SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter

Fei Mei
Huazhong University of Science and Technology
Wuhan, China
meifei@hust.edu.cn

Qiang Cao*
Huazhong University of Science and Technology
Wuhan, China
caoqiang@hust.edu.cn

Hong Jiang
University of Texas at Arlington
Arlington, USA
hong.jiang@uta.edu

Jingjun Li
Huazhong University of Science and Technology
Wuhan, China
jingjunli@hust.edu.cn

ABSTRACT

Key-value (KV) stores based on multi-stage structures are widely deployed in the cloud to ingest 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 models of data structures facilitating KV operations, namely, the multi-stage tree (MS-tree) as represented by LevelDB, and the multi-stage forest (MS-forest) as typified by the size-tiered compaction in Cassandra. We then build a KV store on a novel split MS-forest structure, called SifrDB, that achieves the lowest write amplification across all workload patterns and minimizes space reservation for the compaction. In addition, we design a highly efficient parallel search algorithm 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 large-dataset-driven KV stores.

CCS CONCEPTS

• **Information systems** → *Record and block layout*; *Query reformulation*; Physical data models; Point lookups;

*Corresponding Author.

KEYWORDS

Key-Value, Multi-Stage, LSM-tree, Parallel Search

ACM Reference Format:
Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *SoCC '18: ACM Symposium on Cloud Computing (SoCC '18), October 11–13, 2018, Carlsbad, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3267809.3267829

1 INTRODUCTION

Multi-stage structures maintain a set of B-trees in several stages with increasing capacities, which are efficient for block devices including both HDDs and SDDs [26, 30, 34] by aggregating small random writes in large writes. Various key-value (KV) stores are implemented based on multi-stage structures, such as LevelDB [12], RocksDB [4], Cassandra [29], BigTable [19], HBase [25], LSM-trie[43], PebblesDB [38] etc. However, our in-depth empirical study reveals that these implementations 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 sorted tree in each stage, as represented by LevelDB, and the multi-stage forest (MS-forest) structure that allows multiple trees 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 for compaction, while LevelDB is more efficient for reads and space requirement.

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

[1]The name Sifr comes from letters in the words '**S**pl**i**t' and '**f**o**r**est'.

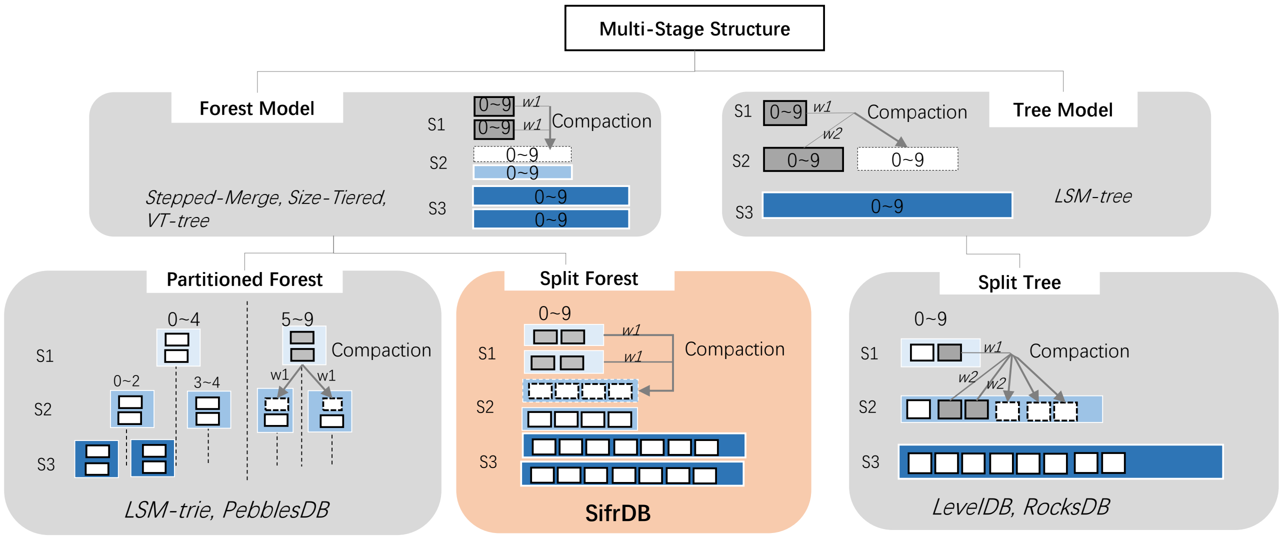


Figure 1: A taxonomy of the popular multi-stage implementations under the tree or forest model. While rewriting in the MS-forest implementations only takes place across stages, it happens both across stages and within each stage in the MS-tree implementations (*w1* indicates data is written across the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

efficiency. Specifically, in its top layer of the multi-stage structure SifrDB performs compaction by a method similar to that used in the stepped-merge [27] or the Size-Tiered [10] implementations that are popular in modern large-scale KV stores [19, 25, 29], to leverage the advantages of MS-forest for random writes. For the layers below, SifrDB splits each tree to fix-sized sub-trees that are stored independently, referred to as the *split storing* in this paper, so as to easily and efficiently detect key-range overlapping to enable sequential write optimization. Meanwhile, based on the split storing of the trees, an early-cleaning technique is proposed to ensure that the space requirement for compaction is kept at a minimal level, which solves a serious problem suffered by the Size-Tiered [3]. 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 to harness the advantages of the MS-forest while avoiding its disadvantages. Moreover, we design a novel parallel-search algorithm for SifrDB to fully exploit the access parallelism of SSDs.

