The Design and Implementation of UniKV for Mixed Key-Value Storage Workloads

Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu.

Abstract—Persistent key-value (KV) stores are mainly designed based on the Log-Structured Merge-tree (LSM-tree), yet they suffer from large read and write amplifications, especially when KV stores grow in size. Existing design optimizations for LSM-tree-based KV stores often make certain trade-offs and fail to simultaneously improve both the read and write performance on large KV stores without sacrificing scan performance. We design UniKV, which unifies the key design ideas of hash indexing and the LSM-tree in a single system. Specifically, UniKV leverages data locality to differentiate the indexing management of KV pairs. It also develops multiple techniques (e.g., merge with partial KV separation, dynamic range partitioning) to tackle the issues caused by unifying the indexing techniques, so as to simultaneously improve the performance in reads and writes. Furthermore, it proposes a parallel optimization scheme to manage partitions in parallel and develops multiple strategies to optimize the scan performance. Experiments show that UniKV significantly outperforms several state-of-the-art KV stores (e.g., LevelDB, RocksDB, PebblesDB and Titan) in overall throughput under read-write mixed workloads.

Index Terms—Key-value storage, LSM-tree, hash index, dynamic range partitioning, parallel optimization

1 Introduction

Persistent key-value (KV) storage organizes data in the form of key-value pairs and forms a critical storage paradigm in modern data-intensive applications, such as web search [10], [18], e-commerce [15], social networking [17], [39], data deduplication [13], [14], and photo stores [7]. Real-world KV workloads become more diversified and are geared toward a *mixed* nature. For example, four out of five Facebook's Memcached workloads are read-write mixed [3]; the read-write ratio of low-latency workloads at Yahoo! has shifted from 8:2 to 1:1 in recent years [38]; Baidu's cloud workload also shows a read-write ratio of 2.78:1 [24]. Such mixed workloads challenge the indexing structure design of KV stores to achieve high read and write performance, while supporting scalable storage.

The Log-Structured Merge-tree (LSM-tree) [33] is a popular indexing design in modern persistent KV stores, including local KV stores (e.g., LevelDB [18] and RocksDB [17]) and large-scale distributed KV stores (e.g., BigTable [10], HBase [2], and Cassandra [25]). The LSM-tree supports three main features that are attractive for KV-store designs: (i) *efficient writes*, as newly written KV pairs are batched and written sequentially to persistent storage; (ii) *efficient range queries (scans)*, as KV pairs are organized in multiple levels in a tree-like structure and sorted by keys in each level; and (iii) *scalability*, as KV pairs are mainly stored in persistent storage and can also be easily distributed across nodes.

The LSM-tree, unfortunately, incurs high *compaction* and *multi-level access* costs. As KV pairs are continuously written, the LSM-tree flushes them to persistent storage from lower levels to higher levels. To keep the KV pairs at each level sorted, the LSM-tree performs regular compactions for different levels by reading, sorting, and writing back the sorted KV pairs. Such regular compactions incur I/O amplification (which can reach a factor of 50× [32]), and hence severely hurt the I/O performance and the endurance of storage systems backed by solid-state drives (SSDs) [1]. Furthermore, as the LSM-tree grows in size and expands to more levels, each lookup may traverse multiple levels, thereby incurring high latency.

Many academic and industrial projects have improved the LSM-tree usage for KV stores to address different performance aspects and storage architectures (see §5). Examples include new LSM-tree organizations (e.g., a trie-like structure in LSM-trie [45] or a fragmented LSM-tree in PebblesDB [36]) and KV separation [9], [32]. However, the indexing structure is still largely based on the LSM-tree, and such solutions often make delicate design trade-offs. For example, LSM-trie [45] trades the scan support for improved read and write performance; PebblesDB [36] relaxes the fully sorted requirement in each level and sacrifices read performance; KV separation [9], [32] keeps only keys and metadata in the LSM-tree and stores values separately, but incurs extra lookup overhead to keys and values in separate storage [9]. Thus, the full performance potentials of KV stores are still constrained by the multi-level design in the LSM-tree.

