Enhancing LSM-tree Key-Value Stores for Read-Modify-Writes via Key-Delta Separation

Jinhong Li[†], Yanjing Ren[†], Shujie Han[*], Patrick P. C. Lee[†]
[†]*The Chinese University of Hong Kong*, [*]*Peking University*

***Abstract*—Read-modify-writes (RMWs) are increasingly observed in practical key-value (KV) storage workloads to support fine-grained updates. To make RMWs efficient, one approach is to write deltas (i.e., changes to current values) to the log-structured merge-tree (LSM-tree), yet it increases the read overhead caused by retrieving and combining a chain of deltas. We propose a notion called *key-delta (KD) separation* to support efficient reads and RMWs in LSM-tree KV stores under RMW-intensive workloads. KD separation aims to store deltas in separate storage areas and group deltas into storage units called buckets, such that all deltas of a key are kept in the same bucket and can be all together accessed in subsequent reads. To this end, we build KDSep, a middleware layer that realizes KD separation and integrates KDSep into state-of-the-art LSM-tree KV stores (e.g., RocksDB and BlobDB). We show that KDSep achieves significant I/O throughput gains and read latency reduction under RMW-intensive workloads while preserving the efficiency in general workloads.**

***Index Terms*—LSM-trees, key-value stores**

I. Introduction

Key-value (KV) storage is a critical paradigm for enabling structured storage at scale, by organizing data as *KV pairs* for reads (i.e., retrieving existing KV pairs from storage) and writes (i.e., inserting KV pairs into storage or overwriting existing KV pairs with new values). While read-intensive [4] and write-intensive [26], [29] workloads have been observed in production, field studies show that *read-modify-writes (RMWs)* are also common, in which existing KV pairs are read from storage, fully or partially modified based on the current values, and written back to storage. For example, 92.5% of all requests are reportedly RMWs [7] in artificial-intelligence/machine-learning (AI/ML) services in RocksDB [16] production at Facebook; more than 90% of transaction requests are RMWs in the transactional database benchmarks [1], [27]; 50% of requests are RMWs in one of the core cloud system workloads (Workload F) in YCSB benchmarks [12]. RMWs are also commonly observed in NoSQL stores [25], real-time data processing [9], [18], and social graph databases [3], [10].

RocksDB [16], which serves as the backend storage for Facebook's AI/ML services with intensive RMWs [7], builds on the widely adopted *Log-Structured Merge-tree (LSM-tree)* [22] for persistent KV storage. The LSM-tree arranges KV pairs in log-structured storage across multiple tree levels and issues sequential writes for new KV pairs for fast writes. It also maintains KV pairs in a sorted manner within each tree level for efficient reads and range queries. To support efficient RMWs, RocksDB [16] provides the `merge()` operator: instead of simply reading and modifying KV pairs, `merge()` writes the *delta* (i.e., the change of the value) of a KV pair without reads, thereby achieving fast RMWs. Thus, the LSM-tree is arguably a good candidate for RMW-intensive workloads.

The `merge()` operator, however, increases the overhead of subsequent reads to a key with deltas. For fast RMWs, `merge()` only writes deltas to the LSM-tree, but does not combine deltas with their associated KV pairs. Thus, reading a KV pair retrieves not only the current value, but also a chain of deltas, so as to reconstruct the latest version of the value, and such deltas may span across tree levels. Our evaluation (Section II-C) shows that compared with simply reading and modifying KV pairs, `merge()` reduces the RMW latency by up to 96.9%, but increases the read latency by up to 74.1%, under RMW-intensive workloads. One possible remedy to mitigating such read overhead is *KV separation* [20], which stores values in separate storage outside of the LSM-tree, so as to reduce the LSM-tree size and hence the read latency. However, the current implementation of KV separation of RocksDB, called BlobDB [14], still stores deltas in the LSM-tree. Our evaluation (Section II-C) shows that while KV separation reduces the read latency to some extent under RMW-intensive workloads, the read latency remains high.

In this paper, we propose *key-delta (KD) separation*, which aims for fast reads, while maintaining high RMW performance, under RMW-intensive workloads. KD separation is inspired by KV separation [20] and takes one step further to store deltas in separate storage outside of the LSM-tree. It builds on three design elements: (i) *bucket-based delta placement*, which groups deltas by *buckets*, such that all deltas of a key are mapped to the same bucket and any subsequent read to a key can be efficiently done by directly accessing a single bucket for all its corresponding deltas; (ii) *delta-based garbage collection*, which reclaims the space of stale deltas from buckets, and further allows buckets to be dynamically split and merged to keep the bucket sizes and the number of buckets manageable; and (iii) *crash recovery*, which ensures that the states of writes and garbage collection can be recovered after crashes.

To realize KD separation, we design and implement KDSep, a middleware layer that integrates KD separation into existing LSM-tree KV stores to efficiently handle RMW-intensive workloads. We demonstrate that KDSep can be feasibly integrated into existing LSM-tree KV stores with or without KV separation, including RocksDB, BlobDB, and vLogDB (the KV separation implementation in [8]). We also conduct asymptotic analysis on the performance of KDSep.

We evaluate KDSep using custom workloads that are

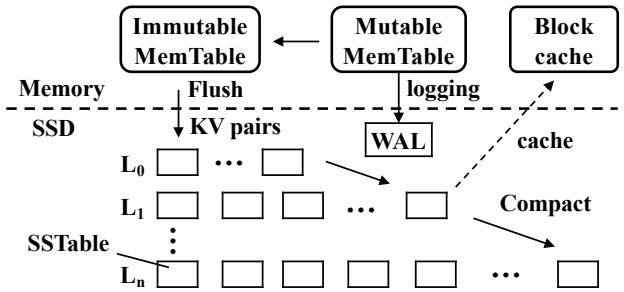


Figure 1: Architecture of an LSM-tree KV store.

synthetically generated with tunable parameters, the production workloads based on Facebook's RocksDB deployment [7], and YCSB core workloads [12]. Experimental results show that under RMW-intensive workloads, our KDSep prototype increases the I/O throughput by 67.0-80.5% and reduces the read latency by 41.9-49.9% compared with the baseline KV stores without KD separation. Our KDSep prototype is open-sourced at **https://github.com/adslabcuhk/kdsep**.

II. Background and Motivation

A. Basics of LSM-Trees

LSM-tree organization. We provide an overview of an LSM-tree KV store, as shown in Figure 1. An LSM-tree organizes KV pairs in $n+1$ levels, denoted by L_0, L_1, $\cdots$, L_n (from lower to higher). Each level stores KV pairs in immutable fixed-size files of several MiB each (e.g., 64 MiB default in RocksDB [16]), called *SSTables*, in persistent storage. Each SSTable groups KV pairs in *data blocks* of several KiB each and keeps an *index block* that stores the key ranges and offsets of all data blocks in the SSTable. For efficient range queries, each SSTable sorts all KV pairs by keys, and all SSTables at the same level (except L_0) have disjoint key ranges. Also, the total size of L_i ($2 \leq i \leq n$) is often multiple times that of its lower level L_{i-1} (e.g., 10× in RocksDB [16]).

Writes. The `put(key, value)` operator writes a KV pair to an LSM-tree. It first appends the KV pair to a write-ahead log (WAL) for crash recovery, and then writes the KV pair in an in-memory mutable structure, called the *MemTable*. When the MemTable is full, it is converted to an immutable MemTable, which is *flushed* and becomes an SSTable in L_0 on disk; the flushed KV pairs are also removed from the WAL. The KV pairs in the SSTables are migrated to higher levels via *compaction* with the following steps: (i) selecting an SSTable in L_i ($0 \leq i \leq n-1$) and multiple SSTables in L_{i+1} with overlapping key ranges, (ii) merge-sorts all the KV pairs and discards any invalid (stale) KV pairs, and (iii) writes the live (non-stale) KV pairs as new SSTables to L_{i+1}. After compaction, all KV pairs in each level (except L_0) remain sorted, so that the query to a key in L_i ($i \geq 1$) can be done via binary search. Note that compaction leads to *write amplification*, as it rewrites the valid KV pairs to new SSTables.

Reads. The `get(key)` operator reads a KV pair of a given key and returns its value. It first searches the currently written MemTable and the immutable MemTables in memory, followed by the SSTables on disk from the low to high levels of the LSM-tree. Thus, it leads to *read amplification*, as it often issues multiple random reads to the SSTables in different levels. To accelerate reads, the index block in each SSTable maintains a Bloom filter [6] to quickly check if a key exists. Also, the frequently accessed index blocks and data blocks are cached in an in-memory *block cache*.

B. Read-Modify-Writes

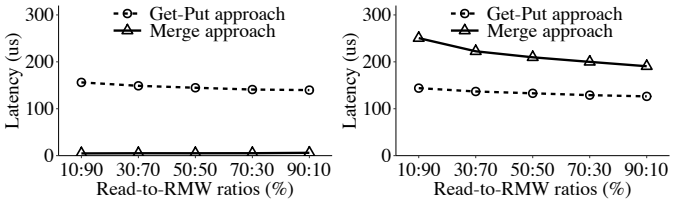
A read-modify-write (RMW) logically reads a KV pair from the LSM-tree to memory, modifies the value in memory, and writes the modified KV pair back to the LSM-tree. We discuss two RMW approaches, namely *Get-Put* and *Merge*.

