

Improving LSM-Tree Based Key-Value Stores With Fine-Grained Compaction Mechanism

Hui Sun , Guanzhong Chen , Yinliang Yue , and Xiao Qin , *Senior Member, IEEE*

Abstract—LSM-tree-based key-value stores (KV stores) render high-performance read/write services to data-intensive applications. KV stores employ an SSTable-based Coarse-Grained Compaction (CGC) mechanism, which involves a huge amount of data that do not need to be updated, thereby bringing a high write amplification (WA) and long tail latency. To address this issue, we propose a Fine-Grained Compaction (FGC) mechanism anchored on a Log-Structured patched-Merge tree (LSPM-tree) - a new data organization that averts rewriting irrelevant data into disks amid compaction. A cluster, the basic unit in FGC, encloses several patches and a redirection table, where each patch has an array of KV regions. We devise three compaction modes powered by the LSPM-tree, and we implement a high-performance key-value store, named FGKV. The extensive experiments show that FGKV improves the random-write throughput by up to 121%, 36.8%, 38.6%, and 15.2% compared with LevelDB, RocksDB, LDC, and ALDC, respectively. FGKV lowers the WA of the alternative KV stores by up to 50%. FGKV boosts read performance by up to 122%, 51.4%, 96.6%, and 368%, respectively, and FGKV curbs the 99th percentile latency of LevelDB, RocksDB, LDC, and ALDC by up to 78.2%, 77.6%, 78.3%, and 73.1% under YCSB A, respectively. Moreover, FGKV is readily extended to the other KV stores.

Index Terms—LSM-tree, key-value stores, fine-grained compaction, KV region, write amplification.

I. INTRODUCTION

LOG-STRUCTURED merge trees (LSM-trees) [1], [2] are the backbone index structures for persistent key-value stores that handle write-intensive workloads. LSM-tree based Key value stores (KV stores) are popularly deployed as a technological underpinning for Big Data and cloud computing applications, including real-time data processing, search engine, graph database, and online transaction processing (OLTP) [3], [4], [5], [6], [7]. KV stores are often utilized in cloud computing because they provide a scalable, reliable, and efficient approach

to data storage and retrieval, which guarantee high performance, availability, and secure data services in cloud computing systems [3], [5], [6], [8]. To achieve high write performance, key-value pairs (KV pairs) are processed in batch and converted into ordered ones in memory before being written into disks. Thus, LSM-tree transfers random writes into sequential ones, providing high performance and space utilization compared with B+ or B-trees [1], [9], [10]. Various KV stores, including but not limit to LevelDB [11], RocksDB [3], Cassandra [8], HBase [12], Dynamo [13], and cLSM [14], are employed in large-scale storage systems. In many applications, there are increasingly high proportion of write operations [15], [16]. The performance of write requests like continuous or burst random writes [17], [18] play a vital role in data storage systems. For this reason, in this study, we aim to optimize the random-write performance of KV stores.

An LSM-tree keeps multiple levels of KV pairs with inclusive key ranges in the disk (Section II). Once level i reaches its size limit, KV pairs at level i merge with that at level $i + 1$ in the merge-sort manner - a process is known as compaction.¹ The compaction procedure guarantees KV pairs to be unique and ordered in each level in LSM-tree, thereby balancing the read performance in KV stores. However, the compaction procedure may inevitably result in severe write amplification (WA), which surges to 10~50 times in the worse case [19], [20]. A plethora of prior solutions were proposed to reduce WA in KV stores [17], [21], [22], [23], [24], [25], [26]. For example, WiscKey [21] was designed with a key-value separation mechanism to reduce the frequent large-size value movement in compaction. dCompaction [23] delays compaction until enough data are accumulated. PebblesDB [17] directly writes data into a lower level, avoiding multiple merge-sorts and data rewrites in the same level. Nevertheless, many of these techniques utilize lazy compaction to accumulate data volume, which not only degrades read performance but also causes high tail latency.

In a modern LSM-tree architecture in a diversity of industrial systems such as LevelDB and RocksDB, sorted string tables (SSTables) are the basic units, which are also used in compaction. We name, throughout this paper, the SSTable-based compaction as a Coarse-Grained Compaction (CGC) mechanism. In CGC, SSTables at level $i + 1$ that overlaps a key range of a compacted SSTable at level i are loaded into memory for merge-sort operations. The newly generated SSTables are

¹In this study, levels i and $i + 1$ involved in a compaction procedure are denoted as upper and lower levels, respectively.

Manuscript received 26 October 2022; revised 20 May 2023; accepted 27 October 2023. Date of publication 2 November 2023; date of current version 6 December 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 61702004 and 62072001. The work of Xiao Qin's work was supported in part by the U.S. National Science Foundation under Grants IIS-1618669, OAC-1642133, and CCF-0845257. Recommended for acceptance by Y. Yang. (Corresponding author: Yinliang Yue.)

Hui Sun and Guanzhong Chen were with the School of Computer Science and Technology, Anhui University, Hefei 230201, China (e-mail: sunhuiworking@gmail.com; e20201057@ahu.edu.cn).

Yinliang Yue was with the Zhongguancun Laboratory, Beijing 100049, China (e-mail: yylhust@qq.com).

Xiao Qin was with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849 USA (e-mail: xqin@auburn.edu).

Digital Object Identifier 10.1109/TCC.2023.3329646

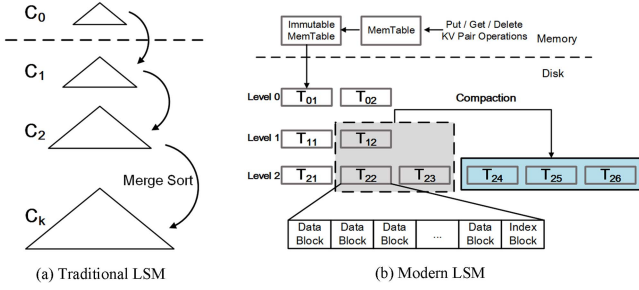


Fig. 1. LSM-tree and LevelDB architectures.

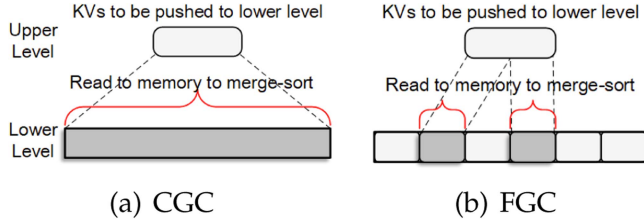
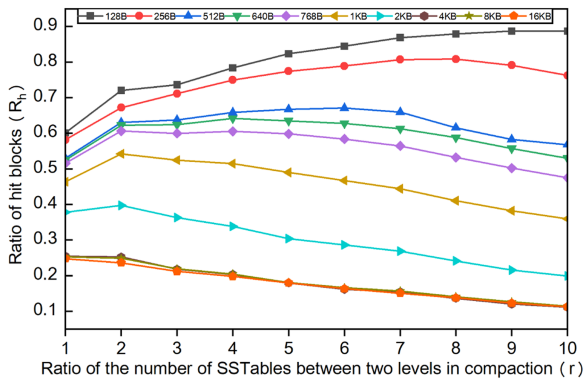


Fig. 2. Compaction mechanism of CGC and FGC.

Fig. 3. Impact of ratio of the number of SSTables in S_i to that in S_u (r) and KV-pair size on block hit ratio (R_h).

rewritten into level $i + 1$. Nevertheless, a majority of KV pairs in SSTables at level $i + 1$ are not updated, resulting in an excessive amounts of unnecessary rewritten data: an enlarged WA is a direct consequence. Moreover, compaction enclosing large amounts of data will block user requests, resulting in long tail latency.

We study data granularity and the SSTable-block data layout in a popular KV store - LevelDB that is outlined in Fig. 1(b). We reckon that the CGC mechanism in traditional KV stores causes severe WA and long tail latency. In Fig. 3, we investigate the proportion of blocks that need to be updated to the total amount of blocks involved in compaction under random-write workload with 40 GB data volume in LevelDB. When a KV pair is larger than 2 KB, there are merely 10%~30% of blocks that ought to be rewritten. Since more than 70% of blocks rewritten in compaction are worthless, the CGC mechanism leads to a high WA.

To address the problems in CGC and to balance write and read performance, we propose a novel data organization - a *Log-Structured patched-Merge tree* or *LSpM-tree*, which inherits the

advantages of disk sequential writes in LSM-tree. In LSpM-tree, a cluster is a basic unit which is referred to as a set of KV pairs in a key range at one level (Section III-A). A cluster includes multiple partitions named region. Each compaction selects a portion of the regions in the cluster, followed by sequentially writing merged key-value pairs to a disk file named a patch. Afterward, the patch is stored in the cluster. Therefore, LSpM-tree can reduce WA while also providing much higher throughput than in-place trees, e.g., B-trees. B-trees require in-place updates that require an abundance of random disk I/O operations [21], [24], [27], [28], [29].

In LSpM-tree, a compaction procedure rewrites only the data that needs to be rewritten in the cluster in a patch-like manner, which reduces WA. Clusters also allow LSpM-tree to find keys efficiently using a built-in redirection table. We design a fine-grained compaction mechanism, named FGC, in KV stores anchored on the LSpM-tree. We design three compaction modes - *patch*, *patch⁺*, and *cluster* modes - for the three types of access granularity in a cluster. Each compaction task in FGKV is merely conducted in a specified compaction mode: FGKV executes compaction mode according to a compaction task type, achieving good trade-off among WA, throughput, and space consumption. FGKV consists of the Compaction Advisor module, Granularity Manager module, Merger module and Cluster Manager module. The Compaction Advisor module change gears among the three compaction modes. The Granularity Manager module is responsible for region partitioning and compaction procedure control. The Merger module performs the merge-sort operations, and produces patches. The Cluster Manager module transfers cluster metadata and data of hit regions to the Compaction Advisor and Merger modules, respectively. The Cluster Manager also assembles produced patches into a cluster.

Our contributions are summarized as follows.

- ▷ We propose a new data organization, called LSpM-tree, in which regions and patches enable compaction procedures to access data at a fine granularity. A redirection table is engaged to ensure data consistency for compaction and efficient point look-ups.
- ▷ Based on access granularity in LSpM-tree, we propose the *Fine-Grained Compaction (FGC)* mechanism, including three compaction modes - *patch* compaction, *patch⁺* compaction, and *cluster* compaction.
- ▷ We design a LSpM-tree-based KV store named FGKV that consists of four modules. The Compaction Advisor is responsible for preprocessing compaction. The Granularity Manager sends control signals and processes intermediate results of CGC. The Merger conducts merge-sort operations to originate compaction results. The Cluster Manager module oversees data in a LSpM-tree.
- ▷ We implement FGKV based on LevelDB. Evaluation results show that FGKV outperforms the CGC-mechanism KV stores (e.g., LevelDB and RocksDB) in YCSB, while achieving 121% higher throughput than these KV stores in random writes. FGKV exhibits the lowest 99th percentile latency under the mixed read/write workloads in YCSB.

The rest of this paper is organized as follows. In Section II, we introduce the LSM-tree and LevelDB, which highlights the

motivation of our work. We elaborate on the design issues of FGKV in Section III. In Section IV, we present the system implementation. Section V describes an experimental environment, followed by thorough comparisons between our approach and the alternative KV stores in Sections VI and VII. We discuss on the crash consistency, memory overhead, and comparison with B-tree issues of FGKV in Section VIII. Prior studies that are related to this work are summarized in Section IX. Finally, we draw a conclusion in Section X.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce the background of LSM-tree and LevelDB. Then, we articulate the CGC mechanism in LevelDB to clarify our research motivation.

A. LSM-Tree and LevelDB

LSM-tree-based key value stores or KV stores have gained popularity in cloud computing, and many researchers have enhanced the performance and reliability of KV stores by optimizing data structures, algorithms, and architectures [30], [31]. LSM-tree, a persistent data structure, converts a significant number of random-write operations into sequential ones to leverage the high-sequential write performance of disks [3], [5], [6], [8], [30], [31] - enabling cloud computing systems to fullest potential. Because of high write performance coupled with low storage consumption, LSM-tree and its variants - including LevelDB [11], RocksDB [3], and Cassandra [8] - have been widely used in the academia and industry.

A traditional LSM-tree (see Fig. 1(a)) consists of multiple components $C_0 \sim C_k$. The C_0 resides in memory, and the other components are located in a disk. A large number of write operations are batched in C_0 of memory. When C_0 is full, C_0 is merged with C_1 in the disk, and the result is written into C_1 . By the same token, a full C_1 performs a merge-sort operation with C_2 . Such a process is referred to as compaction, which leads to WA. In KV stores, term WA is defined as a ratio between the amount of data written into an underlying storage device and the amount of data requested by users. The compaction with large WA blocks I/Os and wastes bandwidth, deteriorating system throughput. Fig. 1(b) shows a modern LSM-tree structure adopted by LevelDB, which is a popular high-performance KV store inspired by BigTable [32]. In LevelDB, each component of LSM-tree is referred to as a *level*. Each level is divided into multiple fixed-size SSTables. Except level 0, key ranges of SSTables in a level are non-overlapping.