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 [7, 24], SifrDB exhibits the best read performance among all the implementations.

2 TREE/FOREST CLASSIFICATION

Before presenting the background and motivation of this paper, it is necessary to introduce the 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. We illustrate in Figure 1 a taxonomy of popular multi-stage implementation models under the tree/forest classification. The defining principles for this classification are *(1) whether overlapped trees are allowed within each stage* and *(2) how compaction is performed.*

For the tree model, in each stage only one sorted 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 tree model is re-written not only across the stages (*w1* in Figure 1), but also within a stage (*w2* in Figure 1). For example, in the tree model demonstrated in Figure 1, when S_1 is full, the compaction process merge-sorts the tree in S_1 and the tree in S_2 to produced 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. On the other hand, for the forest model, multiple trees are allowed in each stage (in Figure 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 forest model is re-written only across the stages (*w1* in Figure 1).

The fundamental virtue of the forest model is that it incurs much lower write amplification than the tree model. However, the latter is more efficient for reads, as will be detailed in the next section (§3) together with the popular multi-stage tree/forest variants (i.e., the split tree and the partitioned forest). We then introduce SifrDB in §4, which is based a new forest variant (i.e., the slit forest).

The 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, while the VT-tree [41] builds on top of a forest model and uses a stitching technique to reduce write amplification, we can 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.

3 BACKGROUND AND MOTIVATION

In this section we analyze the key properties of the multi-stage structures to understand their intrinsic advantages and disadvantages that are revealed in the tree/forest classification above, followed by a review and analysis of the pertinent implementation features of the most representative multi-stage based KV stores to motivate and help simplify the SifrDB design.

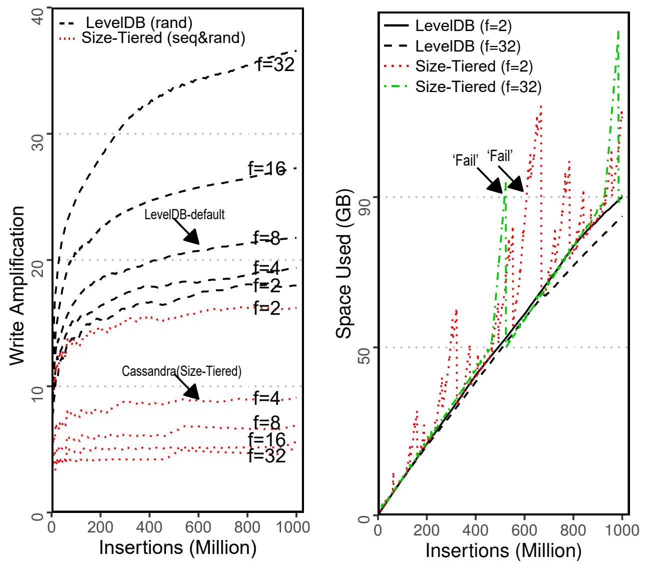
3.1 Write Amplification

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 operation) 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.

The Tree Model. The MS-tree model, originally introduced as the log-structured merge-tree [36], maintains multiple stages of B-trees with increasing capacities. 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 times in each stage by the MS-tree based application, causing significant write amplification.

The Forest Model. The MS-forest model, which has its roots in the stepped-merge approach [27] 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 $\frac{f}{2} \cdot \mathbf{log_f D}$. The partial merge mechanism used in LevelDB (i.e., only selecting a sub-tree 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 sub-trees, for each sub-tree of S_i there will be k overlapped sub-trees in S_{i+1}, and merging a sub-tree causes a re-written ratio of $k+1$, the same as the full merge.



(a) Write Amplification (b) Operational Space Requirement

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

Our experiment result in Figure 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 Space Requirement for Compaction

Unlike the traditional B-trees that usually operate on a single block (e.g., 4KB), the compaction processes in multi-stage structures often operate on several large trees that are stored as Sorted String Table (SST) files, a standard approach in the modern multi-stage-based KV stores. SST is a B-tree-like storing structure introduced by BigTable [19], which is a

[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 x th ($1 \leqslant x \leqslant f$) component is written $f-x$ times in the subsequent merge of the remaining components until S_{i+1} is full.

simple but efficient mechanism for block storage devices. An SST file is immutable and usually consists of two parts: the body composed of sorted KV strings and the tail containing the index data. The index data is built on top of the sorted key-value strings when an SST is produced.

A compaction operates on a set of SSTs by merge-sorting their key-values to new SST file/files. The operated SSTs can not be deleted before the compaction is finished [2, 12]. 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% space efficiency [3]. 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 [3]. Figure 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, which is detailed in §3.4.

3.3 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 topmost stage is moved to lower 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 [15]. For example, searching a 2MB tree needs 3 random IOs, 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 §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, the total number of candidate trees in the MS-forest usually shows 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 Figure 3 that plots the experimental read latency of LevelDB and Size-Tiered in the datasets generated previously with different growth factor configurations.

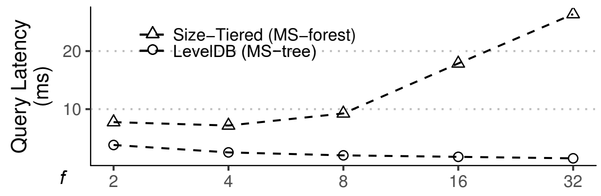


Figure 3: Read latency in the datasets generated previously.