Get-Put. An RMW can be realized with a pair of `get()` and `put()`, by reading the original value via `get()`, modifying the value, and writing the new value via `put()`. The Get-Put approach is used in YCSB benchmarking [12] and modern LSM-tree KV stores. For example, the update in YCSB core workloads A, B, and F implements an RMW by reading a database record with multiple fields via `get()`, modifying one or multiple fields, and writing back the updated database record via `put()`. Another example is the update of a statistical counter (e.g., in machine learning inference for monitoring user activities in social networks at Facebook [7]), which implements an RMW by reading the counter value via `get()`, incrementing or decrementing the counter value, and writing back the new counter value via `put()`. One drawback of the Get-Put approach is that its performance is bottlenecked by `get()`, which incurs read amplification (Section II-A).

Merge. To mitigate the `get()` overhead in the Get-Put approach, RocksDB introduces the `merge(key, delta)` operator [15], which performs RMWs by writing the *delta*, defined as the difference between the original and new values. Referring to the examples above, to modify a field in a database record, the delta represents the offset of the field within the record and the new content of the field; to update a statistical counter, the delta is the numerical difference from the original value to the new value.

Suppose that a client calls `merge()` to issue an RMW for a key. The `merge()` operator first combines the key and delta into a *key-delta (KD) pair*, and then writes the KD pair to the WAL and the MemTable. The MemTable is flushed to the disk as an SSTable when it is full, as in the original write workflow of the LSM-tree KV store; note that each of the MemTables and SSTables now contains a mix of KV pairs and KD pairs. Thus, `merge()` eliminates `get()` as in the Get-Put approach and improves the RMW performance.

RocksDB provides two implementable interfaces under the `merge()` operator: (i) `fullMerge(value, deltaList[])`, which specifies how to merge a list of deltas in `deltaList[]` with the input original value into a new value, and (ii) `partialMerge(delta1, delta2)`, which merges two deltas (i.e., `delta1` and `delta2`) into a single delta to reduce the number of deltas in SSTables. Both interfaces are called during compaction in RocksDB: if a KD pair encounters a KV pair of the same key, the delta of the KD pair



(a) RMWs (b) Reads

Figure 2: Latencies for Get-Put and Merge.

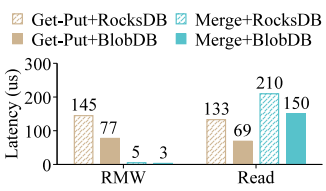


Figure 3: Latencies with and without KV separation.

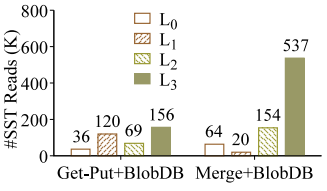


Figure 4: Number of reads at different SSTable levels.

may be merged with the KV pair via `fullMerge()`; if a KD pair encounters another KD pair of the same key, the two KD pairs can be merged into one via `partialMerge()`.

The `get()` operator under Merge is different from the original read path, as it needs to search and combine all deltas of a given key to reconstruct the returned value. Specifically, it searches all MemTables, and the SSTables from low to high levels, and stores all deltas of the given key in memory. It stops the search when it finds the latest KV pair of the given key. It then applies all deltas in memory to reconstruct the latest value via `fullMerge()` and returns the new value. Thus, although `merge()` makes RMWs more efficient, it incurs extra overhead to reads, especially when a KV pair is frequently updated with RMWs and the reads to the KV pair need to aggregate a large chain of deltas.

C. Motivating Experiments

Comparisons of Get-Put and Merge. We compare the read and RMW performance of both Get-Put and Merge to understand their performance trade-offs. We first load 100 M 1-KiB KV pairs to RocksDB in our testbed (Section IV-A), where each KV pair has a 24-byte key and a 1,000-byte value. Each value contains 10 100-byte fields (the default setting of YCSB [12]). We then issue workloads of 50 M operations mixed with reads and RMWs following a Zipf distribution with a Zipfian constant of 0.99, with direct I/O enabled. An RMW randomly selects and modifies one of the 100-byte value fields in a KV pair in both Get-Put and Merge.

Figure 2 shows the average read and RMW latencies of both Get-Put and Merge, in which we vary the read-to-RMW ratio from 10:90 to 90:10. We observe the performance trade-off in Merge and Get-Put. For Merge, it significantly reduces the RMW latencies of Get-Put by 95.7-96.9% (from 139.8-156.1 μs to 4.8-6.0 μs) (Figure 2(a)), as Get-Put issues many pairs of `get()` and `put()` when there exist several RMWs for a key, while Merge directly stores KD pairs in the LSM-tree and avoids multiple calls of `get()`. However, Merge also increases the read latencies of Get-Put by 50.8-74.1% (Figure 2(b)), as Merge needs to search for the value and all deltas of a key, while Get-Put directly reads the up-to-date KV pair. In particular, its read latency is higher for smaller read-to-RMW ratios (e.g., 10:90), as it injects more deltas into the LSM-tree and increases the read overhead. For Get-Put, its RMW and read latencies are almost identical, as they are both dominated by `get()`, while `put()` in RMWs incurs negligible overhead.

Can KV separation mitigate the RMW overhead? *KV separation*, first proposed by WiscKey [20], is known for reducing write and read amplifications by separating the storage of keys and values. Its idea is that values are not needed for indexing. Thus, it stores keys and metadata in the LSM-tree for indexing, while storing keys and values in an append-only circular log called the *vLog* (the keys in the vLog are for efficient value lookups [20]). It significantly reduces the LSM-tree size, so it mitigates the compaction and lookup costs, and hence the write and read amplifications, respectively.

Intuitively, KV separation is expected to also mitigate the overhead of both Get-Put and Merge. We evaluate Get-Put and Merge on BlobDB [14], which realizes KV separation in RocksDB and stores keys and values in dedicated files (called *Blob files*). Note that for KD pairs in Merge, BlobDB still keeps them (in addition to keys and metadata) in the LSM-tree. We focus on the read-to-RMW ratio of 50:50, and evaluate the average read and RMW latencies of both Get-Put and Merge.

Figure 3 shows the average latencies of RMWs and reads with and without KV separation. KV separation (i.e., BlobDB) reduces both RMW and read latencies in Get-Put and Merge compared with no KV separation (i.e., RocksDB). For Get-Put, KV separation (i.e., Get-Put+BlobDB) mitigates the RMW and read latencies by 46.5% and 48.0%, respectively, compared with no KV separation (i.e., Get-Put+RocksDB), but the RMW latency remains high. For Merge, the RMW latencies of both KV separation (i.e., Merge+BlobDB) and no KV separation (i.e., Merge+RocksDB) are low. However, the read latency of Merge remains high; for example, the read latency of Merge+BlobDB is 116.6% higher than that of Get-Put+BlobDB.

To further study the read penalty of Merge, we analyze the number of random reads issued to different LSM-tree levels for both Get-Put and Merge under KV separation. Figure 4 shows the results. We make two observations. First, almost most of the KV pairs in Get-Put appear in the low levels (e.g., totally 156.2 K reads in L_0 and L_1), while the KD pairs in Merge may appear across different levels, especially high levels (e.g., 153.7 K and 536.7 K reads in L_2 and L_3, respectively). Second, Merge has much more reads than Get-Put (773.7 K versus 381.1 K reads), since the LSM-tree also stores a large number of KD pairs. We also find that at the end of the workload, 23.2% of the total SSTable size is occupied by KD pairs in Merge+BlobDB (not shown in the figure).

III. KDSep Design

We propose *key-delta (KD) separation*, which separates the storage of KD pairs from the LSM-tree, so as to further reduce

the size of the LSM-tree and hence the read amplification for retrieving KD pairs. KD separation follows the similar observations in KV separation [20] that keys are only needed in the LSM-tree for indexing, while values and deltas can be kept outside of the LSM-tree. It can be viewed as an LSM-tree optimization technique complementary to KV separation.

To this end, we design KDSep, a middleware layer aiming for enhancing LSM-tree KV stores via KD separation to achieve high-performance RMWs and reads under RMW-intensive workloads, while maintaining high read/write performance for general workloads (even without RMWs). It builds on the Merge approach for RMWs, as Merge outperforms Get-Put in RMWs, while reducing the read overhead in Merge via KD separation. KDSep is designed to support *general* LSM-tree KV stores with and without KV separation (e.g., BlobDB and RocksDB, respectively).

A. Design Challenges

Despite the relevance with KV separation, KD separation differs from KV separation in its design and has the following unique challenges.

Challenge 1: Reducing reads to LSM-tree metadata. Traditional KV separation keeps metadata in the LSM-tree to store the location of the KV pair in the vLog [20]. However, storing metadata in the LSM-tree for KD separation can incur large read overhead, as a key can have multiple deltas that require separate metadata records for the key. This leads to multiple random reads in the LSM-tree and causes significant performance degradations in reads. It is important to consider removing the metadata records for KD pairs from the LSM-tree.