After being initially written into a disk's log, a new KV pair is inserted into a MemTable in memory according to the order of keys. In case of the full MemTable, LevelDB creates a new MemTable to handle the subsequent writes. In the background, the previous MemTable is converted into immutable MemTable and awaits the flush to level 0. When level i is filled with KV pairs, a compaction procedure selects SSTables with an overlapping key range as in the adjacent levels (levels i and $i + 1$) in merge-sort operations.

In Fig. 1(b), for instance, the key range of SSTable T_{12} at level 1 overlaps with those of T_{22} and T_{23} at level 2, and

the KV pairs are involved in compaction. All KV pairs of SSTables T_{12} , T_{22} and T_{23} are read to memory and performed merge-sorting operation; afterward, newly produced SSTables T_{24} , T_{25} , and T_{26} are stored at level 2. The compaction procedure intends to write one SSTable, but three SSTables are written into the underlying disk. Thus, the WA is measured as $WA = Size(T_{24} + T_{25} + T_{26}) / Size(T_{12})$. And if the new SSTables written back triggers a compaction at level 2, WA will increase further. In the LSM-tree, data size grows exponentially between two adjacent levels (e.g., levels i and $i + 1$ from the upper level (level i) to the lower level (level $i + 1$) by an amplification factor (AF, usually 10). If an SSTable moves from level i to level j ($j > i$) through multiple compaction procedures, the average WA factor is $AF \times (j - i)$. A large WA, undoubtedly, consumes I/O bandwidth and greatly reduces the throughput in applications [17], [33]. The main reason behind high WAs is that the compaction procedure is obliged to retrieve all KV pairs in SSTables with overlapping key ranges in a lower level into memory and write the KV pairs back to the same level after merge-sort operations. To mitigate a high WA, we strive to reduce the amount of lower-level compaction data.

B. Motivation

Let us take LevelDB as an example. The compaction procedure selects SSTables with overlapping key ranges at adjacent two levels (e.g., levels i and $i + 1$). As mentioned in Section II-A, the data size of two adjacent levels exponentially surges. For the same key range, the amount of data at level $i + 1$ is r times that of level i .² Under update-intensive workloads, there are merely a small portion of KV pairs in r SSTables at level $i + 1$ being updated.

In CGC, KV pairs that do not require an update are forced to be part of a compaction operation because these KV pairs - residing in the compacted SSTables - are merely rewritten back into the same level. In Fig. 2(a), the CGC mechanism involves a large number of KV pairs that are unnecessary to be updated in merge-sort and rewrite operations. Moreover, the unwanted updates of KV pairs push up WAs, CPU cost, and I/O bandwidth usage, thereby degrading the overall system performance.

To further explore data rewrites in CGC, we delve into the data selection and rewrite operations, and preliminary results verify our speculations. In LevelDB, an SSTable is composed of data blocks, and a block is the smallest unit for read and write. We treat a block as a basic statistical unit in the test, and our primary focus is the relationship between *hit blocks* and all blocks in compaction. *Hit blocks* are defined as blocks that contain KV pairs to be updated in compaction.

Table I presents the notation and symbols used throughout this paper. To simplify the analysis, in this section, the SSTable size S is an integer multiple of block size B ; meanwhile, block size B is an integer multiple of key-value pair size K . Let us consider a block in S_i : when a KV pair in S_u is within its key range, this

²Symbol r is the amplification factor. For instance, in a stable state [34], when an SSTables at level i is involved in compaction, the number of SSTables at level $i + 1$ should be r .

TABLE I
NOTATION AND DESCRIPTION IN CGC

Notation	Description
K	The size of a KV pair
B	The size of a block
S	The size of an SSTable
S_u	A set of compacted SSTables at level i
S_l	A set of compacted SSTables at level $i + 1$
R_h	The ratio of hit blocks in S_l
n_i	Number of SSTables in S_u
N_{Ku}	Number of KV pairs in S_u
N_{Bl}	Number of blocks in S_l
S_{new}	A set of new SSTables after compaction
r	Ratio of the number of SSTables in S_l to that in S_u

block is considered as a *hit block*; otherwise, the block is marked as a *non-hit block*.

When block size B is larger than KV-pair size K ($B > K$), the number of blocks in an SSTable is measured as $\frac{S}{B}$. The number of KV pairs in a block is $\frac{B}{K}$. If the number of SSTables in S_u is ' n_i ' involved in compaction, there will be ($n_i \times r$) SSTables in S_l . Thus, we have

$$N_{Ku} = n_i \times \frac{S}{K}. \quad (1)$$

where N_{Ku} denotes the number of KV pairs in an upper level. And the number of lower-level blocks (N_{Bl}) is

$$N_{Bl} = n_i \times r \times \frac{S}{B}. \quad (2)$$

In the worst case, each key-value pair at the upper level hits a block at the lower level individually. Therefore, a non-hit block only exists when the number of blocks at the lower level in the compaction is greater than the number of key-value pairs at the upper level ($N_{Bl} > N_{Ku}$). We can deduce that the fine-grained compaction has a significant performance optimization only when r is larger than $\frac{B}{K}$. Hence, the block hit ratio (R_h) can be expressed as

$$R_h = \frac{N_{Ku}}{N_{Bl}} = \frac{n_i \times \frac{S}{K}}{n_i \times r \times \frac{S}{B}} = \frac{B}{Kr}. \quad (3)$$

R_h depends on three parameters: block size B , KV-pair size K , and SSTable number ratio r . B is generally a constant, and R_h relies on K and r . Note that $\frac{B}{K}$ is 1 when K is larger than B because each block is comprised of at least one key-value pair. And (3) is rewritten as $R_h = \frac{1}{r}$.

R_h refers to the proportion of blocks in CGC to be updated to the number of blocks that actually participate in the compaction: the CGC procedure rewrites $(1 - R_h)$ extra data that have no need to be updated. This problem motivates us to propose an approach to reducing the volume of rewritten data, which in turn can suppress WA. The degree of WA reduced is inversely correlated with R_h , the larger R_h . The more blocks are hit, and the less amount of compaction data can be reduced (which can be simply regarded as WA). Conversely, a smaller R_h means that there are more blocks that do not need to be updated, so the greater the WA can be reduced. After quantifying rewritten data

in CGC and weighing in on CGC's WA measures, we draw the following conclusions.

(1) When r is smaller than or equal to $\frac{B}{K}$ ($B \geq K$), R_h goes up as the SSTable ratio (r) increases; R_h drops as the KV-pair size (K) is enlarged. The rationale behind this trend is justified as follows. The compaction procedure selects SSTables at level $i + 1$ according to the key range of the SSTable at level i in this procedure. Generally, the key range of SSTables at level $i + 1$ is larger than that at level i ; then, the blocks at the beginning and end of the SSTables at level i are not hit. In case of a small r , the proportion of non-hit blocks cannot be ignored. As r expands, the volume of these non-hit blocks becomes small, enlarging the R_h value. The larger the KV pair size (K) is, the smaller the number of KV pairs in each SSTable becomes. Generally, few blocks at level $i + 1$ will be hit; thus, R_h becomes small as K increases.

(2) When $r > \frac{B}{K}$, R_h decreases as the SSTable ratio (r) or the KV pair size (K) increases. We present the reason as follows. When r is larger than $\frac{B}{K}$, the number of KV pairs involved in the compaction at the level i is smaller than the number of blocks at level $i + 1$. In the worst case, it is guaranteed that not all the blocks at level $i + 1$ cannot be hit. Then, the hit ratio (R_h) of the blocks at level $i + 1$ must be less than 1. According to (3), R_h heavily depends on K and r . When the KV pair size is fixed (K), R_h is shrinking as the ratio of the number of compacted SSTables between levels i and $i + 1$ climbs up: the WA significantly scales down. Similarly, with a constant r , the larger the key pair size (K) is, the smaller the R_h becomes; then, the WA reduces to a greater extent.

To verify above arguments and speculations, we employ LevelDB-1.20 as a KV-store testbed. YCSB is applied to generate 40 GB random-write workloads with different KV sizes – 128 B, 256 B, 512 B, 640 B, 768 B, 1 KB, 2 KB, 4 KB, 8 KB, and 16 KB. In this test, we set the size of an SSTable and a block to 4 MB and 4 KB, respectively. We plot the results in Fig. 3.

Observation 1: With small-size KV pairs such as 128 B and 256 B, raising r value leads to a continuously growing block hit ratio R_h . Fig. 3 presents the change of r over the interval between 1 and 10. Given small-size KV pairs, $\frac{B}{K}$ is larger than 10 (e.g., when $K = 256$ Bytes, $\frac{B}{K} = 16$). Thus, Fig. 3 reveals that r is smaller than $\frac{B}{K}$, and conclusion 1 is held. We also find that R_h goes up with an increasing r , which concurs with conclusion 1.

Observation 2: We find that the block hit ratio (R_h) generally increases and then decreases under workloads with medium-size KV pairs (e.g., 512 Bytes, 1 KB, and 2 KB). For medium-sized KV pairs, a K of 512 Bytes gives rise to the $\frac{B}{K}$ value of 8. Given $K = 512$ Bytes, R_h rises and then declines according to conclusion 1 and 2. In Fig. 3, R_h gradually climbs up as r increases from 1 to 8, but R_h falls after r exceeds 8. Similarly, if $K = 1$ KB, R_h rises gradually as r increases to 4 from 1; then R_h drops as r becomes larger than 4.

Observation 3: When KV-pair size is smaller than or equal to 4 KB ($K \leq 4$ KB), the distinction between each tested item is apparent. With the same r , R_h obviously decreases with an

TABLE II
THE IMPACT OF DECREASING THE SSTABLE SIZE ON PERFORMANCE OF
LEVELDB

KV size	1 KB		4 KB	
SSTable size	4 MB	4 KB	4 MB	4 KB
Random write throughput	33.7 MB/s	0.2 MB/s	33.5 MB/s	0.5 MB/s
Total write data volume	2.08 GB	1.6 GB	2.13 GB	1.05 GB
Numbers of SSTables	95	101021	96	82751
Random read throughput	134.4 MB/s	4.8 MB/s	524.9 MB/s	10.2 MB/s

increment of KV size K . This also fits the conclusions 1 and 2, i.e., under what circumstances the larger the K , the lower the block hit ratio R_h .

Observation 4: The results are largely similar for the case of large-scale KV pairs ($K > B$), and the change of KV size rarely impacts block hit ratio (R_h), which drops with a rising value of r . When $K > B$, the conditions in conclusion 2 above are met, and r must be greater than $\frac{B}{K}$, so according to the variant $R_h = \frac{1}{r}$ of (3), in the case of $K > B$, R_h will only decrease with the increase of r , and the change of K has little effect on R_h .

We observe from the preliminary observations that a large portion of irrelevant data passively involved in merge-sort and are rewritten back to the same level in CGC. Consequently, unnecessary data rewrites prolong compaction time, leading to a high WA and long tail latency. Under some workload conditions with a large data skew, the above procedure is repeatedly processing data multiple times before moving KV pairs to level $i + 1$, which experience degraded KV-store performance. Resizing the size of an SSTable to that of a block of a given data volume ultimately slashes the number of hit blocks without changing the data organization and compaction process, but this approach introduces serious performance issues. The low percentage of data in the SSTable causes subpar metadata management and intensive I/O access due to a huge number of small files on disks. To explore the impact of decreasing the SSTable size on performance, we conduct four tests using *db_bench*, setting the SSTable size to 4 MB and 4 KB, and configuring 1-KB and 4-KB KV pairs in LevelDB. We set up 500 MB random-write data coupled with 500 MB random-read data in workloads. The tested results are summarized in Table II.

Notably, we configure 4-KB KV pairs and 4-KB SSTable in LevelDB, where there is no overlapping key-ranges of two-level SSTables in compaction at level k ($k \geq 1$). Only the metadata of upper-level SSTables is shifted to the lower level (*a.k.a* trivial move), and this trivial move reduces WA. The findings suggest that shrinking the SSTable size to that of a block curbs WA at the cost of sharply downgraded read/write performance. Thus, without improving the LSM-tree structure and the compaction process, it is unfeasible to simply cut down the SSTable size to optimize the performance of modern KV stores.

In this study, we propose a novel Fine-Grained Compaction (FGC) mechanism depicted in Fig. 2. When a compaction is provoked, FGC selects data with a smaller granularity and only updates (or rewrites) data that genuinely need an update (see the dark area in Fig. 2(b)). Then, the amount of undesirable data involved in compaction is trimmed (see the light-colored area in Fig. 2(b)). In a modern LSM-tree-based production environment, there exists a wide range of KV-pair sizes in applications (e.g., from 128 B to 256 KB [5], [14], [35], [36], [37], [38]). Our

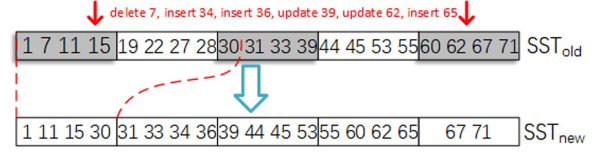


Fig. 4. After the SSTable-block based FGC, the key range of the first block in new SSTable SST_{new} is 1-30, which overlaps with the first three blocks in old SSTable SST_{old} .

