Scavenger: An I/O-Efficient Garbage Collection schema for Key-Value Separated LSM-trees

Jianshun Zhang[†], Fang Wang[†*], Sheng Qiu[‡*], Yi Wang[‡]
Jiaxin Ou[‡], Baoquan Li[†], Junxun Huang[†], Dan Feng[†]
[†]Huazhong University of Science and Technology, China
[‡]ByteDance Inc.
{shunzi,wangfang}@hust.edu.cn,{sheng.qiu,wangyi.ywq,oujiaxin}@bytedance.com,{bqli,junix,dfeng}@hust.edu.cn

ABSTRACT

The Key-Value Store (KVS) based on the LSM-tree has gained widespread applications in storage systems due to its exceptional write performance. Despite this, it has also faced significant criticism due to the severe write amplification issue caused by compaction. Although the KV-separated LSM-tree reduces compaction overhead, its newly introduced GC operation has become a new bottleneck, resulting in severe space amplification issues. We propose Scavenger, an I/O-efficient garbage collection scheme for the KV-separated LSM-tree. Scavenger optimizes the three critical steps of GC operations, including read, lookup, and write, to minimize I/O overhead during the GC process, thereby accelerating GC execution. Simultaneously, Scavenger employs a space-aware approach to throttle foreground writes, achieving a more balanced trade-off between performance and space utilization. The evaluation results indicate that Scavenger significantly enhances write performance across diverse workloads and achieves lower space amplification than other KV separation solutions.

PVLDB Reference Format:
Jianshun Zhang[†], Fang Wang[†*], Sheng Qiu[‡*], Yi Wang[‡] and Jiaxin Ou[‡], Baoquan Li[†], Junxun Huang[†], Dan Feng[†]. Scavenger: An I/O-Efficient Garbage Collection schema for Key-Value Separated LSM-trees. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The persistent key-value store, a fundamental component of modern storage systems, provides storage services for various data-sensitive applications such as distributed databases [5, 6, 10, 20], file system metadata management [1, 19], and streaming data processing engines [7, 9]. As digitalization advances and the scale of data increases, there is a growing demand for persistent key-value storage in modern data centers. LSM-tree-based key-value stores have gained widespread adoption in the industry due to their exceptional performance among the various persistent key-value storage engines.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

* Corresponding authors.

The core idea of the LSM tree is to convert random user writes into sequential writes by maintaining a write buffer in memory. This approach takes advantage of the superior capabilities of persistent devices for sequential data writing and reading [2], thereby improving data processing efficiency. The LSM-tree facilitates efficient data querying, including point and range queries, by maintaining a hierarchical structure and a globally ordered data layout at each level. However, the background compaction operations employed by LSM-tree to maintain the multi-level structure and guarantee data ordering incur substantial I/O overhead. These operations necessitate frequent reads and writes to the storage device, resulting in severe read and write amplification [24, 28, 31], which has garnered significant criticism.

To alleviate the I/O overhead imposed by compaction, the academic community has proposed various optimization techniques for LSM-trees. The key-value (KV) separation scheme devised by WiscKey [24] is a widely adopted and efficient solution. The fundamental principle underlying KV separation is to store values in a distinct log-structured data structure while maintaining only keys and indexes within the conventional LSM-tree [25]. This approach effectively reduces the size of the LSM-tree and alleviates the overhead associated with compaction. Although KV separation can improve the write performance, it presents new problems, such as poor range query performance [23, 30] and new garbage collection (GC) operations with high I/O overhead [8, 23, 29].

Numerous methods for GC optimization have been proposed in the literature to address these issues. Some studies[23, 30] concentrate on enhancing range query performance through improving data orderliness, while others explore alternative approaches, such as triggering GC through compaction [23] or partitioning values [8, 30] to reduce I/O overhead during GC. However, these approaches still encounter two significant challenges.

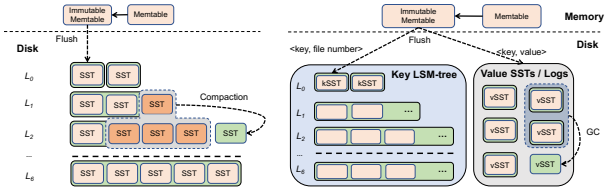
For one thing, existing solutions often overlook the issue of space amplification. Traditional LSM-trees, like RocksDB, employ leveled compaction to tightly control space amplification [13, 15]. Conversely, the KV-separate LSM-trees utilize GC to reclaim space based on the garbage ratio, trading increased space amplification for improved foreground read and write performance. However, this does not mean that space amplification can be disregarded [15]. The recent research [33] shows space amplification is critical for storage costs, especially in embedded systems and cloud computing scenarios. Unfortunately, current KV-separated LSM-trees, like DiffKV [23] and NovKV [29], aim to lower GC costs by triggering GC through compaction but often lead to more severe space amplification due to delayed space reclamation.

For another thing, existing solutions struggle to accommodate diverse workload characteristics. According to Facebook's investigation [6], modern persistent key-value storage systems must address a range of workload types in real-world scenarios, including variable-length and hot-cold workloads. However, in variable-length workloads, the KV-separated LSM-trees, primarily designed for large KVs, often overlook the consequences of small KVs falling below the separation threshold. As the proportion of small values increases, the capacity of the LSM-tree grows, making it challenging to retain most of the LSM-tree data blocks in the cache, resulting in increased overhead for lookup during GC. Besides, in hot-cold workloads, due to the mixture of hot and cold data, performing garbage collection on hot data inevitably leads to frequently reading and writing cold data, significantly reducing the efficiency of GC.

This paper introduces Scavenger, an I/O-efficient GC scheme for KV-separated LSM-trees that realizes balanced performance and space. To this end, Scavenger accelerates GC execution by reducing the I/O overhead of three critical steps in the GC process. First, Scavenger reduces the I/O overhead of GC read operations by performing lazy reads. Second, Scavenger avoids unnecessary record access during GC-Lookup by designing a GC-friendly index storage layout. Third, Scavenger employs a lightweight hotness-aware write strategy to minimize the impact of GC operations on cold data. Furthermore, the Scavenger dynamically schedules GCs and throttles foreground writes to achieve limited space amplification. By accelerating GC execution and dynamically scheduling GCs based on space usage, Scavenger achieves a more favorable balance between performance and space utilization.

Real-world production requirements drive our work. It provides a comprehensive analysis of potential performance bottlenecks in GC operations for KV-separated LSM-trees, distinct from previous research. We are pioneers in addressing the serious issue of spatial amplification in this context, striving to achieve a more balanced trade-off between read-write and space amplification. Our principal contributions are summarized as follows:

- We investigate the impact of GC operations on foreground performance in KV-separated LSM-trees under varying workloads and identify the performance bottlenecks of the GC operations.
- We propose a new storage layout to construct a complete index for all separated values, ensuring GC can only read keys to validate and values lazily after key validation, with minimal I/O overhead to expedite GC operations.
- We propose a new index storage layout that separates keys and records in the index LSM-tree, enabling efficient GC lookup to access index blocks to validate data, reducing the overhead of GC validity checks under variable-length value distributions, and improving the speed of GC operations.
- We propose a lightweight hotness-aware write strategy that accurately identifies data workload hotspots, guides subsequent flush and GC operations and achieves hot-cold data separation to reduce unnecessary GC overhead for cold data.
- We propose a space-aware throttling scheme that effectively manages foreground operations and GC execution to prevent storage leaks, achieve a more balanced trade-off between read-write and space amplification, and provide a fair testing benchmark for KV-separated LSM-trees.



(a) Vanilla LSM-tree (b) Key-Value Separation: TerarkDB

Figure 1: Overview of LSM-tree and TerakDB.

This paper is organized as follows. We begin by analyzing the challenges existing solutions face in Section 2, providing background information and motivations. In Section 3, we delve into the design of Scavenger. Section 4 presents the evaluation results. Next, in Section 5, we discuss related work. Finally, we conclude in Section 6.

2 BACKGROUND AND MOTIVATIONS

2.1 The LSM-tree Basics

The vanilla LSM-tree [26] typically adopts a multi-tree structure, where each tree represents a sorted run, as depicted in the Figure.1a. Popular LSM-trees typically comprise two sorted runs in memory and multiple sorted runs on storage devices. The sorted runs in memory are primarily used for buffering data writes and initial sorting, represented by the Memtable and Immutable Memtable. The sorted runs on storage devices maintain a multi-level structure to guarantee query performance. A single sorted run on a disk consists of one or more SSTables. Within the LSM-tree, the L_0 level consists of multiple sorted runs, each represented by an SSTable, facilitating efficient flush operations. The L_1 to L_N levels correspond to a single sorted run, with multiple SSTables forming a single-level, globally ordered structure for efficient query execution.