Challenge 2: Reducing reads to KD pairs. KD separation writes deltas to some separate storage space outside of the LSM-tree, and creates new challenges in the design of delta placement. Each read to a key now needs to retrieve all its KD pairs from the separate storage space, as opposed to reading a single value from the vLog in KV separation [20]. It is critical to group the deltas together to mitigate random reads.

Challenge 3: Garbage collection of KD pairs. With KD separation, existing KD pairs become *invalid* when a new KV pair for the same key is written via `put()`, as the deltas are derived from the old KV pair. Garbage collection is needed for reclaiming the free space of invalid KD pairs. However, it is critical to effectively locate invalid KD pairs using lightweight data records, since keeping track of the locations of KD pairs in the LSM-tree, as in KV separation [20], can incur high lookup overhead [8].

Challenge 4: Crash recovery. Crash recovery in KV separation should maintain the consistent states for both the LSM-tree and the vLog. KD separation adds one more separate storage component for deltas and requires special attention to crash recovery for all storage components.

B. Design Overview

KDSep realizes KD separation to store KD pairs in the *delta store*, which resides outside of the LSM-tree. It builds on several techniques to resolve the challenges in Section III-A.

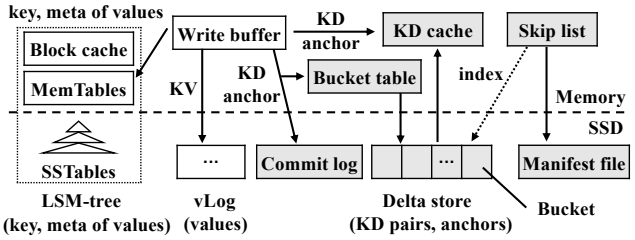


Figure 5: Architecture of KDSep. We show how KDSep extends WiscKey's vLog implementation [20] with new components (shaded) for KD separation.

- **Bucket-based delta placement (Section III-C).** The delta store partitions its storage space into append-only partitions called *buckets*. KDSep maps the KD pairs of a key into the same bucket, so that it does not need to keep the metadata of KD pairs in the LSM-tree, while it can still readily locate all KD pairs of a key (Challenge 1 addressed). It can also readily retrieve all KD pairs of a key as they are grouped together in the same bucket (Challenge 2 addressed).
- **Delta-based garbage collection (Section III-D).** For efficient garbage collection, KDSep adds a special type of data called *anchor* for each `put()`, so as to readily identify and clean the invalid KD pairs issued before the anchor. It also dynamically splits and merges buckets to keep the bucket sizes and the number of buckets manageable (Challenge 3 addressed).
- **Crash recovery (Section III-E).** KDSep maintains crash consistency under KD separation, and further supports the recovery of in-memory data structures under KD separation (Challenge 4 addressed).

Architecture. Figure 5 shows the architecture of KDSep. To make our discussion complete, we assume that KDSep is integrated into WiscKey's vLog implementation, which implements not only KV separation but also fast recovery (Section III-E); recall that KDSep is also applicable for KV stores without KV separation. KDSep keeps KD pairs in the delta store. It also keeps a *commit log* and a *manifest file* in persistent storage for crash recovery. We assume that a solid-state drive (SSD) is used for persistent storage for reasonably high performance as in WiscKey [20].

Like WiscKey, KDSep keeps an in-memory write buffer for fast writes. The write buffer caches both KV pairs and KD pairs and flushes them in batch to the vLog and the delta store, respectively. Also, KDSep introduces three new in-memory data structures: (i) a skip list for indexing buckets with disjoint key ranges in the delta store, (ii) a key-delta (KD) cache for fast access to frequently accessed KD pairs, and (iii) a bucket table for efficient bucket-based data management.

Interfaces. We focus on three interfaces: (i) `put()` for writing a new KV pair (i.e., blind updates), (ii) `merge()` for writing a new KD pair (i.e., Merge-based RMWs), and (iii) `get()` for retrieving the latest version of a KV pair. We describe their workflows under KDSep as follows.

(i) Write workflow of `put()` *and* `merge()`*:* KDSep keeps the newly written KV pairs from `put()` and KD pairs from `merge()` in the in-memory write buffer. If the total size

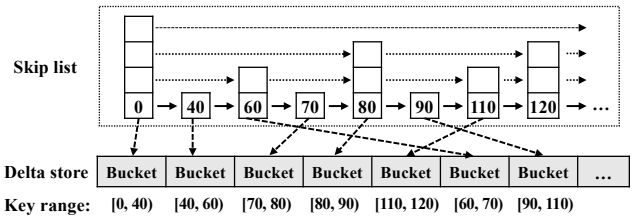


Figure 6: Indexing buckets with a skip list in KDSep.

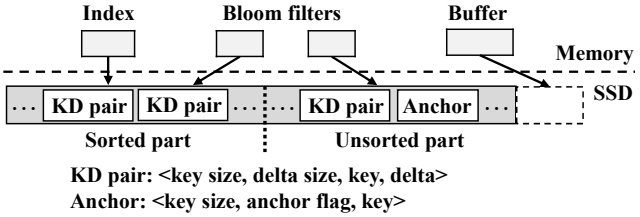


Figure 7: Bucket data organization.

of KV pairs and KD pairs in the write buffer exceeds the buffer size limit, KDSep flushes the write buffer as follows. First, it groups all KV pairs and KD pairs in the write buffer by their keys. If a key has a newly written value, KDSep performs `fullMerge()` on the value and all its deltas (if any) appearing after the value to generate a new KV pair for flushing; otherwise, if a key does not have a newly written value (i.e., only having KD pairs), KDSep performs `partialMerge()` on the deltas to prepare a new KD pair for flushing. Finally, for crash recovery, KDSep updates the vLog, commit log, LSM-tree, and delta store in order (Section III-E). For KV pairs, KDSep flushes them to the vLog, writes the metadata to the LSM-tree, and writes anchors to the buckets in the delta store; for KD pairs, KDSep appends them to the commit log (Section III-E), updates the bucket table, and writes them to the delta store (see Section III-C for details). Note that it does not need to write metadata to the LSM-tree.

(ii) Read workflow of `get()`*:* KDSep reads the latest value by searching in parallel for the value and deltas from the vLog and delta store, respectively. Specifically, it queries the LSM-tree to locate the KV pair in the vLog; meanwhile, it checks the in-memory data structure and retrieves the KD pairs (if any). Finally, it performs `fullMerge()` on the value and deltas, and returns the merged value.

C. Bucket-based Delta Placement

KDSep partitions the delta store into buckets and deterministically maps KD pairs to buckets. It allocates each bucket with fixed-size contiguous storage space (e.g., 256 KiB by default) that starts from a specific offset. Each bucket stores KD pairs with key ranges that are disjoint from any other buckets. KDSep maintains a skip list to organize buckets in an ordered manner, such that all KD pairs *across* buckets are ordered, but may not be ordered *within* a bucket.

Skip list. KDSep maintains an in-memory skip list to index buckets, as shown in Figure 6, so as to support efficient queries and updates over an ordered set of buckets.

The skip list maintains multiple layers of linked lists of nodes, in which each node stores the smallest key of a bucket and the reference to the bucket. The bottom layer indexes all buckets, while the higher layers maintain skipping pointers for the fast search of nodes in the bottom layer. The number of nodes in a layer is roughly half of that of its next lower layer. Let N be the maximum number of buckets in the delta store. Then the number of layers is $O(\log N)$, while the query and update costs are also $O(\log N)$ [24].

Initialization. Before writing any KD pairs, the delta store is empty. KDSep initializes the delta store and the skip list when it writes KD pairs from the write buffer for the first time. Specifically, KDSep sorts the KD pairs and divides them into multiple buckets of small sizes (e.g., 5% of the bucket capacity) with disjoint key ranges, so that it reserves most of the bucket capacity for further appends of KD pairs. It then builds the initial skip list based on the smallest key of each bucket. Note that as KDSep receives the writes of more KD pairs, the buckets will grow in size and are split during garbage collection (Section III-D), so the initial sizes of buckets have negligible impact on the overall performance.

Bucket data organization. Each bucket stores two types of data: KD pairs and anchors, in which the anchors mark the deletion of KD pairs. While it is straightforward to store new data at the end of a bucket in an append-only manner for fast writes, the KD pairs and anchors of a key are scattered across the bucket, thereby degrading the read performance.

To guarantee both efficient reads and writes, KDSep arranges each bucket with two parts: a *sorted* part and an *unsorted* part, as shown in Figure 7. The sorted part stores only KD pairs sorted by keys, without anchors, for efficient reads, while the unsorted part stores both KD pairs and anchors in an append-only manner for efficient writes. KDSep relocates KD pairs from the unsorted part to the sorted part during garbage collection (Section III-D). It also keeps an in-memory *bucket table* to manage per-bucket data. The bucket table contains three components for each bucket: (i) a *per-bucket index*, (ii) two *per-bucket Bloom filters*, and (iii) a *per-bucket buffer*. We elaborate on each component as follows.