FGC based KV store FGKV – offering impressive performance improvement under real-world workloads [14], [35], [38] with large-sized KV pairs – achieves superb read/write performance, short tail latency, and low WA in these scenarios. A description on the FGC mechanism is detailed in Section III.

III. DESIGN AND UNDERPINNINGS

Modern KV stores adopt the SSTable-block data organization to manage data. At each level, there are multiple SSTables with non-overlapping key ranges. The compaction procedure in CGC only loads SSTables at level $i + 1$ that intersect with the picked SSTables at level i into the main memory. These compacted SSTables are deleted after completing merge-sort operations, and new SSTables are inserted into level $i + 1$. This data organization, which is relatively simple yet intuitive, well adapts to the traditional compaction mode (coarse-grained compaction). Nevertheless, the data organization is not expected to facilitate the design of a fine-grained compaction mechanism. We present three problems to be solved as follows.

Problem 1: Blocks are produced in a data-driven manner anchored on the SSTable-block data organization. In FGC, the key ranges of a block generated by compaction can be large and may overlap with the key ranges of non-hit blocks, disturbing the existing block settings. Fig. 4 illustrates that the key range of the first block in the new SSTable SST_{new} overlaps with first three blocks in SST_{old} . The second block of SST_{old} does not participate in compaction.

Problem 2: In FGC, only a portion of KV pairs in SSTable SST_{old} in Fig. 4 participates in merge-sort operations (the data in hit blocks) with the rest KV pairs remaining in SST_{old} . Therefore, SST_{old} cannot be crossed out after compaction; meanwhile some KV pairs in new SSTable SST_{new} come from SST_{old} . After SST_{new} is entering to the same level, a key-range conflict occurs.

Problem 3: Because SST_{old} has not yet been deleted - the data of hit blocks still remains in it, rewritten data exists at the same time as source data. This issue causes the same key corresponds to multiple values, resulting in a serious data inconsistency obstacle.

A. LSpM-Tree

The above problems prompt us to propose a new data storage layout - the LSpM-tree organization - to facilitate the implementation of the FGC mechanism, which is aimed to not only enhance random write throughput but also reduce WA and tail latency. LSpM-tree is a variant of LSM-tree built on *Clusters*,

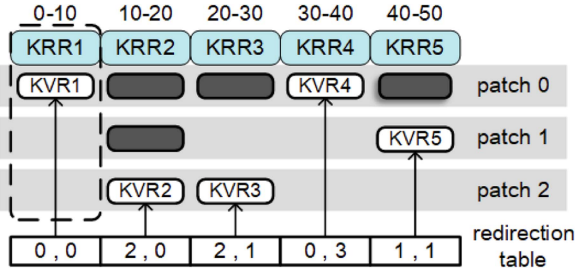


Fig. 5. Layout of a cluster. KRR and KVR denote KR region and KV region, respectively. In the redirection table, the two numbers in each entry represent patch number and KV region number in the patch, respectively. KV regions are sequentially stored in each patch. The dashed box denotes a region in the cluster, including KR region and corresponding KV regions.

the basic unit of LSpM-tree. A cluster consists of *KR regions*, *KV regions*, *patches*, and a *redirection table*. Fig. 5 depicts the layout of a cluster. Based on the region-patch-cluster three-level organizational structure of LSpM-tree, we can implement the region-patch-cluster based FGC without worrying about the SSTable-block-based FGC problems.

A cluster contains multiple partitions named *region*. Each region contains a fixed key range called *KR region*. KV pairs within the same KR region form a *KV region* and are stored consecutively on disk. Data in the region granularity are handled in compaction. Notably, regions that are elected by the compaction procedure are referred to as *hit regions*, and their KV regions are read into memory for merge-sort operations. Then, the data enclosed to the same KR region is written to a new KV region. A region stores data in a *key-driven manner* - regardless of the number of KV pairs existing in a region, new KV pairs can still be stored in it, as long as these new KV pairs fall within its key range. Any changes of the KV-region size have zero effect on the KR region. Applying the idea of separating KR region and KV region, we make data independent of region settings - the underlying disk data cannot affect the existing region settings. As the KV pairs are stashed in the same region before and after compaction, we solve Problem 1 while simplifying the complexity of data management in FGKV.

We design a sub-table file, named *patch*, in the cluster to store compaction results. When cluster_k at level $i + 1$ is forged, all the KV regions are orderly stored in the same physical file (file 0). If cluster_k is involved in compaction, the FGC procedure first marks hit regions in this cluster, and accesses data of hit regions in file 0 to perform merge-sort. Afterward, new KV regions are generated according to the KR regions of hit regions, and are registered in a new physical file (file 1). The data of file 1 derive from cluster_k along with some KV pairs at level i . File 1 is an update and addition to cluster_k, like a patch to cluster_k. Thus, we refer such files to as patches. Each compaction is presumed as a patch to cluster_k, and the updated data is stored in the latest patch, which addresses Problem 2.

FGC only updates the data of hit regions into new patches, which slash the amount of compaction data. After multiple FGC procedures, however, data corresponding to the same region may exist in multiple patches. The same key tends to have more than one value in the old and new KV regions, resulting in

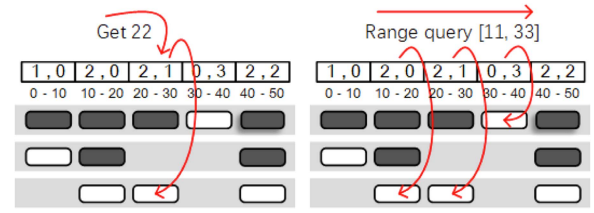


Fig. 6. Get and range queries in a cluster.

data inconsistency in Problem 3. Another problem is that read operations must search a target key in each patch to find the latest value, which degrades read performance. Thus we design a *redirection table* in a cluster to tackle the above problems. The redirection table labels the status and position of the latest KV region corresponding to the KR region and converts the relationship between the KR region and old KV region to that between the KR region and the latest KV region, see Fig. 5. The first entry in the redirection table denotes that the KV region corresponding to KRR1 is located at position 0 in patch 0. The second entry means that the KV region corresponding to KRR2 is updated to position 0 in patch 2 after two FGC procedures. The rest entries have a similar process.

Notably, there is no entry (i.e., metadata of region) wasted. This is because, when a delete operation hits one of the key-value pairs, the amount of data in a region is reduced. When all KV pairs in a region are deleted in the next compaction procedure, FGKV wipes out the redirection entry of this region during the redirection phase. With a redirection table, multiple patches in a cluster can be referred to as a single file externally. It means that a put/get operation accesses a cluster as if it was a contiguously stored file. User requests first access fixed KR regions whether the KV regions change or wherever they are stored, and access valid KV regions according to the relationship in the redirection table.

B. Read Process Under Clusters

Get Operations: Initially, a *Get* operator checks the in-memory memtable. If the key is not found, the operator searches level by level. Amid this stage, in LSpM-tree, the operator searches for a cluster where the key range contains the target key, rely on its region redirection table to acquire the information of the region where the target key is located, and then obtain the corresponding patch according to the location information and reads the KV region, see Fig. 6. Therefore, in each cluster, the *Get* operation will access disk at most once.

Range Queries: Range queries need to get all KV pairs in a given range. LSpM-tree first finds clusters in each level that intersect the given range, and then accesses valid KV regions in those clusters within the intersecting range. Because the regions in the cluster are contiguous, performing a range query in a cluster only requires sequentially accessing the region redirection table and then reading valid KV regions from the corresponding patches, without having to repeatedly read the data in every patch, see Fig. 6.

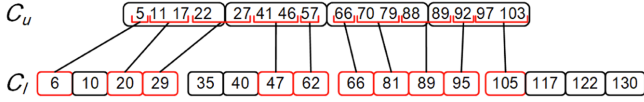


Fig. 7. Example of marking hit regions. Each number of C_u denotes a KV pair. Each number of C_l denotes a region, and the red-frame regions are hit regions.

C. Compaction Modes in LSpM

LSpM-tree selects clusters (C_l) at level $i + 1$ that have overlapping key ranges with level- i clusters (C_u) that sparked compaction. LSpM-tree renders three compaction execution modes for each cluster in C_l to choose (Section III-D for a specific selection mechanism). The *patch* compaction (Section III-C1) is a direct embodiment of FGC. The *patch*⁺ compaction (Section III-C2) is a variant with an early garbage collection of *patch* compaction. The *cluster* compaction (Section III-C3) performs major garbage collections and balances LSpM-tree. We introduce the three compaction modes below.

1) *Patch Compaction*: The *patch* compaction mode is an implementation of FGC, which is one of most important approaches to reducing WA. If a cluster C_i of C_l to be performed *patch* compaction, the FGC procedure marks hit regions for it according to the keys of C_u (see Fig. 7). Afterward, the corresponding KV regions are extracted to memory to conduct merge-sorting. The sorted KV pairs are built in new KV regions according to the KR regions of hit regions. The new KV regions constitute a patch as the up-to-the-minute patch in C_i . Finally, the region redirection table renews the mapping relationship between the hit regions and new KV regions. In the cluster included in PC process in Fig. 8, four regions are marked as hit regions. The valid data of hit regions are involved in merge-sorting and then rewritten back to the same cluster after compaction. A new file (patch 1) is generated as the compaction result.

2) *Patch⁺ Compaction*: The *patch* compaction mode refreshes the data of hit regions and rewrites them back to the cluster, keeping the compacted KV regions as invalid in old patches. Because a patch is the basic unit for creation and deletion operations in LSpM-tree, those invalid KV regions still remain on disks as garbage data - an enlarged space consumption becomes inevitable. Meanwhile, as the number of *patch* compaction surges, the proportion of invalid data in the old patches increases. FGKV is still obliged to allocate memory and disk space to store these patches for a handful of valid KV regions. A large number of patches incurs extra overhead on data management. As a solution to alleviate the side-effect of patches, we design a *patch*⁺ compaction mode to squash space consumption caused by the *patch* compaction, aiming to undertake part of garbage collection.

We introduce a threshold T_{ir} to represent the maximum proportion of invalid data in a patch. The *patch*⁺ compaction procedure marks each patch whose proportion of invalid data is greater than T_{ir} (e.g., 0.8) as a *hit patch*. Then, the remaining valid KV regions in hit patches are also moved into memory to conduct merge-sorting. The subsequent operations are identical to those

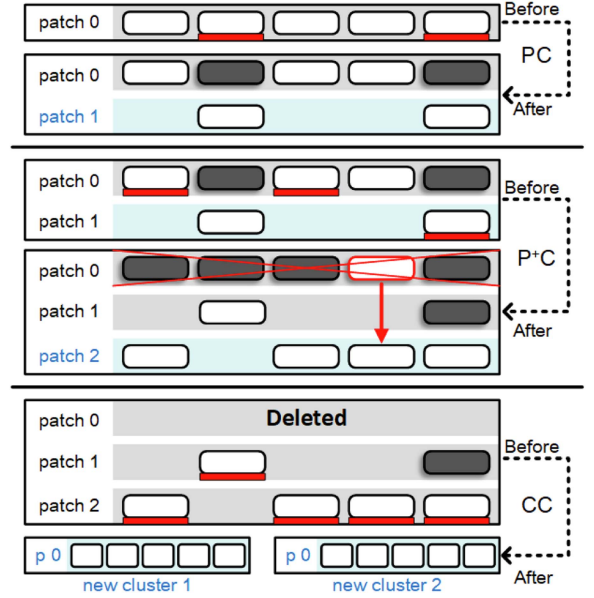


Fig. 8. Compaction example in FGKV. The boxes with red bars denote hit regions. The *patch* compaction (PC) generates a new patch. The *patch*⁺ compaction (P⁺C) forges a new patch and removes hit patches that has few valid data. The *cluster* compaction (CC) constructs new clusters and erases old cluster.

in the *patch* compaction, except that the *patch*⁺ compaction mode eradicates hit patches from the cluster to free up disk space after compaction. The *patch*⁺ compaction concentrates a small number of valid KV regions in multiple patches into one patch: the number of patches drops sharply by the virtue of an early garbage collection in the compaction where system resources are conserved. In the P⁺C process in Fig. 8, there are a lot of invalid data in patch 0, so it is marked as a hit patch. The rest KV region (the region with red frame) in patch 0 is also involved in compaction. Afterwards, patch 2 is generated, and patch 0 is deleted.