Write process. Initially, data requests are recorded in the Write-Ahead Log (WAL) to guarantee crash consistency. Upon insertion, the data is temporarily stored in the memory Memtable. Upon full, the Memtable transforms into an Immutable Memtable, which will be flushed to L_0 as an SSTable (SST). When a layer of L_i becomes full, the LSM-tree schedules the background compaction thread to select a set of overlapping SSTables from two adjacent levels, merge-sort key-value pairs of these SSTables, and create new SSTables that are written to the L_{i+1} level. Compaction involves multiple I/Os of storage devices, leading to significant read and write amplification.

Read process. Query operations are categorized into point and range queries. In terms of point queries, the LSM-tree traverses each sorted run from memory to storage devices in a top-down order. If a sorted run that potentially encompasses the queried key-value pair is identified, a binary search is conducted within it. When querying an SSTable, the LSM-tree employs index and filter blocks to speed up data retrieval. Range queries commence by locating the starting key in each sorted run and then reading key-value pairs within the specified range by moving the iterators of the sorted runs. Ultimately, a collection of key-value pairs will be retrieved.

2.2 Key-Value Separation: A Case Study of TerarkDB

To address the significant write amplification issue caused by compaction operations in vanilla LSM-trees, numerous academic studies [8, 23, 24, 29, 30] have proposed solutions that separate the key and value to reduce compaction overhead in LSM-trees. The industry also has shown strong interest in key-value separation solutions, as evidenced by the adoption of KV-separated LSM-trees in open-source key-value storage engines such as BadgerDB [14], TitanDB [27], and TerarkDB [4]. TerarkDB, specifically, a high-performance key-value storage engine based on the KV separation developed by ByteDance, serves as the underlying storage service for both ByteDance's distributed OLTP database, ByteNDB [10] and its distributed key-value storage, Abase. TerarkDB provides storage engine support for tens of thousands of upper-layer instances in production.

In this paper, we focus on TerarkDB [4, 22] as an illustration of a key-value separation LSM-tree, as depicted in Figure 1b. The central principle of KV separation is to store keys and values in separate locations. Specifically, values are stored in a log-structured structure, while references to keys and values are stored in the vanilla LSM-tree. For workloads with a relatively small key-to-value ratio, KV separation significantly reduces the size of the LSM-tree, reducing the compaction overhead. In contrast to the original WiscKey approach, TerarkDB employs SSTables to store values in an ordered manner, referred to as value SST (simplified as vSST), as shown in the Figure.1b. By adopting this structure, TerarkDB enhances the order of values and improves the performance of range queries. For vSST, TerarkDB schedules background GC threads to reclaim storage space and improve the order of values.

The critical distinction between TerarkDB and other KV-separated LSM-trees lies in the strategy for reducing the index update overhead during the GC process. When performing KV separation, TerarkDB employs a tuple composed of the key and the associated vSST file number, where the value resides, as the index stored in the LSM-tree. Moreover, TerarkDB maintains the inheritance relationship between the latest vSST file number and the outdated vSST file number after GC instead of updating the address of the valid key-value pairs in the LSM-tree index, as most KV-separated LSM-trees do. Therefore, the read operation first queries the LSM-tree to acquire the index tuple and utilizes the vSST file number and the corresponding inheritance tree to retrieve the most recent vSST file number where the value currently resides. As vSST follows the structure of traditional SSTables in LSM-trees, the read operation can retrieve the corresponding data as vanilla LSM-trees. The GC strategy implemented in TerarkDB alleviates write-back overhead and ensures prompt space reclaim.

2.3 Compaction and GC comparison

The compaction operations in vanilla LSM-trees involve reading data, merging sorted data, and writing data, which can negatively impact the bandwidth and performance of the storage device when processing foreground user requests [3]. Meanwhile, the GC operations in KV-separated LSM-tree involve reading, validating, and writing data in value SSTs or value logs, consuming storage device bandwidth, and potentially competing with foreground read and write requests. Both GC and compaction impact the performance of a KV-separated LSM-tree. To evaluate the impact of both the inherent compaction operations on KV-separated LSM-trees and the newly introduced GC operations on foreground read and write performance, we conducted experiments utilizing the widely used KV-separated LSM-tree TerarkDB with a read-while-writing workload. This experiment monitored the disk bandwidth utilization of background and foreground threads for over 2000 seconds. Section 4.1 outlines the experimental setup, where foreground and background threads were set to 32 (maximum 24 threads for the GC to schedule GC frequently).

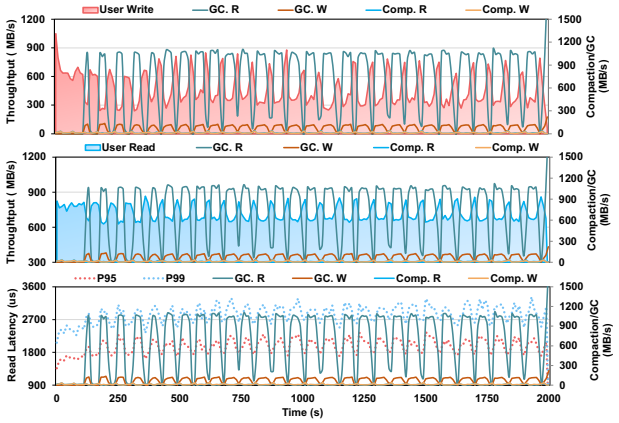


Figure 2: Performance Fluctuations caused by GC

As shown in Figure. 2, the results indicate that KV-separated LSM-trees significantly reduce compaction overhead through value separation. Despite this, there are still fluctuations in read/write throughput, read latency, and performance stability. These fluctuations correlate with variations in the GC operation bandwidth, showing that GC operations compete with foreground read and write threads for storage bandwidth. As a result, the bandwidth of GC operations significantly impacts foreground read and write performance. In short, the performance bottleneck for KV-separated LSM-trees shifts from the original compaction operations to the GC operations. How to reduce GC I/O overhead and mitigate resource contention with foreground threads has become an urgent issue in KV-separated LSM-trees.

2.4 Trade-offs of Existing Solutions

Compared to traditional LSM-trees, the performance enhancement in KV-separate LSM-trees is attributed to their garbage collection (GC) operations, which achieve space reclamation with reduced read-write amplification compared to compaction operations. However, this advantage often results in increased space amplification. To figure out the trade-offs between compaction and GC operations, we evaluated various scenarios: compaction, GC operations triggered at different garbage ratio thresholds (GC-20% and GC-50%, signifying GC activation at 20% and 50% garbage rates respectively), and completely bypassing GC (No-GC) within TerarkDB. Our primary focus was to evaluate the foreground update performance and space amplification.

As depicted in Figure. 3, the results indicate that while GC operations exhibit higher throughput than compaction operations,

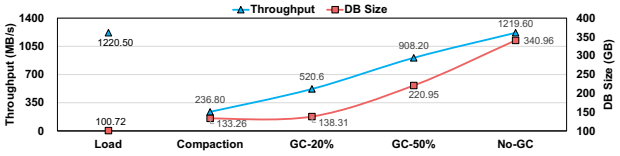


Figure 3: Tradeoffs of Compaction and GC

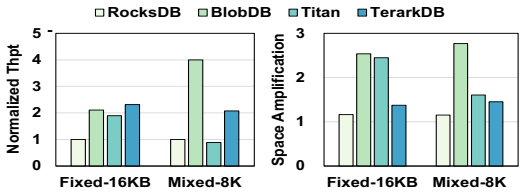


Figure 4: Tradeoffs of Existing Solutions

they also result in greater space amplification. As the garbage ratio threshold increases, indicating a decrease in GC executions, performance improves. Nevertheless, this leads to an increase in space amplification. When the GC is disabled, it behaves like an append-only log structure, leading to optimal performance and maximum space amplification. However, in real-world production environments, space amplification is crucial, particularly in scenarios with cost sensitivity and extensive data volumes. Our proposed approach aims to enhance foreground performance within limited space amplification constraints.

To evaluate the trade-offs of existing solutions in terms of space and performance, as well as their adaptability to diverse workloads, we conducted evaluations on various advanced KV-separated LSM-tree implementations, such as BlobDB, Titan, TerarkDB, and the widely used LSM-tree storage system, RocksDB. We primarily evaluated performance and space amplification under fixed-length (16KB) and variable-length (Mixed) workloads. The variable-length workload, denoted as Mixed, is a typical workload pattern in the internal OLTP database of ByteDance. It features a 1:1 ratio of large values (16KB) to small values (ranging from 100 to 512B and following a uniform distribution). Incremental user updates primarily contribute to smaller values, while larger values stem from updates to original data pages.