(i) Per-bucket index. The per-bucket index maintains the locations of sorted keys in the sorted part of the bucket, so as to quickly retrieve the KD pairs of a key. It is created when the bucket is under garbage collection (Section III-D). To keep the index size small, the per-bucket index only keeps a subset of keys and offsets, such that the offset distance between two adjacent indexed keys is slightly larger than the SSD page size 4 KiB. This allows a read to load at most two SSD pages. Specifically, suppose that KDSep reads the KD pairs of a key K from the sorted part. It first finds the key range in the per-bucket index where K resides. It then loads the SSD pages (at most two) from the sorted part, and searches for the KD pair of K. Let the bucket capacity be 256 KiB, the key size be 24 bytes, and the reference size be 4 bytes. The per-bucket index size is at most $256/4 \times (24+4)$ bytes $= 1.75$ KiB.

(ii) Per-bucket Bloom filters. To quickly determine if a KD pair exists in a bucket, KDSep keeps two small in-memory

per-bucket Bloom filters (of size 2 KiB each), one for the sorted part and one for the unsorted part. The Bloom filter for the sorted part is reset and created when the bucket is under garbage collection, while the Bloom filter for the unsorted part is reset when the bucket is under garbage collection and is updated upon receiving new KD pairs.

(iii) Per-bucket buffer. Since KD pairs are often small (e.g., 100 bytes [7], [12]), flushing KD pairs individually to an SSD page (of size 4 KiB) can lead to substantial small writes and high write amplification [21]. Thus, when KDSep flushes KD pairs from the write buffer to the delta store, it first adds them to the per-bucket buffer. It then flushes the per-bucket buffer to the SSD when the buffer becomes full. We configure the per-bucket buffer size as 4 KiB to match the SSD page size.

KD cache. KDSep uses an in-memory KD cache to cache the recently read/written KD pairs, as they are likely accessed again soon. For each KD pair being read, KDSep either adds it to the KD cache if its key is new, or combines it with any currently cached KD pair of the same key with `partialMerge()`. For each KD pair being written, if the key is already cached, KDSep combines it with the cached KD pair. For each anchor being written, KDSep replaces any existing KD pair with a new one with an empty delta. The KD cache uses least-recently-used eviction if it becomes full.

Bucket writes/reads. We describe how to write KD pairs and anchors to a bucket and read KD pairs from a bucket.

(i) Writes of KD pairs and anchors. KDSep writes KD pairs and anchors when it flushes the write buffer to the delta store (Section III-B). First, it uses the key of the KD pair or the anchor to query the skip list for the associated bucket. Second, it appends the KD pair or anchor to the per-bucket buffer, which is flushed to the unsorted part of the bucket when being full. Third, it inserts the key to the per-bucket Bloom filter for the unsorted part. Finally, it updates the KD cache.

(ii) Reads of KD pairs. KDSep reads KD pairs associated with a key from the bucket during `get()`. First, it queries the KD cache and returns the cached KD pair upon a cache hit. Second, it searches the skip list by the key for the bucket. Third, it checks if the key exists in the sorted and unsorted parts based on the per-bucket Bloom filters. Based on the query results, it searches via the per-bucket index for the KD pair from the sorted part, or searches for the KD pairs that appear after the latest anchor from the unsorted part. It also calls `partialMerge()` to combine the KD pairs being retrieved. Any KD pair being retrieved is also updated into the KD cache.

Note that the per-bucket Bloom filters may return false positives, in which the search to the sorted part or unsorted part does not find the corresponding KD pair. In this case, KDSep returns an empty delta.

D. Delta-based Garbage Collection

The goals of delta-based garbage collection are three-fold. First, it reclaims the free space for the delta store by removing invalid KD pairs and merging KD pairs of the same key. Second, it sorts the KD pairs and adds them to the sorted part (Section III-C). Finally, it controls the bucket sizes and the number of buckets by dynamically splitting and merging buckets based on workload patterns. Due to workload skewness, some buckets may receive more KD pairs than others and become full soon, even though the delta store still has available space. Thus, KDSep re-allocates buckets based on workload patterns.

We elaborate on the steps of garbage collection as follows.

Step 1: Triggering. KDSep triggers garbage collection when it is about to flush the write buffer to the delta store but some buckets are full and cannot store the flushed data. Such buckets are referred to as *victim buckets*, which will be selected for garbage collection.

Step 2: Preparing valid KD pairs. For each victim bucket, KDSep groups the KD pairs and anchors by keys. For each key, KDSep keeps only the KD pairs after the latest anchor (if any) and performs `partialMerge()` on such KD pairs, so as to reduce the number of valid KD pairs.

Step 3: Writing valid KD pairs. KDSep determines if each victim bucket should be split. We define a configurable parameter called the *split threshold* (in bytes), such that KDSep compares the total size of valid KD pairs prepared from Step 2 against the split threshold to decide if a victim bucket should be split. There are three cases:

- *Case 1 (Rewriting a bucket)*: If the total size of valid KD pairs is no larger than the split threshold, KDSep does not split the victim bucket. Instead, it rewrites the valid KD pairs in a sorted manner to the sorted part of a new bucket in the delta store.
- *Case 2 (Splitting a bucket)*: If the total size of valid KD pairs exceeds the split threshold and the current number of buckets is no larger than $N-2$, KDSep splits the victim bucket. We choose $N-2$, since a split temporarily adds two new buckets and we ensure that the split does not increase the number of buckets beyond N. KDSep partitions the valid KD pairs into two new buckets based on their key ranges, and writes the KD pairs in a sorted manner to the sorted parts of the respective buckets.
- *Case 3 (Merging with the vLog)*: If the total size of valid KD pairs exceeds the split threshold and the current number of buckets exceeds $N-2$, it implies that the delta store cannot allocate more buckets. KDSep merges all KD pairs in the victim bucket with the KV pairs in the vLog via `fullMerge()` and frees the victim bucket.

Step 4: Updating index structures. KDSep updates the bucket table and the skip list based on the new locations of the valid KD pairs. Figure 8 shows how KDSep updates the skip list if it splits or merges buckets. If KDSep splits a bucket, it inserts a new node into the skip list; if KDSep merges two buckets, it deletes the node that originally references the bucket with larger keys from the skip list. In both cases, KDSep updates the references of the remaining nodes to point to the newly written buckets.

Step 5: Merging buckets. KDSep merges buckets with only a few KD pairs, so that other frequently appended buckets can be split. Specifically, it checks if the number of buckets exceeds $N-2$ (i.e., no split can be done in the future). If so, KDSep examines all pairs of adjacent buckets along the skip list and

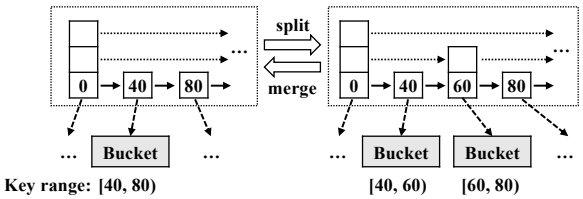


Figure 8: Bucket splits and merges.

selects the pair of buckets whose total size of valid KD pairs is the smallest. If the selected pair of buckets has a total size no larger than the bucket capacity and it is not just generated by the split in Case 2 of Step 3, then KDSep rewrites all valid pairs of the selected pair of buckets in a sorted manner to the sorted part of a new bucket, and updates the index structures as in Step 4.

E. Crash Recovery

KDSep supports crash recovery for the LSM-tree, vLog, and delta store during writes and garbage collection.

Crash recovery for writes. Recall that in writes, KDSep flushes the write buffer and updates the vLog, LSM-tree, and delta store in order (Section III-B). We follow WiscKey [20] to remove the WAL to avoid redundant writes, as the vLog can also serve the purposes of WAL. Like WiscKey, KDSep periodically records the *head pointer* of the vLog (i.e., the offset where the vLog issues new writes) to the LSM-tree. In crash recovery, KDSep scans from the offset specified by the head pointer to the end of the vLog, so as to provide crash consistency between the vLog and the LSM-tree.

To provide crash consistency between the vLog and the delta store, KDSep attaches a global monotonically increasing *sequence number* into each KD pair and anchor during `put()` or `merge()` to identify the occurrence sequence. It also introduces a *commit log* to append the KD pairs and anchors (with sequence numbers). Specifically, KDSep first flushes the write buffer by appending the KV pairs to the vLog and appending the KD pairs and anchors to the commit log (both appends can be done in parallel). Next, it writes a commit message to the commit log. Finally, it updates the LSM-tree (including the metadata and recorded head pointer), bucket table, and delta store. We consider the two crash scenarios to show how KDSep performs crash recovery.