Our insight is that *hash indexing* is a well-studied indexing technique that supports a fast lookup of a specific KV pair. However, combining hash indexing and the LSM-tree is challenging, as each of them makes a different design trade-

- *An earlier version of this paper appeared at the 36th IEEE International Conference on Data Engineering (ICDE2020) [48]. In this journal version, we develop a parallel scheme which operates different partitions in parallel to further optimize the performance, and also address the scan performance issue caused by hashing and partial KV separation by leveraging asynchronous I/O (AIO). We consider more baselines and conduct extensive experiments to validate the effectiveness and efficiency of the optimized designs and UniKV.*
- *Qiang Zhang, Yongkun Li and Yinlong Xu are with University of Science and Technology of China, Hefei, China.*
E-mail: zhgqiang@mail.ustc.edu.cn, {ykli, ylxu}@ustc.edu.cn.
Yongkun Li is the corresponding author.
- *Patrick P. C. Lee is with the Chinese University of Hong Kong, Hong Kong, China. E-mail: pclee@cse.cuhk.edu.hk.*

off. Specifically, hash indexing supports high read and write performance, but does not support efficient scans. Also, a hash index is kept in memory for high performance, but the extra memory usage poses scalability concerns when more KV pairs are stored. The LSM-tree supports both efficient writes and scans as well as high scalability, but suffers from high compaction and multi-level access overheads. Thus, we pose the following question: *Can we unify both hash indexing and the LSM-tree to simultaneously address reads, writes, and scans in a high-performance and scalable fashion?*

We observe that data locality, which also commonly exists in KV storage workloads [3], [4], [29], offers an opportunity to address the above problem. To leverage data locality, we design UniKV, which unifies hash indexing and the LSM-tree in a single system. Specifically, UniKV adopts a layered architecture to realize *differentiated data indexing* by building a light-weight in-memory hash index for the recently written KV pairs that are likely to be frequently accessed (i.e., *hot*), and keeping the large amount of infrequently accessed (i.e., *cold*) KV pairs in a fully sorted order with an LSM-tree-based design. To efficiently unify hash indexing and the LSM-tree without incurring large memory and I/O overhead, UniKV carefully designs four techniques. First, UniKV develops light-weight two-level hash indexing to keep the memory overhead small while accelerating the access to hot KV pairs. Second, to reduce the I/O overhead incurred by migrating the hot KV pairs to the cold KV pairs, UniKV designs a partial KV separation strategy to optimize the migration process. Third, to achieve high scalability and efficient read and write performance in large KV stores, UniKV does not allow multiple levels among the cold KV pairs as in conventional LSM-tree-based design; instead, it employs a scale-out approach that dynamically splits data into multiple independent partitions via a dynamic range partitioning scheme. As different partitions have no overlaps in keys under dynamic range partitioning, we can parallelize the operations on partitions for performance improvements. Specifically, we propose a parallel optimization scheme to manage different partitions in parallel. Finally, we propose multiple strategies (e.g., size-based merge and asynchronous I/O (AIO)) to optimize scan performance. To this end, we show that UniKV simultaneously achieves high read, write, scan performance for large KV stores.

We implement a UniKV prototype atop LevelDB [18] and evaluate its performance via both micro-benchmarks and YCSB [11]. Compared with LevelDB [18], RocksDB [17], PebblesDB [36], and Titan [35], UniKV achieves significant throughput gains in both reads and updates. For the six core YCSB workloads except for scan-dominated workload E (in which UniKV achieves slightly better performance), UniKV also achieves significant throughput gains.

Our UniKV prototype is open-sourced for public use at: https://github.com/ustcadsl/unikv.