Notably, the *patch*⁺ compaction will be executed only when the proportion of invalid data of each patch exceeds T_{ir} in the *patch* compaction. As the *patch*⁺ compaction slightly increases the amount of rewritten data, it is prudent to make a good trade-off between WA and space consumption. We delve in the impacts of the *patch*⁺ compaction and T_{ir} on the system performance in Sections VII-B and VII-C.

3) *Cluster Compaction*: As the execution of *patch* compaction leads to garbage data in a cluster, we introduce *patch*⁺ compaction to perform partial garbage collection. *patch*⁺ compaction can only recycle patches with a high proportion of garbage data, and patches that do not reach threshold T_{ir} will not be reclaimed. With the operation of FGKV, the garbage data in such patches that do not meet the recycling requirements can no longer be ignored. Another problem is that as FGKV runs, the amount of data in each cluster grows, and an oversized cluster sparks compaction and merges with lower-level clusters – increasing the amount of compaction data accompanied by a long tail latency. To mitigate the above two roadblocks, we bring forth the *cluster* compaction mode to reallocate data in oversized clusters while performing garbage collection in compaction to

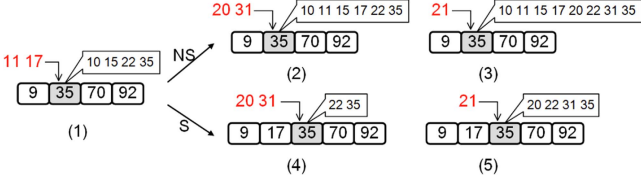


Fig. 9. Impacts of splitting and non-splitting method. A box represents a region, and the red numbers are the KV pairs to be inserted.

avert such additional overhead. Notably, FGKV does not have a cluster at each level. There is a large amount of data involved in each compaction blocking the foreground thread for a long time when we have one cluster per level. Besides, the generated patch is bound to be large, which gives rise to a relatively large amount of garbage data in the patch, but its proportion of the entire patch has not yet reached the threshold. Therefore, FGKV does not perform a *patch*⁺ compaction in this situation, and garbage data are store on disks for a long time before being removed – reducing disk utilization.

In addition, we introduce a threshold T_{cs} to denote the maximum size of a cluster. The *cluster* compaction mode is invoked once the *SizeRatio* of a cluster exceeds threshold T_{cs} . *SizeRatio* is the ratio of the a cluster size to that of its patch 0. All the regions in this cluster are marked as hit regions, and all patches are marked as hit patches. After *cluster* compaction, new clusters are built, and FGKV evenly partitions data in a cluster according to a default KV region size (e.g., 4 KB or 8 KB) instead of the key range. The reason is that there are existing regions that have no data when the key range is referred to as the basic to evenly divide the cluster under skewed workloads (e.g., Zipfian distribution-based workloads). Then, this scheme ensures that each region initially encompasses approximately the same amount of data. The new KV regions form new patches in the default patch size (4 MB). Each patch is packaged into a new cluster as its patch 0. After a redirection table for each new cluster is constructed, the compaction procedure deletes all patches and the redirection table of the source cluster. Then, garbage collection and reallocation of cluster data are accomplished. In the CC process (see Fig. 8), new clusters 1 and 2 are generated after a *cluster* compaction is performed on the source cluster. The source cluster is subsequently deleted.

4) *Splitting*: The key idea behind the FGC mechanism is to only rewrite the hit regions of clusters to suppress the amount of compaction data. Unfortunately, as the number of *patch/patch*⁺ compaction grows, another crucial problem is surfacing. In Fig. 9(1), (2), and (3), compaction data volume increases as the same region is repeatedly hit. After two rewrites issued to region 35, the third compaction requires twice data volume to be processed as the first compaction – such an expanded data volume weakens the advantages of FGC. To address the aforementioned problem, we fashion a splitting optimization: when the data volume in a region after compaction reaches the threshold (e.g., four KV pairs in Fig. 9), the compaction procedure splits large regions into multiple small ones (region 17 and region 35 in Fig. 9(4) and (5)) that have independent KR region and KV

region. Afterward, this procedure renews the change information of the regions to the redirection table. The splitting method in FGKV warrants small data volume of hit regions, which fully utilizes the advantages of the FGC.

D. Compaction Mode Selection

Patch compaction, being adept at reducing WA, may bring forth spatial amplification. Prematurely, *patch*⁺ compaction and *cluster* compaction tend to weaken the performance of LSpM-tree. Thus, it is of paramount importance to elect a suitable compaction mode for each cluster, aiming to well balance WA, read/write performance, and spatial amplification. When a compaction is kick-started, a compaction task is partitioned into multiple sub-tasks, each of which (CT_i) encompasses a cluster (C_i) of C_l and a portion of KV pairs of C_u pertaining to key range of C_i . Given sub-task CT_i , LSpM-tree, according to following steps, judiciously elects a compaction mode and sets the state sets of CT_i .

Step 1: If the *SizeRatio* of C_i in CT_i is larger than threshold T_{cs} , LSpM-tree sets CT_i to the *cluster* mode. The metadata of all regions of C_i enter *hit region set* R . All patches of C_i are inserted into *hit patch set* P . Then, Step 4 is the next move; otherwise, the procedure goes to Step 2.

Step 2: LSpM-tree gears sub-task CT_i to the *patch* mode, marking hit regions that are added into R (see Fig. 7), set P to empty. Then, the procedure goes to Step 3.

Step 3: With R , LSpM-tree obtains the proportion of invalid data in each patch of C_i . If the ratio of invalid data exceeds threshold T_{ir} , LSpM-tree fixes up CT_i to the *patch*⁺ mode, adding this patch to P . After processing all patches, LSpM-tree incorporates the rest regions of patches in P to R . Step 4 will be the next move.

Step 4: The compaction mode identifier of sub-task CT_i , the hit region set R , and the hit patch set P are determined, and compaction procedure is ready to perform compaction based on the above.

IV. IMPLEMENTATION DETAILS

FGKV is implemented based on LevelDB. The extension of RocksDB is outside our radar because RocksDB is extended with a handful of modules and functions compared with LevelDB. We find that most additional components in RocksDB exceed the scope of FGC. To avoid side impacts of irrelevant contents on FGC, we implement FGC using the concise LevelDB (see <https://github.com/AHU-IACS/FGKV>). FGKV implements the APIs of LevelDB, entailing that FGKV is readily applied to the other KV storage systems as an underlying engine.

FGKV is mainly composed of a compaction advisor, a granularity manager, a merger, and a cluster manager. Each module completes a set of operations and communicates with the other modules to jointly realize the FGC mechanism, see Fig. 10. The compaction advisor divides a compaction task into multiple sub-tasks and determines the compaction mode and state sets for each sub-task. The granularity manager sends working signals to other modules and completes region redirection and splitting work. The output of granularity manager is the region update set

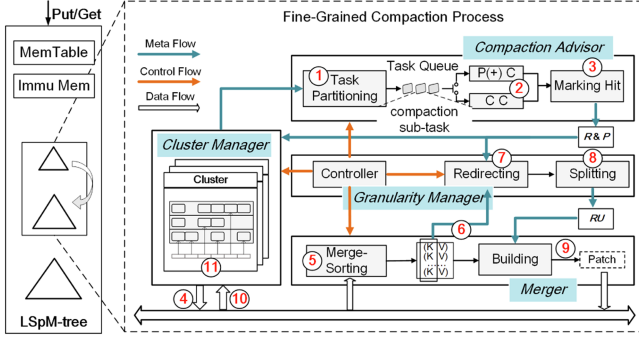


Fig. 10. Overall system design of FGKV.

RU , which will serve as the basis for generating new KV regions. The merger module performs the actual merge-sort operations, then it writes the sorted KV pairs to each new KV region and assembles them into patches according to the regulations in set RU . The cluster manager module mainly manages the organization and storage of data in the LSpM-tree. It needs to update the region redirection table of each cluster involved in compaction according to set RU to ensure data consistency in a cluster.

A. FGC Workflow in FGKV

Fig. 10 depicts the FGC workflow orchestrating 11 steps: ① to partition the compaction task into multiple sub-tasks; ② to choose an appropriate compaction mode for each sub-task; ③ to determine the hit regions and hit patches (obtain set R and set P); ④ to read valid data corresponding to the hit regions to memory; ⑤ to perform the merge-sort operations; ⑥ to collect metadata information of sorted KV pairs; ⑦ to redirect sorted KV pairs to the corresponding hit regions according to the metadata in step ⑥; ⑧ to perform the splitting operations (obtain set RU); ⑨ to generate new KV regions and assemble them into patches; ⑩ to organize new patches into LSpM-tree; ⑪ to update cluster metadata and redirection tables.

Algorithm 1 displays the compaction process in FGKV. We set an identifier for each compaction sub-task. In the *patch* compaction, FGKV decides whether the proportion of invalid data in each patch exceeds threshold T_{ir} , followed by setting compaction identifier to *patch*⁺ (Lines 8–14). After the merge-sort operation, the compaction procedure regulates whether to deposit the generated patches into current clusters or to encapsulate the patches into new clusters according to the compaction identifier (Lines 19–20 and 23–25). Thereafter, the hit patches (in the *patch*⁺ compaction mode) or the source cluster (in the *cluster* compaction mode) is deleted in the clear-up phase (Lines 21 and 26).

B. Bloom Filter in FGKV

Typically, LSM-trees use Bloom filters to avoid I/Os when performing point queries for tuples that do not exist. We also integrate the Bloom filter in FGKV. A cluster contains multiple patches. Each patch only embraces the KV pairs involved in the

Algorithm 1: Compaction Process in FGKV.

Input: Clusters triggered compaction: C_u

```

1  $C_l \leftarrow \text{GetOverlappedLowerClusters}(C_u)$ ;
2  $CT \leftarrow \text{PartitionCompactionTasks}(C_u, C_l)$ ;
3 foreach  $CT_i \in CT$  do
4   if  $\text{SizeRatio}(CT_i) > T_{cs}$  then
5      $\text{SetCompactionMode}(CT_i, \text{CLUSTER})$ ;
6   else
7      $\text{SetCompactionMode}(CT_i, \text{PATCH})$ ;
8    $R_i \leftarrow \text{MarkHitRegions}(CT_i)$ ;
9   if  $\text{compaction\_mode} == \text{PATCH}$  then
10    foreach  $\text{patch}_j \in CT_i$  do
11      if  $\text{InvalidDataRatio}(\text{patch}_j) > T_{ir}$  then
12         $R_i \leftarrow \text{SelectLeftRegions}(\text{patch}_j)$ ;
13         $P_i \leftarrow \text{patch}_j$ ;
14         $\text{SetCompactionMode}(CT_i, \text{PATCH}^+)$ ;
15   $KVs \leftarrow \text{AccessValidKVRegions}(CT_i, R_i)$ ;
16   $\text{SortedKVs} \leftarrow \text{MergeSort}(KVs)$ ;
17   $KV\text{Meta} \leftarrow \text{CollectKVMeta}(\text{SortedKVs})$ ;
18  if  $\text{compaction\_mode} \neq \text{CLUSTER}$  then
19    /* Only patch/patch+ compaction performs splitting */
20     $RU \leftarrow \text{SplitRegion}(CT_i, KV\text{Meta}, R_i)$ ;
21     $\text{BuildPatch}(CT_i, \text{SortedKVs}, RU)$ ;
22     $\text{RemovePatch}(CT_i, P_i)$ ;
23  else
24     $RU \leftarrow \text{DefineNewRegions}(CT_i, KV\text{Meta})$ ;
25    foreach  $RU_j \in RU$  do
26       $\text{BuildCluster}(\text{SortedKVs}, RU_j)$ ;
27     $\text{RemoveCluster}(CT_i)$ ;
28 LogAndApply;

```

compaction. FGKV builds a Bloom filter for each new patch and adds a record to the corresponding Bloom filter to indicate whether the key exists or not when the merged key-value pairs are written into the patch. Thus, each cluster is comprised of multiple Bloom filters, which exist in each patch file in the cluster.

In FGKV, newly generated patch files in each compaction procedure only contains the KV pairs in the hit regions, and these KV pairs have already been accessed. Therefore, FGKV does not read data in the non-hit regions, and the data does not exist in new patches. Thus, FGKV directly calculates the hash value of each key in the new patch to generate a Bloom filter that only targets the key-value pairs participating in the merge-sort. FGKV bypasses complex filter management operations. Consequently, there is no extra memory space cost in FGKV to merge existing filters for building filters.

In point queries, FGKV first determines which cluster's key range the target key is locate at each level. Then, the region redirection table in the cluster is accessed to acquire the patch that stores the valid KV region corresponding to the target key. After achieving the patch information, FGKV reads the Bloom filter of the patch. When the filter indicates that the target key may exist in the patch, FGKV accesses the valid KV region corresponding to the target key. Therefore, FGKV provides high random-write throughput while also allowing for high-throughput point queries.