Our analysis reveals that the existing KV-separated LSM-tree designs exhibit significantly higher space amplification than RocksDB for all evaluated workloads. The KV-separated LSM tree solutions introduce GC operations triggered by garbage ratios to reduce space amplification. However, they often result in space amplification that exceeds these ratios, disregarding actual disk capacity and total cost of ownership (TCO). For example, Titan under the Fixed-16K workload and BlobDB under the Mixed-8K workload shown in Figure 4 exhibit impressive foreground performance (1.9x and 4x that of RocksDB, respectively). However, their space amplification surpasses other KVSs at 2.45 and 2.53. In contrast, TerarkDB achieves a better trade-off between performance and space across both workloads.

Importantly, KV-separated LSM-trees with Mixed-8K workloads exhibit a reduction in foreground performance compared to Fixed-16K workloads, with instances like Titan performing even worse than RocksDB under Mixed-8K workloads (a decrease from 426.5 MB/s to 206.6 MB/s). Meanwhile, TerarkDB exhibits a foreground performance reduction under Mixed-8K workloads (from 520.6 MB/s to 484.7 MB/s) combined with increased space amplification (1.37 to 1.45). BlobDB performs better under Mixed-8K workloads but incurs severe space amplification of up to 2.77.

In summary, the initial design of KV-separate LSM-trees, primarily designed for large values, overlooked the typical variable-length workloads prevalent in real-world environments. Consequently, the adaptability of existing KV-separate LSM-trees to variable-length workloads is suboptimal, preventing them from effectively balancing space amplification and foreground performance. Our objective is to achieve balanced performance and utilization of space across a diverse range of workloads.

2.5 GC Operations Analysis

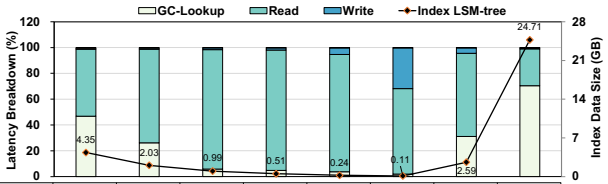
To further investigate the sources of overhead caused by GC operations and the proportion of overhead under different workloads, we take TerarkDB as an example to analyze the workflow of GC operations and then break down the latency of GC operations.

2.5.1 Workflow of GC operations. In traditional key-value (KV)-separated log-structured merge (LSM) trees, such as WiscKey, garbage collection (GC) operations consist of four stages: reading eligible data, verifying its validity, writing valid data to a new value SSTable (vSST), and updating the latest index of the LSM tree. In contrast, TerarkDB stores file number data in the index LSM tree and preserves file number relationships, thus avoiding index updates. The GC process in TerarkDB comprises three steps: reading eligible data, verifying its validity, and writing valid data into a new vSST.

To execute GC read operations, TerarkDB establishes iterators for GC-eligible files and extracts key-value pairs from these files by traversing the entire vSST file. This process is referred to as **Read**. For data validity verification, denoted as **GC-Lookup**, TerarkDB queries the LSM tree-stored value address/reference for a specific key and compares the current GC value address with the LSM tree's stored value address. If the checked value does not match the pair, it is considered garbage and subject to discard. However, if a match exists, the pair is deemed the latest valid data and requires copying and writing into the new vSST, denoted as **Write**.

2.5.2 Performance Bottlenecks in the GC Operations. We conducted a latency breakdown of three stages of the TerarkDB GC process to find the performance bottleneck in the GC process. The evaluations were initiated by loading 100GB of data and then updating 300GB of the base dataset to induce frequent GC operations. We investigated various write request granularities and measured the GC latency under fixed and variable workloads.

As depicted in Figure. 5, the results indicate that **Read** and **Write** latencies increase as the value size increases for fixed-length workloads. **GC-Lookup** is prominent for small key-value pairs. Under fixed large-value workloads, each read and write operation accesses larger data units, increasing the average latency and percentage occupied. Under variable-length workloads, the average latencies and percentages of the **GC-Lookup** surpass those of fixed-length workloads. Upon further evaluation and analysis, we discovered that the index LSM-tree size under the variable-length workloads is significantly larger than that under fixed-length workloads, particularly for workloads following the generalized Pareto distribution



Avg lat.	1K	2K	4K	8K	16K	32K	Mixed-8K	pareto-1K
GC-Lookup	25.57	20.37	7.38	8.06	9.98	10.32	109.41	120.27
Read	28.24	56.57	115.96	156.61	239.81	343.85	227.43	48.84
Write	0.88	1.48	2.58	4.70	23.09	273.56	24.25	2.29

Figure 5: GC Latency Breakdown

(Pareto-1K). We observe a positive correlation between the average **GC-Lookup** latency and the index LSM-tree size. Notably, although the index LSM-tree size under the Mixed-8K workload is comparable to Fixed-2K, the average latency of **GC-Lookup** is increased by 5.4x. Due to the mixed storage layout, we found that its cache hit ratio for **GC-Lookup** is much lower than that under Fixed-2K.

2.5.3 Summary. KV-separate LSM-trees successfully mitigate compaction overhead in traditional LSM-trees but introduce a new bottleneck, GC operations. Despite improving the foreground performance of KV-separate LSM-trees, they often neglect the issue of space amplification. Meanwhile, diverse workloads present challenges in the design of KV-separated LSM-trees, with GC overhead characteristics varying across workloads. We aim to introduce an innovative KV-separate LSM-tree that effectively balances space amplification and performance while accommodating diverse workload characteristics.

3 DESIGN

Our previous analysis revealed three primary sources of overhead in the GC process. We propose an efficient I/O GC scheme named Scavenger to address these issues. The Scavenger scheme was specifically designed to optimize the three critical operations within the GC process. By implementing targeted optimizations, Scavenger aims to reduce the costs of I/O during GC.

3.1 System Overview

Like most KV-separation-based LSM-trees, Scavenger employs a hierarchical structure for data management. It incorporates conventional LSM-tree components such as Memtables, Immutable Memtables, and Write-Ahead Logs, as illustrated in Figure. 6. It separates data into an index LSM-tree for key-index and multiple value SSTs for key-value stored on disk. Scavenger performs key-value separation during background flush operations and maintains order using ordered data structures for all data.

Unlike other schemes, Scavenger employs IndexDecoupledTable for managing small key-value and index data, ensuring physical separation. Scavenger introduces the RecordBasedTable, a GC-friendly structure instead of value logs or block-based tables, for key-value data. Meanwhile, Scavenger introduces the memory cache, DropCache, which efficiently stores frequently written data in the key LSM-tree, thus facilitating hot-cold data separation. Scavenger employs space-aware throttling before writing to limit space amplification.

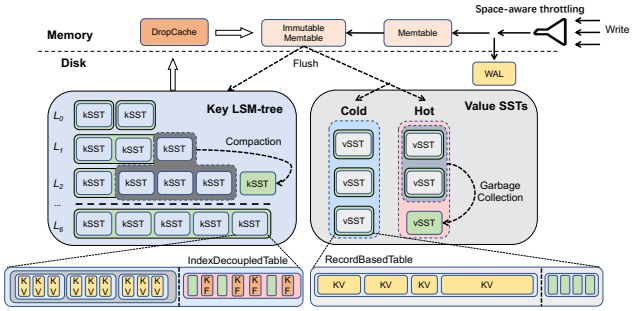


Figure 6: Overview of Scavenger

3.2 Lazy Read of the GC

Our analysis in Section 2.5 reveals that the primary GC overhead for fixed-length large-value workloads originates from GC Read. GC Read experiences increased latencies as the value size increases, with value access costs dominating. From the overall perspective of GC operations, the volume of data accessed during the read step typically surpasses that of the write step. The gap is due to the distinction between the read step, which involves all data in the value SST (vSST), and the write step, which only processes valid data post-validation. Consequently, the read step has accessed significant unnecessary garbage data. Reducing access to garbage data becomes pivotal in mitigating GC Read overhead.

A straightforward approach would be to read data considered valid solely. However, storing adequate validity information as prior knowledge within a limited overhead for large-scale persistent key-value stores is often infeasible. A more practical and feasible approach is to minimize the amount of garbage data read, i.e., to read as little data as possible for validity checks and then read the remaining data after the validity check has passed. In KV-separation-based LSM-trees, the validation (GC-Lookup) typically involves conducting a point query operation on the index LSM-tree for a specific key. Hence, we can validate the entire vSST by examining only the contained keys. After obtaining the validity information, we validate the key-value pairs and avoid unnecessary value data reads for discarded items.