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 require 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.

Bloom filters have been used in the multi-stage structure to improve performance of point queries by consuming a chunk of memory space, a strategy of trading space for time. Even though the Bloom filter is designed for memory efficiency, it consumes significant memory space in a large data store. For example, in the workloads with 100-byte KV pairs, the Bloom filter requires a memory space more than 1% of the dataset in a general setting (i.e., 10 bits for each key). 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 [7, 24] and cause frequent I/Os for swapping. Considering that the KV pairs in real workloads tend to be even smaller than 100 bytes [43] and Facebook has begun to reduce memory provision for its cloud store [24] for economic reasons, third-party indexes such as Bloom filter can only be used to limited extent.

3.4 State of the Art

Split Tree. LevelDB [12] is an MS-tree based implementation that employs a split approach 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. For the sake of consistency, we call the split stored SST files as *sub-trees* that, when combined with the aforementioned global index, form a *logical tree*. A logical tree is still referred to as a tree in the rest of the paper unless specially noted otherwise.

Although LevelDB induces much higher write amplification than MS-forest-based implementations such as Size-Tiered, it has two salient advantages over the latter by adopting the partial merge.

(1) After selecting a victim sub-tree in a stage for compaction, LevelDB first determines whether there are overlapped sub-trees in the next stage. If not, the victim sub-tree 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 [40] collected by a sensor.

(2) 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 sub-trees size of 2MB and a growth factor of 10 (default in LevelDB), about 11 sub-trees are involved in a compaction (one victim sub-tree and 10 estimated overlapped sub-trees). Therefore, an extra reserved space of about 22MB 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 Figure 2b.

Partitioned Forest. LSM-trie [43] is based on a variant of the forest model and is the first of its kind that partitions the trees in each stage to non-overlapped ranges, and compaction on a stage only merges the data within a victim range. As a result, space requirement in LSM-trie is not high as in the Size-Tiered. However, since LSM-trie uses hash to partition the keys, it loses the range query feature. Instead, PebblesDB [38] proposes to use the real keys as the partition boundaries. PebblesDB inherits the low write amplification property of the forest model. 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 [33, 35].

Summary — In this section we have analyzed the two models of MS-structures, the MS-tree and MS-forest models, and introduced their popular implementations. 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 (Figure 1) but attempts to remedy all its deficiencies.

4 SIFRDB

The compaction strategy of SifrDB is based on the MS-forest model which by design incurs much lower write amplification under random writes than the MS-tree model. However, 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 [3]. Our design goals are to provide 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 §4.1 and §4.2. To optimize query performance, we introduce our parallel-search algorithm in §4.3.

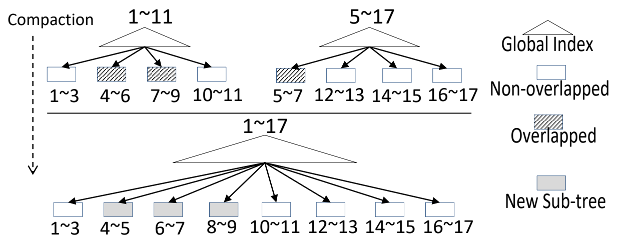


Figure 4: Compaction is performed on logical trees, while the merge is performed on the physical sub-trees.

4.1 Compaction Strategy

With the proposed split MS-forest storing, each tree in SifrDB is composed of a group of non-overlapped and fix-sized sub-trees whose metadata are recorded in a separate global index, which in essence constructs a logical tree. While a compaction is performed on several logical trees, the actual merge is performed at the granularity of sub-trees and only involves sub-trees with overlapped key ranges to eliminate the unnecessary data re-writing under sequential (or sequential-intensive) workloads.

More specifically, take Figure 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 sub-trees with overlapped key ranges are merged and re-written, while metadata of the non-overlapped sub-trees (unshaded) and the newly generated sub-trees (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).

Because SifrDB performs compaction based on the forest model (as illustrated in Figure 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 [29, 38, 43]. In addition, the design of SifrDB is also able to achieve the effect of the stitching technique introduced by VT-Tree [41] that builds a second index on top of each tree of an MS-forest model[3], seeing §6 for an

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

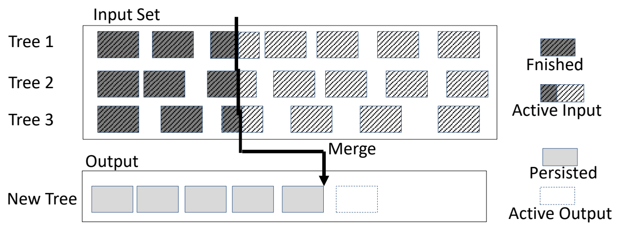


Figure 5: Merge Operation in SifrDB. Early-cleaning can be executed after a new sub-tree is persisted.

extensive discussion. 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 sub-trees. 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.

4.2 Optimize Space Efficiency

In this sub-section, we present the early-cleaning technique in SifrDB 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 'space full'.

The idea behind early-cleaning is to safely delete the sub-trees as soon as they have been successively merged to the new sub-trees, 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 sub-trees 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 sub-trees to reclaim the storage space. As shown in Figure 5, early-cleaning can be scheduled after a new sub-tree is sealed and persisted to delete the finished sub-trees. Since the data in the deleted sub-trees have been written to the new sub-trees, 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 sub-trees in the output and the active sub-trees in the input. In fact, the metadata of the persisted sub-trees are the abuilding and uncommitted global index of the new tree. Note that if an input sub-tree does not overlap with other sub-trees, it is directly moved to the output and is not affected by the cleaning process.

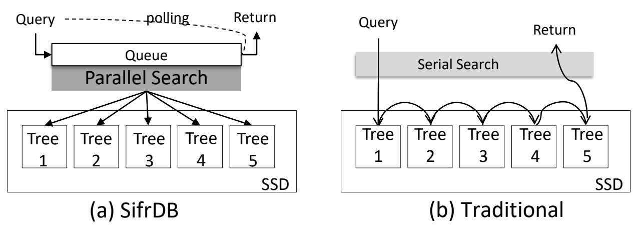


Figure 6: (a) The trees are searched concurrently. (b) The trees are searched serially in order of their priorities.

Although each time when a new sub-tree 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 sub-tree 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 sub-trees, 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 sub-tree. 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 sub-trees may have been deleted. To correctly serve the search requests, SifrDB redirects the requests to the new sub-trees 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 sub-trees. The search for a logical tree first checks the *redirection map*, to determine if the query key falls into one of the new sub-trees. If yes, the request is redirected to access that new sub-tree. Otherwise, the search process will access the original sub-tree 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 sub-tree, a problem could arise if the *redirection map* is updated promptly and then the early-cleaning is scheduled to delete that sub-tree. To prevent such a problem from happening, we design a "twice check" mechanism. After the search process firstly checks the *redirection map* and decides to access the original sub-tree, it tags the sub-tree and then checks the *redirection map* secondly (the early-cleaning will not delete a tagged sub-tree 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 sub-tree, the original sub-tree can be un-tagged and deleted without trouble. Otherwise, since

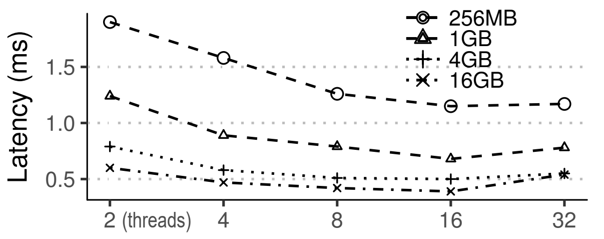


Figure 7: Query latency as a function of the number of the background search threads with different memory provisions (100GB dataset).

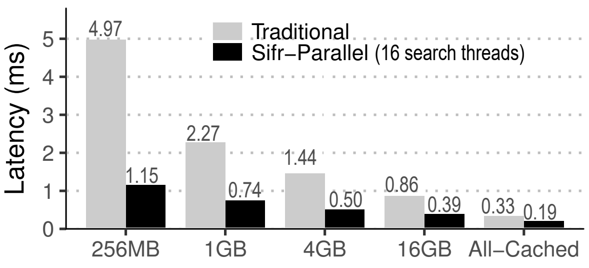


Figure 8: A comparison in query latency between the traditional approach and SifrDB's parallel-search under different memory provisions (100GB dataset).

	Average Core Time (ms)					95 percentile latency (ms)					99 percentile latency (ms)				
Mem	256MB	1GB	4GB	16GB	Cached	256MB	1GB	4GB	16GB	Cached	256MB	1GB	4GB	16GB	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

Table 1: CPU core time and 95/99 percentile latency for queries (*"Tra" is the traditional way, and "Sifr-16" is SifrBD with 16 search threads*).

the search process still decides to access the original sub-tree according to the *redirection map*, it is guaranteed that the key range of the sub-tree is not in the map and the sub-tree is not in the cleaning list before the second check, thus the search process can safely work on the sub-tree and un-tag it after the search finishes.

4.3 Parallel Search for Read

We design a parallel-search algorithm for SifrDB to mitigate the read performance degradation of the MS-forest model. Figure 6 illustrates the parallel-search scheme in SifrDB in contrast to the traditional approach. The search operation is detached from the server thread and is carried out in parallel by a bundle of background threads (referred to as *search threads*) on the candidate trees.

Algorithm In the traditional approach, the server thread serially searches the trees in order of their priorities. If the query key is found in a tree, there is no need to further search the trees with lower priorities. In SifrDB, when a query request arrives, the server thread constructs a query object that is appended to the left end (back) of the query queue and signals the search threads, after which the server thread simply polls on the right end (front) of the query queue. A query object contains three types of state variables to keep track of the search progress: (1) `COUNTer`, indicating how many candidate trees have been selected or searched; (2) `LocalR`, an array of state variables each indicating the search result of its corresponding tree ('Found' or 'NotFound'); (3) `GlobalR`, a signal of whether the final result of the query is determined ('Found' or 'NotFound'). A fingerprint generated by the server thread is also included in an object to help the server thread recognize its object in the polling[4]. The polling operation repeatedly checks the `GlobalR` and the fingerprint of the front object. Once the `GlobalR` is set (no matter 'Found' or 'NotFound') and the fingerprint matches, the server thread removes the object from the queue and returns the result according to the `GlobalR` value. We use polling (a spin lock is also optional) for the server thread because the query will be finished shortly in the near future, and polling can detect the result without context switches.

Each search thread usually performs three sequential steps in the query process (a search job):

i Selection. Lock the queue to select an unfinished object and a candidate tree based on the object's `COUNTer`, then unlock.
ii Search. Search the selected tree in parallel with other search threads.
iii Update. Lock the queue to update the `LocalR`, and if necessary, the `GlobalR`, then unlock.