TABLE III
CHARACTERISTICS OF WORKLOADS

Workload 1				
Type	Feature	Key Size	Value Size (1 KB)	
Load 20 GB	100% writes	16 B	1x, 2x, 4x, 8x, 16x, 64x	
Workload 2 (YCSB)				
Type	Feature	Key Size	Value Size	Count
Load	100% writes	16 B	4 KB	5,000,000
A	50% Reads, 50% Updates			
B	95% Reads, 5% Updates			
C	100% Reads			
D	95% Reads, 5% Inserts			
E	95% Range Queries, 5% Inserts			
F	50% Reads, 50% Read-Modify-Writes			
Workload 3				
Type	Feature	Key Size	Value size	Count
Sp1c	Load+A+B+C+D+E+F	16 B	4 KB	35,000,000
RU	Load+R0U1~R1U0			

V. EXPERIMENTAL ENVIRONMENT

In this section, we introduce workloads, settings, and the alternative KV stores examined in the experiments.

Workload Conditions: We apply YCSB [39] as a workload generator, constructing a diversity of workload conditions listed in Table III. All the YCSB workloads follow the *Zipf* distribution with a Zipfian constant of 0.99 – a default setting in YCSB. Workload 1 randomly writes 20-GB KV pairs under different value sizes. Workload 2 includes typical YCSB workloads - Load, A, B, C, D, E, and F to represent different application scenarios. Workload 3 includes two workloads (Sp1c and RU) with dynamical read/write ratios. Sp1c used in ALDC [40] combines all above typical YCSB workloads. In the typical YCSB workloads except for *Load*, data are initially loaded; then, workloads are replayed based on the loaded data. For example, YCSB B performs 5,000,000 read-write-mixed operations (95% read 5% update) on the basis of Load 20 GB (Load+B). Likely, Sp1c is Load+A+B+C+D+E+F. Inspired by Sp1c, we configure workload RU. After loading 20-GB data, the update operations are executed. Then, the read ratio gradually surges to 100% from 0% with an increment of 20% in each phase, which executes 5,000,000 operations.

Alternative KV Stores: We introduce the alternative KV stores tested in our experiments. *LevelDB* [11] is a baseline solution representing the classic CGC mechanism. *RocksDB* [3], a KV store extended from LevelDB, adopts the CGC mechanism. We opt for RocksDB-6.10.2 with levelled compaction mode. *LDC* [25] slices SSTable at level i that sparks compaction according to the key ranges of SSTables at level $i + 1$. The slices are linked to the SSTables at level $i + 1$. It delays the compaction procedure until the SSTables at level $i + 1$ have sufficient number of linked slices, reducing data rewrite frequency and WA. *ALDC* [40] is an advanced version of LDC, which adaptively adjusts the number of linked slices according to read/write ratios in workloads.

Experiment Settings: We conduct extensive experiments on a Linux server powered by Ubuntu 16.04, Intel Core i7-10700@2.90 GHz, a Samsung 870 EVO 500 GB SSD, a SATA-ST500LM021 500 GB HDD, and 16 GB DRAM. In Section VI,

in all the tested KV stores, the MemTable, SSTable, and the maximum number of open files are 8 MB, 4 MB, and 1,000, respectively. The Bloom filter is configured to 10 bits. These parameters are also the default parameter settings in LDC and ALDC; therefore, these settings make it possible to explore the advantages of the two KV stores. Unlike the default parameters in LevelDB or RocksDB, the configuration parameters of PebblesDB and Wiskey have been already tuned by the developers (see also the extended experiments articulated in Section VII). This setup aims to achieve the optimal performance of the two KV stores in our experiments.

The default configuration – leveling merge policy with a size ratio of 10 – of the tested KV stores are conducive to fair comparison in performance evaluation: this methodology offers three compelling benefits. First, each KV store runs in the same environment, eliminating performance gap resulting from inconsistent parameter configurations in tested KV stores. Second, when the I/O pattern of workloads dynamically changes, it is likely that the current tuning parameters fail in achieving optimal performance of the KV store under new workloads, entailing that the parameters ought to be tuned again. Third, the tuned parameters of one KV store may not be suitable or effective for other KV stores, and experimental results may not be credible because each KV store enjoys its own optimal tuned parameter configuration. In addition, the default parameters deployed in tested KV stores is a viable method to assess the performance of tested KV stores in a large body of works [17], [23], [25], [33], [40], [41], [42]. Therefore, applying the default configuration of the tested KV stores in this study is reasonable and justified. In addition, we use 8-MB *block_cache* in memory in the experiment. We set the data capacity at levels 0 and 1 to 10 MB. Afterward, the data size at level $i + 1$ is 10 times that of level i ($i > 1$). We configure 20 GB and 100 GB KV pairs in workloads. In FGKV, the size of an initial KV region is 4 KB. The KV region size dynamically changes as the compaction is in full swing. To obtain accurate WA, we turned off the compression function in all the KV stores.

VI. PERFORMANCE EVALUATION

In this section, we present the performance of the evaluated KV stores in terms of write performance and WA under random-write and mixed read/write workloads. We conduct experiments more than five times on write throughput and write amplification, and three or more in the extended experiments. The average values of these measures are reported as experimental results. Error bars in the figures represent error ranges of the empirical results. The notation ‘*error ranges*’ is the minimum/maximum measure.

A. Performance Under Various Value Sizes

Fig. 11 unveils the random-write throughput and WA measures of the KV stores under Workload 1. The size of a key (16 B) is smaller than that of a value. Thus, value size, by default, is considered the KV-pair size. The results demonstrate

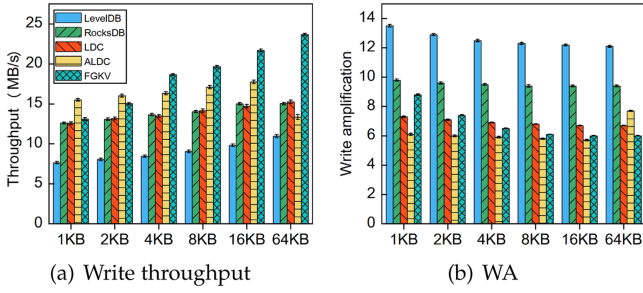


Fig. 11. Impact of different value sizes.

that when value size is large, FGKV has a clear edge over the other KV stores in terms of random write throughput and WA.

Random Write Throughput: FGKV improves the random-write throughput as the value size is enlarged, see Fig. 11(a). We use error bars to show error ranges of the write throughput. Given a small value size (e.g., 1 KB), the write throughput of FGKV is 71.6%, 4.2%, and 4.3% higher than those of LevelDB, RocksDB, and LDC, respectively, but FGKV's throughput is 15.5% lower than ALDC. The write throughput of FGKV is greatly optimized as value size raises. When the value size leaps to 4 KB, FGKV delivers the highest write throughput compared against the other KV stores. With the 16 KB value size, FGKV revamps random write throughput by 121% and 22.2% compared with LevelDB and ALDC, respectively. This is because FGKV only reads/rewrites data of hit regions. The results plotted in Fig. 11 validate our analysis articulated in Section II-B. A large value size lowers the proportion of hit regions in FGKV, thereby cutting back on the amount of compaction data. FGKV is a guaranteed winner in terms of write performance under workloads with large-size KV pairs. The CGC (e.g., LevelDB and RocksDB) selects the same amount of compaction data regardless of the KV-pair size; therefore, the size of KV pairs rarely affects the performance of CGC. When it comes to LDC and ALDC, the number of linked slices largely impacts the amount of compaction data, but the KV pair size rarely affects the number of linked slices. LDC and ALDC share a similar random-write performance.

Write Amplification (WA): Fig. 11(b) shows the WA of the five KV stores under Workload 1. LDC utilizes the link-merge operation to achieve an ideal compaction data ratio in LSM-tree, implying that the ratio of data at levels i and $i + 1$ involved in compaction is 1:1. ALDC lifts the number of linked slices under write-intensive workloads, where the ratio of two-level data can reach 1.2:1. Thus, ALDC obtains the lowest WA. LDC and ALDC, in contrast, are implemented using the lazy compaction mechanism - a solution that manage data updates in a non-real-time fashion. The disadvantage is that the point look-up operations require multiple reads to grab the target value. FGKV, on the other hand, achieves real-time updates as well as reduces WA. The point look-up operation merely accesses the latest KV region to get the target value (see Section VI-B). FGKV has a higher read performance than LDC and ALDC; thus, it is acceptable that the WA of FGKV is not on par with LDC/ALDC in case of small value size.

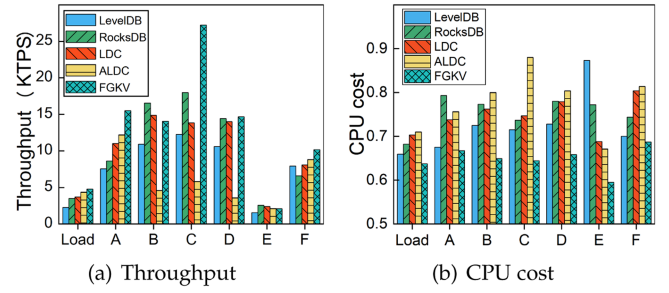


Fig. 12. Performance under YCSB.

The average hit region ratio of FGKV declines as the value size climbs, reducing the amount of compaction data. When the value size is 1 KB, the WA of FGKV and ALDC is 33.1% and 54.3% lower than that of LevelDB, respectively. With 4 KB value size, the WA of FGKV and ALDC reduces by 47.6% and 52.8%. FGKV not only achieves near ideal compaction data ratio (1:1) but also exhibits a higher random-write throughput than ALDC. We conclude that the LSPM-tree and cluster-based FGC mechanism has a performance edge over the link-merge mechanism in ALDC.

B. Performance Under YCSB

In this section, we study the throughput of KV stores under Workload 2 (see Table III).

Among all the five tested KV stores, FGKV has the highest throughput under write-intensive workloads (e.g., Load, A, and F), see Fig. 12(a). Under YCSB A, FGKV improves the throughput by 106%, 79.5%, 40.7%, and 26.9% compared with LevelDB, RocksDB, LDC, and ALDC, respectively. Under the read-intensive workloads (e.g., B and D), FGKV keeps its dominance with a performance improvement of 30% compared with LevelDB. Under workload C, the throughput of FGKV achieves 122%, 51.3%, 96.6%, and 368% improvement over LevelDB, RocksDB, LDC, and ALDC, respectively. The reason is three-fold. First, FGKV integrates all region information of a cluster in a redirection table, which improves data access efficiency. Second, FGKV quickly locates the valid KV regions requested by point look-ups, and accesses KV pairs in this KV region. Third, The cluster size is usually larger than the standard SSTable size, suggesting that the frequency of invoking compaction is low under the read-write-mixed workloads. Thus, FGKV achieves an optimized performance.

Notably, the performance of ALDC is poor under YCSB B, C, and D because these workloads perform a plethora of write operations followed a high read ratio, which makes ALDC unable to adjust the number of linked slices in real-time. ALDC pushes up performance under workloads with a frequent change of read-write ratios. Thus, a single read-write-ratio workload cannot confirm the advantages of ALDC. In Section VI-C, we dive into the performance evaluation of KV stores under workloads with frequently changing read-write ratios. The results plotted in Fig. 12(a) unveil that for workload E, FGKV minimizes WA while improving write performance and maintaining a high-level

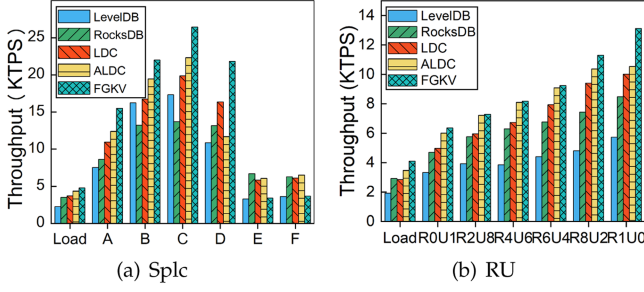


Fig. 13. Performance under workloads with changing read-write ratios.

range query performance. Fig. 12(b) reveals the average CPU cost of KV stores under Workload 2, where FGKV has lower CPU cost than the other KV stores. FGKV curbs CPU costs of LevelDB, RocksDB, LDC, and ALDC by up to 20%, 14.4%, 13.1%, and 16.5%, respectively.

C. Dynamically Changing Read-Write Ratios

We delve in the performance of the KV stores under workloads with read-write ratios that are frequently changing. In a prior work [40], workload Spic (see Table III) is a set of workloads reflecting dynamic changes of read-write ratios. We employ this workload to validate the adaptive adjustment approach of ALDC, aiming to study the dynamic performance of the KV stores.

In Fig. 13(a), under workload Spic, the read ratio varies with a pattern 0%-50%-95%-100%-95% (Load-A-B-C-D). ALDC achieves a high performance in these phases owing to ALDC's adaptive adjustment of the linked slices, improving the performance by an average of 15% compared with the non-adaptive counterpart - LDC. Likewise, FGKV maintains high throughput performance under workload Spic while achieving the highest throughput in the first five phases. Compared with ALDC in the first five phases, FGKV boosts the throughput by 9.6%, 24.9%, 13.2%, 18.4%, and 86.5%, respectively. In phase E and F, FGKV improves the performance of LevelDB by 4.0% and 4.3%.