3.2.1 Storage layout of vSST. The efficient retrieval of all vSST keys before GC validity checks (GC-Lookup) necessitates a redesign of the vSST storage layout. Traditional approaches, such as WiscKey's unordered value log, store keys and values in corresponding KV records, making rapid access to all vLog keys challenging, as depicted in Figure. 7. Specific industrial schemes, such as Titan and TerarkDB, adopt a structure similar to RocksDB's SSTable for improved range query performance in the value log, known as BlockBasedTable. The internal index block of BlockBasedTable can facilitate retrieval, but it only points to specific data blocks that may comprise multiple entries. In short, BlockBasedTable only maintains a sparse index for the key-value pairs, preventing a rapid retrieval of all keys.

Scavenger introduces a novel table structure, **RecordBasedTable**, designed explicitly for value storage in KV-separated LSM-trees. Like BlockBasedTable, RecordBasedTable maintains data in an ordered manner and utilizes additional metadata such as Footer, Meta Index, and Filter to enhance data retrieval within the table. Unlike traditional BlockBasedTable, RecordBasedTable organizes values

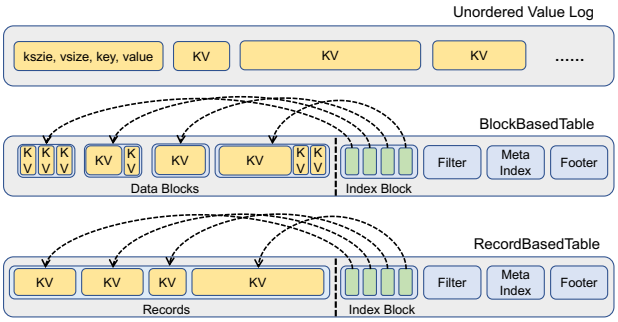


Figure 7: Storage layout of RecordBasedTable

in the form of records (key-value pairs) and creates a <key, offset> tuple as the corresponding index for each record, which is stored in the index block. In essence, RecordBasedTable creates a dense index for the pairs, enabling GC read operations to directly access the index blocks of RecordBasedTable to retrieve all keys without the need to access values. Simultaneously, foreground read operations can efficiently retrieve value addresses directly from the index block and read values without in-block retrieval.

3.2.2 Lazy Read process. The introduction of the **RecordBasedTable** facilitates Scavenger GC operations to access the entire set of keys with minimal I/O. Initially, the GC operation reads the Footer block to obtain the address information of the index block and subsequently retrieves the complete index block from the corresponding address. With the index block, the GC threads can perform validity checks (GC-Lookup) without accessing values. Once a discovers valid keys, the GC threads can proceed to read the corresponding values and execute the subsequent write operations. However, if the key is invalid, the garbage data can be directly skipped, avoiding reading the value corresponding to the garbage keys. Thus, the GC will only read the value once it has confirmed the valid data corresponding to the key. This approach helps minimize read I/O and is known as lazy read.

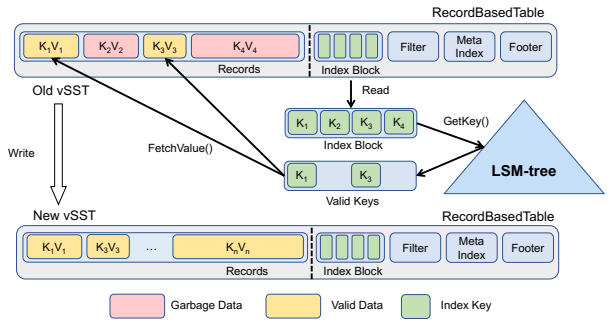


Figure 8: Lazy Read process of GC

Figure. 8 shows an example of the GC process with lazy read. The GC thread initially reads the Index Block of the selected vSST, acquiring key K_1-K_4, and stores the index block in the memory cache with high priority. With all keys, the GC thread invokes the $GetKey()$ function for GC-Lookup to validate the keys. The K_2 and K_4 are identified as garbage data, while K_1 and K_3 are considered valid. Upon completing the reverse lookup, the GC thread shall retrieve the values V_1 and V_3 of valid keys K_1 and K_3 using the address information stored in the index block. The valid key-value pairs K_1V_1 and K_3V_3 are then appended to the newly created vSST's record segment. Upon reaching the RecordBasedTable size threshold, the corresponding IndexBlock and other metadata blocks are generated to finalize the construction of the new vSST.

3.3 Index-Record Separation

The Lazy Read strategy can significantly reduce the GC read overhead, particularly in large key-value workloads. However, our previous analysis in Section 2.5 exposes that GC-Lookup is another significant source of GC overhead, particularly evident in fixed-length small key-value workloads and variable-length workloads. Optimizing GC-Lookup overhead becomes paramount in addressing this challenge. As previously discussed, the essence of the GC-Lookup is the point query operation on the index LSM-tree, enabling optimization techniques such as caching and indexing to be applied to the KV-separated LSM-tree. We aim to investigate how KV-separated LSM trees can more efficiently utilize existing optimization techniques and how to address problems unique to KV-separated LSM-trees.

The evaluations outlined in Section 2.5 demonstrate that the LSM-tree sizes for fixed-length 1KB, 2KB, and variable-length Mix-8K, Pareto-1K workloads surpass the preallocated cache size of 1GB (1% of the dataset). While allocating more memory for caching the index LSM-tree would be advantageous, practical constraints often limit the allocation of substantial memory on high-capacity storage servers, particularly in variable-length workloads like the Pareto-1K distribution. As a result, we have to make the most of the cache space and improve the GC-Lookup efficiency. Unlike traditional point query operations, the GC-Lookup operation does not necessitate accessing value data stored in the index LSM-tree. Instead, it only requires checking the keys for validity information. In existing KV-separated LSM-tree implementations, small KV data and index data for large KV are stored in the same data block. Consequently, the GC-Lookup operation incurs unnecessary data access. Scavenger introduces a novel data table structure termed IndexDecoupledTable to tackle this challenge.

3.3.1 Storage layout of kSST. Some KV-separated LSM trees utilize the default BlockBasedTable structure of open-source LSM tree storage engines such as LevelDB/RocksDB to store both indexes (denoted by KF, <key, file number/address>) and small KV data (also known as Record, denoted by KV, <key, value>) in the same file. This mixed index and key-value storage layout significantly increases the GC-Lookup latency as GC-Lookup reads unnecessary value data. Specifically, GC-Lookup typically only needs to check the indexed data (KFs) stored in the LSM-tree, but the mixed storage layout results in the smallest basic unit of read operations, the data block, containing a large number of key-value data (KVs) irrelevant to GC-Lookup.

Scavenger strives to provide fast access to pure indexes while preserving table data order. A straightforward solution, depicted as the RedundantIndexTable in Figure. 9, involves an additional Index Key Block containing all index data (KF) for the current key SSTable (simplified as kSST). The Index Key Block encompasses all index data items (KF) relevant to the current kSST, enabling GC-Lookup to directly access the Index Key Block without traversing

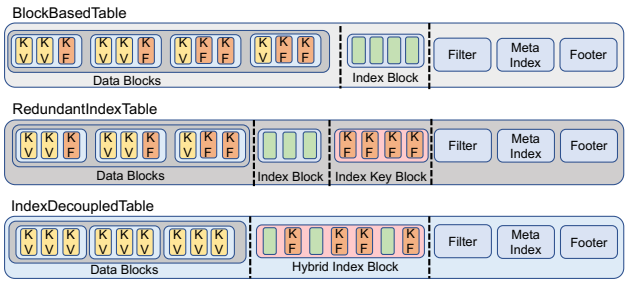


Figure 9: Storage layout of IndexDecoupledTable

the small KV data in the original data blocks. However, this solution introduces extra space and write overheads, involving redundant storage and repeated writing of index-related data.

An alternative approach entails further isolating the Index and Record stored within the index LSM-tree. To this end, Scavenger introduces a more space-efficient table structure called **IndexDecoupledTable**. This structure adopts the conventional BlockBasedTable configuration and enhances the index block. The original index block functions as a sparse index for key-value data, maintaining one index item for each data block. Conversely, the Hybrid Index Block within IndexDecoupledTable maintains both the index of small KV data blocks and the index of large KV (KF), as illustrated. This Hybrid Index Block enables the complete decoupling of KF data from KV data blocks without redundant storage. Data blocks only need to store the corresponding small KV data, ensuring complete physical separation from the index data.

3.3.2 GC-Lookup. With **IndexDecoupledTable**, GC-Lookup operations can perform data validity checks with reduced I/O overhead by directly accessing the Hybrid Index Key Block. Upon obtaining the relevant keys for reclamation, the GC thread can directly invoke the *GetKey* API offered by the index LSM-tree. This API, similar to the traditional LSM-tree *Get* but with a distinct access pattern, solely accesses the Hybrid Index Block within the SST. Unlike traditional *Get* operations, it does not necessitate access to corresponding data blocks. Upon loading the Hybrid Index Block, the GC thread prioritizes this block in the cache, ensuring that subsequent GC-Lookup accesses are efficiently served from the block cache, thereby avoiding I/O operations. As previously discussed, the Hybrid Index Block's size is relatively tiny compared to the entire dataset, making it feasible to cache it in memory for faster access.