- *Case 1: A crash happens before the commit message is written.* In this case, the data in the write buffer is not fully committed. KDSep rolls back the commit log to discard any non-committed data. It also rolls back the vLog by setting the latest head pointer as the recorded head pointer in the LSM-tree (which refers to the last committed writes). Any non-committed data in the vLog can later be overwritten by the newly committed KV pairs from the latest head pointer.
- *Case 2: A crash happens after the commit message is written.* Like WiscKey, KDSep reads the head pointer from the LSM-tree, scans the vLog from the recorded head pointer to the end, and rewrites the scanned keys and their metadata to the LSM-tree.

Table I: Summary of results of asymptotic analysis.

	Without KDSep	**With KDSep**
KV+KD reads	$O(n+m)$	$O(1)$
KV writes	$O(Fn)$	$O(Fn)$
KD writes	$O(Fn)$	$O(\frac{1}{1-p})$ or $O(Fn)$

In both cases, KDSep recovers the bucket table through the commit log. It scans the commit log (after discarding any non-committed data) and recovers the per-bucket buffers from the commit log if they have larger sequence numbers than the data in buckets (i.e., such data is yet written to the buckets before the crash). Finally, it scans all buckets to recover the per-bucket index and Bloom filters.

Crash recovery for garbage collection. Garbage collection may change the bucket references in the skip list. To ensure that the buckets can be located after crashes, KDSep maintains a *manifest file* to record the keys and bucket offsets in the skip list (note that RocksDB also maintains a manifest file to track the version changes of files). The manifest file in KDSep persists a snapshot of the skip list and appends any record that describes the changes of the skip list. During garbage collection, KDSep first updates the index structures and modifies the buckets, and then appends the smallest keys and bucket offsets for the buckets that are to be modified into the manifest file. During crash recovery, KDSep replays the snapshot and appended records in the manifest file to rebuild the skip list.

F. Asymptotic Analysis

We provide an asymptotic analysis on the read and write performance of KDSep and show the effectiveness of KD separation. We show that KDSep improves the read performance and preserves the write performance. Prior studies [5], [11], [20] also provide asymptotic analysis on LSM-trees, but they do not consider KD separation.

Here, we assume that the LSM-tree caches all index blocks and Bloom filters in memory, which is realistic in modern KV store implementations [16]. We also assume that the vLog is disabled, yet introducing the vLog only makes the LSM-tree smaller and does not compromise our analysis. Table I summarizes our analytical results.

Read performance. Recall that $n+1$ is the number of LSM-tree levels (Section II-A), and let m be the maximum number of SSTables in L_0. Each read retrieves all KD pairs and the KV pair for a key. Without KD separation, to retrieve all KD pairs, the LSM-tree reads each SSTable in L_0 and examines all the LSM-tree levels, so the number of reads can be $O(n+m)$; to retrieve the KV pair, the LSM-tree issues an $O(1)$ read, assuming a sufficiently low false positive rate of the Bloom filter [11]. With KD separation, the total read complexity is $O(1)$ only, since KDSep reads a single bucket to find all its KD pairs, and issues an $O(1)$ read in the LSM-tree to retrieve the KV pair associated with the KD pairs. Note that the read size in the bucket may be larger (bounded by the bucket capacity of 256 KiB) than that in the LSM-tree (e.g., a 4 KiB data block). However, the read latency is not proportional to the read size for small reads. For example, a 256 KiB read on our tested drive

[2] has only about 4× latency compared with a 4 KiB read. Also, the read size can be further mitigated by the per-bucket index and Bloom filters.

Write performance. Let F be the ratio of sizes between two adjacent LSM-tree levels (e.g., 10 by default in RocksDB). Without KD separation, each written KV or KD pair can be repeatedly written during compaction for at most $O(Fn)$ times, since the data is written by F times on average in each LSM-tree level. With KD separation, the number of writes for KV pairs is also $O(Fn)$, while the number of writes for KD pairs depends on whether there is sufficient reserved space of the delta store for storing all written KD pairs:

- *Case 1 (Sufficient reserved space)*: Let p ($p < 1$) be the ratio of the split threshold with respect to the bucket capacity. If there is enough space, garbage collection does not cause splits, so the total size of valid KD pairs during garbage collection is upper-bounded by the split threshold. Thus, each KD pair is rewritten by at most p times in one garbage collection operation. In the long run, the average number of repeated writes for a KD pair is upper-bounded by $O(p+p^2+p^3+\cdots) = O(\frac{1}{1-p})$. For example, by setting the split threshold as 80% of the bucket capacity, the fraction is only $\frac{1}{1-p} = 5$, so KD separation does not cause significant degradations in write performance.
- *Case 2 (Insufficient reserved space)*: Each KD pair in the bucket is merged with its corresponding KV pair (see Step 3, Case 3 in Section III-D). It causes an $O(1)$ additional read for the KV pair. After merging, the write complexity of each new KV pair is $O(Fn)$, as in without KD separation.

G. Implementation Details

We implement a KDSep prototype in C++ on Linux with 14.4 K LoC to support KD separation. In particular, we implement the delta store as a large file in user space, in which the buckets start from the file offsets that are aligned with the multiples of bucket capacity. We let KDSep issue reads and writes via `pread` and `pwrite` system calls, respectively.

KDSep serves as a middleware layer that can be integrated into general LSM-tree KV stores (which may not support Merge) with minimal code changes. In this work, we integrate KD separation into RocksDB (v7.7.3) [16] and BlobDB (which is included in RocksDB's source code) [14]; RocksDB does not support KV separation, while BlobDB does. We also integrate KDSep with vLog-based KV store implementation in [8], which supports KV separation and uses RocksDB (v7.7.3) [16] for the LSM-tree.

Multi-threading. KDSep implements multi-threading for I/O parallelization. It manages threads using `asio::thread_pool` in the `boost` library. When flushing the write buffer, KDSep groups the KD pairs and anchors that belong to the same bucket and allocates one thread to write them to the bucket. In particular, KDSep leverages multi-threading to preserve the efficiency of reads by issuing reads to multiple storage components in parallel. It allocates two threads, one for reading KV pairs from the LSM-tree and the vLog, and another for reading KD pairs from the delta store. The two threads exchange data via a lock-free message queue using the `boost` library, and use polling for low-latency data synchronization. Note that our polling implementation only causes slightly higher CPU usage (Section IV-B).

KDSep also issues two threads for range queries, one for reading the keys and metadata from the LSM-tree to retrieve KV pairs from the vLog, and another for retrieving the KD pairs from delta store. It performs `fullMerge()` on the KV and KD pairs, and returns the merged KV pairs.

Memory budget. The memory usage mainly comes from the block cache (in RocksDB), KD cache, and bucket table. To control the overall memory usage and subject to a given memory budget, KDSep dynamically tunes the capacity of the block cache as the KD cache and bucket table sizes grow. Specifically, KDSep sets a memory budget and initially sets the capacity of the block cache equal to the memory budget. During runtime, if the KD cache and bucket table grow in size, KDSep reduces the capacity of the block cache using the `Cache::SetCapacity()` interface in RocksDB to maintain the same memory usage.

Trade-off discussion. Our KDSep implementation makes trade-offs in two aspects. First, it trades additional CPU cycles to support I/O parallelization with multi-threading for faster reads and writes. Nevertheless, such CPU overhead is small compared to I/Os (Section IV-B). Second, it trades extra memory for maintaining per-bucket buffers, yet the extra memory usage is negligible since the major memory usage is for caching (e.g., $2^{15} \times 4$ KiB$= 128$ MiB for per-bucket buffers, equivalent to $\frac{128}{4*1024} = 3.1\%$ of the total memory budget).

IV. Evaluation

A. Methodology

Testbed. We conduct experiments on a single machine running Ubuntu 22.04 LTS with Linux kernel 5.15. The machine is equipped with a 16-core Intel Xeon Silver 4215 CPU, 96 GiB DDR4 memory, and a 3.84 TiB Western Digital Ultrastar DC SN640 NVMe SSD [2].

Default settings. We consider three baseline KV stores, namely RocksDB, BlobDB, and vLogDB (i.e., the vLog-based KV store implementation in [8]). We also add KDSep as a middleware layer to RocksDB, BlobDB, and vLogDB, referred to as RocksDB+KDS, BlobDB+KDS, and vLogDB+KDS.

For all six KV stores, we configure the LSM-tree based on the official tuning guide for RocksDB [17]. In particular, we set the MemTable size as 64 MiB and the SSTable size as 16 MiB. For multi-threading, we allocate eight threads for flushing and compaction. For vLogDB and vLogDB+KDS, we over-provision the vLog with 30% additional space of the total data size for garbage collection of KV pairs [8].

By default, for the KV stores with KD separation, we set the maximum number of buckets as $N = 32{,}768$ and the bucket capacity as 256 KiB (i.e., the capacity of the delta store is 8 GiB). We set the split threshold as 204.8 KiB (i.e., 80% of the bucket capacity). We enable direct I/Os for reads and writes.

We configure all KV stores with the same memory budget as 4 GiB for fair comparisons. For RocksDB, BlobDB, and

vLogDB, the block cache size in the LSM-tree is 4 GiB. For their variants of KD separation, we control the total size of the block cache, KD cache, and bucket table under the same memory budget (Section III-G). We set the default write buffer size as 2 MiB. For KD separation, we set the maximum size of the KD cache as 0.5 GiB.