In Step (i), the `COUNTer` of an object is increased by 1 after the selection and it indicates the number of trees that have been selected or searched for this object. An unfinished object is defined as that its `GlobalR` is not set and its `COUNTer` is smaller than the number of candidate trees. The selection always chooses a tree from the remaining ones with the highest priority to enable the GlobleR to be set as early as possible. Step (ii) is the costly portion that the parallel search attacks. When a search thread gets a result after step (ii) and comes to step (iii), it first sets the `LocalR`, and then decides whether the `GlobalR` should be set by predefined rules based on the `LocalR` values. For example, if a search thread finds that the current `LocalR` is 'Found' and all the `LocalR`s of trees with higher priorities have been set to 'NotFound', it will set the `GlobalR` to 'Found'; if any other thread enters Step (iii) and finds that the `GlobalR` has been set, it simply discards its search result and continues the next job.

Configuration The number of search threads can be configured to any positive integers (default to 16). Configuring 1 search thread reduces SifrDB to the traditional approach. We recommend that this number be configured based on the hardware resources, such as the number of CPU cores and SSD internal access parallelism. Figure 7 shows the query latency on a 100GB dataset as a function of the number of search threads in the environments introduced in §5, in which 16 is an optimized value.

In the parallel-search algorithm, if a query key is found in a high-priority tree quickly, SifrDB can detect and return the result immediately, even though some search threads working

[4]When multiple server threads run simultaneously, fingerprint is unique for each server thread.

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 structure of query object
version_set.cc	Implement the parallel search
db_impl.cc	Implement the compaction and early-cleaning
merge.cc	Detect whether two sub-trees are overlapping

Table 2: Key source files of LevelDB touched to implement SifrDB.

on the trees with lower priorities have not completed, in which case the latter simply discard their results afterwards. It should be noted that the efficiency of the parallel search is impacted by the cached data, since the benefit of our parallel-search algorithm stems from exploiting the internal access parallelism of SSD flash. The more candidate trees are cached in memory, the less time will be spent on the parallel-search portion (i.e., parallel search in memory instead of in SSD flash), thus weakening the benefit. Nevertheless, we find that the parallel algorithm still performs better than the traditional approach even though when all the dataset has been cached in memory. Figure 8 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 1.

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 §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 1. We can see that when the memory provision is larger than 4GB (equivalent to a 25:1 dataset/memory configuration), the CPU cost in SifrDB becomes higher than that in the traditional way. This suggests that one should carefully use the parallel search when the CPU is stressful. We plan to enable dynamically adjusting the background search threads to best adapt the hardware resources and satisfy the user requirement.

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.

4.4 Implementation

SifrDB is implemented by reusing the LevelDB outer code and replacing 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 2.

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 128GB DDR4 memory. The storage subsystem used in the experiments, Intel SSD DC S3520 Series, has a capacity of 480GB with a 400MB/s sequential read and a 350MB/s sequential write speed in raw performance, and 41K IOPS for read.

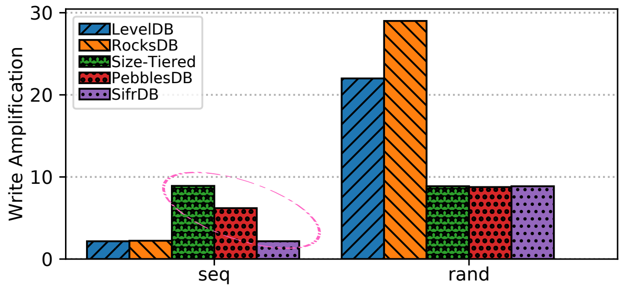
All the microbenchmark and YCSB workloads are replayed by the `db_bench` toolset [11, 13]. Since Cassandra does not support the `db_bench` and running it in the normal mode involves network latency, we re-implement the Size-Tiered by reusing the LevelDB code to provide a fair comparison. We still run 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 118GB 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 [7, 43].

5.2 Write Performance

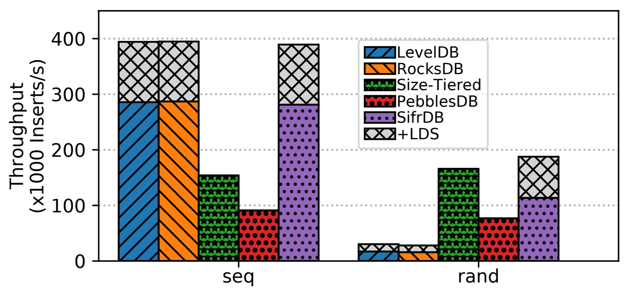
In this sub-section we evaluate the write performance by inserting 1 billion KV pairs to an empty store, with an average KV size of 123 bytes (23 bytes key, and remaining portion containing a number of bytes uniformly distributed in the 1~200), leading to a 118-GB dataset being built at the end. The 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 tree model and the forest model as analyzed in §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[5] (an optimized value in the practical MS-forest implementations).

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

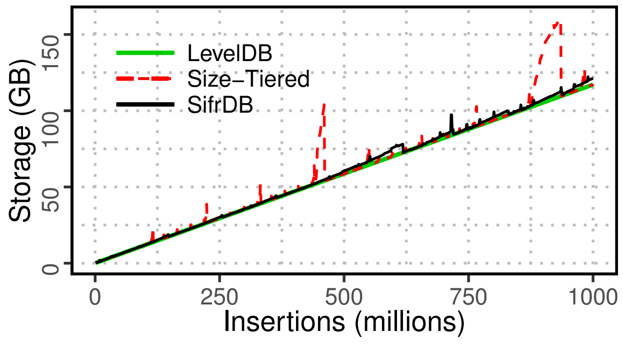
[5]The growth factor in PebblesDB is the number of SSTs in a guard that triggers the compaction.



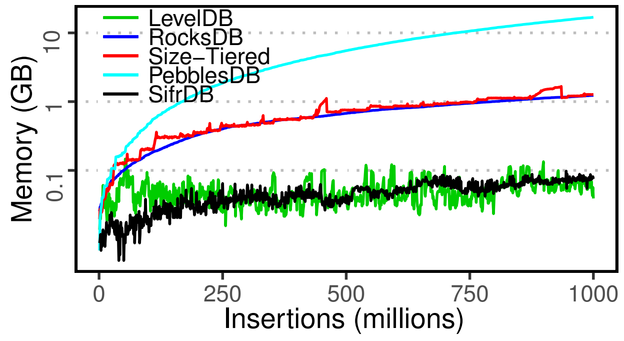
(a) Write Amplification (lower is better)



(b) Write Througput (higher is better)



(c) Storage Requirement



(d) Memory Requirement

Figure 9: Figure (a) shows the overall write amplification of inserting 1 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*).

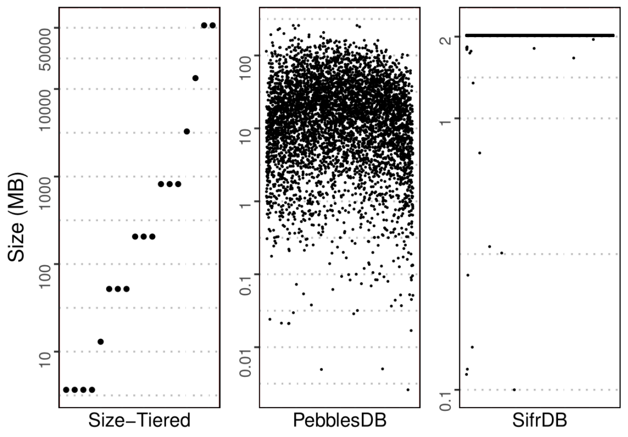
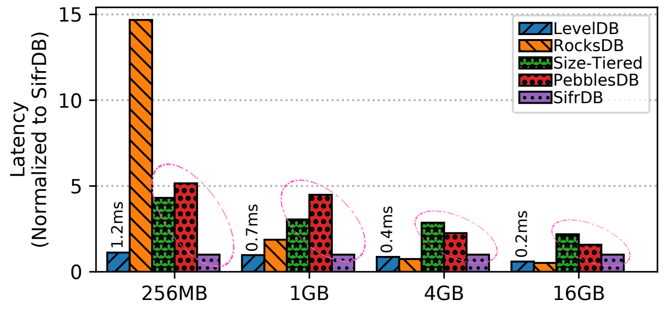


Figure 10: A snapshot of the physical files' size of the three MS-forest implementations under random writes.

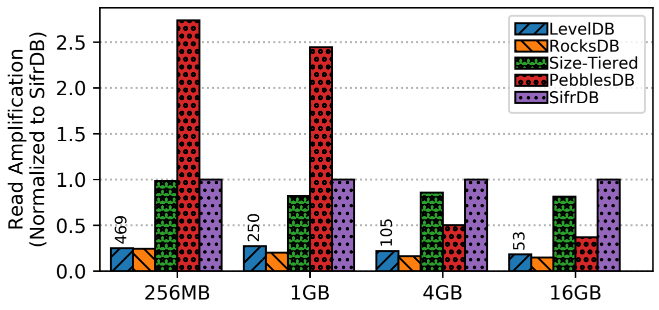
sequential workload, SifrDB induces no write amplification, just like LevelDB and RocksDB do, which is in sharp contrast to the other two MS-forest implementations, Size-Tiered and PebblesDB, that causes 8× and 6× write amplification respectively. This is useful in some cloud environments such as sensor-collected data [40]. 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.

Intuitively, 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 Figure 9b. One is the overhead on the write-ahead log, which is prominent in a sequential write pattern [33]. The other is the file-system overhead, which is more pronounced for small files. Figure 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 [33]. With the aligned storing, SifrDB achieves the highest throughput for the random workload, as shown by the SifrDB+LDS result in Figure 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 through 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.

Figure 9c shows the storage requirement of SifrDB, Size-Tiered and LevelDB. With the early-cleaning mechanism,



(a) Read Latency (Lower is better)



(b) Read amplification

Figure 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 3 bars of each bar group), and Figure (b) shows the read amplification (read_IO_size/queried_data_size).

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

Table 3: Number of candidate trees in different stores *(in PebblesDB the number of trees are various in different guards)*.

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 Figure 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 Read Performance

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 3, which are obtained after the write process is finished. As MS-forest implementation 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.

The evaluation results are presented in Figure 11a in the metric of query latency, along with read amplification presented in Figure 11b. We disable the seek-triggered compaction (referring to §6 for a discussion) and conduct the experiments on each store with sufficiently long time to make sure the performance has become stable. From the results shown in the figures we can draw two conclusions: (1) SifrDB is more efficient under configurations with higher dataset/memory ratios, and among the best performers cross the board at 256MB and 1GB memory provisions; (2) SifrDB consistently performs the best among the three MS-forest implementations, outperforming Size-Tiered and PebblesDB significantly. For example, if the provisioned memory is 256MB, which simulates a configuration of 400:1 dataset/memory ratio [7], SifrDB achieves better performance than Size-Tiered by 4×, and PebblesDB by 5× respectively. This is because at a high dataset/memory ratio, I/O is frequent and the parallel search efficiency of SifrDB is fully utilized. RocksDB performs poorly under low memory provision because it consumes more memory than the provisioned and frequently accesses the swapping space. Note that the I/Os from the swapping space is not accounted in the read amplification. With higher memory provision, the parallel efficiency of SifrDB is weakened because a significant portion of the search operations are serviced by the cache without I/O. However, SifrDB still consistently outperforms other MS-forest implementations.

With the parallel-search algorithm, 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-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 1GB. Comparing SifrDB's read amplification to that of Size-Tiered, we can see that 15% more unnecessary I/Os are incurred by the former (with 16GB 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 SifrDB's design principle, and does not impact the effectiveness of the parallel-search algorithm.

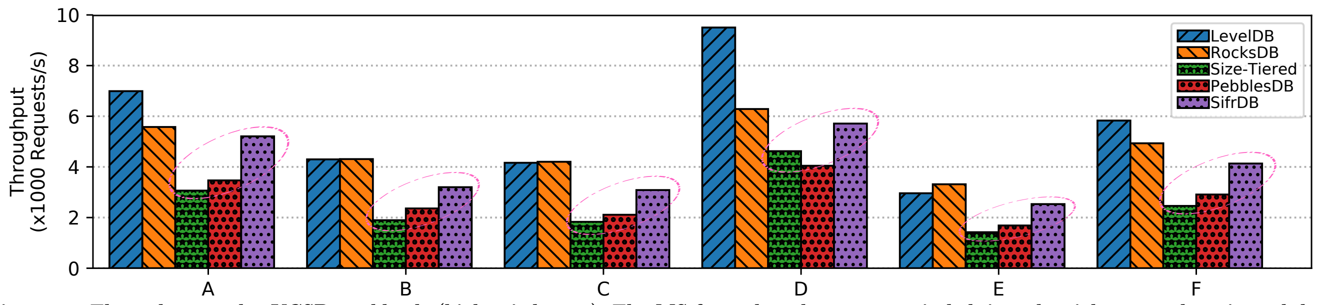


Figure 12: Throughput under YCSB workloads (higher is better). The MS-forest based stores are circled, i.e., the rightmost 3 bars in each bar group.

It should be noted that if enough memory is available and a Bloom filter is used on top of the stores, searches for most candidate trees can be avoided and all the stores will exhibit better read performance. We have performed experiments using an SST-Level Bloom filter with 10bits/key on each store, and found that, for almost all queries, only one candidate tree was actually searched. In such cases, SifrDB has comparable read performance to all other stores. We have an extensive discussion of the Bloom filter in §6.

5.3.2 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 forest model needs to seek more trees than the 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% over the other two forest-based implementations.

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. We have a discussion for the seek-triggered compaction in §6.

5.4 Synthetic Workloads

YCSB [21] provides a common set of workloads [1] for evaluating the performance of cloud systems.For a workload, 4 threads run concurrently and each of them sends 10K 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 Figure 12. While most of the workloads are mixtures of reads and writes, the main performance cost comes from the read requests, as the read-intensive workloads exhibits a lower throughput. SifrDB 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.

6 DISCUSSIONS

Split Unit. The unit size for the split storing of SifrDB, referred to as the *split unit*, is set to 2MB by default. To fully benefit from the special sequential-intensive workloads as demonstrated in VT-tree [41], a smaller split unit would be preferred. The downside of using a smaller split unit is the increased file-system metadata overhead, which can be eliminated with aligned and direct storing [33]. However, the overhead for maintaining smaller key ranges, such as larger index data and scattered key ranges on the underlying storage, is a necessary price to pay for both VT-tree and SifrDB, while the advantage of SifrDB is that there is no need to concern about garbage collection.

Scalability of the Parallel Search. In the parallel-search algorithm, a search thread simply moves on to the next object in the queue if it finds that the current one is not necessary to work on (an FIFO processing policy). Although our current design works well and consistently improves the query performance, the proposed queuing mechanism can be further exploited with a more efficient and economic scheduler. For example, if two query keys exist in the same tree or block, they can be searched in batch by one thread. For another example, if users want a short response time for a special query (such as VIP service), the query object can be scheduled to the front so that all the resources are used for it. A limitation of the parallel-search algorithm is that it is not suitable for KV stores hosted on singular HDD that lacks access parallelism. However, the algorithm provides a foundation to be extended for a RAID storage, be it HDD-based or SSD-based. For example, if the candidate trees are placed in different HDD devices, the parallel search would still take effect.

Seek-Triggered Compaction. LevelDB, as well as other multi-stage based stores, supports the seek-triggered compaction, which compacts the tree/sub-trees that are accessed more than a threshold number of times (e.g., 10), even though the data in the relevant stage has not yet reached its legal capacity. Seek-triggered compaction optimizes the subsequent read requests, with a price of extra compaction operations. This is profitable for a read-intensive workload in the long run. With enough read requests, the data in the store can be merged to a single tree, which is most efficient for read. For this consideration, SifrDB and the other common MS-forest implementations can converge to a single large tree faster, because data is written in each stage only once.