Fig. 13(b) shows the performance of the five KV stores under workload RU. It can be seen that ALDC gains a high performance because its adaptive adjustment mechanism dynamically varies the number of linked slices with read/write ratio. FGKV takes the lead in the performance competition amid all the phases owing to the LSpM-tree organization accompanied by the three compaction modes. We find that FGKV enhances the throughput of LevelDB, RocksDB, LDC, and ALDC by an average of 113%, 40.5%, 27.4%, and 9.6%, respectively. We confirm that FGKV achieves high performance under workloads with dynamically changing read-write ratios.

D. Latency

In this section, we evaluate the average latency and tail latency of the five KV stores under the widely used workload - YCSB A. Fig. 14 illustrates that LDC and ALDC are capable of lowering latency. This is because SSTables are sliced along with compaction handled in a low-level driven manner, which

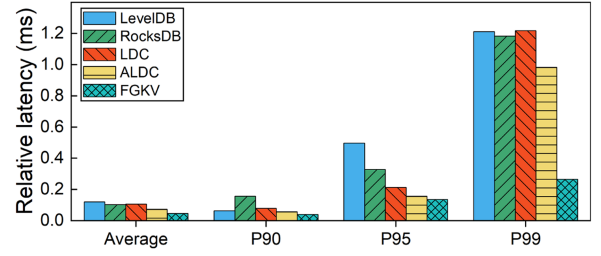


Fig. 14. The average latency and tail latency under YCSB A.

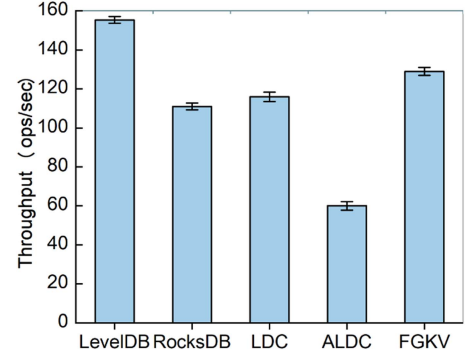


Fig. 15. Performance of long-range queries.

shrinks the granularity of compaction, leading to low average latency and tail latency.

The average latency and the tail latency in all the cases - P90, P95, and P99 - of FGKV are lower than those of the other four KV stores. For example, FGKV shortens the average latency of ALDC by 36.8% while reducing the tail latency of ALDC in P90, P95, P99 by 30.1%, 12.9% and 73.1%, respectively. The reasons behind the improvement offered by FGKV are three-fold. First, FGKV's fine-grained compaction mechanism curbs the amount of compaction data. Second, the *cluster* compaction mode averts large-size clusters. Meanwhile, the splitting method slashes data volume in hit regions. Third, the redirection table enables FGKV to obtain the location of a valid KV region in memory, thereby only requesting one disk I/O to get the target KV pair. Thus, FGKV obviously slashes the tail latency under mixed read/write workloads.

E. Long-Range Queries Performance

In Section VI-B, we employ YCSB-E (95% scan, 5% insert) to explore the KV store performance. The scan operation initially seeks a key, followed by randomly performing 1-100 *Next()* operations. In this test, we gauge the performance of KV stores under workloads with long-range queries. We use *db_bench* to first load 20-GB data with 4-KB key-value pair, then scan 20-GB. The scan operation seeks a random key first and then executes 1,000 *Next()* operations. A scan operation accesses approximately 1,000 disk blocks under this workload. The tested results are plotted in Fig. 15.

Fig. 15 reveals that LevelDB has the highest (155.4 ops/sec) throughput, and FGKV is a runner-up with a throughput of 129 ops/sec. FGKV has a 17% lower long-range-query throughput

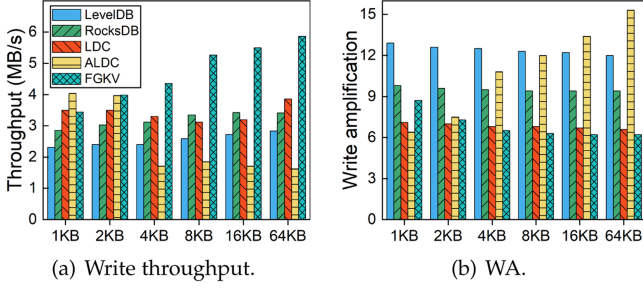


Fig. 16. Impacts of different value sizes on random write performance based on HDD.

than LevelDB but FGKV has a throughput edge over RocksDB, LDC, and ALDC by 16.2%, 10.6%, and 115.0%, respectively. In FGKV, the patch compaction scatters KV regions across multiple patches; therefore, in polygons will have some impacts on the performance of long-range queries, but the magnitude of the impact is insignificant. Because the two KV pairs physically adjacent in an SSTable may not be neighbor in the logical global key orders. In the tested KV stores, the scan operation needs to perform a *Next()* operation on the iterator at each level to obtain the smallest key, so the non-adjacent blocks in FGKV do not have as much impact on long range queries as expected. We observe that RocksDB performs an excessive number of disk I/Os and comparison operations of keys during a long-range query. The capacity of a cluster in FGKV is larger than that of an SSTable in RocksDB, eliminating a lot of disk I/Os to read metadata. Besides, the number of SSTables at Level 0 of RocksDB is more than that of FGKV, which leads to more comparisons in RocksDB. Therefore, FGKV has an edge in terms of long-range query performance over RocksDB. LDC and ALDC perform *Next()* operations on each linked slice; in contrast, FGKV simply reads the region, including the latest data once, according to the redirection table. Thus, FGKV maintains its advantage over the two KV stores in the context of long-range queries.

F. HDD-Based Random Write Performance

HDDs are widely deployed as storage devices for KV stores under write-intensive workloads Workload 1. Thus, we study the random-write performance of FGKV running on HDDs. Fig. 16(a) and (b) show the throughput and WA of the five KV stores. The random-write throughput of FGKV is much higher than those of LevelDB and RocksDB. The WA measures of LevelDB, RocksDB, LDC, and FGKV under HDDs share the similar trend to that under SSDs.

Notably, the random-write throughput of ALDC drops sharply as value size increases. The reason is that the compaction bandwidth is reduced in low-speed storage devices. ALDC lowers the number of linked slices, which frequently triggers compaction, thereby increasing the amount of rewritten data and impairing write throughput. Similarly, Fig. 16(b) reveals that a large value size gives rise to the increasing WA of ALDC. In contrast, the compaction data size of FGKV is determined by the workload characteristics. The proportion of hit regions under the same

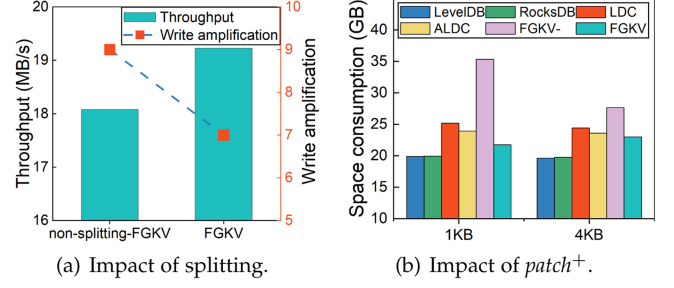


Fig. 17. Impacts of splitting and $patch^+$ compaction.

workload is basically unchanged, so the compaction data size is not nudged - a high robustness featured by FGKV.

VII. EXTENDED EXPERIMENTS

We study the impacts of critical parameters in FGKV on the system performance. The findings obtained from the extended experiments enable users to elect optimal parameter values to find tune performance in various workloads.

A. Impact of Splitting Method

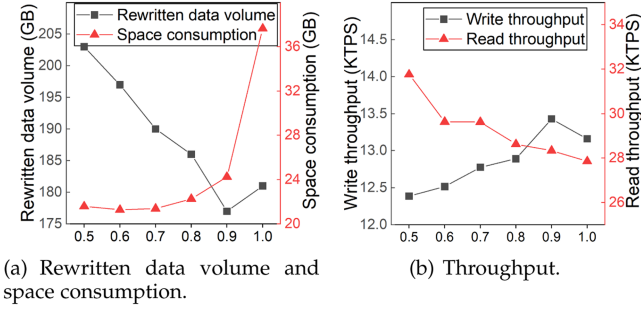
Section III-C4 demonstrates the splitting method of FGKV affects compaction data size. We study the performance of non-splitting FGKV driven by a 20-GB random-write workload with 4-KB value size. *Non-splitting-FGKV* denotes FGKV's variant in which the splitting method is disabled.

In Fig. 17(a), compared with non-splitting-FGKV, FGKV cuts back on WA by 21.7% while boosting the random-write throughput from 18.08 MB/s to 19.23 MB/s. Because data in a KV region is contiguous, the latency to sequentially access to both small- and large-size KV regions is similar. Thus, FGKV presents a higher degree of WA decrement than that of write throughput increment. A lower WA can significantly extend device lifetime. Thus, the findings prove that the splitting method plays a critical role in FGKV.

B. Impact of $patch^+$ Compaction

Recall that we design the $patch^+$ compaction in FGKV for an early active garbage collection, which suppresses space consumption of FGKV and the footprint cost of the cluster management. We study the impact of $patch^+$ compaction on the performance as follows. Fig. 17(b) shows the space consumption of KV stores under 20-GB random-write workloads with 1- and 4-KB values.

It is noteworthy that FGKV is equipped with the $patch^+$ compaction mode, whereas FGKV- denotes a variant of FGKV where the $patch^+$ compaction mode is turned off. With a 1-KB value, FGKV conserves space consumption by 59% compared with FGKV-. FGKV has a 10% higher space consumption than LevelDB and RocksDB, but FGKV saves ALDC's space consumption by 5%. The $patch^+$ compaction removes patches with a high proportion of invalid data, achieving a high proportion of valid KV regions in each patch. In Section VII-C, we study the threshold of the proportion of invalid data T_{irr} in FGKV.

Fig. 18. Impact of T_{ir} .

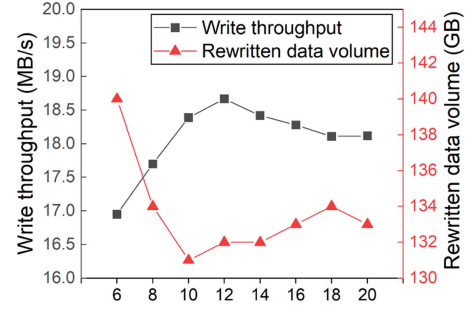
Given a 4-KB value, FGKV saves space consumption by 16.8% compared with FGKV-, which has a lower reduction degree than that of 1 KB. This is because a small value size leads to a large proportion of hit regions, which easily sparks the $patch^+$ compaction.

C. Impact of Invalid Data Ratio Threshold (T_{ir})

Threshold T_{ir} affects the frequency of $patch^+$ compaction in FGKV. It is evident from Section VII-B that 1-KB value as a more obvious impact on $patch^+$ compaction compared with 4-KB value. To clearly illustrate the impact of T_{ir} on FGKV, we apply a 20-GB random-write workload with 1-KB value size in a test.

We observe four phenomena from Fig. 18. First, the space consumption goes up with an increasing T_{ir} . Second, the rewritten data volume declines. Third, the write throughput increases. Forth, the read throughput slightly drops. This is because a large T_{ir} suppresses the frequency of $patch^+$ compaction. Thus, the average number of patches in a cluster hikes, and the proportion of invalid data in each patch becomes high. A portion of invalid data cannot be reclaimed in real-time, causing a large space consumption, see Fig. 18(a). The $patch^+$ compaction selects the remaining valid data in hit patches, even though there are non-hit regions. The $patch^+$ compaction slightly enlarges compaction data volume compared with the $patch$ compaction. With a large T_{ir} , patches have more invalid KV regions reducing the frequency of $patch^+$ compaction. Meanwhile, a high-efficient garbage collection is made by the virtue of switching $patch$ compaction to $patch^+$ compaction. A large T_{ir} threshold reduces compaction data and achieves higher random write throughput. In each cluster, valid data is distributed across multiple patches; therefore, a large number of *Get* operations in a cluster must access multiple patches, impacting read performance. Fig. 18(b) reveals that an increasing T_{ir} marginally decreases the read throughput of FGKV.

By and large, optimal threshold T_{ir} depends on multiple factors. This threshold should be adjusted to meet application requirements. A large T_{ir} should be set under the requirement of high-random-write throughput and low WA in applications. In case of limited storage space or a stable read performance is required, we can set up a low T_{ir} .

Fig. 19. Impact of T_{cs} .