B. Performance under Different Workloads

Exp#1: Custom synthetic workloads. We use YCSB [12] to generate custom synthetic workloads with varying distributions of `get()` and `merge()` for different degrees of RMW-intensive workloads. We load 100 GiB of 1-KiB KV pairs, each with a key size 24 bytes and a value size 1,000 bytes, to each KV store. We then issue 200 M operations of `get()` or `merge()` to the KV pairs. We set the read-to-RMW ratio as 10:90, the Zipfian constant as 0.99, and the value field size as 100 bytes (i.e., each value has 10 fields). Each `merge()` modifies a randomly selected field as in Section II-C. We evaluate the throughput, average and 99th-percentile read and RMW latencies, I/O sizes, total storage space, and CPU usage.

Figure 9(a) shows the throughput. KDSep increases the throughput of RocksDB, BlobDB, and vLogDB by 80.5%, 67.0%, and 68.2%, respectively, due to the reduction of average read latencies (i.e., the main design goal of KDSep).

Figures 9(b) and 9(c) show the average read and RMW latencies, respectively. From Figure 9(b), KDSep reduces the read latencies of RocksDB, BlobDB, and vLogDB by 49.9%, 41.9%, and 46.3%, respectively, since it avoids reading a chain of deltas in the LSM-tree during reads. From Figure 9(c), KDSep reduces the RMW latencies of RocksDB, BlobDB, and vLogDB by 6.7%, 33.3%, and 7.4%, respectively. The reason for the high reduction in BlobDB is that when BlobDB performs LSM-tree compaction, it retrieves the values from the Blob files and merges them with any associated KD pairs in the SSTables selected for compaction. By removing KD pairs from the LSM-tree, KDSep reduces the compaction overhead of BlobDB and hence the RMW latency.

Figures 9(d) and 9(e) show the 99th-percentile read and RMW latencies, respectively. Compared with RocksDB, BlobDB, and vLogDB, KDSep reduces the 99th-percentile read latencies by 48.2%, 42.7%, and 36.5%, respectively (Figure 9(d)) and the 99th-percentile RMW latencies by 62.4%, 23.6%, and 49.9%, respectively (Figure 9(e)). In particular, KDSep shows a higher reduction of 99th-percentile RMW latencies for RocksDB and vLogDB than for BlobDB. The reason is that BlobDB has the smallest LSM-tree compared with vLogDB and RocksDB by cleaning KD pairs during LSM-tree compaction, while vLogDB keeps KD pairs in the LSM-tree and RocksDB maintains both KV and KD pairs in the LSM-tree. Thus, BlobDB has fewer compaction operations and hence fewer latency spikes (i.e., its 99th-percentile RMW latency is smaller), even though its average RMW latency is higher. By offloading KD pairs to the delta store, KDSep eliminates the differences among the KV stores and keeps the 99th-percentile RMW latency low in all cases (7.49-7.93 μs).

Figure 9(f) shows the amount of I/Os incurred during the operations. KDSep reduces the I/O sizes of RocksDB, BlobDB, and vLogDB by 27.4%, 41.1%, and 36.9%, respectively. The reason is that KDSep reduces random reads in the LSM-tree with the reduced LSM-tree size; in particular, it also reduces the compaction overhead of BlobDB.

Figure 9(g) shows the storage size of each KV store at the end of the workloads. KDSep slightly increases the storage sizes of RocksDB and vLogDB by 6.4% and 6.1%, respectively, due to additional storage in the delta store (e.g., anchors and invalid KD pairs). On the other hand, KDSep reduces the storage size of BlobDB by 12.5%. The reason is that BlobDB merges KD pairs in SSTables with the KV pairs in Blob files during compaction, and creates additional Blob files to accommodate the new merged KV pairs. Note that the original KV pairs (which now become invalid) are still stored in Blob files and wait for being reclaimed, so the total size of Blob files remains large. On the other hand, KDSep eliminates such operations and reduces the total size of Blob files in BlobDB.

Figure 9(h) shows the average CPU usage during runtime, in which we measure the CPU usage using the `top -b -n 1 -p processID` command every second and obtain the average. KDSep incurs 2.0×, 1.7×, and 2.0× CPU usage for RocksDB, BlobDB, and vLogDB, respectively, since KDSep uses one extra thread to perform parallel reads in the delta store while accessing the LSM-tree and vLog (Section III-B). Nevertheless, the total CPU usage of KDSep is no more than 15% and has significantly less overhead than I/Os (the major overhead of KV stores).

Exp#2: UP2X workloads. We consider the production workloads of AI/ML services of UP2X, a distributed KV store based on RocksDB, at Facebook [7]. Since we do not have access to the real workloads, we synthesize the workloads based on the statistics reported in [7], in which the workloads contain 92.53% `merge()`, 7.46% `get()`, and less than 0.01% `put()`. The key and value sizes follow normal distributions, with averages of 10.45 bytes and 46.8 bytes, and standard deviations of 1.4 bytes and 11.6 bytes, respectively. We first load 1 billion KV pairs into each KV store. We then issue 200 M operations based on the distributions of `merge()`, `get()`, and `put()`.

Figure 10 shows the results. Figure 10(a) shows that KDSep increases the throughput of RocksDB, BlobDB, and vLogDB by 51.3%, 30.3%, and 34.2%, respectively. Figure 10(b) shows that the average read latency decreases by 28.5-39.0% due to the reduced read overhead to deltas. Figure 10(c) shows that the average RMW latency reduces by 0.4-6.4%, showing that KDSep does not degrade the RMW performance. Note that RocksDB outperforms BlobDB and vLogDB even though the latter two apply KV separation. The reason is that the UP2X workloads are dominated by small values, in which the extra I/Os to the LSM-tree and the value storage under KV separation are more severe for small values [19].

Exp#3: YCSB workloads. We consider the six YCSB core workloads [12] to show that KDSep preserves high performance in general workloads: A (50% reads and 50% updates), B (95% reads and 5% updates), C (100% reads), D (95% latest reads

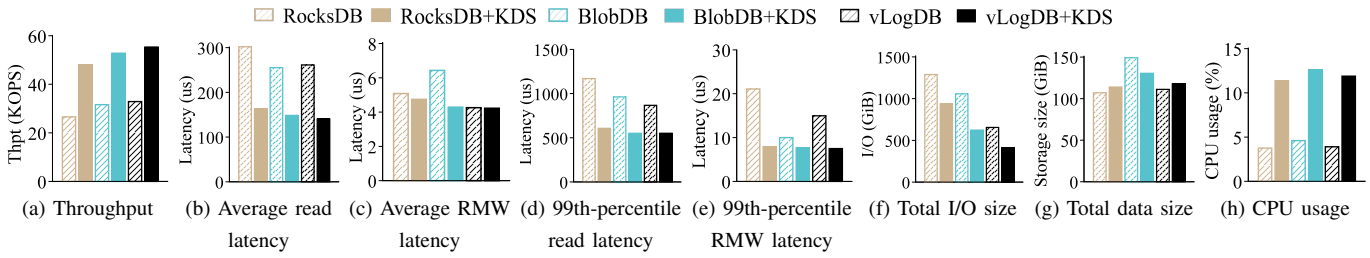


Figure 9: Exp#1: Synthetic RMW-intensive workloads.

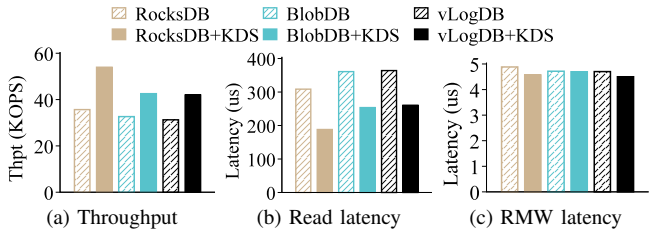


Figure 10: Exp#2: UP2X workloads.

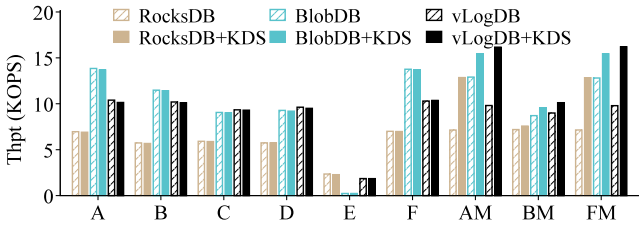


Figure 11: Exp#3: YCSB workloads.

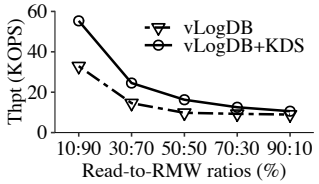


Figure 12: Exp#4: Impact of read-to-RMW ratios.

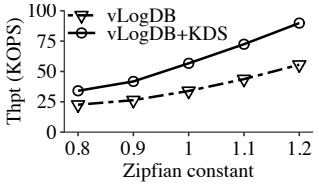


Figure 13: Exp#5: Impact of skewness.

and 5% writes), E (95% range queries and 5% writes), and F (50% reads and 50% RMWs). Note that the updates and RMWs in the original workloads A, B, and F are based on Get-Put. Thus, we also implement three additional YCSB workloads for A, B, and F, whose updates and RMWs are based on Merge (referred to as workloads AM, BM, and FM, respectively). All workloads, except workload D, follow a Zipf distribution with a Zipfian constant 0.99 (default in YCSB), while workload D reads the latest written keys. We issue 200 M operations to each KV store for each of workloads except workload E, and 1 M operations for workload E. For each core workload, we start with a clean KV store, load the whole KV store with 100 GiB of 1-KiB KV pairs (with a key size 24 bytes and a value size 1,000 bytes, in which each value has 10 100-byte fields), and run the workload.

Figure 11 shows the throughput results. For workloads A-F (without Merge), KDSep incurs almost no extra overhead for RocksDB, BlobDB, and vLogDB, even though KD pairs are not included. The throughput differences with and without KDSep for the KV stores are within 2.4%. Also, when KDSep is used, BlobDB and vLogDB increase the throughput of workloads A-D and F by 53.0-101.2% and 47.0-65.7% compared with RocksDB, respectively, meaning that KDSep still preserves the benefits of KV separation; for workload E (i.e., range-query-intensive), BlobDB and vLogDB have 89.8% and 18.9% lower throughput compared with RocksDB, respectively. Note that the overhead of KV separation in range queries is consistent with the findings of prior studies [8], [19], [20]. Note that BlobDB is much slower than vLogDB, as it does not support multi-threaded range queries. The range query overhead of KV separation can be mitigated via the sorting of KV pairs [19], [28], which is orthogonal to our work.

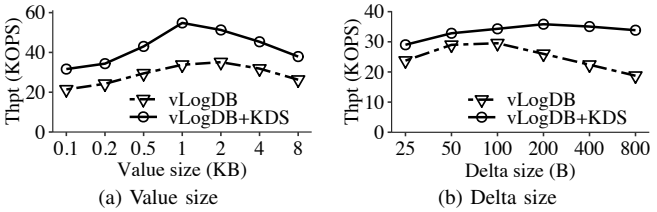
We further examine workloads AM, BM, and FM, in which we modify updates and RMWs to use Merge. For both workloads AM and FM with 50% `merge()`, KDSep increases the throughput of RocksDB, BlobDB, and vLogDB by 79.6-80.1%, 19.5-20.5%, and 64.7-65.4%, respectively. Note that the increase is the lowest in BlobDB, as BlobDB can leverage compaction to merge values in Blob files with the deltas and reduce the number of KD pairs in the LSM-tree. Compared with Exp#1, in which BlobDB has large compaction overhead, BlobDB in workloads AM and FM has fewer KD pairs (only 50% `merge()` instead of 90%) and hence lower compaction overhead. For workload BM with only 5% `merge()`, KDSep still increases the throughput of RocksDB, BlobDB, and vLogDB by 5.3%, 9.7%, and 12.4%, respectively.

C. Impact of Workload Settings

We consider different workload settings based on the custom synthetic workloads (see Exp#1). By default, we configure the read-to-RMW ratio as 10:90, the Zipfian constant as 0.99, the value size as 1,000 bytes, and the value field size of 100 bytes (i.e., each value has 10 fields). We focus on vLogDB with and without KDSep.

Exp#4: Impact of read-to-RMW ratios. We first vary the read-to-RMW ratio from 10:90 to 90:10. We issue 200 M operations for each setting.

Figure 12 shows the throughput. Overall, KDSep has higher performance gains at low read-to-RMW ratios (i.e., more RMW-intensive); for example, when the read-to-RMW ratios are 10:90, 30:70, and 50:50, KDSep increases the throughput of vLogDB by 68.2%, 68.4% and 66.0%, respectively, while the throughput gain drops to 35.1% and 18.6% when the read-to-RMW ratios become 70:30 and 90:10, respectively. The reason is that in RMW-intensive workloads, the read overhead for combining deltas increases. Nevertheless, KDSep maintains throughput gains even with limited RMWs.



(a) Value size (b) Delta size

Figure 14: Exp#6: Impact of value and delta sizes.

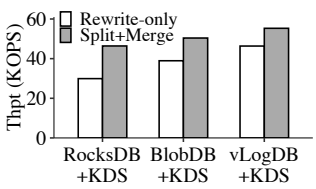


Figure 15: Exp#7: Bucket splits and merges.

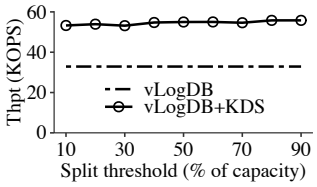


Figure 16: Exp#8: Impact of the split threshold.

Exp#5: Impact of skewness. We examine the impact of skewness by varying the Zipfian constant from 0.8 to 1.2 (a larger value means higher skewness), given that practical workloads have a Zipfian constant no larger than 1.2 [30].

Figure 13 shows the throughput. Overall, KDSep increases the throughput of vLogDB by 50.8-67.2%. The gain is the lowest for the lowest skewness (i.e., the Zipfian constant is 0.8). The reason is that with low skewness, the access covers more keys and hence there exist more KD pairs. The delta store merges the KD pairs with KV pairs in the vLog in garbage collection and causes higher read and write overhead.

Exp#6: Impact of value and delta sizes. We consider how the value and delta sizes affect the performance. We fix the read-to-RMW ratio as 10:90. We first vary the value size from 1,00 bytes to 8,000 bytes and fix the field size as 100 bytes (i.e., each value has 1 to 80 fields). Figure 14(a) shows that KDSep increases the throughput of vLogDB by 20.5-68.2%. The throughput is the highest for a medium value size, since KV separation performs worse for smaller values, while `get()` have larger read sizes on the vLog for larger values.

We next vary the field size from 25 bytes to 800 bytes and fix the value size as 8,000 bytes. Each delta is a modification to a field. To accommodate the different delta sizes, we also vary the number of buckets and the (in-memory) bucket table size proportionally. Figure 14(b) shows that KDSep's throughput gain for vLogDB increases from 13.3% to 80.6% as the delta size increases. This shows that KDSep has higher performance gains for larger delta sizes, as larger deltas occupy more space in the LSM-tree without KDSep.

D. Performance of Delta-based Garbage Collection

We evaluate delta-based garbage collection by considering the performance impact of enabling bucket splits and merges, using the default custom synthetic workloads in Section IV-C.

Exp#7: Bucket splits and merges. We consider two schemes for bucket management under KD separation. The first scheme is *Rewrite-only*, in which the delta store simply rewrites a victim bucket to a new bucket without bucket splits and merges during garbage collection; during the rewrites, any invalid KD pairs are removed and the valid KD pairs for the same key are merged. Since Rewrite-only disables bucket splits, we let the delta store distribute KD pairs to as many buckets as possible (up to N buckets) during initialization so as to utilize all available delta store space. The second scheme is *Split+Merge*, in which the delta store performs bucket splits and merges during garbage collection (i.e., our design).

Figure 15 shows the results. Split+Merge shows 55.1%, 29.4%, and 19.3% higher throughput than Rewrite-only for RocksDB+KDS, BlobDB+KDS, and vLogDB+KDS, respectively. Rewrite-only statically maps KD pairs to buckets and causes some buckets to receive more KD pairs than others; such buckets are frequently selected for garbage collection and hence lead to high rewrite overhead. In contrast, KDSep re-allocates KD pairs to buckets via bucket splits and merges based on the workload patterns, so as to limit the garbage collection overhead.

Exp#8: Impact of the split threshold. We vary the split threshold from 10% to 90% of the bucket capacity (recall that our default is 80% of the bucket capacity). Figure 16 shows the results (note that vLogDB has the same throughput independent of the split threshold). We show that the throughput is insensitive to the split threshold. Even though a smaller split threshold makes KDSep split buckets more aggressively, the number of buckets will finally converge to N (i.e., the maximum number of buckets) and the performance becomes stable.

E. Performance of Crash Recovery

We evaluate crash recovery of KDSep based on the custom synthetic workloads.

Exp#9: Overhead of crash recovery. We evaluate the performance of KDSep with and without the support of crash recovery; for the latter case, we disable the commit log and manifest file. We consider four synthetic workloads: W1 (90% `merge()` and 10% `get()`), W2 (100% `merge()`), W3 (50% `put()` and 50% `merge()`), and W4 (100% `put()`). For each workload, we start with an empty KV store, pre-load 100 GiB of 1-KiB KV pairs, and issue 200 M operations.

Figure 17 shows the normalized throughput with respect to no crash recovery support. In W1-W3, the throughput with crash recovery support only drops by 2.5-6.9%, while in W4, the throughput only drops by 0.6%. It shows that the crash recovery support incurs limited overhead for workloads with and without `merge()`.

Exp#10: Recovery performance. We evaluate the recovery time of KDSep upon crashes. We run the four synthetic workloads described in Exp#9. For each workload, we pre-load 100 GiB of 1-KiB KV pairs and issue 200 M operations. We randomly crash the KV store using the `kill -9 processID` command in the midst of workload execution and recover the KV store. We conduct ten rounds for each workload, and report the average recovery time with error bars showing standard deviations.

Figure 18 shows the results. In W1-W3, the absolute average recovery times of vLogDB are 1.07-1.82 s. Compared with vLogDB, the absolute average time of KDSep increases by 4.08-5.02 s. KDSep has slower recovery since it needs to scan

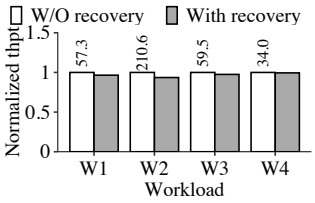


Figure 17: Exp#9: Overhead of crash recovery.

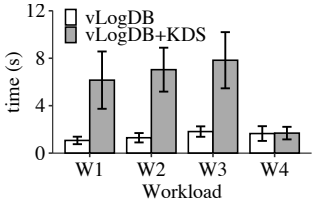


Figure 18: Exp#10: Recovery performance.

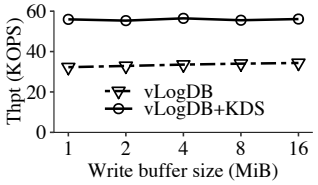


Figure 19: Exp#11: Impact of write buffer size.

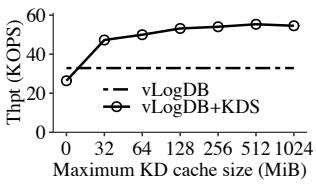


Figure 20: Exp#12: Impact of the maximum KD cache size.

all buckets to recover the bucket table; the average total sizes of all buckets to recover in W1, W2, and W3 are 5.84 GiB, 5.16 GiB, and 4.62 GiB, respectively (not shown in the figure). Note that we leverage multi-threading (16 threads) to fully utilize the CPU resources and SSD bandwidth to mitigate the overhead of bucket table recovery. KDSep has a larger standard deviation of recovery time than vLogDB since its recovery time depends on the size of the delta store during recovery, which varies in a large range. In W4, KDSep has a comparable recovery time (1.69 s) with vLogDB (1.65 s) because it does not introduce deltas in the delta store.

F. Performance of Caching

We study the impact of the write buffer and KD cache on the performance of KDSep. We use the default custom synthetic workloads in Section IV-A and focus on vLogDB and vLogDB+KDS.

Exp#11: Impact of the write buffer size. We evaluate the impact of the write buffer size. We vary the write buffer size from 1 MiB to 16 MiB. Figure 19 shows the results. We see that the throughput differences of KDSep are within 2.1% only, instead of seeing increasing throughput with a larger write buffer size. The reasons are two-fold. First, the performance under RMW-intensive workloads is mainly determined by reads, and increasing the write buffer size has limited impact on read performance gains. Second, when flushing the KD pairs from the write buffer, KDSep needs to search the skip list. A larger write buffer incurs more overhead on processing KD pairs and causes slightly longer stalls for reads. Thus, a small write buffer size is suitable for our design.

Exp#12: Impact of the maximum KD cache size. We evaluate the impact of the maximum KD cache size. We increase the maximum KD cache size from zero to 1 GiB, subject to the total memory budget of 4 GiB. Figure 20 shows the results. When the maximum KD cache size is zero, KDSep's throughput is 80.2% of vLogDB's. Nevertheless, a small maximum KD cache size (e.g., 32 MiB) sufficiently reduces reads to buckets, and KDSep has higher throughput than vLogDB by up to 68.2% when the maximum KD cache size is 512 MiB. Increasing the maximum KD cache size to 1 GiB shows slight performance drops, as the block cache size decreases and the cache misses for accessing the LSM-tree become more frequent.

V. Related Work

Some KV stores focus on optimizing RMWs [9], [11], [18]. FASTER [9] optimizes RMWs by performing latch-free in-place updates in mutable partitions and appending deltas as partial updates into append-only partitions. Kalavri *et al.* [18] improve FASTER by applying multiple backends with workload-aware configurations. Note that the indexing of FASTER is based on hash tables, while KDSep uses the LSM-tree. SplinterDB [11] optimizes KV operations on the B^ε-tree and supports delta-based RMWs, but the delta may still be separated to different storage components and lead to high read overhead. In contrast, KDSep groups KD pairs of the same key in the same bucket to mitigate read overhead.

Many studies enhance KV separation (proposed by WiscKey [20]) to address different scenarios in LSM-tree optimization. HashKV [8] deterministically maps values by keys into dedicated logs with hash functions, so as to reduce the garbage collection overhead in value storage; note that KDSep shares similar ideas of hash-based data placement, yet it applies the idea to delta placement. Bourbon [13] builds a lightweight index with machine learning for KV separation to reduce the query overhead of KV pairs. DiffKV [19] differentiates value management by putting small values in the LSM-tree, medium-size values in tree-based value storage, and large values in the vLog as in WiscKey. Kreon [23] introduces a spill mechanism to reduce CPU overhead of reorganizing key indices under KV separation. FenceKV [28] maps values to different storage areas based on their key ranges to improve update and range query performance. KDSep takes one step further and proposes KD separation for delta-based RMW-intensive workloads.

VI. Conclusion

RMW-intensive workloads are commonly found in production LSM-tree KV stores. We propose KD separation, inspired by the idea of the well-known KV separation technique, to store KD pairs outside of the LSM-tree. KD separation has specific challenges that are distinct from KV separation, and builds on several design elements: bucket-based delta placement, delta-based garbage collection, and crash recovery. We implement KDSep, a middleware layer that realizes KD separation and is designed to support general LSM-tree KV stores with and without KV separation. Extensive experiments show that KDSep achieves throughput gain and read latency reduction compared with various baselines without KD separation under different workloads.

Acknowledgements

This work was supported in part by Research Grants Council of Hong Kong (GRF 14214622 and AoE/P-404/18), Innovation and Technology Commission of Hong Kong (ITS/205/21), and Research Matching Grant Scheme. The corresponding author is Patrick P. C. Lee.

REFERENCES

[1] TPC-C. http://www.tpc.org/tpcc/.

[2] Western Digital Ultrastar DC SN640. https://www.westerndigital.com/en-gb/products/internal-drives/data-center-drives/ultrastar-dc-sn640-nvme-ssd.

[3] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A database benchmark based on the facebook social graph. In *Proc. of ACM SIGMOD*, pages 1185–1196, New York, New York, USA, June 2013.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, pages 53–64, London, England, UK, June 2012.

[5] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An introduction to bε-trees and write-optimization. *USENIX login; magazine*, 40(5):22–28, October 2015.

[6] Burton Howard Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 12(7):422–426, 1970.

[7] Zhichao Cao, Siying Dong, and Sagar Vemuri. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proc. of USENIX FAST*, pages 209–223, Santa Clara, CA, USA, February 2020.

[8] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. of USENIX ATC*, pages 1007–1019, Boston, MA, July 2018.

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proc. of ACM SIGMOD*, pages 275–290, Houston, TX, USA, May 2018.

[10] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *Proc. of USENIX FAST*, pages 17–32, February 2021.

[11] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tail, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *Proc. of USENIX ATC*, pages 49–63, July 2020.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, pages 143–154, Indianapolis, Indiana, USA, June 2010.

[13] Yifan Dai, Yien Xu, Aishwarya Ganesan, and Ramnatthan Alagappan. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *Proc. of USENIX OSDI*, pages 155–171, November 2020.

[14] Facebook. BlobDB in RocksDB. http://rocksdb.org/blog/2021/05/26/integrated-blob-db.html.

[15] Facebook. Merge operator in RocksDB. https://github.com/facebook/rocksdb/wiki/Merge-Operator.

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

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

[18] Vasiliki Kalavri and John Liagouris. In support of workload-aware streaming state management. In *Proc. of USENIX HotStorage*, July 2020.

[19] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced I/O performance. In *Proc. of USENIX ATC*, pages 673–687, July 2021.

[20] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, pages 133–148, Santa Clara, CA, February 2016.

[21] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random write considered harmful in solid state drives. In *Proc. of USENIX FAST*, pages 1–16, San Jose, CA, February 2012.

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[23] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. Kreon: An efficient memory-mapped key-value store for flash storage. *ACM Trans. on Storage*, 17(1):1–32, January 2021.

[24] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[25] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, pages 497–514, Shanghai, China, October 2017.

[26] Russell Sears and Raghu Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proc. of ACM SIGMOD*, pages 217–228, Scottsdale, Arizona, USA, May 2012.

[27] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. COBRA: Making transactional key-value stores verifiably serializable. In *Proc. of USENIX OSDI*, pages 63–80, August 2020.

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

[29] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, pages 191–208, August 2020.

[30] Yue Yang and Jianwen Zhu. Write skew and Zipf distribution: Evidence and implications. *ACM Trans. on Storage*, 12(4):1–19, 2016.