Bloom Filter. Bloom filter is a third-party index that can be applied to the KV stores to improve read performance (not applicable to range query). There are two main types of Bloom-filter based implementations: the block-level filter and the file-level filter. The difference is that using the block-level filter the block index is accessed first, while using the file-level filter the filter is tested first. The total bits needed are the same no matter the filter types. Monkey [22] studies how to efficiently configure the filter bits with respect to the access frequency of different stages. Although SifrDB supports both types of filters, in the evaluation we focus on the query performance without the Bloom filter, in order to have a clear understanding of the raw performance of the stores. Nevertheless, using the Bloom filter would give SifrDB flexible trade-offs, e.g., allowing more overlapped trees in a stage so as to boost write throughput without sacrificing the query QoS.

7 RELATED WORK

Write optimized structures have become popular in large-scale data stores [5, 8, 14, 20, 23, 28, 39, 44]. In general, there are two families of write optimized structures. One is the fractal-trees [16–18] that buffer the 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 [4, 12, 19, 25, 29].

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 [22, 31, 33, 38, 41, 43] and most of them focus on the write amplification of the 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 [9, 38, 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 [41] 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 forest model, since it uses a tree-based implementation (LevelDB) as the baseline for evaluation.

As the overheads introduced by the storage stacks becoming outstanding in high performance storage environments [35, 37], aligned storing is used to optimize the file-system overheads [33] 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 [42] propose to expose the internal channels of the SSDs and schedule requests on the channels to fully exploit the SSD bandwidth. Wisckey [32] 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 [22, 24, 31]. 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 [6] to ingest the large data generated in offline or migrated from other data stores.

These work 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.

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 a parallel-search algorithm 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

We thank our shepherd Peter Alvaro for advising to improve the final version of this paper, and thank all the anonymous reviewers for their constructive comments. This work is supported in part by Nature Science Foundation of China under Grant No. 61872156 and No.61821003, the Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, the US NSF under Grant No.CCF-1704504 and No.CCF-1629625, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] 2010. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads/.

[2] 2011. Improved Memory and Disk Space Management. https://www.datastax.com/dev/blog/whats-new-in-cassandra-1-0-improved-memory-and-disk-space-management.

[3] 2011. Introduction to Compaction. https://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra.

[4] 2013. RocksDB. http://rocksdb.org/.

[5] 2016. LSM Design Overview. https://www.sqlite.org/src4/doc/trunk/www/lsm.wiki.

[6] 2017. Bulkloading by ingesting external SST files. http://rocksdb.org/blog/2017/02/17/bulkoad-ingest-sst-file.html.

[7] 2017. From Big Data to Big Intelligence. https://www.purestorage.com/products/flashblade.html.

[8] 2017. The Modern Engine for Metrics and Events. https://www.influxdata.com/.

[9] 2018. Compaction subproperties. http://docs.datastax.com/en/cql/3.1/cql/cql_reference/compactSubprop.html.

[10] 2018. Configuring compaction. https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_configure_compaction_t.html.

[11] 2018. LevelDB benchmark. https://github.com/google/leveldb/blob/master/db/db_bench.cc.

[12] 2018. LevelDB project home page. https://code.google.com/p/leveldb/.

[13] 2018. RocksDB benchmark tools. https://github.com/facebook/rocksdb/blob/master/tools/db_bench_tool.cc.

[14] Daniel Bartholomew. 2014. *MariaDB cookbook*. Packt Publishing Ltd.

[15] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.

[16] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. 2007. Cache-Oblivious Streaming B-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 81–92.

[17] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 546–554.

[18] Adam L Buchsbaum, Michael H Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. 2000. On External Memory Graph Traversal. In *ACM-SIAM Symposium on Discrete Algorithms*. 859–860.

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

[20] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment* 1, 2 (Aug. 2008), 1277–1288.

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

[22] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 79–94.

[23] Shakuntala Gupta Edward and Navin Sabharwal. 2015. MongoDB Explained. In *Practical MongoDB*. Springer, 159–190.

[24] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 42, 13 pages. https://doi.org/10.1145/3190508.3190524

[25] Lars George. 2011. *HBase: the definitive guide*. "O'Reilly Media, Inc.".

[26] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-Based Solid State Drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM.

[27] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *International Conference on Very Large Databases*, Vol. 97. 16–25.

[28] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*. 301–315.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[30] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. 2014. Assert (! Defined (Sequential I/O)). In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*.

[31] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. 149–166.

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

[33] F. Mei, Q. Cao, H. Jiang, and L. Tian. 2018. LSM-tree Managed Storage for Large-Scale Key-Value Store. *IEEE Transactions on Parallel and Distributed Systems* (2018). https://doi.org/10.1109/TPDS.2018.2864209

[34] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*.

[35] J. Mohan, R. Kadekodi, and V Chidambaram. 2017. Analyzing IO Amplification in Linux File Systems. *ArXiv e-prints* (July 2017). arXiv:cs.OS/1707.08514

[36] Patrick Oneil, Edward Cheng, Dieter Gawlick, and Elizabeth Oneil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* (1996).

[37] Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2017. Iris: An Optimized I/O Stack for Low-latency Storage Devices. *ACM SIGOPS Operating Systems Review* 50, 1 (2017), 3–11.

[38] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 497–514.

[39] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 145–156.

[40] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. LittleTable: A Time-Series Database and Its Uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 125–138.

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

[42] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM.

[43] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*.

[44] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A Bender, et al. 2016. Optimizing Every Operation in a Write-optimized File System. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*. 1–14.