as will be detailed in the next section (Section 3) together with the popular multi-stage tree/forest variants. The MS-tree/forest classification not only indicates a high-level design preference, but also helps pinpoint in benchmark results the individual impacts of the implementations. For example, VT-tree [19] builds on top of a MS-forest model and uses a stitching technique to reduce write amplification. Using the classification, we can clearly figure out which part of the performance improvement in the benchmark result is from the structure effect and which part is from the stitching technique.

2.2 Existing Implementations

Many existing MS-tree and MS-forest implementations often use some mechanism to divide each logical tree into SSTs in different ways to adjust the granularity of compaction. These mechanisms also have a great impact on the space requirement for compaction operation (specifically discussed in Section 3.3). In this subsection, we just focus on a *split* mechanism in MS-tree and a *partition* mechanism in MS-forest. We called the two implementations as called *Split Tree* and *Partitioned Forest*.

Split Tree. Split Tree is a common MS-tree based implementation that employs a split approach to divide the data in each tree into a series of fix-sized SSTs with exclusive key ranges, such as LevelDB [1].

LevelDB is an MS-tree based implementation that employs the split mechanism to store the trees, where each tree is stored as independent SST files with a global index used to position a query key to a candidate SST. LevelDB has two salient advantages over the latter by adopting the partial merge. After selecting an SST in a stage during compaction, LevelDB first determines which SSTs are overlapped in the next stage. If not, the SST is pushed to the next stage without rewriting its data by only updating the global index. As a result, LevelDB is optimized for sequential workloads. This is useful for some special workloads, such as the time-series data [20] collected by a sensor. RocksDB [2] is based on a version of LevelDB and has done many optimizations on it. Although the current version of RocksDB can already support switching between MS-tree and MS-forest models in each stage, we only discuss the MS-tree version of RocksDB, which has many same features with LevelDB, including the definition of SST and compaction way.

Partitioned Forest. Partitioned Forest is a common MS-forest based implementation that partitions the trees in each stage to a set of non-overlapped key ranges and the compaction on a stage only merges the data within a target key range into a new SST. Compared to merging all the trees in one stage for a compaction, partitioned forest has a smaller compaction granularity, and reduces time and space cost in each compaction operation.

LSM-trie [6] is the first implementation of this kind and uses the hashed prefix as the partition boundary to build a trie index. ForestDB [8] also uses hash prefix to partition and combines the advantages of trie and B+tree to optimize the long key. Using hashed prefix as the edge of the partition guarantees the fairness of the query, but also invalidates the range query operation.

PebblesDB [7] proposes to use the real keys as the partition boundaries to improve range query. However, since the compaction in PebblesDB generates SST files with strict respect to the boundaries, an SST file is created even only one key falls into a partition. As a result, PebblesDB produces SST files with variable and unpredictable sizes that can be quite small, hence introducing I/O overheads on the block storage [21], [22].

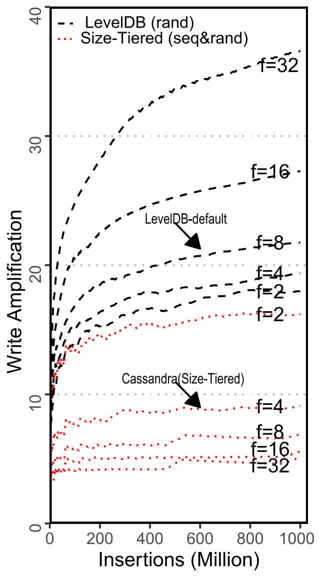
3 Motivation

In this section, three key properties are used to characterize the holistic performance in a multi-stage structure as write performance, read performance and space requirement. We analyze these properties to understand the intrinsic advantages and disadvantages of the MS-tree/forest classification above. By reviewing and analyzing the pertinent implementation features of the most representative multi-stage based KV stores, we are motivated to build SifrDB to improve holistic performance in a multi-stage structure.

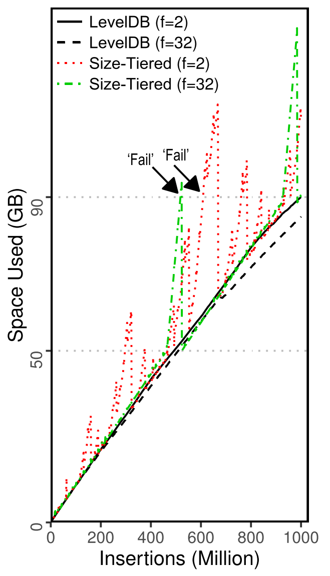
3.1 Write Amplification

Reducing write amplification is the most important research objective for the multi-stage structures. In a typical write process, a user sends data (i.e., *user data*) to the KV store application that then persists a version of that data (i.e., *app data*) to the underlying storage system. However, the application may purposefully rearrange the data on the storage periodically (e.g., compaction) and generates another kind of app data, hence amplifying the write traffic relative to the user data. The ratio of the size of the *app data* to that of the *user data* is called *write amplification*, which not only adversely affects the write performance, but also impacts the lifetime of the flash-based storage devices.

MS-Tree Model. In MS-tree, when a stage S_i is full, a compaction process is triggered to merge its tree to that on the next stage S_{i+1}, which entails rewriting the content of the tree from stage S_i to stage S_{i+1} as well as rewriting the content of stage S_{i+1}'s existing tree. After a number of compactions that move data from S_i to S_{i+1}, S_{i+1} becomes full, which triggers a compaction process to merge S_{i+1} to S_{i+2}. This process repeats itself iteratively from the top stage all the way to the bottom one. As a result, while the user data is sent by the user only once, this data is written multiple



(a) Write Amplification



(b) Operational Space Requirement

Fig. 2. Randomly writing to stores configured to different growth factors. *(The size of the KV pair is 116 bytes. In (b), for a 90 GB storage provision, Size-Tiered will fail at the arrows where the user data is much less than 90 GB.).*

times in each stage by the MS-tree based application, causing significant write amplification.

MS-Forest Model. The MS-forest model, which has its roots in the stepped-merge approach [13] that serves as an alternative structure to MS-tree, allows multiple trees to coexist within each stage. A compaction on a full stage S_i merges the multiple trees in this stage to produce a new tree that is directly written to the next stage S_{i+1} as an additional tree, without interfering with the existing trees of S_{i+1}. That is, in MS-forest, the same data is written only once in each stage, hence incurring lower amplification than the MS-tree model.

Generally, for both the MS-tree and MS-forest models, the first stage's capacity c_1 is predefined and other stages' capacities increase geometrically by a constant *growth factor* f. Assuming there are N stages and the last stage is full, if we denote the size of dataset as D, since $D = c_1 \cdot \frac{f^N-1}{f-1}$, we get $N \approx \log_f \frac{D}{c_1} + \log_f(f-1)$, i.e., $N = O(log_f D)$. Because of the geometric increase of the stage capacities, the number of rewritten times of the data in the last stage can approximately represent the overall write amplification. For MS-forest, the data in the last stage has been written once in each stage, leading to a write amplification of N, or $\mathbf{log_f D}$. For MS-tree, the data has been written $\frac{f}{2}$ times on average in each stage,[2] incurring a write amplification of $\frac{f}{2} \cdot N$, or $\mathbf{\frac{f}{2} \cdot log_f D}$.

The partial merge mechanism used in LevelDB (i.e., only selecting an SST instead of the whole tree to merge) based on the split tree does not influence the write amplification because the ratio of the re-written data to the merged data (re-written ratio) does not change. For example, assuming the existing data in S_{i+1} is k times larger than that in S_i, a full merge will cause a re-written ratio of $k+1$. If both S_i and S_{i+1} are split to SSTs, for each SST of S_i there will be k overlapped SSTs in S_{i+1}, and merging an SST causes a re-written ratio of $k+1$, the same as the full merge.

2. For a stage S_{i+1}, it becomes full after receiving f components from S_i. Each of the components is written once when it is first merged to S_{i+1}, and the xth ($1 \leq x \leq f$) component is written $f-x$ times in the subsequent merge of the remaining components until S_{i+1} is full.

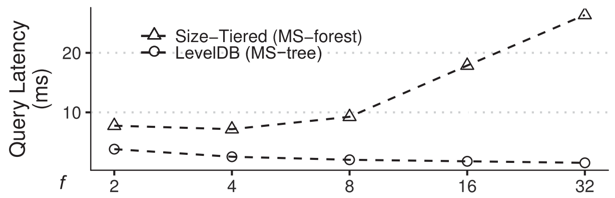


Fig. 3. Read latency in the datasets generated previously.