3.3.3 Discussion. Compared to the RedundantIndexTable, the IndexDecoupledTable offers a more comprehensive view of valid data for kSST, preserving additional space. By employing this comprehensive view, more precise and up-to-date validity information for keys can be obtained, thereby enhancing GC efficiency. As depicted in Figure.10, a large key-value pair K_1V_1 is inserted into vSST. Subsequently, the user writes a new small key-value pair K_1V_2, leading to the presence of K_1F_1 and K_1V_2 data associated with K_1 in L_i and L_{i+1} of the index LSM-tree. During the GC process targeting the vSST housing K_1V_1, the GC thread obtains the key K_1 via lazy read (①,❶) and navigates the data table in the index LSM-tree. The GC-Lookup in the RedundantIndexTable cannot locate the latest K_1V_2 in L_i (②), only finding the K_1F_1 in L_{i+1} (③). Consequently,

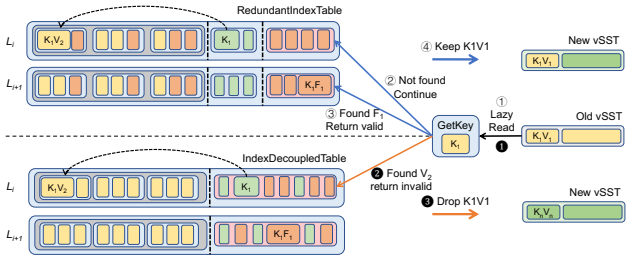


Figure 10: Comparison of RedundantIndexTable and IndexDecoupledTable

the GC thread keeps outdated K_1V_1 (④), transferring it to a new vSST. Conversely, with the IndexDecoupledTable, the GC-Lookup operation can identify K_1V_2 through the hybrid index block (❷), thereby empowering the GC thread to drop K_1V_1 (❸) and bypass this pair.

3.4 Hotness-aware writing

The existing workloads display hot and cold characteristics, excluding variable length characteristics. These workload characteristics significantly impact the efficiency of GC execution. Specifically, the GC operation primarily reclaims garbage data produced by repetitive data writes, denoted as hot-write data, which includes outdated data resulting from updates and deletes. A higher percentage of hot-written data leads to more effective garbage collection, reducing the I/O overhead of individual GC operations. Scavenger exploits the cold-hot characteristics of the workload to introduce a hotness-aware write strategy. This approach identifies data hotspots and leverages this insight to steer the write operations, encompassing Flush and GC operations.

3.4.1 DropCache. Scavenger provides an in-memory cache named DropCache to identify and record data hotspots. DropCache is essentially an LRUCache, which employs a partitioned hash table to store data and employs the LRU algorithm to manage data eviction. DropCache aims to store hot data keys in minimal memory efficiently. As previously noted, hot-write data is predominantly generated from user actions such as deletion and overwrite. Scavenger records data during compaction when performing data merges or deletions, storing related keys in the DropCache to streamline identifying hot-write data. After multiple compactions, the DropCache accumulates many hot data keys. If a key in the DropCache is hit, it is deemed to be a hot data key.

It is essential to mention that DropCache requires only a tiny amount of memory space as hot data in a workload usually occupies a small percentage of the workload. Besides, DropCache only requires recording the keys of hot data (24B per KV). In scenarios where the hot keys may be larger than the available memory space for the DropCache, space-efficient probabilistic data structures such as CuckooFilter [18] may be considered an eventual extension.

3.4.2 Hotness-aware Flush and GC.. Building upon DropCache, Scavenger introduces a hotness-aware write strategy aimed at Flush and GC operations for value SST generation. Scavenger verifies data keys scheduled for flushing if hit in the DropCache, as Figure. 11 shows. Suppose the key hits in DropCache, the data associated with the key is hot and needs to be written to the hot vSST. Otherwise,

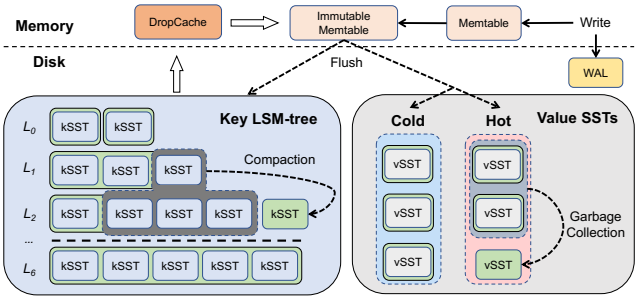


Figure 11: Hotness-aware writing

it is cold and needs to be written to the cold vSST. Finally, the cold and hot vSSTs with temperature tags are generated separately.

Enhancing GC efficiency requires more than just the hotness-aware Flush. Scavenger prioritizes hot vSSTs during GC, as they usually contain recently updated or deleted data, making them prime targets for GC. In contrast, cold vSSTs often hold data that seldom undergoes subsequent changes, rendering GC efforts on them less effective. This hotness-aware separation applies not only to Flush but also to GC operations. Before rewriting valid data to a new vSST during GC, Scavenger verifies its presence in the DropCache and segregates it into hot or cold vSSTs. This approach reduces read and write amplification during GC by prioritizing hot vSSTs.

With the DropCache, Scavenger can enhance the efficiency of the GC-Lookup process. However, this necessitates the implementation of DropCache to either store additional sequence information or encode it into hot data keys. During the GC-Lookup, the DropCache is first queried. We can determine whether the current data is outdated by comparing the GC data sequence with the key sequence stored in the DropCache. Suppose the GC data sequence is lower, indicating outdated data. In that case, the GC-Lookup will immediately return an INVALID result, thereby avoiding any I/O overhead and enabling Scavenger to reduce I/O further and expedite GC execution.

3.5 Space-aware throttling

Although we can expedite the reclaim of storage space through optimized GC operations, the issue of GC being unable to keep up with foreground writes may still occur when the foreground write pressure is high, leading to a rapid expansion of storage space and degradation of space amplification. However, in production environments, storage space is usually limited, and KVS cannot continue to provide its services once the storage space becomes full. Therefore, we must limit the space amplification of the KV-separated LSM-tree to improve the availability of KVS. It is also worth noting that existing performance tests for LSM-tree lack a fair comparison mechanism and tend to ignore the actual storage space used by the workload running process, especially the KV-separated LSM-trees, which often exhibit a significant disparity in space amplification. Hence, a more accurate evaluation mechanism for LSM-tree is necessary.

To address the above issues, Scavenger proposes a space-aware throttling strategy that limits the actual space usage of KV-separation LSM-trees during runtime. Scavenger provides tunable parameters for determining the actual space usage. With these options, Scavenger dynamically handles foreground requests and schedules GC operations based on actual space utilization. This policy works as follows: Users first customize two space amplification thresholds, *low_space_ratio* and *high_space_ratio*, which are usually expressed as a multiple of the dataset size based on the storage quota of their production environment. Upon completion of the configuration, if the actual space utilization reaches *low_space_ratio* during the running process, Scavenger throttles the foreground write requests to reduce the write rate while increasing the background GC thread to expedite the execution of the GC. When the actual space utilization reaches *high_space_ratio*, Scavenger pauses foreground writes and allocates a pre-set maximum number of GC threads to perform GC while simultaneously lowering the garbage ratio threshold for triggering GC, thus ensuring that the maximum space amplification is limited to the configured *high_space_ratio*.

With the space-aware throttling strategy, we can regulate the space utilization of LSM-trees below a specific value, which is more applicable to actual production environments. When comparing KV-separated LSM-trees utilizing the same workload, setting a reasonable threshold ensures that the space amplification of each KVS is consistent, thus enabling a more equitable performance comparison. The space-aware throttling strategy proposed by Scavenger finds a harmonious balance between GC efficiency and foreground read and write, enabling KV-separation LSM-trees to operate with a more reasonable space overhead in practical production environments.

4 EVALUATION

In this section, we evaluate Scavenger and compare it with several state-of-the-art KV-separation LSM-tree solutions: BlobDB, Titan, and TerarkDB. Furthermore, we use the widely adopted LSM-tree-based key-value store, RocksDB, as our baseline. We have implemented Scavenger on top of TerarkDB, an open-source RocksDB fork, to fully utilize TerarkDB's compatibility and stability.

4.1 Setup

Testbed. Our testing environment consists of a system equipped with a 32-core Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz and 64GB of DDR4 memory. We employed a KIOXIA NVMe solid-state drive (SSD) with a capacity of 3.5TB. The operating system kernel for the server is Linux 5.4, with an operating system version of Debian 10. We formatted the NVMe SSD using the ext4 file system.

System Configuration. For all evaluated KVSs, we standardized the configuration parameters by setting the Memtable size to 64MB, the key Sorted String Tables (kSST) size to 64MB, the value Sorted String Tables (vSST) to 256MB, and configure Bloom filters by setting 10 bits/key. Both foreground and background threads are set to 16, with a 1GB BlockCache (approximately 1% of the 100GB dataset). Flushing and compaction utilize direct I/O and direct read to mitigate page cache impact. The GC process is triggered at a garbage ratio of 0.2. The space-based throttling strategy dictates that storage is limited to 1.5 times the dataset size unless otherwise specified.

Workload. We support fixed and variable-length workloads using a modified dbbench (from RocksDB), and YCSB-C [21]. The key

size was set to a constant 24B. Regarding the value lengths, unless specified otherwise, the fixed-length workload utilized 16KB values. In variable-length workloads, a 1:1 ratio of small (100-512KB) to large (16KB) values simulates the ByteDance OLTP database pattern, simplified as Mixed. The average size of the Mixed workload is approximately 8KB. For key distribution, we employ Zipfian distribution, reflecting real-world hotspot scenarios unless otherwise noted.

4.2 Microbenchmarks

We evaluated key-value operation throughputs across various workloads (Mixed-8K and Fixed-16K), including inserting 100GB of key-value pairs, updating 300GB of pairs, reading 300GB, and performing 40 million range query requests. Scan lengths followed a uniform distribution from 1 to 1000 for the scan workload. The testing procedure involved randomly loading 100GB of data and executing the update workload to generate substantial garbage data and trigger GC operations. Subsequently, the read and scan workloads were executed to evaluate the impact of GC operations. To guarantee equitable comparisons between different KVS implementations, we applied a space-based throttling policy, limiting the disk space utilization of each KVS to 1.5 times the size of the dataset, corresponding to a storage quota of 150GB.

Mixed-8K workload. The Figure. 12a demonstrates that Scavenger significantly improves update performance, surpassing RocksDB by 2.8x and outperforming BlobDB, Titan, and TerarkDB by 2.4x, 2.6x, and 2.0x, respectively. Scavenger exhibits similar insertion, read, and scan performance to TerarkDB's. In terms of insertion, multiple KV-separated LSM-trees achieve about a 5x enhancement over RocksDB. In scan performance, the frequent compaction operations of RocksDB contribute to its superiority over several KV-separated LSM-tree implementations. Notably, Scavenger and TerarkDB perform better than the others, as efficient GC operations in TerarkDB enhance data orderliness, leading to superior scan performance.

Given Scavenger's remarkable update performance boost, we evaluated the read and write I/O impact under the update workload, as shown in Figure. 12b. Scavenger significantly reduces I/O operations by 82%. Compared to other KV-separated LSM-trees, Scavenger reduces read I/O by 32% to 47%. Regarding write I/O, Scavenger reduces 74% and 30% compared to RocksDB and BlobDB while behaving slightly better than Titan and TerarkDB. Additionally, we investigated the read-and-write I/O associated with GC operations within KV-separated LSM-trees. This analysis reveals that the Scavenger reduction in disk read and write I/O is primarily attributed to minimized I/O during GC operations.

Fixed-16KB workload. We also tested the performance and read-write I/O of various operations under the fixed-length large KV workload (Fixed-16K), and the results were presented in Figure. 12c and Figure. 12d. The conclusions drawn from the fixed-length large KV workload are comparable to those of the Mixed workload. Specifically, under the update workload, Scavenger improves performance by 1.61x to 5.21x, with read I/O reductions ranging from 32% to 87% and write I/O reductions from 6% to 79%. The significant advantage of Scavenger is its exceptional ability to accelerate GC execution by reducing read and write I/O during the GC process. It is worth mentioning that the better read performance of TerarkDB and Scavenger is mainly due to the higher cache utilization under fixed-length 16KB loads compared to other KV-separated LSM-trees.

4.3 YCSB Evaluation

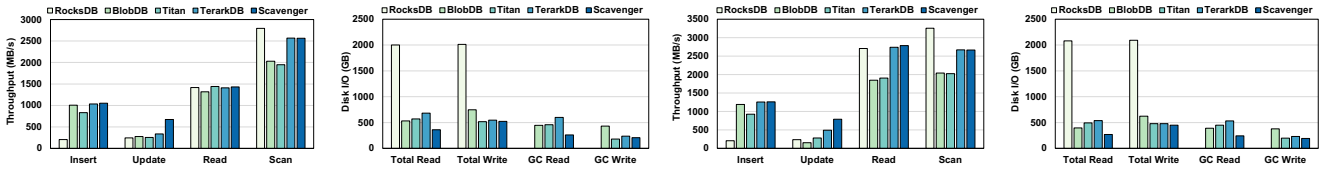
We evaluated Scavenger's performance using YCSB workloads [11], focusing on the KV store's performance following many updates. We followed the steps for each YCSB workload run: initializing 100GB of uniformly random data and then applying 300GB of updates to guarantee GC operations running in all KVS. Subsequently, we executed YCSB workloads A-F on the updated 300GB dataset, maintaining consistent configuration as previously described.

Mixed-8K workload. Initially, we evaluated performance under a mixed-value-size workload (Mixed), as Figure. 13a. Our results revealed notable improvements in write-intensive workload performance with Scavenger. Specifically, Scavenger exhibited a 2.0x to 2.8x enhancement in the update workload, 2.0x to 3.2x in the YCSB-A read-write balanced workload, and 1.5x to 3.0x in the YCSB-F workload, compared to other KV stores. Even in read-intensive scenarios with limited writes like YCSB-B and YCSB-D, Scavenger maintained its performance advantage, retaining the benefits inherited from TerarkDB. In workloads of pure reading, such as YCSB-C, these KVSs exhibited comparable performance levels. Under the YCSB-E workload, Scavenger and TerarkDB maintained proximity to RocksDB's performance, surpassing other KV-separation LSM-tree solutions significantly.

For write-intensive workloads, Scavenger demonstrated superior performance. We conducted read-and-write amplification analyses for YCSB A and YCSB-F workloads to pinpoint the origins of Scavenger's performance improvements (the update workload analysis was conducted previously in microbenchmarks). As depicted in Figure. 13b, our findings indicate that Scavenger's reduced read I/O is more significant than other KV-separated LSM-trees. Regarding write amplification, Titan, TerarkDB, and Scavenger performed similarly, with Scavenger exhibiting a significant reduction of 31% to 39% compared to BlobDB. Our unified space-based throttling strategy ensured consistent space amplification at 1.5 for all cases, with Scavenger consistently outperforming other KV-separation LSM-tree solutions under the same space amplification factor.

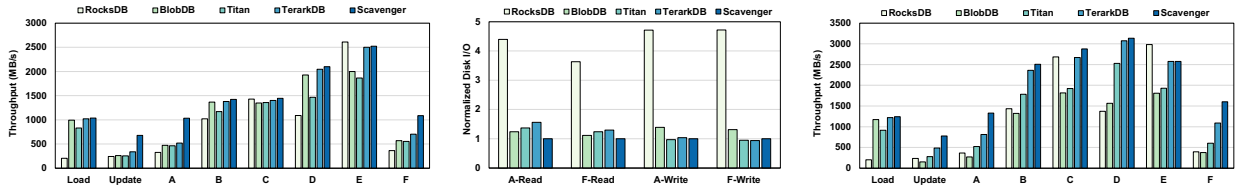
Fixed-16KB workload. We conducted tests under the Fixed-16K workload. Like the microbenchmarks, Scavenger demonstrated more pronounced improvements in performance in this scenario, as Figure 13c. In write-intensive workloads such as YCSB-A and YCSB-F, Scavenger demonstrated notable improvements of 1.7x to 5.0x and 1.5x to 4.4x, respectively. For read and write I/O, the reduction is even more significant compared to Mixed-8K loads. For example, under YCSB-A load, Scavenger reduces read I/O by 32%-65% and write I/O by 5%-44% compared to other KV-separated LSM-trees.

Scavenger consistently demonstrated superior performance in Fixed-16K and mixed-size value (Mixed) YCSB workloads. The performance benefit was particularly evident in write-intensive scenarios, where GC executions occurred more frequently. Scavenger effectively reduced I/O overhead during the GC process, contributing to its overall improvement in performance.



(a) Throughput under Mixed-8K (b) I/O under Mixed-8K (c) Throughput under Fixed-8K (d) I/O under Fixed-16K

Figure 12: Microbenchmarks under Mixed-8K and Fixed-16K



(a) Throughput under Mixed-8K (b) I/O under Mixed-8K (c) Throughput under Fixed-8K

Figure 13: YCSB under Mixed-8K and Fixed-16K

4.4 Features

We evaluated essential GC optimization techniques in Scavenger: lazy read (TDB-R for read optimization), index-record separation (TDB-L for lookup optimization), and hotness-aware writing (TDB-W for write optimization). TerarkDB served as the baseline for comparison. We combined read and write optimization into TDB-RW and integrated all three techniques into TDB-RWL (Scavenger). Our objective was to evaluate the impact of optimization techniques on update operation performance through a write-intensive workload of 300GB updates. We set a 1.5x space constraint to ensure fairness to maintain consistent space amplification.

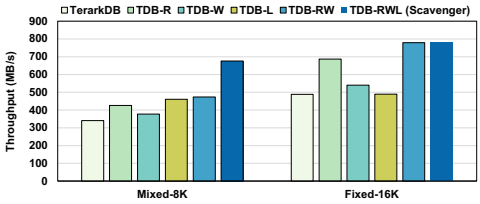


Figure 14: Throughput of Different Features

Throughput. We evaluated Scavenger's performance separately using a mixed-size value workload (Mixed-8K) and a fixed-size large value workload (Fixed-16K). As depicted in Figure.14, each optimization technique yielded corresponding improvements in the mixed-size value workload (Mixed), with enhancements of 25% and 35% from read optimization (TDB-R) and lookup optimization (TDB-L), respectively. In the fixed-length workload scenario, as identified in Section 2.5, the primary GC performance bottlenecks were read and write operations. As a result, optimizations focused on enhancing GC process read operations (TDB-R) and hotness-aware writing (TDB-W) showed promising results, with improvements of 40% and 11%, respectively.

Disk I/O. We analyzed the I/O statistics of the respective GC threads to determine the performance gains from each optimization technique, shown in Figure.15. In the Mixed-8K workload, each optimization technique notably reduced the GC read I/O, collectively leading to a substantial reduction in the Scavenger's overall GC read I/O. Only the hotness-aware writing strategy (TDB-W) lowered GC

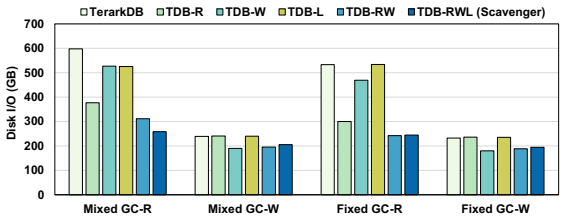


Figure 15: I/O of Different Features

write operations among these techniques. Notably, optimizations targeting GC reads (TDB-R) and lookup operations (TDB-L) did not decrease GC writes and might have slightly increased them. It can be attributed to increased GC executions, as shown in Table.1 indicates. The more garbage data, the faster the GC execution tends to result in more GC executions.

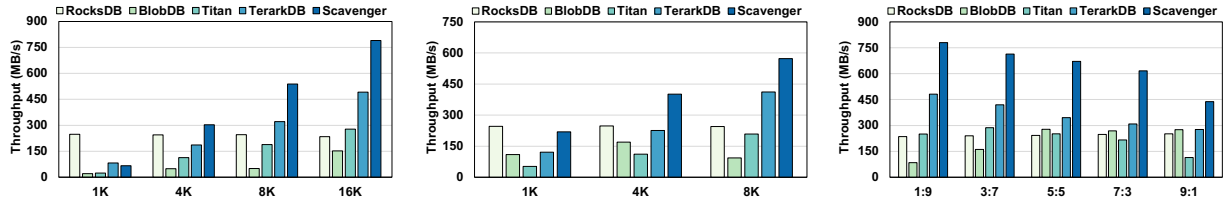
Table 1: GC Count

	TDB	TDB-R	TDB-W	TDB-RW	Scavenger
Mixe-8K	1104	1106	1455	1460	1473
Fixe-16K	1068	1083	1447	1458	1463

For large fixed-length workloads, the lazy read optimization (TDB-R) significantly decreased GC read I/O. The hotness-aware writing strategy (TDB-W) also effectively curtailed unnecessary cold data reads. However, the optimization aimed at GC-Lookup operations (TDB-L) had limited read I/O impact. In this specific workload, the size of the LSM tree's index is sufficiently small to be accommodated within the Block Cache, thereby limiting the optimization potential of the lookup operation. Similar to the Mixed-8K workload, optimizations in the Fixed-16K scenario aimed at improving GC read (TDB-R) and GC-Lookup operations (TDB-L) slightly increase GC writes due to increased GC executions.

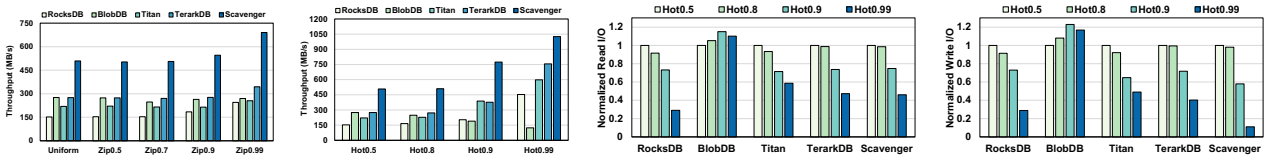
4.5 Performance under Different Workloads

4.5.1 Varing value size. To assess the adaptability of Scavenger across diverse workloads, we first evaluated its performance under varying value-size distribution workloads. Our primary focus was



(a) Throughput under Fixed Workload (b) Throughput under Pareto Workload (c) Throughput under Mixed Workload

Figure 16: Update Performance under workloads with varying size



(a) Throughput under Zipfian (b) Throughput under Hotspot (c) Read I/O under Hotspot (d) Write I/O under Hotspot

Figure 17: Throughput and I/O under workloads with varying skewness

on update operation performance, particularly in write-intensive scenarios involving 300GB of data updates.

Fixed size. We evaluated different value sizes of 1KB, 4KB, 8KB, and 16KB, as depicted in Figure 16a. We have observed that, except for the 1KB small value workload, Scavenger outperformed all other KVS solutions. As the value size increased, the performance advantage of Scavenger became more pronounced. For Fixed-1KB, all KV-separated LSM-trees exhibited poorer performance than RocksDB for two reasons. Firstly, KV-separated LSM-trees struggle in scenarios involving small KVs, as they incur more data writes compared to vanilla LSM-trees. Secondly, while RocksDB enables read-ahead during compaction, most KV-separated LSM-trees disable read-ahead for GC operations, leading to a slower read during GC. By enabling read-ahead for Scavenger in Fixed-1K, the performance gap was reduced from 73% to 12%, better than other KV-separated LSM-trees. Besides, Scavenger's performance declined slightly in Fixed-1K compared to TerarkDB because of the read amplification brought by the alignment size (4KB) of Direct I/O. Although Scavenger reduces the number of read I/O operations, the overall read I/O bandwidth of the disk remained unchanged. Additionally, Scavenger maintains a default index entry for each value to facilitate GC lazy reads, slightly increasing the data volume compared to TerarkDB.

It is worth mentioning that such small value workloads are uncommon for KV-separated LSM-trees. Existing KV-separated LSM-trees typically set higher KV separation thresholds to store small KVs within their LSM-tree components, e.g., 1KB/4KB for Titan and 4KB for BadgerDB. TerarkDB defaults to 512B, resulting in a substantial number of small KVs being stored in the value SST, negatively impacting the GC and foreground performance.

Pareto distribution. We employed a workload with value sizes following a Pareto distribution [17] for variable-length key-value sizes, resulting in average value sizes of approximately 1KB, 4KB, and 8KB, as shown in Figure16b. Similar to Fixed-1K, the performance of Scavenger under Pareto-1K was not as competitive as that of RocksDB. Nevertheless, Scavenger demonstrated a distinct advantage in this particular scenario over TerarkDB. The main difference is that the index LSM-tree under the Pareto workload stores many small KVs, significantly slowing GC execution. In contrast, Scavenger's optimization of GC-Lookup speeds up GC execution, resulting in an 81% performance improvement over TerarkDB. The overhead associated with GC reads and writes increases as the average size increases. Scavenger performs significantly better in this scenario due to its reduced GC I/O.

Mixed size. We tested the Mixed workload by varying the proportions of small and large key-value pairs, with ratios of 1:9, 3:7, 5:5, 7:3, and 9:1, as shown in Figure 16c. Small values followed a uniform distribution with sizes between 100 and 512B, while large values were consistently 16KB. The analysis revealed that as the proportion of small values increased, Scavenger's performance advantage diminished. This trend resembled our observations in the 1KB workload scenario, where the impact of small value workloads on the GC process led to reduced Scavenger performance. Despite the decline, Scavenger consistently outperformed the other five KVS solutions. Its performance remained superior, especially in scenarios where large values constituted a substantial portion.