D. Impact of Cluster Size Threshold (T_{cs})

To study the impact of T_{cs} on FGKV, we employ a workload with write-random 20-GB data volume and 4-KB value size. As shown in Fig. 19, the random-write throughput of FGKV increases and then decreases as T_{cs} goes up, while the data volume to be rewritten presents the converse trend. Recall that T_{cs} determines the frequency of the *cluster* compaction (Section III-C). A large T_{cs} implies that a cluster performs multiple $patch/patch^+$ compactions before the *cluster* compaction, thereby reducing compaction data. However, an oversized cluster invoking compaction leads to an increasing amount of compaction data, which inflates WA. Thus, Fig. 19 shows that FGKV achieves the highest throughput as well as the smallest rewritten data volume when T_{cs} reaches 12. Subsequently, increasing T_{cs} degrades the write performance of FGKV. When it comes to the point look-ups performance, FGKV attains the latest location of valid KV regions via the region redirection table and; then, the target KV pair is accessed. Therefore, the cluster size has little impact on the point look-up performance.

E. Comparing With PebblesDB and WiscKey

In modern big-data environments, frequent range query operations are performed on databases for data analysis, where the performance of scan operations is paramount and critical. Therefore, KV stores are expected to maintain high performance under a large number of range query operations. For this reason, we compare FGKV with two leading-edge KV stores, WiscKey [21] and PebblesDB [17], under random-write and range-query workloads. We make use of *db_bench* to randomly construct 5,000,000 write operations followed by performing 5,000,000 scan or mixed write-scan operations (50%-50% and 10%-90%). The KV pair size is 4 KB and the scan length is 1,000. Fig. 20(a) and (b) show the performance of the three KV stores under a 100% random write and 100% scan workloads, respectively. Fig. 20(c) sketches the throughput of the KV stores under the mixed write-scan ratios of 50%-50% and 10%-90%, respectively.

WiscKey deploys a key-value separation strategy, where only merge-sort operations are performed on keys in compaction. Thus, WiscKey has the highest performance under a random write workload compared to the other two without key-value separation; however, it achieves the lowest throughput under

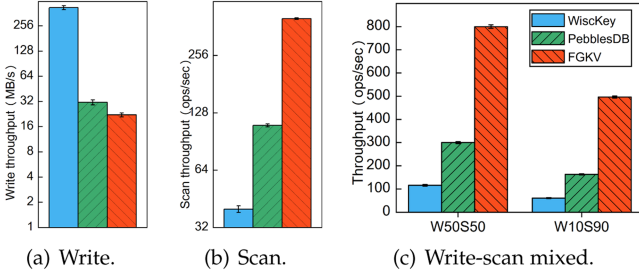


Fig. 20. Performance balancing of write and scan operations among FGKV, PebblesDB, and WiscKey.

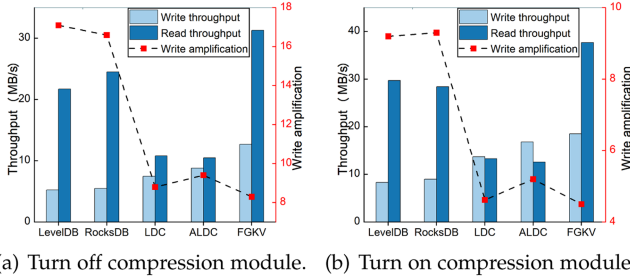


Fig. 21. Performance of KV stores with compression module under 100-GB dataset.

workloads including scan operations. FGKV boosts the throughput of PebblesDB and WiscKey by a factor of 2.3 and 7.6 under a mixed workload of 50% random write and 50% range query. FGKV optimizes the throughput of PebblesDB and WiscKey by a factor of 3.0 and 8.2 when the scan ratio is raised to 50%. The experimental results confirm that FGKV gains a good write/scan balance while enhancing write performance; thus, FGKV is an ideal choice under mixed write/scan workloads.

F. Impact of Large-Size Dataset and Compression

In this section, we make use of the fillrandom in *db_bench* to generate a 100-GB dataset, where encompasses 25,000,000 4-KB-sized KV pairs. We issue a total of 25,000,000 readrandom operations. We set the compression ratio to 0.5. Fig. 21(a) sketches the throughput and WA of KV stores with compression disabled. We evaluate the performance of FGKV when the snappy compression is enabled under fillrandom in *db_bench*, see Fig. 21(b). The bars in the figures represent the throughput, and the lines capture the WA.

Large-Size Data Volume: The 100-GB data, in this test, covers levels 0–5 in the LSM-tree. Fig. 21(a) shows that FGKV exhibits an advantage over the other KV stores in terms of random write/read performance. For instance, the random write performance of FGKV is 142.0%, 131.0%, 69.4%, and 44.0% higher than those of LevelDB, RocksDB, LDC, and ALDC, respectively. FGKV’s random read throughput is 44%, 27.8%, 190.0%, and 198.0% higher than those of LevelDB, RocksDB, LDC, and ALDC, respectively. Meanwhile, the WA of FGKV, being 8.3, is the lowest among its peers: FGKV’s WA measure is 51.5%, 50.0%, 6.3%, and 12.7% smaller than those of LevelDB, RocksDB, LDC, and ALDC, respectively. Interestingly, ALDC

performs compaction more frequently under the workloads with high read ratios; thus, ALDC has higher WA than LDC under this workload.

Compression Module: The compression module is often used in modern KV stores to reduce the data volume written to disks and curb WA. Notably, the compression module works out of the box with FGKV. The snappy compression module has been integrated into LevelDB and RocksDB with the compression-module option to be turned on or off. In LevelDB, RocksDB, and FGKV, the snappy compression module compresses data blocks governed by the compression algorithm. The compression procedure is invoked after a data block is originated; therefore, the compression does not affect the compaction procedure. We turn on the snappy compression and set the compression ratio to 0.5. Fig. 21(b) reveals that compression-enabled FGKV outperforms the other KV stores. After switching on the compression option, FGKV still maintains the highest performance among its peer KV stores. We reckon that turning on compression has little impact on the performance comparison of the KV stores; meanwhile, this strategy just cuts back the amount of data that the system eventually writes to disk.

VIII. DISCUSSION

Crash Consistency: The FGKV system, in case of a crash, recovers failures with a fresh restart. FGKV is derived from LevelDB - a well designed system with a recovery mechanism. To recover the system, we extend the original recover mechanism using the following two operations in FGKV: (1) associating each cluster with its patches and (2) rebuilding the region redirection table for each cluster. In the first operation, FGKV records new and deleted patch numbers of an associated cluster in the *Manifest* after each compaction. In the latter one, FGKV associates the maximum keys of the hit regions in each sub-compaction task with the newest patch number and the KV region offset; meanwhile, $\langle \text{key}, \text{patch number}, \text{KV region offset} \rangle$ is registered in *Manifest*. Thus, when the system restarts, FGKV traverses *Manifest* to apply the update information of the current version, thereby recovering the crash.

Memory Overhead: The *patch* compaction causes invalid data to remain on disks, so the problem of space amplification is not a surprise. The *patch*⁺ compaction early gives rise to garbage collection, achieving the proportion of valid data in patches. In the experiment, we reckon that space consumption of FGKV is around 10% larger than LevelDB and RocksDB, but FGKV mostly improves the performance of KV stores in terms of write throughput and WA.

A region redirection table in FGKV registers the location of valid data in each region; thus, each region needs a redirection entry. The format of one entry in the region redirection table is $\langle \text{key} \rangle \langle \text{patch number} \rangle \langle \text{KV region offset} \rangle$. We set the key size and the patch number to 16 B and 1 B. Notably, one byte represents up to 256 patches, which is sufficient for a cluster. The KV region offset is 8 Bytes. If there are D Bytes data being inserted into FGKV, the initial region size will be configured to R Bytes. The space occupied by one redirection entry is 25 Bytes, and the memory overhead of the region redirection

TABLE IV
MEMORY OVERHEAD OF REDIRECTION TABLES

Data volume	10 GB	20 GB	30 GB	40 GB	50 GB
Memory overhead	74 MB	136 MB	172 MB	183 MB	191 MB

table is measured as $(\frac{D}{R} \times 25)$ Bytes. The region redirection table of FGKV in each cluster is stored on disk instead of main memory. To mitigate memory overhead, the redirection tables are loaded to memory when clusters are accessed. To ensure read performance, we use the LRU algorithm to keep the most recently accessed redirection tables in main memory, where there are redirection tables for 1,000 clusters being stored simultaneously.

We also conducted experiments to study the memory space occupied by the redirected tables in FGKV. We use the fillrandom of *db_bench* randomly write 10 GB, 20 GB, 30 GB, 40 GB, and 50 GB data, and the KV pair size is 4 KB: the space occupied by the redirection table has little to do with the KV pair size because it only depends on the number of regions contained in each cluster. The initial cluster size and region size are set to 8 MB and 4 KB. The test results are listed as Table IV. After running FGKV for a period of time, we observe that the average size of valid data in a cluster is approximately 20 MB. Given 4 KB regions, a cluster requires around 5,200 region redirection entries, which cost around 200 KB of main memory. If the redirection table in memory reaches 1,000, the memory overhead becomes 200 MB which is smaller than the 16-GB memory in our tested platform.

Comparison With B-Tree: The write amplification of LSM-tree is gauged as $O(T * L)$, where T and L are the ratio of data volume size between two adjacent levels and the number of levels. Regardless of the changing size ratio, the compaction procedure of FGKV only elects data in KV regions that ought to be updated. When the number of levels in the LSpM-tree is L , the write amplification is capped at $O(L)$. As the size of key-value pairs expands, B-tree lowers write amplification compared with LSM-tree. However, the in-place update mode in B-tree leads to numerous random-write operations, downgrading write throughput [21], [24], [27], [28], [29]. B-tree is primarily applicable to applications with small batches of random reads along with update data. In addition, B-tree's insertion and deletion operations are complex, making it crucial to carefully consider tree balancing.

Various applications store and manage large-scale data in cloud computing environments, where write/read throughput is one of the critical performance metrics for storage engines. In this context, LSM-tree is superior to B-tree by virtue of converting random writes into sequential ones. LSM-tree becomes a mainstream data structure in KV store systems. Moreover, the valid data of B-tree are all stored in leaf nodes, and the increment of the leaf nodes is linearly related to B-tree size. When data volume becomes large, it is unlikely to write multiple synchronized requests to a single leaf node with an increasing number of leaf nodes. Thus, write operations issued to external storage devices (SSD or HDD) become more discrete – affecting system performance. B-tree structure brings out a large number

of random I/Os, whereas LSpM-tree with the sequential-write mode renders high-throughput capacity in HDDs thanks to because performance speedup of sequential writes over random ones. Even though SSD bolsters the performance of B-tree, the in-place update mode of B-tree inevitably causes the coexistence of numerous invalid and valid pages in one physical block, leading to hefty garbage collection and wear-leveling [43], [44].

In contrast, LSM-tree and LSpM-tree eliminate in-place updates on SSDs, and append writes enable LSM-tree and LSpM-tree to curtail the overhead of garbage collection inside an SSD, in which erasure count and WA are lowered [43], [44], [45]. Thus, LSM-tree or LSpM-tree is proved to be a better choice for SSDs than B-tree.

IX. RELATED WORK

Tiering Compaction Policy: An important way to reduce WA is to adopt a tiering compaction policy, which reduces the frequency of compaction by allowing multiple sorted runs to exist in one level. Inspired by skip-list [46], PebblesDB [17] adopts a partitioned tiering design with vertical grouping [2]. It introduces the guard, and groups SSTables according to key ranges. The SSTables in the same guard can have overlapping key ranges, thereby reducing the frequency of merge and WA. LSpM-tree in FGKV uses the leveling compaction policy. A whole group is merged in the vertical grouping scheme, whereas LSpM-tree orchestrates three granularities – region, patch, and cluster – in the merge operation. Thus, LSpM-tree offers a good trade-off in terms of WA, space amplification and the amount of compacted data compare with vertical grouping scheme in tiering compaction policy. In the vertical grouping scheme, each group organizes multiple files logically. KV pairs in an SSTable of a group are internally ordered; however, KV pairs remain unordered within the group. A point lookup operation in a group should access each SSTable continually until the targeted KV pair is obtained. However each cluster in the LSpM-tree is a logically single file and internally sorted. The region redirection table ensures that required data are accessed at most once in a cluster rather than look up each patch file as designed in vertical grouping scheme. Therefore, the cluster outperforms the group in terms of access performance and efficiency. In addition, some recent works [27], [47], [48], [49] also adopt tiering compaction policy to achieve better write performance.

Key-Value Separation: The key-value separation mechanism is a prominent way of mitigating WA. WiscKey [21] stores only keys and pointers of values in LSM-tree, and merge-sort operations are performed on keys rather than moving values amid compaction: WA is significantly reduced. HashKV [22] uses hash-based data grouping to optimize key-value separation, thereby boosting read performance. Atlas [38] applies a key-value separation mechanism in cloud storage. Keys and metadata in LSM-tree are separately stored, keeping data and implementing low-redundancy fault tolerance through erasure code encoding. With the key-value separation mechanism, however, KV pairs are non-sequentially stored on disks, immensely improving the write performance and reducing read performance

such as range queries. Meanwhile, KV stores with the key-value separation should incorporate garbage collection to conserve space consumption.