Our experiment result in Fig. 2a, which demonstrates the write amplifications of LevelDB (representing the MS-tree model) and Size-Tiered (representing the MS-forest model) as a function of number of insertions with configuration of different growth factors, traces the above theoretical analysis well: larger growth factor leads to higher write amplification in LevelDB while that has the opposite effect on Size-Tiered.

3.2 Read Degradation

In multi-stage structures, a write request (i.e., insertion, update, or deletion) is converted to a new insertion operation, and the data from the bottommost stage is moved to higher stages gradually in batch to avoid random writes on the underlying storage device. The trees in different stages have their respective priorities, and all of them are candidates for a query request. A point query processing is implemented by searching all the candidate trees serially according to their relative priorities until the query key is found or all the trees have been searched without finding one. A high-priority tree (i.e., containing the latest insertions) must be searched first to guarantee the validity of the search result. In general, the multi-stage structures trade off the read performance for write performance.

The latency of searching the trees in different stages increases slowly due to the $logN$ complexity of the B-tree structure [23]. For example, searching a 2 MB tree needs 3 random I/Os, while searching a tree that is a hundred times larger only increases one more I/O. Hence, the number of trees a read request needs to search is critical to the query latency. In Section 3.1 we have known that with larger growth factor less stages are maintained. Since in MS-tree each stage only allows one tree, less stages means less candidates trees to search for a query. However, for the MS-forest model, while the number of stages decreases logarithmically, the number of trees that are allowed in each stage increases linearly. As a result, in the MS-forest, the total number of candidate trees, which cause read amplification, usually has a positive correlation with the growth factor. In this respect, the MS-tree model is more advantageous for read than the MS-forest model because the former needs to search only one tree in each stage, while the latter maintains and requires searching multiple trees in each stage, which is validated in Fig. 3 that plots the experimental read latency of LevelDB and Size-Tiered in the datasets generated previously with different growth factor configurations.

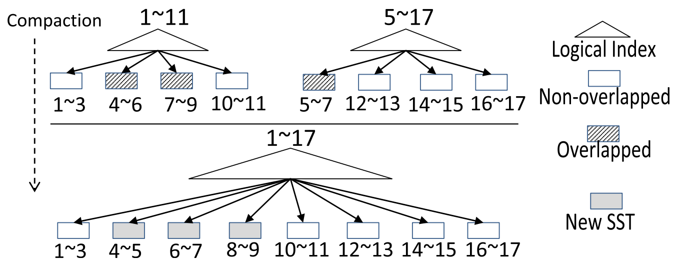


Fig. 4. Compaction is performed on logical trees, while the merge is performed on the physical SSTs.

Range query is an important feature of KV stores. A range query is processed by first serially seeking each candidate tree to find the start key of the range, and then advancing across the trees with an election mechanism that selects the smallest key among all the trees in each step. Performance of range query in MS-forest also degrades more seriously than that of MS-tree because more candidate trees take more time to seek and select. For the common range queries that scan tens to hundreds of keys, the seek process dominates the range query overheads because it usually involves I/Os that load blocks of KVs to memory for the subsequent selection process.

3.3 Space Requirement for Compaction

A compaction operates on a set of SSTs by merge-sorting their key-values to new SST file/files. The operated SSTs cannot be deleted before the compaction is finished [1], [24]. Because the operated SST files and the new SST files both hold the storage space, *more storage space than the actual user data, up to twice as much, must be reserved in order to guarantee the successful processing of the compaction*. Although the space held by the operated SST files can be reclaimed eventually, the reservation is a necessity to prevent the system from failures caused by "insufficient space", hence leading to a low space efficiency. For example, with the size-tiered compaction in Cassandra, in the worst case exactly twice as much free space as is used by the SSTs being compacted would be needed, which results in only 50 percent space efficiency [15]. A problem that can arise from this space inefficiency is that a server could fail when the user writes only a dataset half the size of the provisioned storage space, which is becoming severe in Cassandra [15]. Fig. 2b shows the storage space requirement in the insertion process, which indicates that Size-Tiered will fail when the user data set is only half of the storage capacity. The partial merge based on the split tree used in LevelDB enables low space requirement for a compaction. Because a partial merge involves only a small part of the trees in the next stage regardless of the tree's size, the aforementioned high space reservation problem is significantly mitigated in LevelDB, and its sibling implementation of RocksDB. For example, with SSTs size of 2 MB and a growth factor of 10 (default in LevelDB), about 11 SSTs are involved in a compaction (one selected SST in the lower stage and 10 estimated overlapped SSTs). Therefore, an extra reserved space of about 22 MB is sufficient for a compaction on any stage, a negligible size compared to the space required by Size-Tiered for a KV store of hundreds of GBs, as can be seen in Fig. 2b.

TABLE 1
Performance Differences of Different KV Implementations

	Split-tree	Partitioned-forest	Sifr
Write performance	×	✓	✓
Read performance	✓	×	✓
space utilization of compaction	✓	●	✓

(✓ *means acceptable,*× *means deficient,*● *means acceptable at some cost)*

Space requirement in LSM-trie as partitioned forest is not high as in the Size-Tiered. However, since LSM-trie uses hash to partition the keys, it loses the range query feature.

Because of the different compaction mechanism, MS-forest has a longer turn-around time of compaction, and MS - tree has a higher compaction frequency. In both model, the latency of responses can be affected by compaction processing. But we do not discuss this problem in this paper. Because of the different compaction mechanism, in general, a compaction process of MS-forest has a longer turn-around time but lower frequency than that of MS-tree. The compaction processing in both such two models potentially increase the request latency, but can be scheduled to mitigate its neglect impact [25].

Summary — In the two sections above we have analyzed the two models of MS-structures, the MS-tree and MS-forest models, and introduced their popular implementation models as shown in Table 1. This analysis clearly suggests that there is not a one-size-fits-all solution. In addition to the different levels of severity of read performance degradation, the difference between representative MS-tree implementation LevelDB and MS-forest implementation Size-Tiered, discussed above, implies very divergent performance between them in write performance and space efficiency. In what follows we now present SifrDB, a KV store that is based on the MS-forest structure (Fig. 1) but attempts to remedy the deficiencies of MS-forest.

4 SIFRDB

To combine the performance advantages of partitioned forest and split tree implementations, we proposed SifrDB which is a unified implementation of write-optimized KV-store based on split forest.

SifrDB is based on a split forest storing. Each stage in SifrDB has multiple independent logical trees, each of which is composed of a group of non-overlapped and fix-sized SSTs. While a compaction is performed on several logical trees, the actual merge is performed at the granularity of SSTs and only involves SSTs with overlapped key ranges to eliminate the unnecessary data re-writing under sequential (or sequential-intensive) workloads. Note that each logical tree of stage 1 consists of a single SST. When data is flushed to disk, SifrDB writes these data to an SST and put it into stage 1 as a logical tree.

More specifically, take Fig. 4 as an example, where a compaction is performed on the two logical trees with key ranges of 1∼11 and 5∼17. In this example, only the three shaded SSTs with overlapped key ranges are merged and re-written, while metadata of the non-overlapped SSTs (unshaded) and the newly generated SSTs (solidly shaded) are added to the global index of the new logical tree. The compaction is finished by committing the information about the deletion of the two compacted logical trees (with key ranges of 1∼11 and 5∼17) and the generation of the new logical tree (with key range of 1∼17).

SifrDB performs compaction based on the MS-forest model (as illustrated in Fig. 1), it naively inherits the forest's advantage, i.e., low write amplification for random writes. On the other hand, with the split storing mechanism, SifrDB simultaneously obtains LevelDB's advantage for sequential writes, which is lost in other popular forest implementations [3], [6], [7]. In addition, the design of SifrDB is also able to achieve the effect of the stitching technique introduced by VT-Tree [19] that builds a second index on top of each tree of an MS-forest model.[3] Nevertheless, VT-Tree must deal with the garbage on the tree files that are eventually collected by rewriting the valid data (i.e., the data that is not rewritten in the compaction) in new places. On the contrary, SifrDB does not introduce garbage by splitting each tree to independently stored SSTs. Moreover, SifrDB enables early cleaning to keep the reserved space at a minimum, as detailed next. In other words, SifrDB avoids not only the additional space that VT-tree must reserve for its compaction, but also a space that is not reclaimed timely after the compaction.

SifrDB is based on the MS-forest model which by design incurs much lower write amplification under random writes than the MS-tree model. But it causes higher read amplification because of the tolerance of overlapping key-indexing range. To optimize read performance, SifrDB introduces SST-level filter and parallel-search mechanism in Sections 4.1 and 4.2, respectively.

Besides, the need for MS-forest-based stores to merge small tree files to larger ones (to reduce the number of candidate trees for reads) not only causes unnecessary rewriting of data of non-overlapped key ranges, but also requires high operational space reservation when compacting large files [15]. Therefore, SifrDB also provides the sequential-workload advantage and operational space efficiency the common MS-forest implementations lack, without sacrificing the random-workload advantage of the MS-forest model. We achieve these goals by leveraging the split storing mechanism while overcoming the challenges imposed by the fact compactions are still performed by full merge on the logical trees, as presented in Section 4.3.

4.1 SST-Level Filter to Reduce Read I/O

The MS-forest model allows several logical trees with overlapped key range in each stage. For one query operation, compared to MS-tree model which has only one SST per stage to involve the target data, MS-forest model scans one SST on each logical tree. It means that MS-forest model has k times the read amplification of MS-tree model (k is the number of logical trees in each stage) and causes read performance degradation. To reduce such read amplification, we implement an extra bloom filter approach called SST-level filter as the secondary indexes of logical tree indexes to further improve read performance. Specifically, when an SST is built, SifrDB builds an SST-Level filter correspondingly. When processing a query request, the candidate SSTs that have the matched key ranges are determined first. Then the corresponding SST-level filters can exclude most of non-target SSTs to reduce read IOs.

3. VT-tree is implemented based on an MS-forest model with a growth factor of 2.

Note that existing multi-stage structures have employed block-level Bloom filter as a secondary index to quickly filter out the non-target data blocks within an SST. The block-level bloom filter is stored in the tail of each SST file. In SifrDB, the SST-level filter differs from the block-level bloom filter in two respects: location and granularity.

Location. An SST-level filter is generated once its corresponding SST is created. The filter will not be updated because SSTs are immutable. Differ from block-level filters stored in the form of metablocks in SSTs, SST-level filters are in-memory structures in default to reduce as much read IOs as possible. For persistence, SifrDB stores all SST-level filters in a specific file. So, the bloom filters can be sequentially read from the file into the memory quickly when the system restarts or recovers. When memory space is not enough, caching only the most frequently accessed SST-Level filters in memory is an optional tradeoff. Because the size of each filter is fixed, the filters of requested SSTs can be easily located in the file according to their corresponding SST number.

Granularity. Although bloom filters can be applied in a variety of granularity (such as SST, tree and stage), we consider SST as an appropriate granularity because SST is the largest immutable unit in SifrDB. The filters can be generated or deleted as SST is generated or deleted. With larger granularity, each compaction operation results in an overwrite of the entire filter because bloom filter cannot be partially modified. Also, using SST granularity brings an acceptable dataset/memory ratio.

In SifrDB, we set SST-level filters with 10 bits for each key and the max size of each SST with 2 MB in default. In the workloads with 100 byte KV pairs, the SST-level filters require 1 percent memory space of the dataset in a default setting while the false positive rate is 0.819 percent. Adjusting the bits/key ratio can also be used for trading off memory usage and performance gain.

Some existing partitioned forest implementations (such as PebblesDB [7]) have used SST-level filter to improve read performance, but this technology works better in SifrDB which is based on split forest

Even though Bloom filter is designed for memory efficiency, it also consumes significant memory space for a large data store. As a result, a bloom filter could easily take up all the memory in a high dataset/memory ratio that is becoming popular on SSD-based storage systems [16], [17] and cause frequent I/Os for swapping. Considering that the KV pairs in real workloads tend to be even smaller than 100 bytes [6] and Facebook has begun to reduce memory provision for its cloud store [17] for economic reasons, third-party indexes such as bloom filter can only be used in a limited extent. Therefore, we design a parallel-search mechanism for SifrDB to further exploit the read I/O capability of high IOPS of modern SSDs, while relieving the dependence of memory consumption.

4.2 Parallel Search to Improve Read Performance

In this sub-section, we present a parallel-search mechanism for SifrDB. Fig. 5 illustrates the *Fix-count* (*FC*) and *Priority-aware* (*PA*) parallel-search schemes in SifrDB, compared

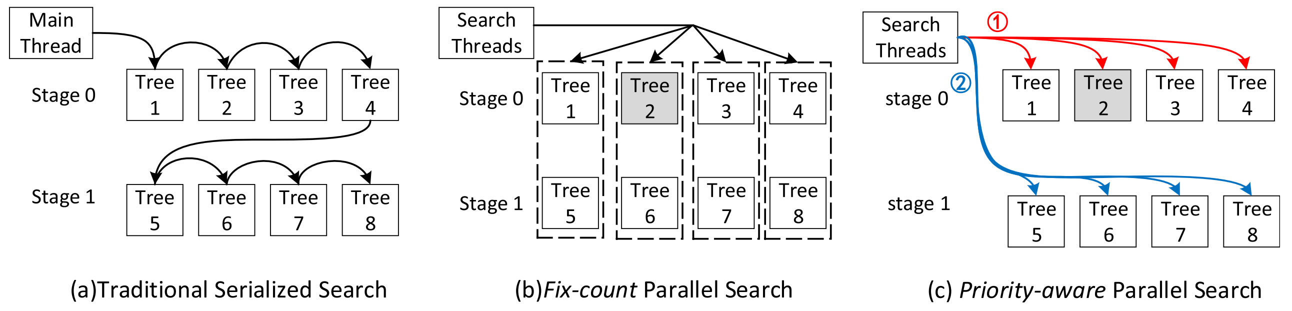


Fig. 5. Different mechanisms for threads to search the trees. (a) The trees are searched serially in order of their priorities. (b) All of the trees are searched concurrently by four search threads and each thread is assigned the task of searching two trees. (c) The trees are searched twice by four search threads, each time for four trees in a stage.

TABLE 2
CPU Core Time and 95/99 Percentile Latency for Queries (*"Tra" is the Traditional Way, and "Sifr-16" is SifrDB with 16 Search Threads*)

	Average Core Time (ms)					95 percentile latency (ms)					99 percentile latency (ms)				
Mem	256 MB	1 GB	4 GB	16 GB	Cached	256 MB	1 GB	4 GB	16 GB	Cached	256 MB	1 GB	4 GB	16 GB	Cached
Tra	0.052	0.030	0.023	0.014	0.012	12.3	5.46	3.12	1.85	0.376	18.7	6.22	3.75	2.17	0.398
Sifr-16	0.047	0.024	0.027	0.030	0.034	2.7	1.65	1.07	0.78	0.232	3.75	1.91	1.34	0.87	0.243

with the traditional serialized approach where the server thread serially searches the trees in order of their priorities and returns once the first result is found. The serialized approach is well suitable for hard disks with only one disk head. In MS-forest models, the trees in lower stages and the newer trees have higher priority.

Modern SSDs with high I/O parallelism can offer the potential to support parallel-search approaches. In the parallel search approach, a *server thread* spawns a bundle of background threads (referred to as *search threads*) on the candidate trees. When a query operation is processed, the server thread only puts its candidate trees into a queue in order of priorities, and then invoke search threads to query the SSTs of the candidate tree. Once the querying key is found in a high-priority tree by a *search thread*, the *server thread* can detect and return the result immediately, even though other search threads may be still processing querying on the rest of SSTs. In this case, the results of the latter *search threads* are simply discard afterwards. The achievement from the parallel search is not always come free. There are two kinds of costs introduced by the parallel search. One is the unnecessary I/Os on the low-priority trees when reading the keys that reside in high-priority trees, which is evaluated in Section 5.3. The other is the CPU cost in a low dataset/memory configuration environment, as shown by the average CPU core time in Table 2. We can see that when the memory provision is larger than 4 GB (equivalent to a 25:1 dataset/memory configuration), the CPU usage in SifrDB becomes higher than that in the traditional way.

To effectively trade off performance and resource wastage, we further present two parallel search schemes *FC* and *PA*. The former is configured to use a fixed number of search threads. The latter dynamically adjusts the background search threads according to the number of candidate trees in current priority.

Fix-Count Scheme. *FC* configures a fixed number of search threads. For example, as shown in Fig. 5b, if the query key is in tree 2 of the stage 0. In *FC* scheme with 4 threads, a search thread returns when the key is found in tree 2, but the other 3 search threads still work in both trees in stage 0 and stage 1. The choice of the number is important for trading off efficiency and performance and relies on the hardware resources, such as the number of CPU cores and SSD internal access parallelism. When this number is too small, the parallelism of SSD cannot be fully exploited and the query latency could increase. On the contrary, the larger number could lead to the parallelism wastage of both CPU and SSD. Although in this case the delay of a query is not affected, these search threads also cause a amount of I/Os and memory footprint. Fig. 6 shows the query latency on a 100 GB dataset and different memory provisions as a function of the number of search threads in the environments introduced in Section 5. The legends represents memory configurations.

Priority-Aware Scheme. To leverage the characteristic of multi-stage structure and save hardware resource, we design a priority-aware parallel search. In *PA* scheme, the server thread allocates a search thread to each candidate tree in current priority. Only when the query key misses in current trees, the next batch of searchings are triggered. As shown in Fig. 5c, the 4 trees in the same stage are considered in the same priority and will be searched simultaneously. If

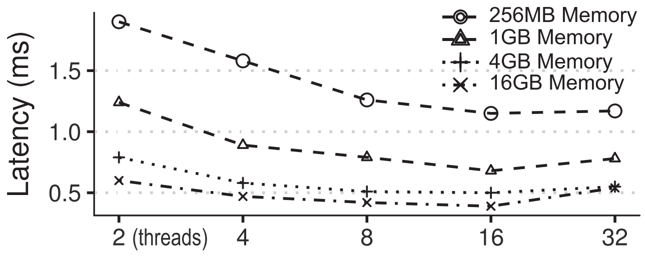


Fig. 6. Query latency as a function of the number of the background search threads with different memory provisions (100 GB dataset).

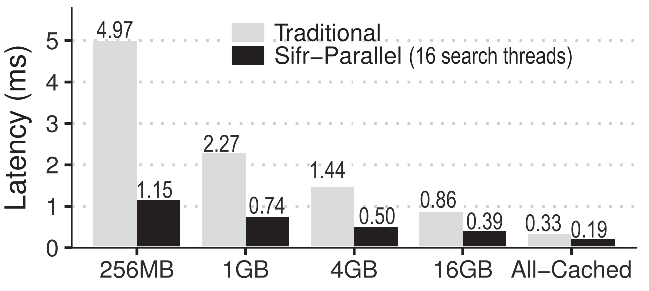


Fig. 7. A comparison in query latency between the traditional approach and SifrDB's parallel-search(*FC*) under different memory provisions (100 GB dataset).

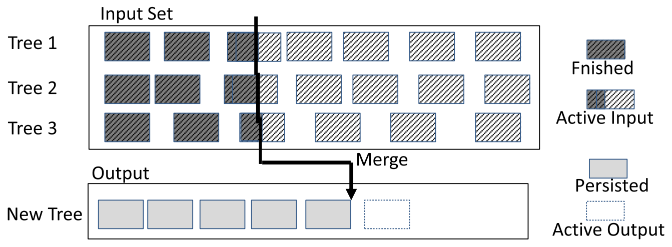


Fig. 8. Merge operation in SifrDB. Early-cleaning can be executed after a new SST is persisted.

the query operation is finished in the first batch of searching on stage 0, three extra searching of the trees on stage 1 are avoided. However, when the target key in the tree with low priority, *PA* might perform several turns to find the key, thus increasing the latency.

FC and *PA* schemes exhibit their own advantages, depending on the workload behaviors detailed in Section 5.3.3.

Additionally, we find that the parallel algorithm still performs better than the traditional approach even though when all the dataset has been cached in memory. Fig. 7 illustrates how the efficiency of the parallel-search algorithm degrades when the provisioned memory increases as expected. We can see that when all the data is cached, the parallel-search scheme is shown to still improve the performance by 1.7×. Tail latency is also an important metric users care about. Long tail latency in a multi-stage structure is mainly caused by queries that need to search multiple candidate trees and the queried data is not in cache, which is what the parallel search attacks. Therefore, the tail latency is likewise improved notably, as shown by the 95 and 99 percentile latency in Table 2.

Range Query. Range query can benefit from the parallel search in a straightforward way. For example, the process can start with executing a point-query for the start key of the range to load the blocks containing the to-be-scanned keys of the candidate trees to memory concurrently, which can significantly speed up the scan performance. In other words, the range query in SifrDB is composed of a parallel point-query and a traditional range query.

4.3 Early-Cleaning to Optimize Space Efficiency

In this sub-section, we present the early-cleaning technique designed to reclaim the operational space held by the merged trees as early as possible, even when the compaction is still under way, so that the data store service would not fail because of 'out-of-space'.

The idea behind early-cleaning is to safely delete the SSTs as soon as they have been successively merged to the new SSTs, i.e., their data have been persisted as new copies elsewhere. Nevertheless, to safely enforce early-cleaning to ensure data integrity and consistency, we must answer the following two questions.

1) When an unexpected crash happens, how to recover the data and guarantee data consistency?
2) How to process the read requests coming to the SSTs that have been deleted by early-cleaning?

The answers are *compaction journal* and *search request redirection* respectively, as explained below.

Compaction Journal. In the merge process, early-cleaning is called periodically to delete the input SSTs to reclaim the storage space. As shown in Fig. 8, early-cleaning can be scheduled after a new SST is sealed and persisted to delete the finished SSTs. Since the data in the deleted SSTs have been written to the new SSTs, if a crash happens halfway through the merge process, data consistency can be achieved by keeping the state information of the merge process, which is continued after the recovery. We use a small journal to record the merge state information before executing the early-cleaning, called compaction journal, which contains the metadata of the persisted SSTs in the output and the active SSTs in the input. In fact, the metadata of the persisted SSTs are the abuilding and uncommitted global index of the new tree. Note that if an input SST does not overlap with other SSTs, it is directly moved to the output and is not affected by the cleaning process.

Although each time when a new SST is persisted provides an opportunity for early cleaning, it can be ineffective and wasteful to clean too frequently. As a default, SifrDB sets the cleaning threshold to 10. That is, every time when 10 SST files are persisted an early-cleaning process is scheduled, which results in an operational space requirement equivalent to that of LevelDB. Users can configure a larger cleaning threshold value, and SifrDB is able to dynamically adjust the setting according to the amount of available storage space.

In the recovery process, SifrDB reads the latest merge state information from the compaction journal and continues the compaction merge by seeking to the correct positions of the active input SSTs, instead of the conventional approaches that simply discard the work that had been done before the crash. The correct positions are determined by the biggest key of the last newly persisted SST. Continuing-compaction brings extra benefit for a full-merge compaction crashed in operating on very large trees, since it can save a significant amount of time from a restarting-compaction approach that does the work from the beginning.

Search Request Redirection. With the early-cleaning technique, it is a challenge to serve the search requests that come to the logical trees for which compaction is currently ongoing because some of the SSTs may have been deleted. To correctly serve the search requests, SifrDB redirects the requests to the new SSTs by exploiting the abuilding global index introduced above, referred to as *redirection map*. Each time a compaction journal is committed, the *redirection map* is updated to cover the newly produced SSTs. The search for a logical tree first checks the *redirection map*, to determine

TABLE 3
Key Source Files of LevelDB Touched to Implement SifrDB

Source file	Functionality
version_edit.h	Define the structure of the logical tree
version_edit.cc	Encode and decode the logical tree
version_set.h	Define the structures about SST-level filter and parallel search
version_set.cc	Implement SST-level filter and parallel search
db_impl.cc	Implement the compaction and early-cleaning
merge.cc	Detect whether two SSTs are overlapping

if the query key falls into one of the new SSTs. If yes, the request is redirected to access that new SST. Otherwise, the search process will access the original SST that possibly contain the query key as in the usual way.

In a case that after the search process has checked the *redirection map* and decides to access the original SST, a problem could arise if the *redirection map* is updated promptly and then the early-cleaning is scheduled to delete that SST.

To prevent such a problem from happening, we design a "twice check" mechanism. After the search process first checks the *redirection map* and decides to access the original SST, it tags the SST and then checks the *redirection map* second (the early-cleaning will not delete a tagged SST until it is un-tagged). After the second check, if the search process is instead redirected by the promptly updated map to access the new SST, the original SST can be un-tagged and deleted without trouble. Otherwise, since the search process still decides to access the original SST according to the *redirection map*, it is guaranteed that the key range of the SST is not in the map and the SST is not in the cleaning list before the second check, thus the search process can safely work on the SST and un-tag it after the search finishes.

4.4 Implementation

SifrDB is implemented based on LevelDB but replaces the core data structures and functions with the SifrDB design. Any applications that use a store compatible with LevelDB can replace the existing storage engine with SifrDB seamlessly, as the exported operation interfaces are not changed. The key source files that are touched for the implementation of SifrDB are listed in Table 3.

5 Evaluation

In this section we present the evaluation results of SifrDB, with comparisons to a broad range of multi-stage based KV stores, including popular MS-tree implementations LevelDB and RocksDB, and representative MS-forest implementations Size-Tiered (used in Cassandra) and PebblesDB (the latest research based on the partitioned MS-forest).

5.1 Experiment Setup

The evaluation experiments are conducted on a Linux 4.4 machine equipped with two Intel E5 14-core CPUs and 128 GB DDR4 memory. The storage subsystem used in the experiments, Intel SSD DC S3520 Series, has a capacity of 480 GB with a 400 MB/s sequential read and a 350 MB/s sequential write speed in raw performance, and 41K IOPS for read.

All the microbenchmark and YCSB workloads are replayed by the `db_bench` toolset [26], [27]. Since Cassandra does not support the `db_bench` and running it in the normal mode involves network latency, we re-implement the Size-Tiered by reusing the LevelDB code to provide a fair comparison. We still run Cassandra for latency irrelevant metrics such as write amplification and space requirement, and verified that the results are consistent with our re-implementation. The dataset is 118 GB in the experiments, and the available memory is varied in the read experiments to simulate different memory provisions for the same dataset, as a large storage system can be configured to have very high storage/memory ratios [6], [16].

Note that our test program uses a single thread to send requests to KV store without batching, therefore, the average latency of the requests can be nearly inversely-proportional to the actual throughput. In order to evaluate the actual read performance between of different KV stores, the SST-level filter mechanism is disabled except for the experiment in Section 5.3.2 and the second experiment and Section 5.4.

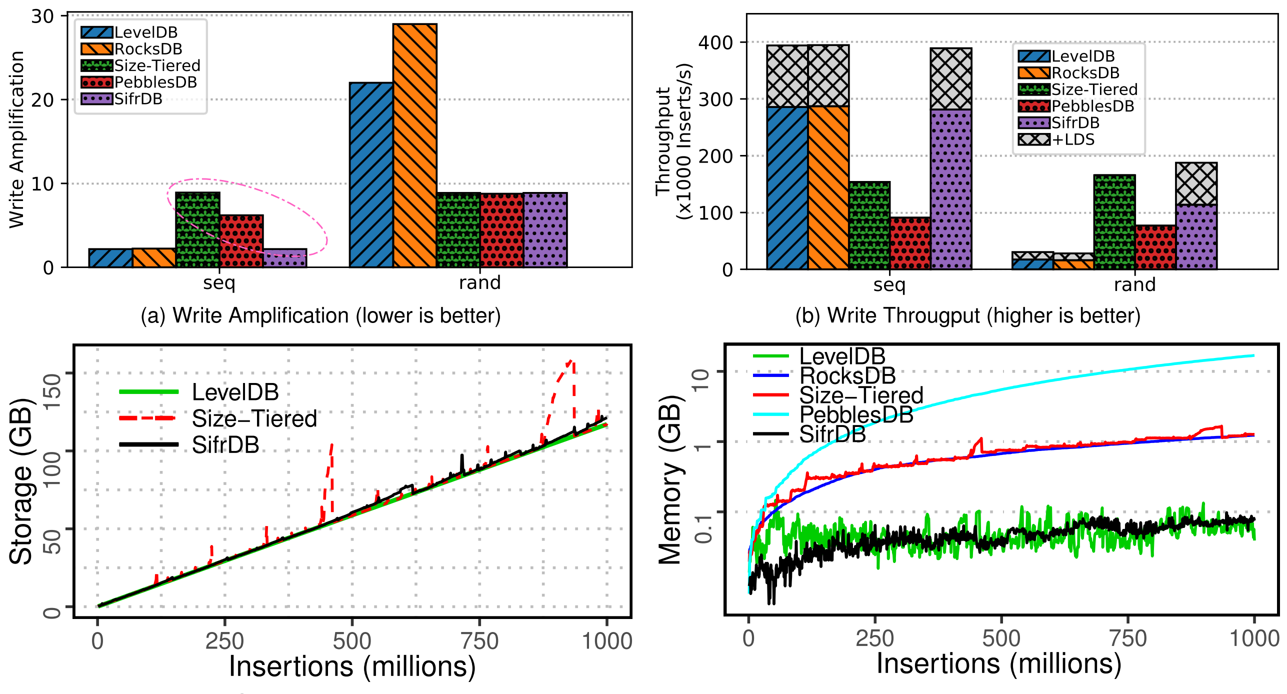
5.2 Write Performance

In this sub-section we evaluate the write performance by inserting 1 billion KV pairs to an empty store, with an average KV size of 123 bytes (23 bytes key, and remaining portion containing a number of bytes uniformly distributed in the 1$\sim$200), leading to a 118 GB dataset being built at the end. The write buffer size is set to the default value of LevelDB for all stores. It should be noted that, while using a larger write buffer can lower the write amplification to some extent, this effect is uniform to all stores and does not alter the overall performance trend. The growth factor has different impacts in the MS-tree model and the MS-forest model as analyzed in Section 3, so it is not set to the same value for implementations based on different models. In the experiments, we set the growth factor to 10 for the MS-tree-based stores (an optimized value in the practical MS-tree implementations), and to 4 for the MS-forest-based stores[4] (an optimized value in the practical MS-forest implementations).

Fig. 9a shows the write amplification of different stores under random and sequential writes respectively. A more extensive set of results can be seen in Fig. 2a.

First, for the sequential workload, SifrDB induces no write amplification, just like LevelDB and RocksDB do, which is in sharp contrast to the other two MS-forest implementations, Size-Tiered and PebblesDB, that causes 8$\times$ and 6$\times$ write amplification respectively. This is useful in some cloud environments such as sensor-collected data [20]. Second, for the random workload, SifrDB, having inherited the main advantage of the MS-forest model in random writes, exhibits that same level of write amplification as the other two MS-forest implementations Size-Tiered and PebblesDB and substantially better than the MS-tree implementations.

4. The growth factor in PebblesDB is the number of SSTs in a guard that triggers the compaction.



(a) Write Amplification (lower is better) (b) Write Througput (higher is better)

(c) Storage Requirement (d) Memory Requirement

Fig. 9. Figure (a) shows the overall write amplification of inserting one billion KV pairs (the circled are MS-forest based implementations), and Figure (b) shows the overall throughput. Figure (c) and Figure (d) show the actual storage requirement and memory requirement respectively in the process of inserting (*In Figure (c) we omit the results of RocksDB and PebblesDB for clarity as their lines are overlapped in large with LevelDB and SifrDB. System failure would happen if the storage or memory provision cannot meet the requirement*).

Intuitively, the write throughput of a system is inversely proportional to its write amplification. However, there are two other factors that can lower the write throughput, which causes the write throughput less proportionally tied to write amplification, as indicated in Fig. 9b. One is the overhead on the write-ahead log, which is prominent in a sequential write pattern [22]. The other is the file-system overhead, which is more pronounced for small files. Fig. 10 illustrates a snapshot of the file size distribution of the three MS-forest implementations under random writes. Clearly, Size-Tiered writes extra-large files to the underlying storage, which is file-system friendly and leads to a higher throughput than SifrDB and PebblesDB despite of the same write amplification they induce. Nonetheless, with the unified file size, SifrDB can take advantage of the aligned write to eliminate the file-system overhead, a technique proposed in LDS [22] (LDS is an LSM-tree Direct Storage system that manages the storage space based on the LSM-tree objects and provides simplified consistency control by leveraging the copy-on-write nature of the LSM-tree structure. It reduces extra IOs caused by filesystem, but only works on fix-sized SSTs). With the aligned storing, SifrDB achieves the highest throughput for the random workload, as shown by the SifrDB+LDS result in Fig. 9b. As LDS naturally support LevelDB, we also show the LevelDB+LDS result (RocksDB is similar to LevelDB). We can see that under the random workload, the write performance of LevelDB+LDS is still much lower than that of the forest implementations, even though LDS improves the performance a great deal. Note that PebblesDB is not able to take advantage of the technique LDS provides because of its variable and unpredictable file size.

5.3 Read Performance

Fig. 9c shows the storage requirement of SifrDB, Size-Tiered and LevelDB. With the early-cleaning mechanism, SifrDB resolves the problem the Size-Tiered is facing and achieves the same space efficiency as LevelDB. Note that the implementations based on the partitioned MS-forest such as PebblesDB also does not suffer from the high space requirement problem. Additionally, we measure the memory requirement in the process of writing (inserting) and present the results in Fig. 9d. RocksDB and Size-Tiered consume much more memory than SifrDB and LevelDB, by an order of magnitude, while PebblesDB consumes two orders of magnitude more memory than SifrDB. In fact, we set the `top_level_bits` to 31 and `bit_decrement` to 2 in PebblesDB, a setting designed to optimize writes; otherwise PebblesDB will fail to complete the 1-Billion insertions in the default setting (process killed by the system for exhausting all the system memory as well as the swapping space).

5.3.1 *Point Query*

In this subsection, we evaluate the read performance, i.e., point-query of random keys. The dataset used is the one generated in the write performance evaluation under random workload. The numbers of the candidate trees are listed in Table 4, which are obtained after the write process is finished. As MS-forest implementations are required to search more trees for a query than MS-tree ones, the former generally have longer query latency than the latter. However, the actual result varies depending on the specific stores and available memory provisions.

We disable the seek-triggered compaction and conduct the experiments on each store with sufficiently long time to

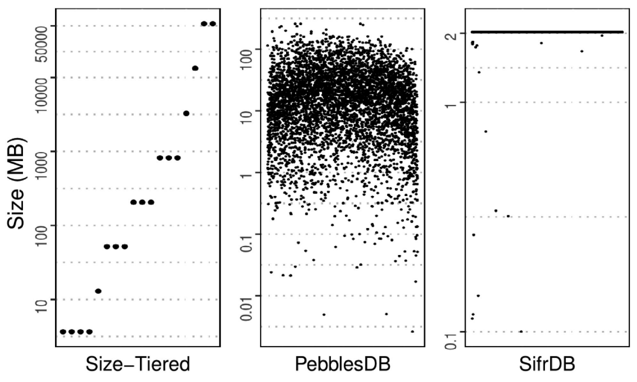


Fig. 10. A snapshot of the physical files' size of the three MS-forest implementations under random writes.

make sure that the performance has become stable. To fairly compare the performance of SifrDB with the other systems with the same memory usage, we also disable the SST-level bloom filter but employ *FC* parallel scheme with 16 threads.

The evaluation results are presented in Fig. 11a in the metric of query latency, along with read amplification presented in Fig. 11b. From the results, we can draw two conclusions: (1) SifrDB is more efficient under configurations with higher dataset/memory ratios under both 256 MB and 1 GB memory provisions; (2) SifrDB consistently performs the best among the three MS-forest implementations, outperforming Size-Tiered and PebblesDB significantly. For example, when the provisioned memory is 256 MB, as a configuration of 400:1 dataset/memory ratio [16], SifrDB achieves better performance than Size-Tiered by 4×, and PebblesDB by 5× respectively. At a higher dataset/memory ratio, because the memory consumption for indexes and data are limited, the intensive requests induce cache replacement that causes read IO. But the parallel search of SifrDB can fully utilize I/O ability of the underlying storage. RocksDB in the default configuration performs poorly under low memory provision due to frequently launching swapping I/Os. With sufficient memory provision, the effect of parallel search of SifrDB is weakened because a significant portion of the search operations are serviced by the cache without I/Os. However, SifrDB still consistently outperforms other MS-forest implementations.

With the parallel-search mechanism, SifrDB achieves comparable performance to LevelDB even though it needs to search 2.4× more trees than the latter. It should be noted the bandwidth of the underlying media could limit the query throughput. Nonetheless, we find that the parallel-

TABLE 4
Number of Candidate Trees in Different Stores *(in PebblesDB the Number of Trees are Various in Different Guards)*

	LevelDB	RocksDB	Size-Tiered	PebblesDB	SifrDB
Trees	7	9	17	11∼19	17

search algorithm consistently improves the read throughput when the candidate trees are the same, and is able to fully exploit the access parallelism of SSDs to provide speedy responses to requests. This is particularly suitable for cases when either request arrivals are sparse or serving high-priority and time-critical requests, in which the requests are served expeditiously as the bandwidth potential of SSDs can be utilized to the fullest.

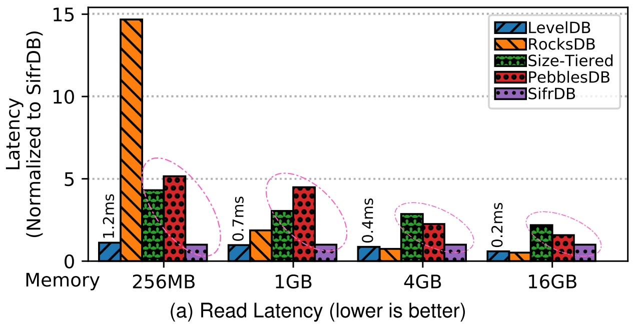
Since the MS-forest implementations need to search more trees than the MS-tree implementations, the former incur higher read amplification than the latter in general. PebblesDB incurs extremely high read amplification in low memory provisions, which cools off when the provision is larger than 1 GB. Comparing read amplification of SifrDB to that of Size-Tiered, we can see that 15 percent more unnecessary I/Os are incurred by the former (with 16 GB memory), which is caused by fact that some search threads of the parallel-search in SifrDB may access the low-priority trees for a small portion of keys that exist in a high-priority tree. Such unnecessary accesses are expected from design principle of SifrDB and does not impact the effectiveness of the parallel-search algorithm.

5.3.2 SST-Level Bloom Filter

When there is enough memory available, the SST-level filter on top of the stores can be used in the memory to improve the read performance. We have performed experiments using the SST-Level filter with different bits/key. The results in Fig. 12 show that in the default configuration (10 bits/key) the read performance improved by 3.7× with 106 MB memory (10 GB dataset). The benefit is contributed to that only one candidate tree needs to be actually searched using SST-level filter. In such cases, SifrDB is unnecessary to trigger the parallel search mechanism. Besides, the higher bits/key merely slightly improves the latency but has a larger memory footprint.

5.3.3 Parallel Search Schemes

With limited memory provision, even if SST-level bloom filter has to be turned off, SifrDB still effectively grasps the high



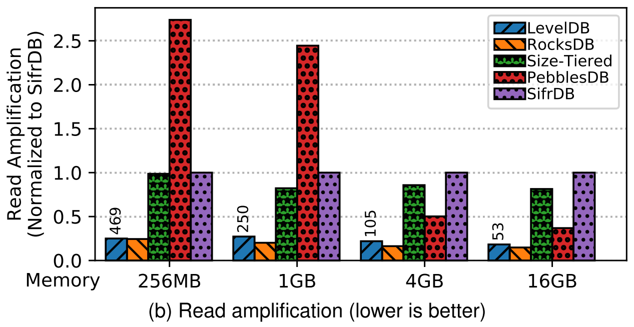


Fig. 11. Figure (a) shows the query latency of different stores as a function of memory provisions (MS-forest based stores are circled, i.e., the rightmost three bars of each bar group), and Figure (b) shows the read amplification (read_IO_size/queried_data_size).

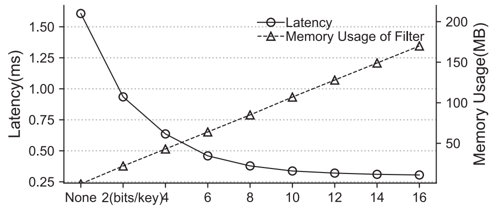


Fig. 12. Latency and memory usage of SST-level filter in different bits/key configuration under 10 GB dataset.

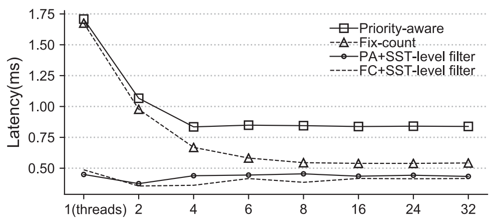


Fig. 14. Read performance of *FC* and *PA* scheme with different number of search threads.

IOPS nature of modern SSDs by parallel search. To compare the differences between *FC* and *PA* schemes, we evaluated the average latency when the query key is in 6 different stages (as illustrated in Fig. 13). The results showed that *FC* exhibits a stable latency regardless of which stage the target key resides in, while in *PA* scheme the latency increases significantly as the target stage increases. However, *PA* scheme exhibits better when the key is at the lower stage. It means hot data could benefit from *PA* scheme.

To clearly understand the combined effect of the parallel search and SST-level filter, we enable SST-level filter in the following experiment.

As shown in Fig. 14, compared to the traditional serial read, *FC* scheme makes full use of the parallelism of SSD devices and achieves 2.1× read performance improvement while in *PA* scheme it achieves only 1.0× read performance improvement because of the latency caused by multiple-turn thread executions. It also verifies that the default number (i.e., 16) of search threads is the appropriate value in *FC* scheme. Besides, using SST-level filter, the performance difference of these schemes is very small. It means the SST-level filter is more effective for the performance than the parallel search.

5.3.4 Range Query

Range query in the common multi-stage structures is processed by (1) seeking all candidate trees one by one to find the start key of the query; (2) comparing the keys and advancing the search across all the candidate trees step by step until the given number of keys are obtained. The MS-forest model needs to seek more trees than the MS-tree model for the start key, in addition to performing more comparing operations in each step. As a result, the forest-based implementations have worse performance than the tree-based implementations in general. In our experiments, the forest-based implementations (Size-Tiered, PebblesDB) exhibit about half the performance of the tree-based implementations. Because SifrDB leverages the parallel-search algorithm to speed up the tree-seeking process, it is able to improve the range query performance by 42 percent over the other two forest-based implementations.

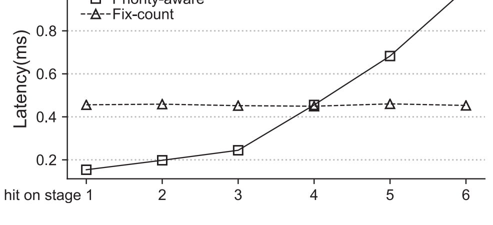


Fig. 13. Average Latency of *PA* and *FC* scheme when the query key is found on different stage.

Another approach to improving the read performance (both point query and range query) is to enable seek-triggered compaction. However, that is only efficient for read-intensive workloads. While multi-stage structures are often used to ingest the massive user data in write-intensive environments, seek-triggered compaction has limited usage.

5.4 Synthetic Workloads

YCSB [28] provides a common set of workloads [29] for evaluating the performance of cloud systems. For a workload, 4 threads run concurrently and each of them sends 10 K requests, with the overall throughput being measured as the performance metric. The memory provision is sufficiently high to ensure that the hot accessed keys (in the Zipfian and Latest distribution) are cached. In range queries of the workload E, the scanned number of keys is uniformly distributed between 1 and 100.

The results of the YCSB benchmarks are shown in Fig. 15. While most of the workloads are read-dominated, when disabling SST-level filters, SifrDB exhibits a lower throughput than MS-tree implementations (RockDB and LevelDB), but consistently outperforms the other two MS-forest implementations (Size-Tiered and PebbleDB). Specifically, workload E helps demonstrate the range query performance. For each range query, SifrDB simply executes a point-query to boost the following seek operations, which proves to be quite efficient. Note that PebblesDB also implements a different parallel-seek algorithm specially for the range query, which is shown to be less efficient than SifrDB's.

To evaluate the effectiveness of SST-level filter and parallel search mechanism under the workload, we execute the YCSB workloads with enabling/disabling the SST-level filter respectively. As illustrated in Figs. 15 and 16, the SST-level filter boosts the overall performance consistently and dramatically compared to disable it. Without SST-level filter, *FC* scheme still performs better on most loads. However, *PA* scheme performs better performance under D workload in which 95 percent operations are reading latest values that are stored in low stages with high-priority. As a result, the SST-level filter can effectively improve the read performance to make up for the read defects of MS-forest. Enabling SST-level filter, SifrDB can achieve better the overall performance than MS-tree because of its superior write performance.

6 Related Work

Write optimized structures have become popular in large-scale data stores [30], [31], [32], [33], [34], [35], [36], [37]. In general, there are two families of write optimized structures. One is the fractal-trees [38], [39], [40] that buffer the

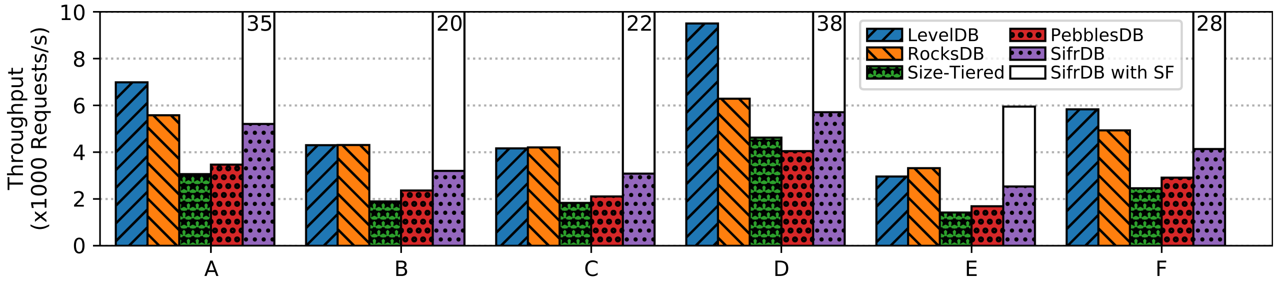


Fig. 15. Throughput under YCSB workloads (higher is better). SifrDB uses FC parallel scheme.To ensure fairness, SST-level filter in SifrDB is disabled for the comparison of the five implementations(two MS-tree and three MS-forest). The results of SifrDB with SST-level filter is also shown for reference.

data in each intermediate node of a B-tree. The other is the multi-stage structures that maintains a set a B-trees in different stages, which are widely used in key-value stores [1], [2], [3], [4], [5].

In this paper we have analyzed and classified the practical implementation models for multi-stage structures, and presented their advantages and disadvantages for different performance considerations.

In fact, a large body of existing studies have contributed to the popularity of multi-stage-based KV stores [6], [7], [19], [22], [41], [42] and most of them focus on the write amplification of the MS-tree model under random writes. The essential reason for the MS-tree's high write amplification is that data must be rewritten within each stage repeatedly in order to keep the data sorted, which is good for reads. On the contrary, the MS-forest model allows overlapped trees (i.e., forest) to avoid rewriting data within a stage. Researches that optimize the random-write problem faced by the MS-tree model often make use of this property of MS-forest and allow overlapped key ranges within a stage [6], [7], [43], hence turning into a variant of MS-forest and suffering from the disadvantages such as a less-compact structure with degraded read performance. VT-tree [19] proposes a stitching technique based on the MS-forest model to reduce the write amplification under sequential/sequential-intensive workloads by applying a secondary index, which introduces garbage on the storage and faces high space requirement for compaction. In addition, it does not distinguish the effect of the stitching technique from the effect of the MS-forest model, since it uses a tree-based implementation (LevelDB) as the baseline for evaluation.

Whether in MS-tree or in MS-forest, request response times can be significantly affected because of the long turnaround time of the compaction. SILK [25] has a thorough analysis of latency spikes caused by compaction in the MS-tree model and introduce the notion of an I/O scheduler to reduce this interference. For the MS-forest model, we used the *Search Request Redirection* to ensure the SST file read requests can be quickly redirected to new SST generated by compaction. In addition, when a compaction operation affects the overall performance of SifrDB, the compaction process can be suspended to prioritize requests processing. It is easy to continue the compaction process later by using *Compaction Journal*.

As the overheads introduced by the storage stacks becoming outstanding in high performance storage environments [21], [44], aligned storing is used to optimize the file-system overheads [22] for the implementations based on the split-forest. Many work also exploit the new storage medias to improve the performance of key-value stores based on the multi-stage structures. LOCS [45] propose to expose the internal channels of the SSDs and schedule requests on the channels to fully exploit the SSD bandwidth. Wisckey [46] and PebblesDB design specific algorithms to exploit the SSD parallelism for range queries.

Other related researches have in-depth studies on the evaluation and configuration of existing multi-stage-based KV stores to find a better performance trade-off [17], [41], [42]. In addition, practical systems have tried to provide customized interfaces to reduce write amplification for special use cases. For example, RocksDB provides the bulkloading scheme [47] to ingest the large data generated in offline or migrated from other data stores.

These works substantially advance the knowledge of multi-stage structures, and motivate the work in this paper, i.e., the taxonomy of current MS-structure based KV stores and the SifrDB design.

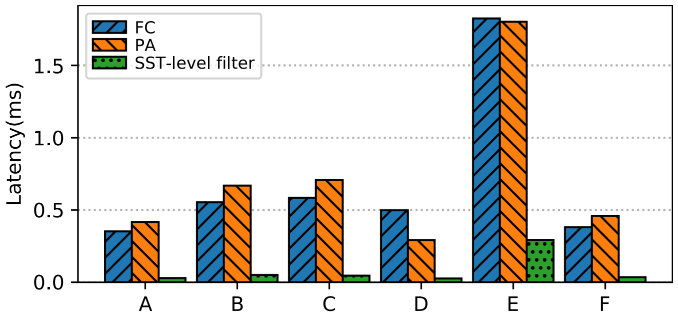


Fig. 16. Latency of SifrDB in different configurations under YCSB workloads (lower is better).

7 Discussion

As discussed in Section 3, MS-tree and MS-forest differ in read and write performance due to their inherent structures. MS-trees such as LevelDB and RocksDB are more suitable for workloads with high read ratio. Cassandra, PebblesDB and other MS-forests are preferable for write-dominant workloads.

SifrDB is based on MS-forest optimizing for write workloads, but uses multiple techniques to enhance read performance and reduce space occupancy. For example, when the memory is sufficient, SifrDB can enable SST-level filter at the cost of memory (about 1 percent of dataset) to gain a great improvement in reading performance, making its overall performance higher than other KV stores. When the memory is limited and the SST-level filter is disabled, by exploiting parallelism of underlying storage, SifrDB still can improve the read performance comparable to MS-tree implementation and also have the highest read performance in MS-forest implementations. Therefore, SifrDB can be a better uniform solution for both write and read workloads.

8 Conclusion

We identified two multi-stage structures, MS-tree and MS-forest, that have opposing trade-offs for important performance metrics. The SifrDB store we have proposed is based on and inherits the advantage of the MS-forest model, and avoids its disadvantages by imposing a split storing mechanism. Additionally, we designed SST-level filter that reduces read I/O and a parallel-search mechanism that fully exploits the SSD access parallelism to boost the read performance. Evaluation results show that SifrDB is exceedingly competitive in large data stores.

Acknowledgments

This work is supported in part by NSFC No. 61872156, Creative Research Group Project of NSFC No. 61821003, National key research and development program of China (Grant No. 2018YFA0701804), Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, the US National Science Foundation under Grant No. CCF-1704504 and No.CCF-1629625, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] LevelDB project home page, 2018. [Online]. Available: https://code.google.com/p/leveldb/
[2] RocksDB, Facebook, 2013. [Online]. Available: http://rocksdb.org/
[3] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[4] F. Chang et al., “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008, Art. no. 4.
[5] L. George, *HBase: The Definitive Guide*. Sebastopol, CA, USA: O’Reilly Media, Inc., 2011.
[6] X. Wu, Y. Xu, Z. Shao, and S. Jiang, “LSM-trie: An LSM-tree-based ultra-large key-value store for small data,” in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 71–82.
[7] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, “PebblesDB: Building key-value stores using fragmented log-structured merge trees,” in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 497–514.
[8] J.-S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng, “ForestDB: A fast key-value storage system for variable-length string keys,” *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 902–915, Mar. 2016.
[9] 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.
[10] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, “SFS: Random write considered harmful in solid state drives,” in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 12.
[11] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, “Write amplification analysis in flash-based solid state drives,” in *Proc. SYSTOR: Israeli Exp. Syst. Conf.*, 2009, Art. no. 10.
[12] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim, “Assert (! defined (sequential I/O)),” in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst.*, 2014, pp. 10–10.
[13] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, “Incremental organization for data recording and warehousing,” in *Proc. Int. Conf. Very Large Databases*, 1997, pp. 16–25.
[14] Configuring compaction, 2018. [Online]. Available: https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_configure_compaction_t.html, DataStax
[15] Introduction to compaction, 2011. [Online]. Available: https://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra
[16] From big data to big intelligence, 2017. [Online]. Available: https://www.purestorage.com/products/flashblade.html, Pure Storage
[17] A. Eisenman, et al., “Reducing DRAM footprint with NVM in Facebook,” in *Proc. 13th EuroSys Conf.*, 2018, pp. 42:1–42:13.
[18] P. Oneil, E. Cheng, D. Gawlick, and E. Oneil, “The log-structured merge-tree (LSM-tree),” *Acta Informatica*, vol. 33, pp. 351–385, 1996.
[19] P. J. Shetty, R. P. Spillane, R. R. Malpani, B. Andrews, J. Seyster, and E. Zadok, “Building workload-independent storage with VT-trees,” in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 17–30.
[20] S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer, “LittleTable: A time-series database and its uses,” in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 125–138.
[21] J. Mohan, R. Kadekodi, and V. Chidambaram, “Analyzing IO Amplification in Linux file systems,” *CoRR*, vol. abs/1707.08514, 2017. [Online]. Available: http://arxiv.org/abs/1707.08514
[22] F. Mei, Q. Cao, H. Jiang, and L. Tian, “LSM-tree managed storage for large-scale key-value store,” *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 2, pp. 400–414, Feb. 2019.
[23] R. Bayer and E. McCreight, “Organization and maintenance of large ordered indexes,” in *Software Pioneers*. Berlin, Germany: Springer, 2002, pp. 245–262.
[24] Improved memory and disk space management, 2011. [Online]. Available: https://www.datastax.com/dev/blog/whats-new-in-cassandra-1-0-improved-memory-and-disk-space-management
[25] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, “SILK: Preventing latency spikes in log-structured merge key-value stores,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 753–766.
[26] LevelDB benchmark, 2018. [Online]. Available: https://github.com/google/leveldb/blob/master/db/db_bench.cc, LevelDB
[27] RocksDB benchmark tools, 2018. [Online]. Available: https://github.com/facebook/rocksdb/blob/master/tools/db_bench_tool.cc, RocksDB
[28] 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.*, 2010, pp. 143–154.
[29] Core workloads, 2010. [Online]. Available: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads/
[30] B. F. Cooper, et al.“PNUTS: Yahoo!’s Hosted data serving platform,” *Proc. Very Large Data Bases Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.
[31] S. G. Edward and N. Sabharwal, “MongoDB explained,” in *Practical MongoDB*. Berlin, Germany: Springer, 2015, pp. 159–190.
[32] SQLite4: LSM Design Overview, 2016. [Online]. Available: https://www.sqlite.org/src4/doc/trunk/www/lsm.wiki, SQLite
[33] The modern engine for metrics and events, 2017. [Online]. Available: https://www.influxdata.com/, InfluxData, Inc.
[34] W. Jannen, et al., “BetrFS: A right-optimized write-optimized file system,” in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 301–315.
[35] J. Yuan, et al., “Optimizing every operation in a write-optimized file system,” in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 1–14.
[36] K. Ren and G. Gibson, “TABLEFS: Enhancing metadata efficiency in the local file system,” in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 145–156.
[37] D. Bartholomew, *MariaDB Cookbook*. Birmingham, U.K.: Packt Publishing Ltd, 2014.
[38] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook, “On external memory graph traversal,” in *Proc. ACM-SIAM Symp. Discrete Algorithms*, 2000, pp. 859–860.
[39] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, “Cache-oblivious streaming B-trees,” in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit.*, 2007, pp. 81–92.
[40] G. S. Brodal and R. Fagerberg, “Lower bounds for external memory dictionaries,” in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2003, pp. 546–554.

[41] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 79–94.

[42] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 149–166.

[43] Compaction subproperties, DataStax, Inc., 2018. [Online]. Available: http://docs.datastax.com/en/cql/3.1/cql/cql_reference/compactSubprop.html

[44] A. Papagiannis, G. Saloustros, M. Marazakis, and A. Bilas, "Iris: An optimized I/O stack for low-latency storage devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 50, no. 1, pp. 3–11, 2017.

[45] P. Wang, et al., "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 16.

[46] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 133–148.

[47] Bulkloading by ingesting external SST files, 2017. [Online]. Available: http://rocksdb.org/blog/2017/02/17/bulkoad-ingest-sst-file.html

Ziyi Lu received the BS degree in computer science from the Huazhong University of Science and Technology, Wuhan, China in 2018. He is working toward the PhD degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His major research interest includes key-value store on new media.

Qiang Cao received the BS degree in applied physics from Nanjing University, Nanjing, China in 1997, the MS degree in computer technology, and the PhD degree in computer architecture from the Huazhong University of Science and Technology, Wuhan, China in 2000 and 2003, respectively. He is currently a full professor of the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology. His research interests include computer architecture, large scale storage systems, and performance evaluation. He is a senior member of the China Computer Federation (CCF), a senior member of the IEEE, and a member of the ACM.

Fei Mei received the BS degree in Software Engineering from Wuhan University in 2010, and the PhD degree in Computer Science from Huazhong University of Science and Technology in 2019. In his career of PhD student from 2014 to 2019, he focused the research work on key-value stores and file systems. He also has interest in distributed storage and intelligent storage. Currently he works in Huawei Technologies as a senior engineer.

Hong Jiangl received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. He is currently chair and Wendell H. Nedderman endowed professor of Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director with National Science Foundation (2013.1-2015.8) and he was with the University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of Computer Science and Engineering. He has graduated 16 PhD students who upon their graduations either landed academic tenuretrack positions in PhD-granting US institutions or were employed by major US IT corporations. His present research interests include computer architecture, computer storage systems and parallel I/O, highperformance computing, big data computing, cloud computing, performance evaluation. He recently served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 300 publications in major Journals and international Conferences in these areas, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of IEEE*, *ACM Transactions on Architecture and Code Optimization*, *ACM Transactions on Storage*, the *Journal of Parallel and Distributed Computing*, ISCA, MICRO, USENIX ATC, FAST, EUROSYS, LISA, SIGMETRICS, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by NSF, DOD, the State of Texas and the State of Nebraska, and industry. He is a fellow of the IEEE, and a member of the ACM.

Jingjun Li received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2016 and 2019, respectively. His research interests include key-value store and computer architecture.