Evaluations conducted under varying workloads reveal that Scavenger performs best in scenarios where the KV-separated LSM-tree is appropriate, and even in small KV scenarios, it exhibits some benefits.

4.5.2 Varying skewness. We evaluated Scavenger's update operation performance under varying workload skewness conditions, including Zipfian and hotspot workloads with different coefficients.

Throughput. We systematically adjusted the Zipfian constants for Zipfian workloads to represent varying skewness. A higher constant indicates a more skewed workload. The uniform workloads were also tested. As shown in Figure 17a and Figure 17b, Scavenger consistently outperforms other key-value stores across all workloads. As workload skewness increases, exemplified by a Zipfian constant of 0.99, Scavenger's performance demonstrates significant enhancement.

Regarding hotspot workloads, we varied the ratios of hot data and proportions of access operations to simulate different levels of

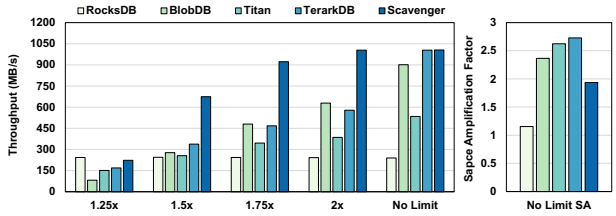


Figure 18: Throughput under varying space limits

data distribution skewness. Four distinct hotspot scenarios were examined: hot0.5, hot0.8, hot0.9, and hot0.99. These labels indicate the percentage of access operations that target hot data. For instance, in hot0.8, 80% of operations involve accessing 20% of data. As depicted in the Figure, the test outcomes confirm Scavenger's superiority as the best-performing KVS across all hotspot scenarios evaluated.

Disk I/O. We tested the corresponding disk read and write I/Os under the hotspot distribution to investigate the source of Scavenger's performance gains under skewed workloads, as depicted in Figure 17c and Figure 17d. Our analysis revealed a decreased read and write I/O for varying workload skewness across various KVS. Scavenger significantly reduces read and write I/O as the workload skew increases.

Through throughput tests and read-write statistics across various skewnesses, it becomes evident that Scavenger is highly adaptable to skewed workloads. Scavenger effectively leverages the advantages of data distribution with concentrated hotspots.

4.6 Varying space limit

To evaluate Scavenger's performance under varying space limitations, simulating diverse production environment constraints, we tested with space limits set at 1.25x, 1.5x, 1.75x, and 2x the dataset size, along with a no-limit scenario. The Figure18 illustrates the evaluation outcomes.

As space constraints ease and storage space quotas increase, the performance gap in update operations among KV-separated key-value stores decreases. The performance of RocksDB, which benefits from low space amplification, remains relatively unaffected. Under enforced space limitations, Scavenger consistently performs exceptionally well, particularly with stricter quotas, such as 1.25x and 1.5x. At 1.25x space constraints, only Scavenger achieves performance close to RocksDB among the KV-separated LSM-trees. It reveals that, under stringent space constraints, the advantage of KV separation decreases. Nevertheless, Scavenger's ability to minimize GC-related I/O enhances space reclamation, aligning it with foreground writes and achieving performance comparable to RocksDB.

Under the no-limit scenario, BlobDB, TerarkDB, and Scavenger exhibit comparable performance, with notable disparities in space amplification between these different KVS solutions becoming apparent. Among the evaluated LSM-trees, Scavenger exhibits the lowest space amplification rate at 1.94x. With no space constraints, Scavenger attains superior front-end performance. It achieves the most minor space amplification among LSM-tree-based KV separation solutions, effectively balancing optimal front-end performance and minimal space amplification.

5 RELATED WORK

LSM-tree-based Key-Value Stores have gained widespread recognition for their exceptional write performance. Numerous studies have been conducted to enhance the read-and-write performance of LSM-tree-based KVS.

Compaction Optimization. Efforts to optimize LSM-tree performance often entail targeting the overhead of compaction. One approach is to weaken existing restrictions on storage layout orderliness. For instance, PebblesDB [28] loosens the constraint of single-layer global orderliness and adopts the structure of FLSM to reduce read and write amplification, albeit at the cost of read performance and space amplification. Other schemes, such as UniKV [32], WipDB [34], and REMIX [35], utilize partitioning to minimize compaction overhead. However, they also experience the drawbacks of read and space amplification associated with tiering compaction. Consequently, the most widely used LSM-tree-based KVS, such as RocksDB, still employ leveled compaction by default to guarantee consistent performance and space usage.

Key-Value Separation. KV separation has gained industry recognition for reducing LSM-tree read and write amplification. WiscKey [24], and HashkV [8] first explored KV-separation but overlooked the query performance, particularly for range queries. Despite implementing learning indexes atop WiscKey by Bourbon [12] to improve read performance, it did not fundamentally alter the unordered storage layout. TerarkDB [4], Titan [27], DiffKV [23], and BlobDB [16] have improved the value orderliness by utilizing structures similar to the original SST. Despite this, they still experience substantial fluctuations in performance due to substantial GC overhead. Titan, in particular, is encountering increased contentions with foreground write operations due to the requirement for write-back during GC. DiffKV and BlobDB have adopted a Compaction-triggered GC strategy to reduce overhead in the GC process, which further couples compaction and GC operations, significantly diminishing GC space reclamation efficiency. In comparison, Scavenger, built on the foundation of TerarkDB, significantly reduces GC overhead and accelerates space reclamation while incurring lower IO costs. Consequently, Scavenger attains a more favorable balance between read-write performance and space amplification.

6 CONCLUSION

In this paper, we propose Scavenger, an I/O-efficient garbage collection scheme for the KV-separated LSM-tree. We analyze the bottlenecks of GC operations under diverse workloads and devise I/O-friendly storage layouts for the KV-separated LSM-tree to accelerate GC. By reducing I/O costs during GC and utilizing space-aware throttling, Scavenger achieves a more favorable balance between performance and space utilization. The evaluation results indicate that Scavenger significantly improves update performance and limits space amplification.

REFERENCES

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.

[2] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2018. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, LLC.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. {SILK}: Preventing Latency Spikes in {Log-Structured} Merge {Key-Value} Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.

[4] Bytedance. [n.d.]. TerarkDB. https://github.com/bytedance/terarkdb.

[5] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} Meets Computational Storage: Efficiently Support Analytical Workloads in {Cloud-Native} Relational Database. In *18th USENIX conference on file and storage technologies (FAST 20)*. 29–41.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[8] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. {HashKV}: Enabling Efficient Updates in {KV} Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[9] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*. 1087–1098.

[10] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

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

[12] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From {WiscKey} to Bourbon: A Learned Index for {Log-Structured} Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[13] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.

[14] Dgraph. [n.d.]. Badger. https://github.com/dgraph-io/badger.

[15] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 33–49.

[16] Facebook. [n.d.]. RocksDB. https://github.com/facebook/rocksdb.

[17] Facebook. [n.d.]. RocksDB Trace, Replay, Analyzer, and Workload Generation. https://github.com/facebook/rocksdb/wiki/RocksDB-Trace-Replay-Analyzer-and-Workload-Generation.

[18] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[19] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. Analysis of {HDFS} Under {HBase}: A Facebook Messages Case Study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*. 199–212.

[20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[21] J. Ren. [n.d.]. YCSB-C. https://github.com/basicthinker/YCSB-C.

[22] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and stable compaction for lsm-tree: A faas-based approach on terarkdb. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3906–3915.

[23] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated {Key-Value} Storage Management for Balanced {I/O} Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 673–687.

[24] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[25] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. *ACM SIGMOD Record* 23, 2 (1994), 233–242.

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[27] Pingcap. [n.d.]. Titan. https://github.com/tikv/titan.

[28] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[29] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. [n.d.]. NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores. ([n. d.]).

[30] Chenlei Tang, Jiguang Wan, and Changsheng Xie. 2022. Fencekv: Enabling efficient range query for key-value separation. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3375–3386.

[31] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. {LSM-trie}: An {LSM-tree-based}{Ultra-Large}{Key-Value} Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[32] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward high-performance and scalable KV storage in mixed workloads via unified indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 313–324.

[33] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM: optimize data layout for LSM-tree based storage using survival analysis. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2161–2174.

[34] Xingsheng Zhao, Song Jiang, and Xingbo Wu. 2021. WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1404–1415.

[35] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. {REMIX}: Efficient Range Query for {LSM-trees}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 51–64.