Compaction Granularity: The compaction granularity was also studied in VT-tree [24] and Spooky [50]. We present the discrepancies between FGKV and the two KV stores.

First, VT-tree performs fine-grained compaction, but its compaction granularity is merely at the block level. FGKV elects an appropriate compaction granularity among the region, patch, and cluster, delivering a good trade-off among WA, space amplification, and garbage collection overhead. Second, VT-tree's stitching technology leads to complex Quotient filter management. However, FGKV directly calculates the hash value of each key in the new patch to construct a Bloom filter that targets the KV pairs participating in merge-sort. FGKV bypasses the complex Bloom-filter management as devised in VT-tree. Third, VT-tree merely employs the key-range overlap among blocks as a criterion to select data in merge-sort, which is likely to incorrectly elect some blocks and spark the superfluous sorting, causing extra I/Os. FGKV elects blocks involved in merge-sort based on the key of each KV pair in compaction, reducing data amount compared with VT-tree. Fourth, VT-tree frequently conducts rewriting fragments to reclaim garbage data and; thus, the foreground write throughput under update-intensive workloads is downgraded. FGKV simultaneously undertakes garbage collection and compaction. In addition, VT-tree's large-scale data causes high memory overhead because of secondary indexes existing in memory. FGKV simply stores the redirection tables that are most frequently accessed into memory, so the footprint is relatively small.

Previous research [45] has proposed a hybrid-merge strategy, in which full merge and partial merge are alternately conducted according to the amount of data in the two-adjacent layers to attain high compaction performance. Spooky also features the full- and partial-merge compaction modes. The partial merge may lead to WA because the key range of compacted SSTables does not perfectly match. Spooky divides data at the lowest level evenly into several files, the key boundaries of which are used to partition the upper-level files. Then, Spooky performs a full merge at the upper levels, and one file is merged at a time at lower levels. Nevertheless, there exist a handful of open issues. At the lowest level, even though the key ranges of these compacted SSTables have a perfect match, the file sizes at two adjacent levels are unequal because data volume at level $i + 1$ is k times that of level i . There will still be non-hit blocks within the files involved in compaction, thereby incurring WA. FGKV, however, employs region for data selection granularity within clusters: fewer regions that need to be updated in lower-level clusters participate in compaction, cutting back the WA caused by non-hit blocks in the Spooky case. FGKV has the advantage over Spooky thanks to FGKV's fine-grained compacted data selection strategy offering lower WA, and FGKV embraces the optimization methods to further gain the performance edge over Spooky.

Filter Optimization: Bloom filters bolster the read performance of LSM-tree, and a large body of prior studies improve the query performance of KV stores through optimizing Bloom

filters. For example, Monkey [51] unfolds that it is not reasonable to allocate memory space for Bloom filters at each level using the same bits per key. Because there are an amount of data at the lower levels of the LSM-tree, most memory is occupied for the Bloom filters while saving a limited number of disk I/Os. Monkey allocates customized memory sizes for the Bloom filter at multiple levels, thereby achieving the lowest false positive rate and improving query performance. In ElasticBF [28], smaller Bloom filters are created for each SSTable, and the number of Bloom filters loaded into memory can be dynamically adjusted according to the access popularity of each SSTable, thereby achieving better point-query performance. Range filters such as SuRF [52] and Rosetta [53] were proposed to enhance range queries by filtering out tables that do not contain any keys in the requested range. SlimDB [54] uses multiple levels of Cuckoo filters [55] to optimize the performance of tiering merge policy. Chucky [56] adopts Cuckoo filters, reducing filtering memory I/Os with a low false rate.

Notably, there exist the other methods to balance WA, read performance, and space amplification in LSM-tree.

For instance, Nova-LSM [31] separates storage from processing and employs a divide-and-conquer approach to divide the key range. Then, Nova-LSM focuses on data transmission, allocating data to different nodes, and optimizing compaction at level 0. In contrast, FGKV aims to revamp the compaction performance; meanwhile, we design three compaction methods to select three-granularity data in clusters to be involved in compaction, thereby reducing WA. GearDB [57] proposes a novel gear compaction algorithm and a new on-disk data layout to achieve both good performance and high space utilization. SkipStore [58] employs skip-tree, which slashes the number of data rewrites by allowing KV pairs to skip several components and storing the KV pairs in a target level. SifrDB [27], applying the split forest model, inherits the advantages of the forest model (e.g., Cassandra [8]), lowering the WA measures. SifrDB also adopts a parallel search algorithm to enhance read performance. LSM-trie [36] uses a prefix tree to organize keys and allows overlapping key ranges in a partition to cut back on WA. TRIAD [42] reduces WA by separating hot and cold data while deferring compaction at the memory, disk, and log levels, respectively. The design primitives of compaction strategies are classified into four categories [59], namely, (1) the compaction trigger, (2) the data layout, (3) the compaction granularity, and (4) the data movement policy. The extensive experiments were performed to analyze multiple compaction strategies in the prior studies, and the experimental results illustrate how the LSM system navigates the compaction design space.

A handful of studies [51], [60], [61] validated that tuning parameters in RocksDB boost system performance. In contrast, the focal point of FGKV is on optimizing compacted data selection and compaction granularity rather than relying on a parameter tuning method. The parameter tuning method, which bolsters the performance of KV stores such as LevelDB and RocksDB, overlooks performance bottlenecks and design issues inside KV stores. The bottlenecks are alleviated by well-crafted data structures, data selection algorithms in compaction [2], [50], [59], [62]. To address this technological gap, our proposed

FGKV – focusing on compacted data selection algorithms and data structures for compaction – is an integral technique that cannot be replaced by parameter tuning methods. Our optimized approach in FGVK is complementary and orthogonal to the parameter tuning methods, and both FGVK and parameter tuning have potential to revamp system performance.

X. CONCLUSION

The compaction procedure, implemented in traditional LSM-Tree based Key-Value stores, reads data in a coarse-grained data selection fashion (SSTable). This technique that is referred to as coarse-grained compaction, which leads to high WA, low throughput, and long tail latency. To tackle the problems of the coarse-grained compaction, we designed a fine-grained compaction mechanism, in which only updated data are processed by merge-sort operations, averting a large number of irrelevant data to be rewritten in storage devices. We devised a novel data organization (LSPM-tree) to favor the fine-grained compaction mechanism, ensuring prominent random read performance and low WA measures. We implemented the fine-grained compaction mechanism-based KV store - FGVK. The results unveil that FGVK reduces WAs, improves read/write performance, and cuts down long tail latency. In addition, FGVK is readily extended to the other LSM-tree based KV stores to serve as a versatile solution for applications in Big Data and cloud computing systems.

REFERENCES

- [1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
- [2] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, Jul. 2019.
- [3] Facebook, "RocksDB, A persistent key-value store for fast storage environments," 2019. [Online]. Available: <http://rocksdb.org/>
- [4] T. G. Armstrong, V. Ponnkanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the Facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.
- [5] Z. Cao and S. Dong, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–224.
- [6] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2021, pp. 33–49.
- [7] G. Huang et al., "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 651–665.
- [8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
- [9] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang, "Closing the B-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression," 2021, *arXiv:2107.13987*.
- [10] V. Jain, J. Lennon, and H. Gupta, "LSM-trees and B-trees: The best of both worlds," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1829–1831.
- [11] LevelDB, "Fast and lightweight key/value database library by Google," 2018. [Online]. Available: <http://code.google.com/p/leveldb>
- [12] T. Harter et al., "Analysis of HDFS under HBase: A Facebook messages case study," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 199–212.
- [13] G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.
- [14] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–14.
- [15] R. Sears and R. Ramakrishnan, "BLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 217–228.
- [16] N. Dayan and S. Idreos, "The log-structured merge-bush & the wacky continuum," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 449–466.
- [17] 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.
- [18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 1–13.
- [19] 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, pp. 1–16.
- [20] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
- [21] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, 2017.
- [22] H. H. Chan et al., "HashKV: Enabling efficient updates in KV storage via hashing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 1007–1019.
- [23] F. Pan, Y. Yue, and J. Xiong, "dCompaction: Delayed compaction for the LSM-tree," *Int. J. Parallel Program.*, vol. 45, no. 6, pp. 1310–1325, 2017.
- [24] 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.
- [25] Y. Chai, Y. Chai, X. Wang, H. Wei, N. Bao, and Y. Liang, "LDC: A lower-level driven compaction method to optimize SSD-oriented key-value stores," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 722–733.
- [26] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, FTL-aware key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 207–219.
- [27] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 477–489.
- [28] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 739–752.
- [29] J. Liang and Y. Chai, "CruiseDB: An LSM-tree key-value store with both better tail throughput and tail latency," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 1032–1043.
- [30] H. Yoon, J. Yang, S. F. Kristjansson, S. E. Sigurdarson, Y. Vigfusson, and A. Gavrilovska, "Mutant: Balancing storage cost and latency in LSM-tree data stores," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 162–173.
- [31] H. Huang and S. Ghandeharizadeh, "Nova-LSM: A distributed, component-based LSM-tree key-value store," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 749–763.
- [32] F. Chang et al., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.
- [33] T. Yao et al., "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 17–31.
- [34] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 505–520.
- [35] 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.
- [36] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 71–82.
- [37] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
- [38] C. Lai et al., "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–14.
- [39] Yahoo!, "YCSB-C, A C version of YCSB," 2019. [Online]. Available: <https://github.com/brianfrankcooper/YCSB>
- [40] Y. Chai, Y. Chai, X. Wang, H. Wei, and Y. Wang, "Adaptive lower-level driven compaction to optimize LSM-tree key-value stores," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 6, pp. 2595–2609, Jun. 2022.

- [41] D. Teng et al., "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 68–79.
- [42] O. Balmau et al., "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 363–375.
- [43] S. Lee et al., "Application-managed flash," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 339–353.
- [44] J. Guo, Y. Hu, B. Mao, and S. Wu, "Parallelism and garbage collection aware I/O scheduler with improved SSD performance," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1184–1193.
- [45] R. Thonangi and J. Yang, "On log-structured merge for solid-state drives," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 683–694.
- [46] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
- [47] H. Amur, D. G. Andersen, M. Kaminsky, and K. Schwan, "Design of a write-optimized data store," Georgia Inst. Technol., Atlanta, Georgia, Tech. Rep. GIT-CERCS-13-08, 2013.
- [48] T. Yao et al., "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.*, 2017, pp. 1–13.
- [49] W. Zhang, Y. Xu, Y. Li, and D. Li, "Improving write performance of LSMT-based key-value store," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst.*, 2016, pp. 553–560.
- [50] N. Dayan, T. Weiss, S. Dashevsky, M. Pan, E. Bortnikov, and M. Twitto, "Spooky: Granulating LSM-tree compactions correctly," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 3071–3084, 2022.
- [51] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 79–94.
- [52] H. Zhang et al., "SuRF: Practical range query filtering with fast succinct tries," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 323–336.
- [53] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2071–2086.
- [54] 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, 2017.
- [55] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, 2014, pp. 75–88.
- [56] N. Dayan and M. Twitto, "Chucky: A succinct cuckoo filter for LSM-tree," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 365–378.
- [57] T. Yao et al., "GearDB: A GC-free key-value store on HM-SMR drives with gear compaction," in *Proc. 19th USENIX Conf. File Storage Technol.*, 2019, pp. 159–171.
- [58] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 961–973, Apr. 2017.
- [59] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and analyzing the LSM compaction design space," 2022, *arXiv:2202.04522*.
- [60] 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.
- [61] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal bloom filters and adaptive merging for LSM-trees," *ACM Trans. Database Syst.*, vol. 43, no. 4, pp. 1–48, 2018.
- [62] W. Kim et al., "ListDB: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 161–177.



Hui Sun received the PhD degree from the Huazhong University of Science and Technology, in 2014. He is an associate professor of computer science with Anhui University. His research interests include computer systems, edge computing, performance evaluation, Non-Volatile Memory-based storage systems, file systems, and I/O architectures.



Guanzhong Chen is currently working toward the MS degree with Anhui University. His main research interests include KV store systems, SSD storage, and intelligent cache management.



Yinliang Yue received the PhD degree in computer architecture from the Huazhong University of Science and Technology, China, in 2011. He is a professor with Zhongguancun Laboratory. His research interests include computer architecture and storage systems. He has more than 10 publications in major journals and international conferences including the *IEEE Transactions on Computers*, *ICDCS*, *IPDPS*, etc.



Xiao Qin (Senior Member, IEEE) received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, in 1992 and 1999, respectively, and the PhD degree in computer science from the University of Nebraska-Lincoln, in 2004. He is currently an Alumni professor and the director of Graduate Programs with the Department of Computer Science and Software Engineering, Auburn University. He won an NSF CAREER award, in 2009. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation.