Differentiated Key-Value Storage Management for Balanced I/O Performance

Yongkun Li[1], Zhen Liu[1], Patrick P. C. Lee[2], Jiayu Wu[1], Yinlong Xu[1,3]
Yi Wu[4], Liu Tang[4], Qi Liu[4], Qiu Cui[4]
[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC* [4]*PingCAP*

Abstract

Modern key-value (KV) stores adopt the LSM-tree as the core data structure for managing KV pairs, but suffer from high write and read amplifications. Existing LSM-tree optimizations often make design trade-offs and are unable to simultaneously achieve high performance in writes, reads, and scans. To resolve the design tensions, we propose DiffKV, which builds on KV separation to carefully manage the ordering for keys and values. DiffKV manages keys using the conventional LSM-tree with *fully-sorted ordering* (within each level of the LSM-tree), while managing values with *partially-sorted ordering* with respect to the fully-sorted ordering of keys in a coordinated way for preserving high scan performance. We further propose fine-grained KV separation to differentiate KV pairs by size, so as to realize balanced performance under mixed workloads. Experimental results show that DiffKV can simultaneously achieve the best performance in all aspects among existing LSM-tree-optimized KV stores.

1 Introduction

Key-value (KV) storage, which abstracts data as KV pairs, becomes a critical storage paradigm for supporting a variety of applications, such as search engines [4, 11, 23, 53], distributed file systems [1, 28], data deduplication [18, 41], graph stores [7, 22, 37, 64], and OLTP/OLAP databases [3, 8, 11, 19, 30, 31, 33, 43, 50, 65]. Such storage paradigms mainly provide three operations: (i) *writes*, which insert KV pairs, (ii) *reads*, which retrieve the value of a single key, and (iii) *scans*, which retrieve the values over a key range. In particular, scans are essential operations for KV stores in various applications; for example, graph computation tasks may traverse multiples nodes or edges [2, 7, 37], and OLAP databases traverse multiple entries in a table for data filtering or aggregation [8, 30, 33, 50]. Thus, optimizations for scans have also been specifically studied in the literature [43, 55, 62, 65].

To better leverage the efficiency of sequential I/Os and preserve the data ordering for fast scans, modern KV stores (e.g., [10, 24, 27, 42, 52]) often adopt the *Log-Structured-Merge-tree (LSM-tree)* [46]. At a high level, the LSM-tree builds on three properties (see §2.1 for details): (i) it organizes KV pairs in a log-structured layout, in which updating KV pairs is treated as issuing sequential writes of KV pairs to persistent storage with high performance; (ii) it adopts a multi-level structure that keeps KV pairs *fully sorted* within each level, so as to support efficient reads and scans without specialized in-memory index structures; and (iii) it mitigates the write overhead by gradually moving KV pairs from lower levels to higher levels via *compaction*, so as to maintain high storage scalability. However, the compaction incurs substantial amounts of I/Os, as the KV pairs in adjacent levels need to be retrieved and written back to maintain sorted ordering. It is well known that LSM-tree KV stores suffer from severe write and read amplifications, especially when the number of levels of the LSM-tree increases with the growing volume of KV pairs being managed [42, 52, 58].

To reduce the compaction overhead, a number of LSM-tree optimization techniques have been proposed in the literature. One direction of work is to relax the fully-sorted nature in each level of the LSM-tree, thereby alleviating the compaction overhead [17, 52, 55]. However, as the degree of fully-sorted ordering is relaxed, the scan performance also degrades. Another direction of work is based on *KV separation*, which separates the storage of keys and values by keeping only keys (in fully-sorted ordering) in the LSM-tree and performing value management in a dedicated storage area [10, 20, 38, 42, 49, 51, 63]. KV separation alleviates the compaction overhead as the LSM-tree size now significantly decreases without storing the values, and is particularly suited for practical KV workloads whose KV pairs are composed of large-size values [5, 9, 30, 38, 44, 59] (see §2.2 for details). However, it degrades the scan performance, especially for the values with small-to-medium sizes that are also common in practice [5, 9], since each scan needs to issue random I/Os to the values over a key range that are no longer fully sorted (in contrast, the random I/O overhead is amortized for large values). Also, KV separation incurs extra garbage collection (GC) overhead [10], thereby triggering additional I/O overhead beyond the compaction in the LSM-tree. In short, existing LSM-tree optimizations are still limited by tight performance tensions between reads/writes and scans, in terms of (i) the degrees of ordering in keys and values, and (ii) the management of KV pairs of varying sizes.

To this end, we design a novel KV store, DiffKV, that realizes balanced I/O performance on commodity storage devices (e.g., solid-state drives (SSDs)). Its main idea builds on the differentiated KV management in two aspects. First, DiffKV differentiates the storage management of keys and values as in conventional KV separation, and takes one step further to carefully coordinate the differentiated management of the ordering for keys and values. Specifically, it keeps keys fully sorted in each level of the LSM-tree as in conventional KV separation for fast reads and scans, while keeping values *partially sorted*, such that the (partially-sorted) ordering of values is coordinated with respect to the (fully-sorted) ordering of keys. We design a new LSM-tree-like structure called the *vTree* for value management, such that the sorting of values in the vTree is triggered by the compaction of the LSM-tree. In this way, we limit the overhead of sorting values, yet we still maintain high scan performance via the partially-sorted ordering for values. Second, DiffKV differentiates the management of values via fine-grained KV separation, such that the KV pairs of different size groups are specifically managed for maintaining balanced performance under mixed workloads.

To the best of our knowledge, this is the first work that examines the differentiated ordering for KV pairs. Traditional LSM-tree KV stores without KV separation always couple keys and values together and only realize a single kind of ordering [24, 27, 52]. While existing KV separation designs [10, 20, 38, 42, 49, 51, 63] decouple keys and values, the values are unsorted, and they cannot be tuned to realize different degrees of ordering for balanced performance. Our main contributions are summarized as follows.

- We present DiffKV, which coordinates the differentiated management of ordering for keys and values, so as to simultaneously improve the performance of writes, reads, and scans. Specifically, DiffKV manages values in the vTree structure for the partially-sorted ordering of values.
- We propose multiple merge optimization techniques to reduce the sorting overhead in the vTree, and also develop a state-aware lazy GC scheme to realize high space efficiency and high performance.
- We propose fine-grained KV separation and differentiate the management of small, medium, and large KV pairs for optimizing mixed workloads. In addition to the LSM-tree and the vTree, we also propose a hotness-aware multi-log design for efficiently managing large KV pairs.
- We implement a DiffKV prototype atop Titan [51], an open-source KV store that implements KV separation with optimized techniques. Evaluation results show that DiffKV achieves the best performance in writes, reads, scans, and space utilization compared to state-of-the-art KV stores, including RocksDB [24], PebblesDB [52], and Titan [51].

We release the source code of our DiffKV prototype at **https://github.com/ustcadsl/diffkv**.

2 Background and Motivation

We first introduce the basics of an LSM-tree KV store. We then discuss the strengths and weaknesses of different LSM-tree optimizations to motivate our DiffKV design.

2.1 LSM-tree KV Store

Storage structure. Figure 1(a) depicts a simplified storage structure of a conventional LSM-tree KV store (e.g., LevelDB [27] and RocksDB [24]). An LSM-tree KV store comprises $n+1$ levels on disk, denoted by $L_0, L_1, \cdots, L_n$ (from lowest to highest), and the capacity of a higher level L_i is a multiple (e.g., $10\times$ by default in LevelDB [27]) of that of a lower level L_{i-1} (where $1 \leq i \leq n$). It stores KV pairs in entirety as multiple disk files, called *SSTables*, in multiple levels. It also has two in-memory write buffers, namely *MemTable* and *Immutable MemTable*, and flushes the Immutable MemTable to level L_0 on disk with append-only writes. One main feature of the LSM-tree KV store is that all KV pairs in each of the levels from L_1 to L_n are *fully sorted* by keys to support fast scans, while KV pairs in L_0 are unsorted across different SSTables for fast flushes.

Write process. To write a KV pair, an LSM-tree KV store first inserts the KV pair into the MemTable, which keeps all buffered KV pairs sorted by keys in a skip-list. When the Memtable is full, it becomes an Immutable MemTable, which is then flushed to level L_0 as a new SSTable. The SSTable comprises the sorted KV pairs and some metadata (e.g., Bloom filters) for indexing. Meanwhile, the KV store generates a new MemTable to receive the subsequent new writes. When L_i is full (where $i \geq 0$), its SSTables will be integrated into L_{i+1} by a *compaction* operation. To compact an SSTable S from L_i into L_{i+1}, the KV store reads S and all SSTables in L_{i+1} that have overlapped key ranges with S, then sorts all KV pairs by keys and creates new SSTables. It finally writes them back into L_{i+1}. Thus, compaction induces to severe write amplification, which can reach up to a factor of $50\times$ [42].

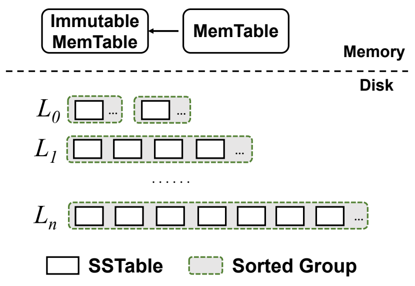
Read process. To read a KV pair, an LSM-tree KV store first searches for the KV pair in memory. If the KV pair does not exist, the KV store performs binary search in each level of the LSM-tree, starting from the lowest level L_0 to the higher levels. For each level, it identifies an SSTable and checks its Bloom filter to see if the KV pair exists.

To scan KV pairs, the KV store first finds the starting key in each level. It then sequentially reads the KV pairs that fall in the queried range. It finally returns the set of KV pairs.

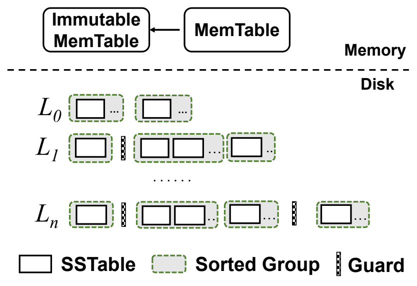
2.2 LSM-tree Optimizations and Limitations

We discuss two major classes of LSM-tree optimizations on how they reduce the compaction overhead of LSM-tree KV stores: (i) relaxing fully-sorted ordering [17, 52, 55] and (ii) KV separation [10, 20, 38, 42, 49, 51, 63]. Other types of optimizations are reviewed in §6.

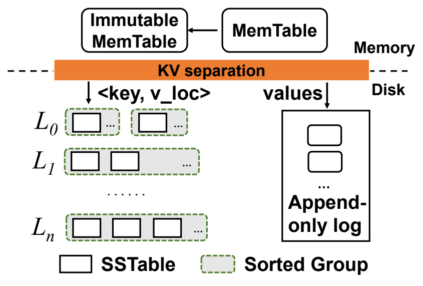
Relaxing fully-sorted ordering. We consider PebblesDB [52], which realizes a *fragmented* LSM-tree, as a representative example, as shown in Figure 1(b). PebblesDB divides each level into several disjoint groups by *guards*, in which the key ranges of SSTables in the same group may overlap with each other. To compact a group of SSTables from L_i to



(a) Conventional LSM-tree



(b) Fragmented LSM-tree



(c) KV separation

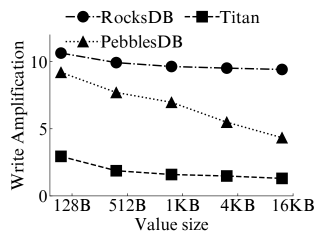
Figure 1: Overview of LSM-tree KV storage structure: (a) fully sorted within each level; (b) partially sorted within each level via multiple non-overlapping segments with guards; and (c) fully sorted for keys, unsorted for values.

L_{i+1}, PebblesDB only reads the SSTables within the group in L_i, sorts them to create new SSTables, and finally writes the new SSTables into L_{i+1}. In this case, a compaction operation in PebblesDB does not need to read SSTables from L_{i+1}, thereby greatly alleviating compaction overhead and write amplification. However, because of the overlapped key ranges of SSTables within each group, PebblesDB sacrifices scan performance. Although it can leverage multi-threading to issue reads in parallel, it incurs more CPU resources, while the improvement remains limited.

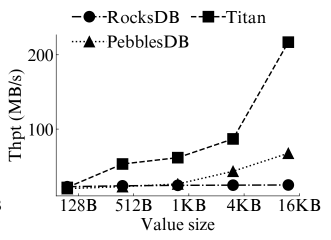
KV separation. Figure 1(c) illustrates the typical structure of KV separation in Wisckey [42] and Titan [51]. KV separation stores keys and values separately, in which keys and their references to values are treated as new KV pairs and stored in the LSM-tree, while the original values are separately stored in an append-only log, which is implemented as multiple blob files in Titan. For medium-to-large value sizes, as the keys and value references often have much smaller sizes than the original values. In this case, the total data volume in the LSM-tree is significantly reduced, so the compaction overhead and hence the write amplification are alleviated. In addition, a smaller LSM-tree reduces read amplification and hence improves read performance.

KV separation is a well-known optimization technique for LSM-tree KV stores, as KV pairs with large values sizes are commonly found in real-world KV workloads. For example, TiDB [30], a transactional database built atop a KV storage layer, maps every row of a database table into a KV pair whose value size can grow to hundreds of kilobytes. Atlas [38] maintains KV pairs of cloud storage with value sizes larger than 128 KB. Also, field studies [5, 44, 59] indicate that large-size values contribute to a considerable fraction of traffic in practical KV workloads, and the recent study by Facebook [9] shows that KV pairs for social graph data can have an average value size as large as 1 KB. Finally, the values with medium-to-large sizes (e.g., from 1 KB to 16 KB) are often chosen in the evaluation benchmarks of KV stores (e.g., [10, 42, 52]).

However, since KV separation writes values into an append-only log, the values of a consecutive range of keys are now scattered in different positions in the log. Thus, the scan operations need to issue random reads to values, thereby leading to low scan performance in KV separation especially for the KV workloads whose values have small-to-medium sizes [5, 9] (e.g., in OLTP applications [5], more than 90% of values are smaller than 1 KB). Furthermore, KV separation needs to perform GC to reclaim the space of invalid values in the append-only log, and frequent GC operations incur extra I/O overhead [10].



(a) Write amplification



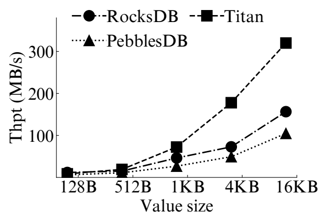
(b) Write throughput

Figure 2: Write performance of RocksDB, PebblesDB, and Titan.

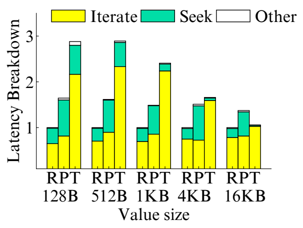
2.3 Trade-off Analysis

We evaluate the design trade-offs of existing LSM-tree designs and optimizations. We consider three open-source KV stores: (i) RocksDB [24], which represents the state-of-the-art LSM-tree KV store with optimized performance; (ii) PebblesDB [52], which relaxes the fully-sorted ordering of each LSM-tree level for reduced compaction overhead; and (iii) Titan [51], which realizes KV separation with optimized implementation (e.g., managing values in multiple small Blob files instead of a large append-only log and using multi-threading to reduce GC overhead). We evaluate write amplification (i.e., the ratio of total write size to user write size) and performance, based on the testbed and configurations in §5.1.

Write performance. Figure 2 shows the write performance of loading a 100 GB database with different value sizes (varying from 128 bytes to 16 KB). From Figure 2(a), both PebblesDB and Titan significantly reduce the write amplification of RocksDB, and their write amplification ratios decrease as the value size increases. For example, for a value size of 16 KB, the write amplification ratios of RocksDB, PebblesDB, and Titan are 9.4×, 4.3×, and 1.3×, respectively.



(a) Read throughput



(b) Scan latency breakdown

Figure 3: Read and scan performance of RocksDB ('R'), PebblesDB ('P'), and Titan ('T').

From Figure 2(b), both PebblesDB and Titan show higher write throughput than RocksDB due to the reduced write amplification. For example, for a value size of 16 KB, the write throughput gains of PebblesDB and Titan over RocksDB are 2.7× and 8.7×, respectively. In short, the LSM-tree optimizations of relaxing fully-sorted ordering and KV separation are effective to reduce write amplification and hence increase write throughput, especially for large-size values.

Read and scan performance. Figure 3 shows the read and scan performance of RocksDB, PebblesDB, and Titan. For read performance, we issue read requests (i.e., random point queries) to a randomly loaded 100 GB KV store; for scan performance, we issue scan requests to 100 KV pairs. From Figure 3(a), relaxing fully-sorted ordering degrades the read performance, so the read throughput of PebblesDB is lower than that of RocksDB. Titan uses KV separation, which largely reduces the LSM-tree size, so its read throughput is evidently higher than that of RocksDB; for example, its throughput 2.0× of that of RocksDB for a value size of 16 KB.

For scans, both PebblesDB and Titan perform worse than RocksDB, especially for small-to-medium size values. Figure 3(b) provides a latency breakdown for scans. For example, for a value size of 1 KB, the scan latencies of PebblesDB and Titan are 1.5× and 2.4× of that of RocksDB, respectively. Most of the scan time is spent on iteratively reading values (e.g., more than 90% for Titan). As the value size becomes larger (e.g., 16 KB), the differences of scan latencies among the KV stores are smaller, as accessing large-size values has smaller random read overhead. For the KV workloads that are dominated by small-size values [5, 9], the scan performance of PebblesDB and Titan will be limited in practice.

In short, existing LSM-tree designs and optimizations are subject to the performance trade-offs between reads/writes and scans. While relaxing the degree of ordering for values (e.g., using the fragmented LSM-tree or an unsorted append-only log in KV separation) reduces write amplification and improves write throughput, it sacrifices the scan performance, especially for values with small-to-medium sizes.

3 DiffKV Design

We present DiffKV, a novel LSM-tree KV store that aims for balanced performance in writes, reads, and scans.

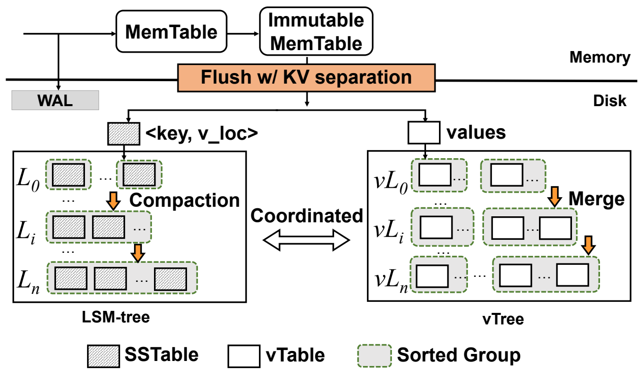


Figure 4: System overview.

3.1 System Overview

Figure 4 depicts the system architecture of DiffKV. DiffKV builds on KV separation for on-disk KV pairs, such that keys and values are decoupled during the flush of KV pairs from memory to disk and then they are separately stored. It also realizes *partially-sorted ordering* for values, so as to maintain high scan performance. To achieve this, DiffKV leverages a new LSM-tree-like multi-level tree, called the *vTree*, to manage values. As in the LSM-tree, the vTree comprises multiple levels, each of which can only be written in an append-only way. The difference between the vTree and the LSM-tree is that the vTree only stores values which are not necessarily fully sorted by their keys within each level; instead, they are allowed to be partially sorted for high scan performance.

To realize partially-sorted ordering for values, the vTree also necessitates a compaction-like operation as in the LSM-tree, which we call a *merge* operation to differentiate itself from a compaction operation in the LSM-tree. To reduce the merge overhead, DiffKV makes the compaction of the LSM-tree and the merge of the vTree be executed in a *coordinated* manner so as to reduce the overall overhead.

To make DiffKV compatible with existing LSM-tree store designs, it still follows the same in-memory data management as in conventional LSM-tree KV stores with an on-disk write-ahead log (WAL). Specifically, as depicted in Figure 4, DiffKV first writes KV pairs to the WAL, inserts them into the MemTable in memory, and finally flushes the Immutable MemTable to disk with KV separation.

3.2 Data Organization

The vTree adopts hierarchical data organization. It consists of multiple levels. Each level consists of *sorted groups*, and each sorted group further consists of multiple *vTables* (Figure 4). In the following, we elaborate their design details.

vTable. DiffKV organizes values as vTables, each of which has a fixed size (e.g., 8 MB by default). Note that each flush of an immutable MemTable may generate multiple vTables depending on the value size and MemTable size.

A vTable includes a *data area* that stores the values of KV pairs in a sorted order based on their keys, as well as a *metadata area* that records the necessary metadata, such as the data size of the vTable, and the smallest and largest keys of the values in this vTable. Note that the Bloom filter is not required in the vTable (as opposed to the SSTable in the LSM-tree), since the values are still indexed by their keys in the LSM-tree. Thus, the metadata in each vTable has a very small size and brings limited storage overhead.

Sorted group. Each sorted group is a collection of vTables, and all vTables in a sorted group are fully sorted according to their corresponding keys. In other words, the key ranges of any two vTables in a sorted group have no overlaps. For ease of presentation, we also apply the concept of a sorted group for the LSM-tree, such that any set of SSTables in the LSM-tree that are fully sorted are also called a sorted group (e.g., all SSTables in the same level in the LSM-tree form a sorted group). In DiffKV, all vTables generated in one flush form a sorted group, so as to preserve the ordering of values in each immutable MemTable. We use the number of sorted groups as an indicator to measure the degree of ordering in the vTree. As the number of sorted groups increases, the degree of ordering decreases. In one extreme, if all SSTables/vTables form one sorted group, then we have the maximum degree of ordering, as all KV pairs are fully sorted.

vTree. The vTree consists of multiple levels, each of which is formed by multiple sorted groups. While the values in each sorted group are fully sorted, the values in a level of the vTree are not necessarily fully sorted, as they may belong to multiple sorted groups that have overlaps in the key range. As the vTree allows each level to have multiple sorted groups, a merge operation does not need to sort all values in successive two levels in the vTree; this alleviates the I/O overhead as opposed to the compaction in the LSM-tree.

3.3 Compaction-Triggered Merge

The vTree regularly performs merge operations to have partially-sorted ordering for values. Each merge reads a number of vTables and checks which values in the vTables remain valid. This can be done by querying the LSM-tree to retrieve the latest value location. Also, each merge needs to update the LSM-tree for the latest value locations of the valid values. To limit the merge overhead in the vTree, the merge operations in the vTree are not executed independently, but are triggered by the compaction operations in the LSM-tree in a coordinated manner. We call such a merge operation to be a *compaction-triggered merge* operation.

To explain the idea, we consider a simplified case in which we let each level in the vTree be coupled with only one level in the LSM-tree, and use L_i and vL_i to denote level i of the LSM-tree and vTree, respectively. If L_i and vL_i are correlated, then it means that for each KV pair, if the key is stored at level L_i in the LSM-tree, then the value of the KV pair can only be stored at level vL_i in the vTree. However, level vL_i in the vTree is not required to preserve the same fully-sorted ordering with its correlated level L_i in the LSM-tree.

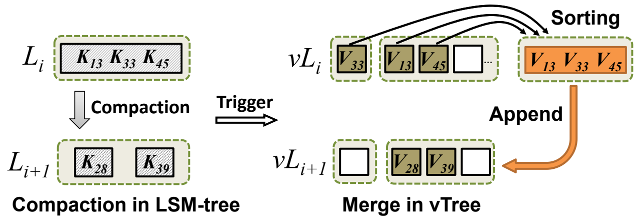


Figure 5: Compaction-triggered merge.

Figure 5 shows the idea of a compaction-triggered merge, which operates as follows. When a compaction operation is triggered to compact keys from L_i to L_{i+1} in the LSM-tree, it also triggers a merge operation that moves the corresponding values from vL_i to vL_{i+1} in the vTree. In the merge operations, there are two major issues: (i) which values should be involved in the merge and (ii) how to write back these values to level vL_{i+1} in the vTree. First, we merge only the values at level vL_i, and their corresponding keys must participate in the compaction in the LSM-tree. We call the value whose key participates in the compaction to be the *compaction-related value* for ease of presentation. Second, we reorganize all compaction-related values at level vL_i to generate new vTables and write these vTables to level vL_{i+1} in an append-only manner. For the example in Figure 5, the values $V_{33}, V_{13}, V_{45}, V_{28}, V_{39}$ are compaction-related values, and V_{33}, V_{13}, V_{45} will be involved in the merge, and finally appended to level vL_{i+1} after sorting.

Note that all values in the generated vTables in each merge are fully sorted; that is, they form only one single sorted group. However, we point out that the merge operation does not require to reorganize all vTables in both levels of vL_i and vL_{i+1} in the vTree. That is, when merging, a new sorted group is created in level vL_{i+1} with all vTables from level vL_i it. This avoids the rewrites of all values at level vL_{i+1}, thereby mitigating write amplification. In addition, the old vTables at vL_i will not be deleted during a merge, as they may still contain valid values, and they will be reclaimed later by GC.

The benefits of a compaction-triggered merge are two folds. First, merging only compaction-related values makes it very efficient to identify which values are still valid, as the corresponding keys also need to be read out from the LSM-tree during a compaction. In contrast, if the vTree is independent of the LSM-tree and triggers merge operations independently, then it needs to query the LSM-tree and compare the value locations to determine the validity of values, thereby inevitably incurring large query overhead. Second, as the locations of valid values are changed when generating new vTables during a merge operation, the LSM-tree needs to be updated accordingly to maintain the latest value locations. Since only the compaction-related values are merged, updating the value locations in the LSM-tree can be executed by directly updating the KV pairs participating in the compaction. Thus, the overhead of updating value locations can be hidden in the compaction operation as the compaction itself needs to rewrite the KV pairs in the LSM-tree.

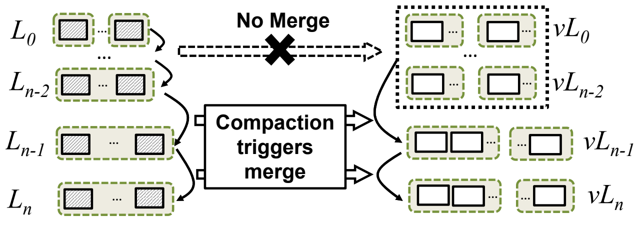


Figure 6: Lazy merge.

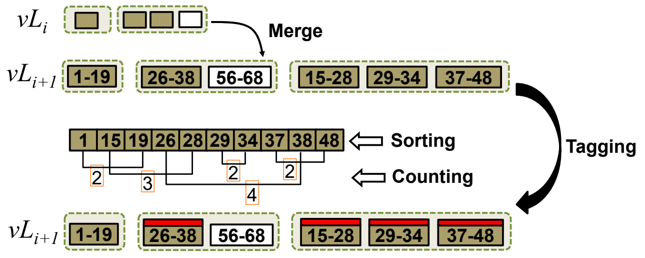


Figure 7: Scan-optimized merge.

3.4 Merge Optimizations

Compaction-triggered merge operations incur limited merge overhead caused by checking the validity of values and writing back new value locations into the LSM-tree. However, letting each compaction operation trigger a merge operation may cause frequent merge operations. For example, if each level in the vTree is correlated with only one level in the LSM-tree, then each compaction operation must trigger a merge operation in the vTree. To further reduce the merge overhead in the vTree, we propose two merge optimizations.

Lazy merge. We propose *lazy merge* to limit the merge frequency, and hence the merge overhead, in DiffKV. Our idea is to aggregate multiple lower levels in the vTree as a single level, and correlate the aggregate level with multiple levels in the LSM-tree. Specifically, as depicted in Figure 6, we aggregate all levels $vL_0, \cdots, vL_{n-2}$ in the vTree as a single level and correlate the aggregate level with levels from L_0 to L_{n-2} in the LSM-tree. Thus, any compaction between level $L_0, \cdots, L_{n-2}$ will not trigger a merge operation; in other words, the merge operations between level $vL_0, \cdots, vL_{n-2}$ will be delayed unless the values need to be merged into level vL_{n-1}.

Lazy merge significantly reduces the number of merge operations and the amount of data size being merged, but sacrifices the degree of ordering for the values in lower levels in the vTree. Nevertheless, we argue that the sacrifice poses limited degradation to the scan performance. Recall that the LSM tree increases its capacity toward higher levels (e.g., the size of L_i is $10\times$ of that of L_{i-1} in LevelDB [27]) (§2.1). Thus, the last two levels L_{n-1} and L_n contain the majority of KV pairs. The uneven data distribution across levels implies that most values are actually retrieved from the last two levels of the vTree for scans, so the degree of ordering of values in the last two levels is the dominant factor that determines the scan performance. In other words, the low levels of the vTree only have limited impact on scan performance, and the frequent merge operations in the low levels do not help scan performance but instead incur large merge overhead.

Scan-optimized merge. We adjust the degree of ordering for values in the vTree via scan-optimized merge, so as to maintain high scan performance. Recall that in a compaction-triggered merge operation, say merging the values from vL_i to vL_{i+1}, only the values in the lower level vL_i are reorganized and appended to the higher level vL_{i+1}, while the values in level vL_{i+1} are not involved in the merge operation and will not be sorted (§3.3). This append-only merge policy mitigates write amplification, but may result in too many sorted groups, which may have overlapped key ranges as values are not sorted across sorted groups. Thus, our idea is to find out the vTables which have overlapped key ranges with many other vTables, and also make them participate in the merge process regardless of which level they reside. With this, we can preserve a higher degree of ordering for values in the vTree, and thus benefit the scan performance.

Figure 7 depicts the idea of scan-optimized merge. After the normal compaction-triggered merge, we further check the vTables that contain compaction-related values at level vL_{i+1}. Our goal is to identify the set of vTables that satisfy two conditions: (i) at least one vTable in the set has overlapped key range with others, and (ii) the number of vTables (i.e., the set size) is larger than a pre-defined threshold, denoted by max_sorted_run. The rationale is that if such a set of vTables exists, then the scan performance may degrade as these vTables are not in a sorted order. We add a *scan optimization tag* for these vTables, so that they will always participate in the next compaction-triggered merge and increase the degree of ordering for values in tagged vTables.

To identify the set of vTables for tagging, we first retrieve the start and end keys of each vTable containing compaction-related values at level vL_{i+1}, and sort these keys. For each checked vTable, we count the number of vTables that have overlapped key ranges with it, and this can be done by scanning once the sorted key string. For example, as in Figure 7, consider a checked vTable [26-38]. By scanning the sorted string, we can count the number of start keys before key 38 (i.e., five in this case) and the number of end keys before key 26 (i.e., one in this case). By subtracting the two numbers, we can obtain the number of vTables that have overlapped key ranges with vTable [26-38], which is four including itself (i.e., vTables [15-28], [29-34], [37-48], and [26-38]). Finally, if the number of vTables with overlapped key ranges is larger than the threshold max_sorted_run, then we add a scan optimization tag for all these vTables to include them in the next compaction-triggered merge.We persist the optimization tags in a *manifest file*, which is already used by existing systems to track the version changes of KV pairs after each compaction; the persistence overhead is negligible.

Scan-optimized merge is an enhancement to compaction-triggered merge: a compaction-triggered merge operation only appends values in a lower level to its next higher level, while a scan-optimized merge operation further includes certain values in the higher level into the merge to increase the degree of ordering of values in the vTree. Note that scan-optimized merge incurs limited merge overhead (see Figure 15 in §5.4) for two reasons. First, we allow each level in the vTree to have multiple sorted groups (i.e., the whole level is not necessarily fully sorted). Second, not all values in a tagged vTable participate in the merge, but instead only the compaction-related values are merged.

3.5 Garbage Collection

The vTree rewrites compaction-related values to new vTables (§3.3), so it necessitates garbage collection (GC) to reclaim the space of invalid values (in the LSM-tree, its invalid data is reclaimed via compaction). To reduce the GC overhead, we propose a *state-aware lazy approach* based on the amount of invalid values in each vTable.

State awareness. DiffKV tracks the amount of invalid values in each vTable in a hash table. Each time when a vTable participates in a merge operation, DiffKV counts the amount of values being retrieved from the vTable and updates the amount of invalid values in the old vTable in the hash table. It also inserts an entry for any new vTable in the hash table. The updates to the hash table are executed during a merge operation, so the overhead is limited. Also, each entry in the hash table only occupies few bytes for each vTable, so the memory overhead of the hash table is limited.

Lazy GC. DiffKV takes a lazy approach to limit the GC overhead. It selects a vTable as a GC candidate if the vTable has a fraction of invalid values greater than a predefined threshold (denoted by gc_threshold). Note that DiffKV does not immediately reclaim the candidate vTables; instead, it simply marks a *GC tag* for each candidate, and delays the GC until the next compaction-triggered merge. Specifically, if a vTable with a GC tag is involved in a compaction-triggered merge, the values contained in this vTable will always be rewritten to the next higher level (similar to scan-optimized merge).

Lazy GC avoids the extra overhead of querying the LSM-tree for the validity of values in the candidate vTable and updating the LSM-tree for the new locations of the valid values. It is now delayed to be executed together with the merge operation, so that the overhead of querying and updating the LSM-tree can be hidden within the merge operation.

3.6 Discussion

Optimizing compaction at L_0. As KV separation is executed during flushes, SSTables at level L_0 in the LSM-tree may be very small, especially when KV pairs are large, so we propose a simple optimization called *selective compaction* to aggregate small SSTables at L_0. Specifically, we trigger intra-level compaction which simply combines multiple small SSTables at L_0 to generate a new large one without merging with SSTables at L_1. With selective compaction, the size of SSTables at L_0 will eventually become comparable to that of SSTables at L_1, and hence no extra compaction overhead will be introduced due to KV separation.

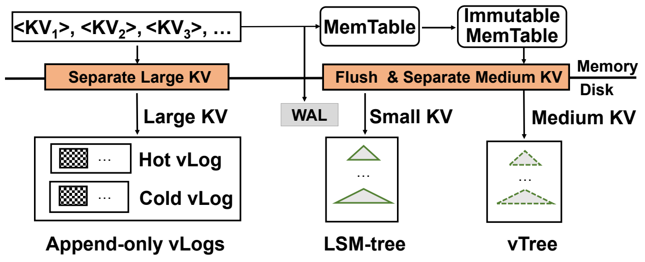


Figure 8: Fine-grained KV separation.

Crash consistency. DiffKV is implemented on Titan (which builds on RocksDB) (§5), and provides the same level of consistency as RocksDB after system crashes. To guarantee data consistency, DiffKV uses a write-ahead log (WAL), and writes KV pairs to the WAL before writing to the MemTable. Also, DiffKV provides crash consistency for the hash table that records the amount of invalid data in each vTable (§3.5). As the hash table is updated after compaction, DiffKV appends the update information into the manifest file (§3.4) after compaction, so it can be restored when DiffKV recovers from a crash.

4 Fine-grained KV Separation

The benefit of KV separation is significant for large KV pairs, but diminishes for small KV pairs (§2.3). However, mixed workloads with varying value sizes are also common; for example, the value size may vary in a large range under the generalized Pareto distribution [25]. In this section, we further enhance DiffKV via fine-grained KV separation by distinguishing KV pairs by value sizes, so as to achieve balanced performance under mixed workloads.

4.1 Differentiated Value Management

DiffKV classifies KV pairs into three groups based on the value size, by using two parameters, namely value_small and value_large, as depicted in Figure 8. For KV pairs whose value size is larger than value_large (i.e., large KV pairs), DiffKV adopts KV separation, and manages values with a hotness-aware multi-log design called *vLogs*. For KV pairs whose value size is between value_small and value_large (i.e., medium KV pairs), DiffKV stores values in the vTree and keep both keys and value locations in the LSM-tree. For KV pairs whose value size is smaller than value_small (i.e., small KV pairs), DiffKV bypasses KV separation and stores the whole KV pairs directly in the LSM-tree. As a result, DiffKV achieves balanced performance for different value sizes.

Note that for large KV pairs, DiffKV adopts the workflow as in Wisckey [42], in which KV separation is performed before writing to MemTable (see Figure 8). Specifically, DiffKV directly flushes the values of large KV pairs into vLogs, treats their keys and value locations as new KV pairs, and writes

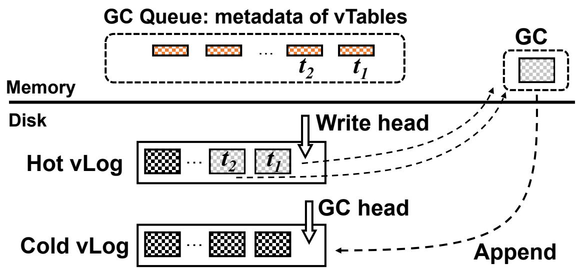


Figure 9: GC for vLogs.

them to MemTable as regular user data. The benefits of performing KV separation early for large KV pairs are two-fold. First, by directly flushing large-size values into vLogs and keeping only small-size value locations in the MemTable, we can save substantial memory space, while still guaranteeing high write performance due to large sequential I/Os. Second, as large-size values are first written to disk, there is no need to write them to the WAL, and this reduces the amount of I/Os. Note that for small and medium KV pairs, as well as the keys and value locations for large KV pairs, they still need to be written to the WAL so as to guarantee consistency (§3.6).

4.2 Hotness-aware vLogs

Structure of vLogs. A vLog is designed as a simple circular append-only log, which consists of a set of *unsorted vTables*. Unsorted vTables share a similar storage format with vTables described in §3.2, and the only difference is that values are written to *unsorted vTables* with append, so they are not sorted even within each unsorted vTable. The reason why we do this is because KV separation for large KV pairs is performed before writing to the MemTable, and values for large KV pairs are flushed to disk immediately after KV separation so as to avoid writing to WAL, so there is no way to sort the values in each vTable (see Figure 8). In fact, there is no need to sort these values as they have a large size, and hence they can already benefit from large I/Os without batched writes.

GC for vLogs. To reduce GC overhead, we leverage a hotness-aware design by employing a simple yet efficient parameterless hot-cold separation scheme. As shown in Figure 9, we adopt two vLogs, namely a *hot vLog* and a *cold vLog*, to store hot and cold values, respectively. Each vLog has its own write frontier, and we call them *write head* and *GC head*, respectively. To realize hot-cold separation, the data from user writes are appended to the write head in the hot vLog, and the data from GC writes (i.e., the valid values that need to be written back after GC) are appended to the GC head in the cold vLog. The rationale is that the values reclaimed by GC are usually accessed less frequently than the recently written user data, so they can be regarded as cold data. One benefit of this design is that it is simple to implement, as no parameter is required to realize hot-cold identification. Clearly, we can also apply alternative hotness-aware classification schemes.

DiffKV employs a *greedy* algorithm to reduce GC cost, and the idea is to reclaim the unsorted vTables which have the largest amount of invalid values. Specifically, DiffKV monitors the ratio of invalid values of each unsorted vTable during compaction, and maintains a GC queue in memory to track all candidate vTables, which are the unsorted vTables with the ratio of invalid values being greater than the threshold gc_threshold. Note that the GC queue only keeps the metadata of each unsorted vTable, and it is maintained in a descending order according to the ratio of invalid values. When GC is triggered, DiffKV simply selects unsorted vTables tracked in the queue head (e.g., t_1, and t_2 in Figure 9), then appends the valid values in the selected vTables to the GC head in the cold vLog. For performance consideration, DiffKV implements GC as a background process with multiple GC threads.

5 Evaluation

In this section, we evaluate and compare DiffKV with the three state-of-the-art KV stores introduced in §2.3: RocksDB [24], PebblesDB [52], and Titan [51]. For fair comparisons with Titan, we also implement DiffKV based on it and our modifications contain around 2.1K lines of code.

5.1 Setup

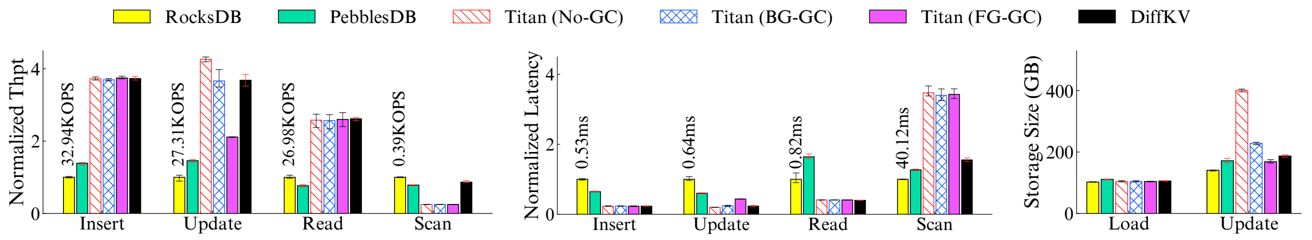
Testbed. We conduct experiments on a single machine equipped with an 12-cores Intel Xeon E5-2650v4 CPU, 16 GB memory, and Samsung 860 EVO 480 GB SSD. The machine runs Ubuntu 18.04 LTS with Linux kernel 4.15.

Workload. We modify YCSB-C [48, 54], a C++ version of YCSB [14], to generate workloads based on the workload statistics in [9]. We fix the key size as 24 bytes, and configure the value size with the generalized Pareto distribution [29], whose probability density function is:

$$f(x) = (1/\sigma)(1 + k(x - \theta)/\sigma)^{(-1-1/k)} \quad (1)$$

where x represents the value size, k, θ, and σ are adjustable parameters. By default, we limit the maximum value size as 128 KB, and set $k = 0.92, \sigma = 226, \theta = 0$ by using the most common workload setting in [25]. Under this setting, the average value size is 1 KB.

System configuration. We refer to the official tuning manual for experimental verification [26]. For all KV stores, we use the recommended configuration by setting MemTable size as 64 MB, SSTable size as 16 MB, configure Bloom filters by setting 10 bits/key. Since our testbed machine has 12 cores, to speedup compactions and also limit overall CPU usage, we use 8 background threads for compaction by following the optimization tuning guide. We also allocate 8 GB memory for block cache and leave the rest for page cache in operating system. For Titan, as GC affects both space usage and foreground write performance, we consider three cases: (i) *no GC (No-GC)*, which achieves the best write performance, but incurs the largest space overhead; (ii) *background GC (BG-GC)*, in which GC is executed in the background without blocking foreground writes; and (iii) *foreground GC (FG-GC)*, in which a limit on space usage is set and GC may block



(a) Throughput of different operations.
(b) Average latency of different operations.
(c) Space usage.

Figure 10: Microbenchmarks on RocksDB, PebblesDB, Titan, and DiffKV.

foreground writes if the free space drops below the predefined limit. For DiffKV, the two thresholds for differentiating small, medium, and large KV pairs are set as 128 bytes and 8 KB, respectively. We limit the vTable size as 8 MB, and set a GC tag to a vTable if it contains more than 30% invalid data. We study the impact of these parameters in §5.6. Each experiment was run at least five times.

5.2 Microbenchmarks

We first study the throughput and latency performance of various KV operations with the following workloads: (i) insert 10 GB KV pairs, (ii) update 300 GB KV pairs, (iii) read 10 GB KV pairs, and (iv) scan 10 GB KV pairs. We first randomly load 100 GB KV pairs, then issue the requests of each workload, and finally clear all KV pairs from the KV store to avoid interference. For scans, we use the widely used configuration for performance evaluation [8, 14, 43, 65], i.e., we issue 16 scan threads, each of which reads 100 KV pairs. We also study the impact of other scan settings in §5.5. By default, we use a Zipf distribution with the skewness parameter 0.9 for each workload. We also run our evaluation under uniform workloads and observe similar conclusions, so we omit the results in the interest of space.

Throughput. Figure 10(a) shows the throughput results, normalized with respect to the throughput of RocksDB for ease of comparison. Compared to RocksDB and PebblesDB, DiffKV achieves 3.8× and 2.7× insert throughput, 3.7× and 2.3× update throughput, 2.6× and 3.4× read throughput, respectively. Thus, DiffKV significantly improves both write and read throughput, with comparable scan performance.

Compared to Titan, DiffKV always improves the scan performance significantly, regardless of the GC policy used in Titan. For example, DiffKV achieves 3.2× scan throughput over Titan. For write performance, even compared to the case of NO-GC (i.e., the case of best write performance for Titan), DiffKV still has similar write performance. Furthermore, compared to the case of using foreground GC in Titan, DiffKV achieves 1.7× update throughput. Note that since the throughput results are evaluated in OPS, the throughput of scans is much lower than that of other operations.

Average latency. Figure 10(b) shows the latency results. Compared to RocksDB and PebblesDB, DiffKV reduces the latency of inserts, updates, and reads by up to 63.8%-78.1%, with a similar scan latency. Compared to Titan, DiffKV has similar latencies for inserts, writes, and reads, while reducing the scan latency by up to 43.2%.

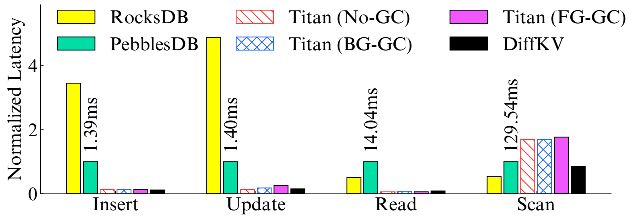


Figure 11: Tail latency of different operations.

Workload	Statistics
A (Update Heavy)	50% updates, 50% reads
B (Read Mostly)	5% updates, 95% reads
C (Read Only)	100% reads
D (Read Latest)	5% inserts, 95% reads
E (Scan Mostly)	5% inserts, 95% scans
F (Read-Modify-Write)	50% read-modify-write, 50% reads

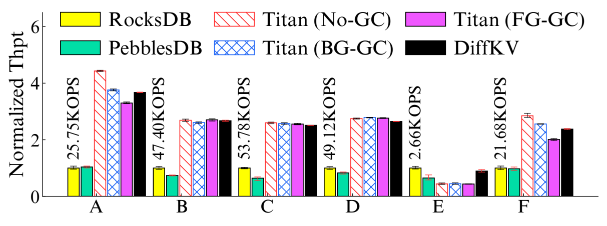
Table 1: YCSB core workloads.

Space usage. Figure 10(c) shows the space usage under GC, in which we randomly load 100 GB KV pairs and update 300 GB KV pairs with a Zipf distribution. As no GC is triggered in the load phase, all KV stores show the same space usage. However, after the update phase, Titan incurs large space usage if it disables GC or uses background GC only. DiffKV reduces space usage by up to 18%-53.7% compared to Titan (NO-GC) and Titan (BG-GC).

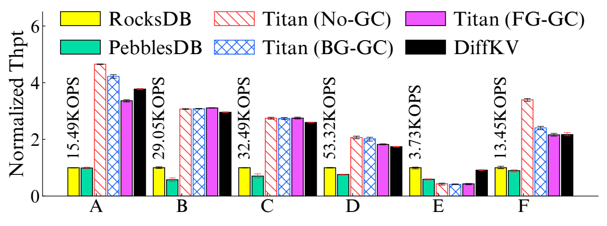
Tail Latency. We evaluate the 99-th percentile tail latency as shown in Figure 11. We normalize the results with respect to PebblesDB. Compared to RocksDB and PebblesDB, DiffKV has a much lower tail latency for inserts, updates, and reads. For example, it reduces the tail latency of inserts, updates, and reads of RocksDB by 96.5%, 94.3%, and 82.7%, respectively, while keeping similar tail latency for scans. Compared to Titan, DiffKV has a similar tail latency for inserts, updates and reads, but reduces the scan tail latency by 50.4%.

5.3 YCSB Evaluation

We show the performance of DiffKV under the YCSB workloads (Table 1). Each workload performs 100M operations on a randomly loaded 100 GB data store, other settings are the same as before. We consider both uniform distribution



(a) Zipfian workload.



(b) Uniform workload.

Figure 12: Performance under YCSB workloads.

and Zipf distribution with parameter 0.9, except workload D which reads the latest data as defined by the benchmark [14].

Throughput. Figure 12 shows the throughput results, normalized with respect to the throughput of RocksDB. Compared to RocksDB, DiffKV achieves 1.7-4.5× throughout for all read- or write-intensive workloads (i.e., all except workload E), and achieves similar performance under the scan-dominant workload E. Note that RocksDB has KV pairs fully sorted in each level, so it represents the best case in terms of scan performance. Compared to Titans, DiffKV achieves 2× scan throughput, while keeping similar performance under other workloads regardless of whether background GC or foreground GC are used. In short, DiffKV achieves balanced performance in all aspects.

Tail latency. Figure 13 shows the tail latency results. As YCSB workloads are mixed with different operations with highly different latencies, we show the tail latency of each type of operations. We normalize the results with respect to Titan (NO-GC). We observe a similar conclusion as in the case of microbenchmarks. That is, compared to RocksDB and PebblesDB, DiffKV significantly reduces the tail latency of inserts, updates and reads; compared to Titan, DiffKV reduces the tail latency of scans, so it always performs almost the best in all performance aspects.

Space usage. We show the space usage under YCSB workload A, which has 50% updates. We observe similar results as in Figure 10(c), i.e., DiffKV increases the space usage by 11.9% and 0.7% compared to RocksDB and PebblesDB, respectively. Also, the LSM-tree, vTree, and vLogs incur 5%, 63.5%, and 31.5% of space in DiffKV, respectively.

5.4 Analysis on Merge Optimizations

We evaluate the merge optimization techniques of DiffKV (§3.3-§3.4) and show how they address the write-scan trade-

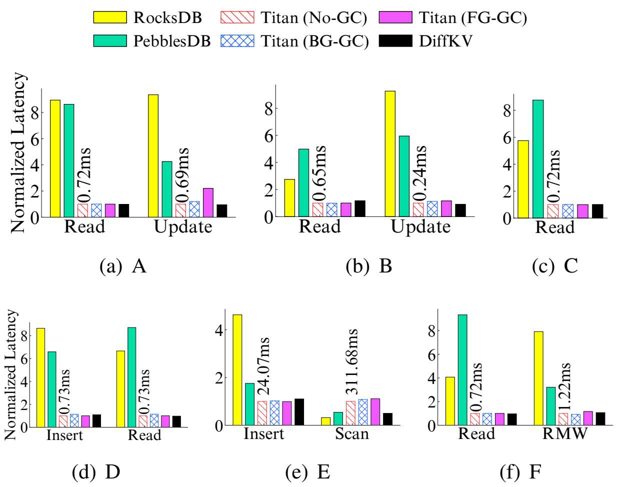
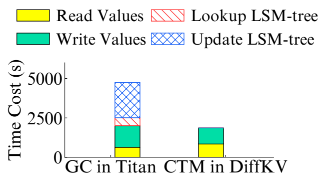
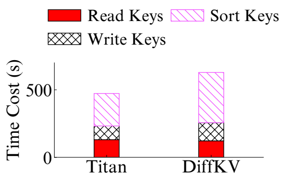


Figure 13: Tail latency of different ops under YCSB workloads.



(a) Value GC/merge overhead.



(b) Key compaction time.

Figure 14: Impact of coordinated value management in DiffKV: it significantly reduces the value merge overhead, and slightly increases key compaction overhead.

off and realize the balanced performance for DiffKV.

We first compare the merge overhead in DiffKV with the GC overhead in Titan. Here, we deploy only the compaction-triggered-merge (CTM) in DiffKV and focus on the effectiveness of the coordinated design. Figure 14(a) shows the time cost breakdown for GC in Titan and merge in DiffKV. We issue a workload of updating 300 GB KV pairs with a Zipf distribution on a randomly loaded 100 GB KV store, using the default background GC for Titan. DiffKV costs much less time than Titan for value management, with a 60.7% reduction of time cost. The time saving mainly comes from the coordinated merge design, in which the overhead of looking up the LSM-tree and updating the new value locations to LSM-tree can be avoided via the compaction-triggered merge. However, as compaction must wait for the completion of the triggered merge, the compaction time slightly increases. Such an overhead can be mitigated via the merge optimizations (i.e., lazy merge and scan-optimized merge).

Figure 15 studies the impact of the merge optimizations. Lazy merge (LM) further reduces the number of merge operations and the amount of merged data size compared to using only the compaction-triggered merge (CTM), by 65.5% and 66.2%, respectively. However, it increases the number of sorted groups in vTree, which is the key factor of influencing scan performance. For example, lazy merge adds up to 20% sorted groups in the last two levels. By further in-

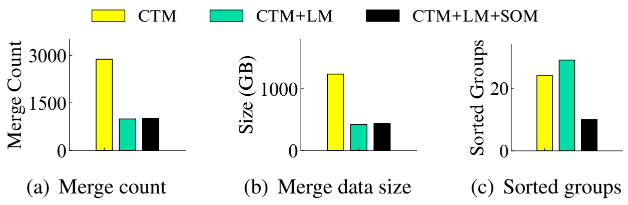


Figure 15: Impact of merge optimizations on merge overhead.

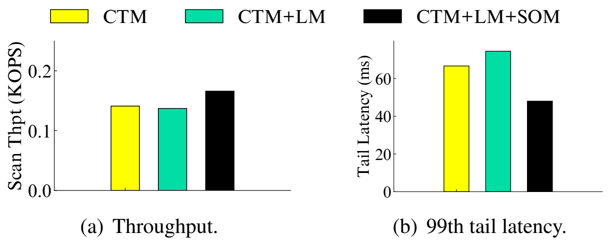


Figure 16: Scan performance.

corporating scan-optimized merge (SOM), the number of sorted groups reduces significantly, by up to 68% compared to lazy merge. Meanwhile, the merge overhead keeps almost the same. Due to the reduced number of sorted groups, the scan performance improves. For example, as shown in Figure 16, scan-optimized merge increases 18% of scan throughput and reduces 27% of the 99th tail latency compared to lazy merge. In summary, by combining lazy merge and scan-optimized merge with the coordinated design, DiffKV guarantees desired ordering for values with limited merge overhead, and hence achieves balanced performance in all aspects.

5.5 Scan Performance

Recall that the main benefit of DiffKV over Titan is its scan improvement. We further examine the scan performance by varying the *scan length* (i.e., the number of KV pairs read by each scan) and the number of scan threads (i.e., the number of threads initiated by clients for concurrent access).

Figure 17(a) shows the scan performance versus the scan length (from 20 to 10000), while fixing the total read size as 10 GB. As GC does not influence scan performance for Titan, we focus on only the case without GC. DiffKV significantly outperforms Titan, and as the scan length increases, the performance gain further increases. The reason is that Titan stores values in an unsorted manner and hence has poor scan performance, while DiffKV maintains a partial ordering for values. Even compared to RocksDB, which represents the optimum for scan, DiffKV still reaches 83% of scan throughput when the scan length is 20, and the ratio further increases to 96% when the scan length increases to 10000. Thus, DiffKV achieves similar scan performance with RocksDB.

Figure 17(b) studies the impact of the number of scan threads (from 8 to 64), while fixing the scan length as 100. DiffKV still significantly outperforms Titan, and achieves comparable performance with RocksDB under all settings.

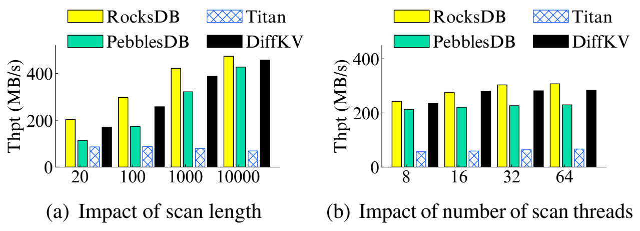


Figure 17: Scan performance under different settings.

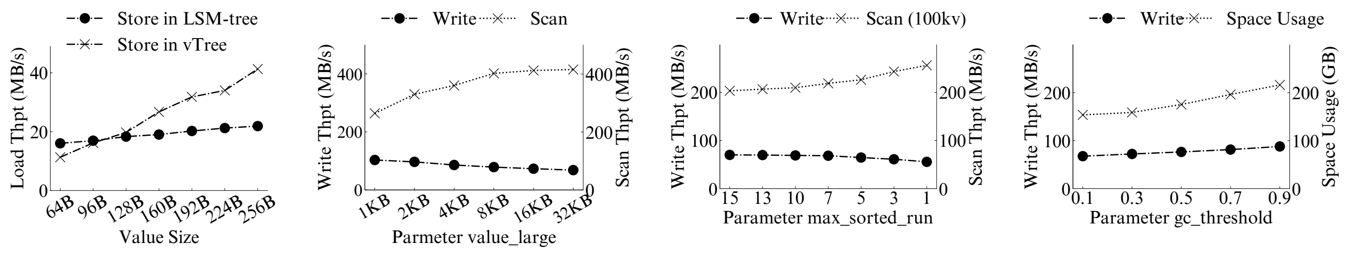
5.6 Tunable Parameters

We analyze the sensitivity of the parameters in DiffKV. First, the parameter value_small is used to differentiate small and medium KV pairs. In other words, for each value size, if KV separation brings no benefit to write, then we should treat this kind of KV pairs as small ones and keep them in the LSM-tree; otherwise, we should leverage KV separation and store values in vTree. Thus, to find an appropriate setting for parameter value_small, we compare the write performance of using the LSM-tree and the vTree under different value sizes. Figure 18(a) shows the load throughput versus the value size. When the value size is at least 128 bytes, using the vTree improves write performance (i.e., KV separation is beneficial). Thus, we set value_small as 128 bytes by default.

Second, we justify how to set an appropriate value for value_large, which influences both the write and scan performance. A smaller value_large means more KV pairs are regarded as large ones and stored in vLogs. We focus on a mixed workload generated with the default parameters (§5.1), and show the write and scan performance by varying value_large from 1 KB to 32 KB. As shown in Figure 18(b), when value_large increases from 8 KB to 32 KB, the marginal improvement of the scan performance is very limited. This implies that managing KV pairs whose value sizes are larger than or equal to 8 KB with vLogs only slightly degrades the scan performance. On the other hand, for write throughput, compared to the case of setting value_large as 1 KB, it already achieves around 80% of the throughput when the parameter is set as 8 KB, Thus, we set value_large as 8 KB by default.

Third, the parameter max_sorted_run controls the number of sorted groups in each of the last two levels in vTree, so it influences the scan performance and merge overhead, i.e., it determines the write-scan trade-off. Figure 18(c) shows the write-scan trade-off by varying max_sorted_run from 15 to one. Here, we run a workload which scans 10 GB data on a randomly loaded 100 GB data store and each scan requests 100 KV pairs. When there are more than 10 sorted groups in each level, both the write and scan performance change slowly, so we set max_sorted_run as 10 by default.

Finally, we study the impact of gc_threshold on write performance and space usage. Recall that a vTable will be set with a GC tag if its invalid data exceeds the threshold defined by gc_threshold, so a smaller gc_threshold means more frequent GC, and both the write performance and space usage should



(a) Analysis of value_small (b) Impact of value_large (c) Impact of max_sorted_run (d) Impact of gc_threshold

Figure 18: Empirical study on tunable parameters in DiffKV. Figure (a) justifies the setting of value_small by comparing the write performance of using LSM-tree and vTree under different value sizes. Figure (b) justifies the setting of value_large by studying its impact on write and scan performance. Figure (c) justifies the setting of max_sorted_run. Figure (d) shows the impact of gc_threshold.

decrease. Figure 18(d) shows the results. As gc_threshold increases, the increased write throughout becomes smaller. This implies that with the lazy GC, the impact of GC on write performance is limited. On the other hand, the space usage significantly increases (from 158.8 GB to 216.2 GB) when we increase gc_threshold from 0.3 to 0.9. Thus, we set gc_threshold as 0.3 by default.

6 Related Work

Research on KV stores has received a lot of attentions. In addition to building KV stores on new hardware like non-volatile memory [12, 13, 21, 34–36, 39, 44, 61] or characterizing real-world KV workloads [5, 9], many studies focus on optimizing the read/write performance of LSM-tree KV stores [6, 10, 15, 16, 32, 40, 42, 52, 56, 58, 62, 63].

Extensive efforts focus on reducing the compaction overhead of LSM-tree KV stores. One line of studies follows the idea of relaxing the fully-sorted ordering of KV pairs. PebblesDB [52] proposes a fragmented LSM-tree that relaxes the fully-sorted ordering of KV pairs by dividing each level into multiple non-overlapped segments and allowing KV pairs within each segment to be unsorted. dCompaction [47] delays compaction by constructing virtual SSTables that contain only metadata for multiple SSTables with overlapped ranges, and it can also be regarded as a relaxation of the fully-sorted ordering of KV pairs. Dostoevsky [17] introduces a lazy-leveling merge policy by adopting the leveling policy only for the last level and using tiering for other levels. The ideas of relaxing the fully-sorted ordering are also found in VT-Tree [57], LWC-tree [60], SlimDB [55], and SifrDB [45] .

Another line of studies of mitigating the write amplification problem is KV separation. WiscKey [42] is the first work that proposes this technique by storing values in a separate append-only circular log, and Bourbon [15] extends WiscKey by integrating a learning approach for indexing the values. Based on KV separation, a lot of efforts are made to reduce the overhead of log management for values, especially the GC overhead [10, 20, 49, 51]. HashKV [10] leverages hash-based data grouping so as to reduce the GC cost of the circular log. BadgerDB [20] reuses the write-ahead log (WAL) as a value log, so as to save the data flush overhead. Titan [51] adopts KV separation and leverage BLOB files for value management. We point out that all these studies follow the design of append-only logs for value management. In contrast, DiffKV proposes the vTree for maintaining the partially-sorted ordering for values, so as to realize balanced I/O performance.

Multiple studies also leverage hash indexing or optimize the Bloom filter to improve read and write performance. For hash indexing, LSM-trie [58] utilizes a hash-based trie structure to improve the performance for small KV pairs, while UniKV [63] unifies hash indexing and the LSM-tree to simultaneously improve both read and write performance. For the Bloom filter optimization, Monkey [16] proposes to differentiate Bloom filters between different levels; ElasticBF [40] further develops a fine-grained elastic Bloom filter scheme to improve read performance; SuRF [62] introduces a succinct range filter to optimize both reads and scans.

Unlike existing studies that usually possess performance trade-offs, DiffKV aims to realize balanced I/O performance by simultaneously improving the performance of writes, reads, and scans. DiffKV adopts KV separation, and takes one step further to adopt a new vTree structure with a coordinated design to realize the differentiated ordering for keys and values. It also adopts fine-grained KV separation, so as to realize balanced performance under mixed workloads.

7 Conclusion

In this paper, we propose to leverage differentiated ordering for keys and values to simultaneously achieve high performance for writes, reads, and scans. We develop DiffKV, which follows KV separation and utilizes a new structure vTree for value management with a partial ordering. By leveraging a coordinated design and multiple merge optimizations, DiffKV achieves efficient writes, reads, and scans with low storage cost, and thus realizes balanced performance in all aspects.

Acknowledgments: We thank our shepherd and the anonymous reviewers for their comments. This work is supported by National Key R&D Program of China (2018YFB1003204), NSFC (61772484, 61772486), Youth Innovation Promotion Association CAS, and USTC Research Funds of the Double First-Class Initiative (YD2150002003). Yongkun Li is USTC Tang Scholar. Yinlong Xu is the corresponding author.

References

[1] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proc. of of ACM SOSP*, 2019.

[2] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *Proc. of USENIX ATC*, 2017.

[3] Apache. Cassandra. `http://cassandra.apache.org/`.

[4] Apache. HBase. `https://hbase.apache.org/book.html`.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[6] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. of USENIX ATC*, 2019.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc. of USENIX ATC*, 2013.

[8] C. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *Proc. of USENIX FAST*, 2020.

[9] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-value Workloads at Facebook. In *Proc. of USENIX FAST*, 2020.

[10] H. Chan, Y. Li, P. Lee, and Y. Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proc. of USENIX ATC*, 2018.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[12] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proc. of ACM ASPLOS*, 2021.

[13] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proc. of USENIX ATC*, 2020.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[15] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. From Wisckey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proc. of USENIX OSDI*, 2020.

[16] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Bavigable Key-Value Store. In *Proc. of ACM SIGMOD*, 2017.

[17] 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. of ACM SIGMOD*, 2018.

[18] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of ACM SIGMOD*, 2011.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of of ACM SOSP*, 2007.

[20] Dgraph. BadgerDB. `https://github.com/dgraph-io/badger`.

[21] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proc. of ACM EuroSys*, 2018.

[22] N. Elyasi, C. Choi, and A. Sivasubramaniam. Large-Scale Graph Processing on Emerging Storage Devices. In *Proc. of USENIX FAST*, 2019.

[23] Facebook. Memcached. `http://memcached.org`.

[24] Facebook. RocksDB. `http://rocksdb.org/`.

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

[26] Facebook. RocksDB Tuning Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`.

[27] Google. LevelDB. `https://github.com/google/leveldb`.

[28] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook Messages Case Study. In *Proc. of USENIX FAST*, 2014.

[29] J. R. Hosking and J. R. Wallis. Parameter and Quantile Estimation for the Generalized Pareto Distribution. *Technometrics*, 29(3):339–349, 1987.

[30] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. TiDB: A Raft-Based HTAP Database. *Proc. of VLDB Endow.*, 13(12):3072–3084, 2020.

[31] HyperDex. HyperLevelDB Performance Benchmarks. `http://hyperdex.org/performance/leveldb/`.

[32] J. Im, J. Bae, C. Chung, and S. Lee. PinK: High-Speed In-Storage Key-Value Store with Bounded Tails. In *Proc. of USENIX ATC*, 2020.

[33] X. Jiang, Y. Hu, Y. Xiang, G. Jiang, X. Jin, C. Xia, W. Jiang, J. Yu, H. Wang, Y. Jiang, J. Ma, L. Su, and K. Zeng. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. of VLDB Endow.*, 13(12):3272–3284, 2020.

[34] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proc. of USENIX FAST*, 2019.

[35] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proc. of USENIX ATC*, 2018.

[36] K. Kourtis, N. Ioannou, and I. Koltsidas. Reaping the Performance of Fast NVM Storage with uDepot. In *Proc. of USENIX FAST*, 2019.

[37] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. of USENIX OSDI*, 2012.

[38] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's Key-Value Storage System for Cloud Data. In *Proc. of IEEE MSST*, 2015.

[39] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proc. of ACM SOSP*, 2019.

[40] 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. of USENIX ATC*, 2019.

[41] G. Lu, Y. J. Nam, and D. H. Du. BloomStore: Bloom-Filter Based Memory-Efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *Proc. of IEEE MSST*, 2012.

[42] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proc. of USENIX FAST*, 2016.

[43] 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. of ACM SIGMOD*, 2020.

[44] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu. ROART: Range-query Optimized Persistent ART. In *Proc. of USENIX FAST*, 2021.

[45] F. Mei, Q. Cao, H. Jiang, and J. Li. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proc. of ACM SoCC*, 2018.

[46] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[47] F. Pan, Y. Yue, and J. Xiong. dCompaction: Delayed Compaction for the LSM-tree. *International Journal of Parallel Programming*, 45(6):1310–1325, 2017.

[48] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proc. of USENIX ATC*, 2016.

[49] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proc. of ACM SoCC*, 2018.

[50] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. Fast Scans on Key-Value Stores. *Proc. of VLDB Endow.*, 10(11):1526–1537, 2017.

[51] PingCAP. Titan. `https://github.com/tikv/titan`.

[52] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building Key-value Stores using Fragmented Log-Structured Merge Trees. In *Proc. of ACM SOSP*, 2017.

[53] RedisLib. Redis. `https://redis.io`.

[54] J. REN. YCSB-C. `https://github.com/basicthinker/YCSB-C`.

[55] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proc. of VLDB Endow.*, 10(13):2037–2048, 2017.

[56] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proc. of ACM SIGMOD*, 2012.

[57] 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. of USENIX FAST*, 2013.

[58] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proc. of USENIX ATC*, 2015.

[59] J. Yang, Y. Yue, and K. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[60] T. Yao, W. Jiguang, H. Ping, H. Xubin, G. Qingxin, W. Fei, , and X. Changsheng. A Light-Weight Compaction Tree to Reduce I/O Amplification Toward Efficient Key-Value Stores. In *Proc. of IEEE MSST*, 2017.

[61] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proc. of USENIX ATC*, 2020.

[62] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proc. of ACM SIGMOD*, 2018.

[63] Q. Zhang, Y. Li, P. P. Lee, Y. Xu, Q. Cui, and L. Tang. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proc. of IEEE ICDE*, 2020.

[64] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proc. of USENIX FAST*, 2015.

[65] W. Zhong, C. Chen, X. Wu, and S. Jiang. REMIX: Efficient Range Query for LSM-trees. In *Proc. of USENIX FAST*, 2021.