2 Background and Motivation

2.1 Background: LSM-tree

To support large-scale storage, many KV stores are designed based on the LSM-tree (e.g., [9], [17], [18], [32], [36]). Figure 1 depicts the design of an LSM-tree-based KV store, using LevelDB as an example. Specifically, it keeps two

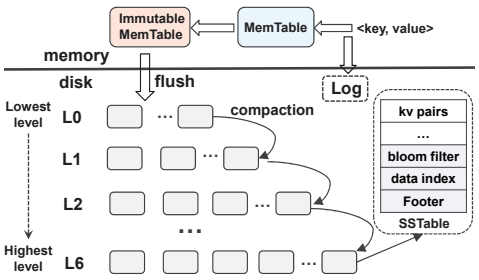


Fig. 1. Architecture of an LSM-tree-based KV store (LevelDB).

sorted skiplists in memory, namely *MemTable* and *Immutable MemTable*, multiple *SSTables* on disk, and a manifest file that stores the metadata of the SSTables. SSTables on disk are organized as a hierarchy of multiple levels, each of which contains multiple sorted SSTables and has fixed data capacity. The data capacity in each level increases from lower to higher levels in the LSM-tree. The main feature of the LSM-tree is that data flows only from lower to higher levels and KV pairs in each level are sorted to balance the trade-off between read and write performance.

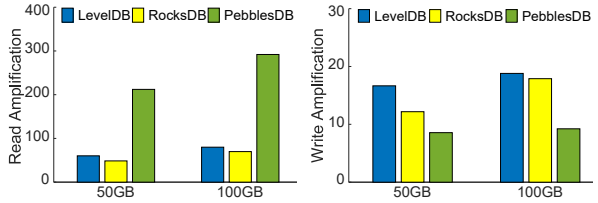
The insertion of a new KV pair works as follows. The new KV pair is first appended to an on-disk log file to enable recovery. It is then added to the MemTable, which is sorted by keys. Once the MemTable becomes full, it is converted to the Immutable MemTable that will be flushed to level L_0 on disk. Thus, SSTables in L_0 could have overlaps. When L_i becomes full, the KV pairs at L_i will be merged into L_{i+1} through compaction, which first reads out data in the two adjacent levels and then rewrites them back to L_{i+1} after merging. Clearly, this process introduces extra I/Os and causes *write amplification*.

The lookup of a KV pair search multiple SSTables. Specifically, it searches all the SSTables at level L_0, and one SSTable from each of the other levels until the data is found. Although Bloom filters [43] are adopted, due to their false positives and limited memory for caching Bloom filters, multiple I/Os are still needed for each read, thereby causing *read amplification*.

Limitations. The main limitations of LSM-tree-based KV stores are the large write and read amplifications. Prior studies show that the write amplification factor of LevelDB could reach up to 50× [32]. For example, when merging a SSTable from L_i to L_{i+1}, LevelDB may read up to 10 SSTables from L_{i+1} in the worst case, and then write back these SSTables to L_{i+1} after merging and sorting the KV pairs. Thus, some unique KV pairs may be read and written many times when they are eventually migrated from L_0 to L_6 through a series of compactions. This degrades write throughput, especially for large KV stores.

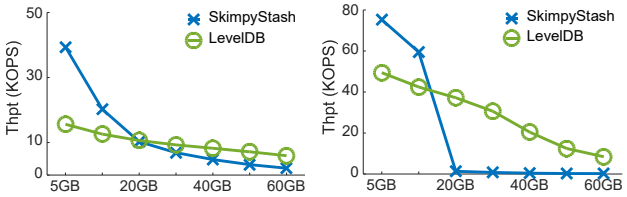
The read amplification may be even worse due to two reasons. First, there is no in-memory index for each KV pair, so a lookup needs to search multiple SSTables from the lowest to highest level. Second, checking an SSTable also needs to read multiple index blocks and Bloom filter blocks within the SSTable. Thus, the read amplification will be more serious when there are more levels, and it may even reach over 300× [32].

We evaluate various LSM-tree-based KV stores, including LevelDB, RocksDB, and PebblesDB, to show the read



(a) Read amplification (b) Write amplification

Fig. 2. Read & write amplification of LSM-tree-based KV stores.



(a) Write throughput (b) Read throughput

Fig. 3. Performance comparison of hash-index-based and LSM-tree-based KV stores (SkimpyStash [14] and LevelDB [18]).

and write amplifications (see Figure 2). We test the read and write amplifications for two different database sizes, 50 GB and 100 GB (the KV pair size is 1 KB). For each KV store, we use the default parameters, and do not use extra `SSTable_File_Level` Bloom filters for PebblesDB. We see that the read and write amplifications increase to up to 292× and 19×, respectively, as the database size increases to 100 GB. Thus, the read and write performance degrades significantly in large KV stores.

2.2 Motivation

In-memory hash indexing. Given the high write and read amplifications in LSM-tree-based KV stores, we can leverage in-memory hash indexing for efficient lookups. Hash indexing has been widely studied in KV stores [13], [14]. For each KV pair, a key tag is computed from the key via certain hash functions and is mapped to a pointer that points to the KV pair on disk. Both the key tag and the pointer serve as the index for an KV pair and stored in a bucket of an in-memory hash index for fast lookup. Note that multiple keys can be hashed to the same bucket, in which case a hash chain can be used to resolve hash collisions. Thus, hash indexing allows both single-key write (i.e., PUT) and read (i.e., GET) operations to be efficiently issued.

Design tradeoffs. Hash indexing and the LSM-tree make different trade-offs in read performance, write performance, and scalability. Hash indexing, while being efficient in PUT and GET, has the following limitations. First, it is not scalable. For large KV stores, both the read and write performance may significantly drop due to hash collisions and limited memory, and become even worse than the LSM-tree. We compare the read and write performance of SkimpyStash [14], an open-source KV store using hash indexing, and LeveLDB, using the default settings and evaluation setup in §4. Figure 3 shows that the read and write throughputs of SkimpyStash drop by 98.9% and 82.5% when the KV store size increases from 5 GB to 40 GB, respectively; its performance is worse than LevelDB. Also, hash indexing maps keys to different buckets, so it cannot efficiently support scans (i.e., reading a range of keys in a single request).

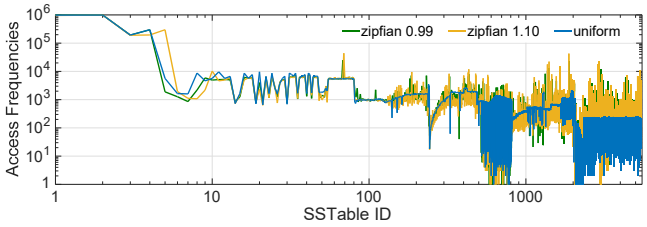


Fig. 4. Access frequency of SSTables in LevelDB.

Unlike hash indexing, the LSM-tree supports large data stores without extra in-memory indexing overhead. However, it suffers from serious read and write amplifications (see §2.1). Some efforts are made to optimize the LSM-tree, but unavoidably make a trade-off between read and write performance. For example, PebblesDB [36] improves the writes by relaxing the fully sorted feature in each level of the LSM-tree, but sacrifices the reads (see Figure 2). As the data size of the KV store increases [45], both the read and write amplifications become more severe when storing all data in the LSM-tree, as it needs more compactions to move data from lower levels to higher levels and searches more levels in reads.

Workload characteristics. Our insight is that the above challenges can be addressed by leveraging workload locality. Real-world workloads of KV stores are not only read-write mixed [3], [24], [38] (see §1), but also have high data access skewness [3], [4], [29], in which a small percentage of keys may receive most of the requests. This workload skewness is also validated by our experiments with LevelDB as shown in Figure 4, which shows the access frequency of each SSTable. The x-axis represents the identities of SSTables numbered sequentially from the lowest level to the highest level, while the y-axis shows the number of accesses to each SSTable. On average, the SSTables in lower levels with smaller IDs that are recently flushed from memory have much higher access frequency than those in higher levels. For example, the last level contains about 70% of SSTables, but they only receive 9% of requests.

Main idea. Based on the design trade-offs of hash indexing and the LSM-tree, as well as the presence of workload locality, our idea is to unify hash indexing into an LSM-tree-based KV store to address their fundamental limitations. Specifically, we use hash indexing to accelerate single-key access on a small fraction of frequently accessed (i.e., hot) KV pairs; meanwhile, for the large fraction of infrequently accessed (i.e., cold) KV pairs, we still follow the original LSM-tree-based design to provide high scan performance. Also, to support very large KV stores so as to provide good scalability, we propose *dynamic range partitioning* to expand KV stores in a scale-out manner, and propose a parallel optimization scheme to perform operations between partitions in parallel to further improve performance. Besides, we carefully manage the merge process when converting data index from hashing to the LSM-tree, so as to guarantee high scan performance. In short, by unifying hash indexing and the LSM-tree in a single system with careful designs, we can achieve high performance in reads, writes, and scans in large KV stores.

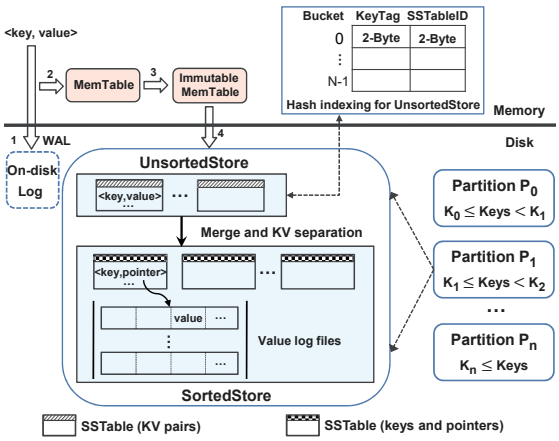


Fig. 5. UniKV architecture.

3 UniKV Design

3.1 Architectural Overview

UniKV adopts a two-layer architecture as shown in Figure 5. The first layer is called the *UnsortedStore*, which keeps the SSTables that are recently flushed from memory in an unsorted manner (as in LevelDB, we organize data in memory as Memtables, and refer to them as SSTables when they are flushed from memory to disk). The second layer is called the *SortedStore*, which stores the SSTables merged from the UnsortedStore in a fully sorted order. Our insight is to exploit data locality (see §2.2) that recently written KV pairs are hot and account for only a small fraction of all KV pairs, so we keep them in the UnsortedStore (without sorting) and index them with in-memory hash indexing for fast reads and writes. Meanwhile, we treat the KV pairs that have been written for a long time as cold KV pairs, and keep the large amount of cold KV pairs in the SortedStore in a fully sorted order for efficient scans and scalability. UniKV realizes the idea via the following techniques.

- **Differentiated indexing.** UniKV unifies hash indexing and the LSM-tree to differentiate the indexing of KV pairs in the UnsortedStore and the SortedStore. It designs lightweight two-level hash indexing to balance memory usage and hash collisions (see §3.2).
- **Merge with partial KV separation.** To efficiently merge KV pairs from the UnsortedStore to the SortedStore, UniKV proposes a partial KV separation scheme that stores keys and values separately for the data in the SortedStore to avoid frequent movement of values during the merge process (see §3.3).
- **Dynamic range partitioning.** To improve scalability, UniKV proposes a range partitioning scheme that dynamically splits KV pairs into multiple partitions that are independently managed according to key ranges, so as to expand a KV store in a scale-out manner (see §3.4).
- **Optimizations.** UniKV proposes a parallel optimization scheme that parallelizes the operations on partitions (see §3.5). It also optimizes the scan performance with multiple strategies such as asynchronous I/O (see §3.6).

3.2 Differentiated Indexing

Data management. We elaborate the layered architecture of UniKV. The first layer stores SSTables in an append-only way without sorting across SSTables, and relies on in-memory hash indexing for fast key lookups. The second layer organizes SSTables in an LSM-tree with keys being fully sorted. It also employs KV separation by storing values in separate log files, while keeping keys and value locations in SSTables. For data organization within each SSTable, UniKV follows the LSM-tree design, in which each SSTable has a fixed size and contains a number of KV pairs.

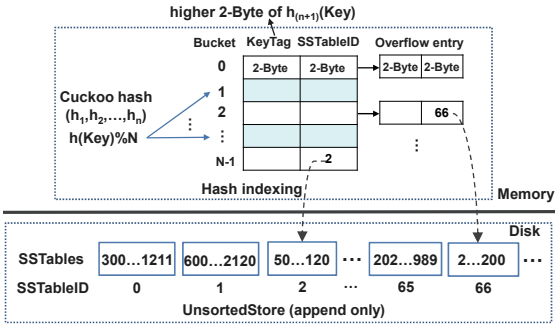


Fig. 6. Structure of the UnsortedStore and in-memory hash indexing.

UniKV *removes* Bloom filters from all SSTables to save memory space, with the following rationale. For a KV pair in the UnsortedStore, we can obtain its location via in-memory hash indexing; for a KV pair in the SortedStore, we can also efficiently locate the SSTable that contains the KV pair via binary search by comparing the key with boundary keys of SSTables that are kept in memory, as all keys in the SortedStore are fully sorted across SSTables. Thus, we do not need to search KV pairs level-by-level in the SortedStore, and the Bloom filters can be removed. Even for looking up a non-existing key, UniKV only needs to check one SSTable in the SortedStore. This incurs only one extra I/O to read the *unnecessary* data from the SSTable to confirm the non-existence of the key, because we can directly decide which data block (usually 4 KB) within the SSTable needs to be read out by using the metadata in the index block, which is usually cached in memory. In contrast, existing LSM-tree-based KV stores may need up to 7.6 reads to SSTables and incur 2.3 reads on average for a key lookup due to false positives of Bloom filters and multi-level searching [28].

For data management in memory, UniKV uses similar ways as in conventional LSM-tree-based KV stores and ensures data durability via write-ahead logging (WAL). That is, KV pairs are first appended to a log on disk for crash recovery, and then inserted into the Memtable that is organized as a skip list in memory. When the MemTable is full, it is converted to the *Immutable MemTable* and a new MemTable is generated to handle the following writes. A background thread then flushes the Immutable MemTable to the UnsortedStore in an append-only manner and records the addresses of these KV pairs in the hash index.

Hash indexing. SSTables in the UnsortedStore are written in an append-only manner and indexed with an in-memory hash index. Note that the hash index of UniKV only stores the location of KV pairs, in contrast to storing KV pairs in the hash table (e.g., in FloDB [6]). To reduce memory usage, we build a light-weight hash index with two-level hashing that combines cuckoo hashing and linked hashing to solve hash collisions. As shown in Figure 6, the hash index contains N buckets. Each bucket stores the index entries of KV pairs with cuckoo hashing, so it may append one

or several overflowed index entries due to hash collisions. When we build an index entry for a KV pair, we search the buckets according to the hash results computed by n hash functions (from the general hash function library [34]), i.e., $(h_1, h_2, \cdots, h_n)$ `(key)%N`, until we find an empty one. Note that we use at most n hash functions in this cuckoo hashing scheme. If we cannot find an empty bucket in the n buckets, we generate an overflowed index entry and append it to the bucket located by h_n `(key)%N`.

After locating a bucket, we record the keyTag and SSTable ID of the KV pair into the selected index entry. Each index entry contains three attributes: ⟨`keyTag, SSTableID, pointer`⟩. The `keyTag` stores the higher 2-Byte of the hash value of the key computed with a different hash function, i.e., h_{n+1} `(key)`, and is used to quickly filter out index entries during key lookup (see below). The `SSTableID` uses 2 bytes to store an SSTable ID, and we can index 128 GB of SSTables of size 2 MB each in the UnsortedStore. The `pointer` uses 4 bytes to point to the next index entry in the same bucket.

Key lookup. The key lookup process works as follows. First, we compute the `keyTag` using h_{n+1} `(key)`. Then we search the candidate buckets from h_n `(key)% N` to h_1 `(key)% N`, until we find the KV pair. For each candidate bucket, since the newest overflow entry is appended to the tail. Thus, we compare `keyTag` with the index entries belonging to this bucket from the tail of overflow entries. Once we find a matched `keyTag`, we retrieve the metadata of this SSTable with the `SSTableID` and read out the KV pair. Note that the queried KV pair may not exist in this SSTable due to hash collisions on h_{n+1} `(key)` (i.e., different keys have the same `keyTag`). Then we continue searching the candidate buckets. Finally, if the KV pair is not found in the UnsortedStore, we search the key in the SortedStore via binary search as all keys are fully sorted.

Memory overhead. We analyze the memory overhead of hash indexing. Each KV pair in the UnsortedStore costs one index entry, and each index entry costs 8 bytes in memory. Thus, for every 1 GB data in the UnsortedStore with 1 KB KV pair size, it has around 1 million index entries, which take around 10 MB memory given that the utilization of buckets is about 80% in our experiments. This memory usage is less than 1% of the data size in the UnsortedStore. However, hash indexing may incur large memory overhead for very small KV pairs; for example, when the KV pair size is 128 B, the memory usage of hash index is about 80 MB per 1 GB data in the UnsortedStore (i.e., 8× the memory usage for 1 KB KV pairs). To reduce memory overhead, one solution is to differentiate the management for KV pairs of different sizes [27], by using the conventional LSM-tree to manage small KV pairs and UniKV to manager large KV pairs. Based on its analysis in [27], the threshold to differentiate KV pairs could be set as 128 B.

Our hash indexing scheme makes a design trade-off. On one hand, hash collisions may exist when we allocate buckets for KV pairs, i.e., different keys have the same hash value h `(key)` and are allocated to the same bucket. Thus, we need to store the information of keys in index entries to differentiate keys during lookup. On the other hand, storing the complete key wastes memory. To balance memory usage and read performance, UniKV uses two hashes and keeps only 2 bytes of the hash value as a `keyTag`. This significantly reduces the probability of hash collision (e.g., less than 0.001% [8]), as also shown in our experiments. Even if hash collisions happen, we can still resolve them by comparing the keys stored on disk.

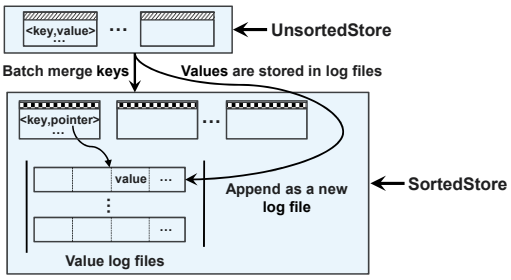


Fig. 7. Merge with partial KV separation.

3.3 Merge with Partial KV Separation

Recall that KV pairs in the UnsortedStore are indexed with an in-memory hash index, which incurs extra memory usage. To limit memory overhead, UniKV limits the UnsortedStore size. When the size reaches a predefined threshold `UnsortedLimit` (i.e., the maximum UnsortedStore size), UniKV triggers a merge operation to merge KV pairs from the UnsortedStore into the SortedStore. The parameter `UnsortedLimit` determines the trade-off between write performance and memory overhead. A large `UnsortedLimit` implies that more data is stored in the UnsortedStore. It triggers fewer merge operations for higher write performance, yet the memory usage also increases. To guide the setting of `UnsortedLimit`, we evaluate the impact of `UnsortedLimit` on write performance and memory overhead (see §4.4).

Merging KV pairs from the UnsortedStore into the SortedStore may incur large I/O overhead, as existing KV pairs in the SortedStore are also required to be read out and written back after merging and sorting. Thus, how to reduce the merge overhead is a critical but challenging issue in UniKV. Here, UniKV proposes a partial KV separation strategy, which keeps KV pairs without KV separation in the UnsortedStore but separates keys from values in the SortedStore. The rationale is as follows. The KV pairs in the UnsortedStore are flushed from memory recently, so they are likely to be hot due to data locality. Thus, we keep keys and values in KV pairs for efficient access. However, the KV pairs in the SortedStore are likely to be cold, and the amount of data is very large that causes high merge overhead. Thus, we adopt KV separation for the SortedStore so as to reduce the merge overhead.

Figure 7 depicts the partial KV separation design. When merging KV pairs from the UnsortedStore to the SortedStore, UniKV merges keys in batches, while keeping the values in a newly created log file in an append-only manner. It also records the value locations with pointers that are kept together with the corresponding keys. Note that each pointer entry contains four attributes: ⟨`partition,logNumber,offset,length`⟩, representing the partition number, the log file ID, the value location and length, respectively.

Garbage collection (GC) in SortedStore. First, we point out that GC in UniKV implies to reclaim the storage space which

is occupied by invalid values in log files. Note that invalid keys in SSTables are deleted during compaction, which is independent with GC. In UniKV, GC operates in units of log files, and is triggered when the total size of a partition is above a predefined threshold. Specifically, the GC process first identifies and reads out all valid values from log files in the partition, then writes back all valid values to a new log file, and generates new pointers to record the latest locations of values, and finally deletes invalid pointers and obsolete files after GC. We need to address two key issues: (i) Which partition should be selected for GC? (ii) How to quickly identify and read out valid values from log files?

Unlike the previous KV separation scheme (e.g., [32]) that performs GC in a strict sequential order, UniKV can flexibly choose any partition to perform GC as KV pairs are mapped to different partitions according to their key ranges and the operations between partitions are independent. We adopt a greedy approach which selects the partition with the largest amount of KV pairs to perform GC. Also, to check the validity of values in log files of the selected partition, UniKV only needs to query the keys and pointers in the SortedStore that always maintains the valid keys and latest locations of valid values. Thus, for each GC operation, UniKV only needs to scan all keys and pointers in the SortedStore to get all valid values, and the time cost only depends on the total size of SSTables in the SortedStore. Note that GC and compaction operations are performed sequentially in UniKV as they both modify SSTables in the SortedStore, so GC operations also occur along the way of data loading and the GC cost is also counted in measuring write performance.

3.4 Dynamic Range Partitioning

As the SortedStore grows in size, if we simply add more levels for large-scale storage as in most existing LSM-tree-based KV stores, it will incur frequent merge operations that move data from lower to higher levels during writes and also trigger multi-level accesses during reads. Also, each GC needs to read out all values from log files by querying the LSM-tree and write valid values back to disk, so the GC overhead becomes substantial as the number of levels increases. Thus, UniKV proposes a dynamic range partitioning scheme to expand storage in a scale-out manner. This scheme maps KV pairs of different key ranges into different partitions that are managed independently, and each partition has its own UnsortedStore and SortedStore.

The dynamic range partitioning scheme works as follows (see Figure 8). Initially, UniKV writes KV pairs in one partition (e.g., P_0). Once its size exceeds the predefined threshold `partitionSizeLimit` (which is configurable), UniKV splits the partition into two partitions with equal size according to the key range and manages them independently (e.g., P_0 is split to P_1 and P_2). With range partitioning, the key feature is that partitions should have no overlaps in keys. To achieve this, KV pairs in both the UnsortedStore and the SortedStore need to be split.

To split the keys in both the UnsortedStore and the SortedStore, UniKV first locks them and stalls write requests. Note that the lock granularity is a partition, i.e., UniKV locks the whole partition and stalls all writes to this partition during splitting. Then it sorts all the keys to avoid overlaps between partitions. It first flushes all in-memory KV pairs

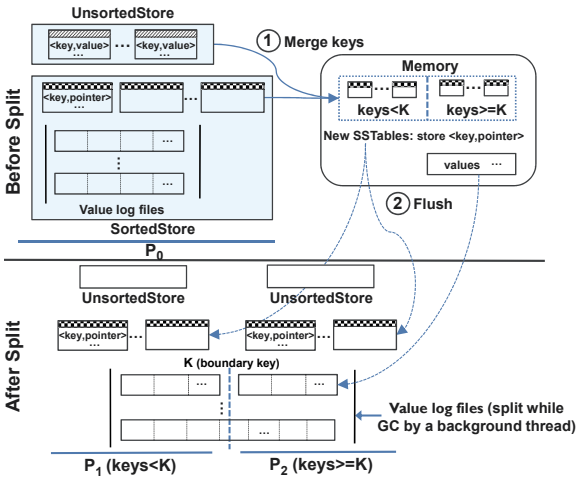


Fig. 8. Dynamic range partitioning.

into the UnsortedStore, and reads out all SSTables in the UnsortedStore and SortedStore to perform merge sort as in LSM-tree-based KV stores. It then divides the sorted keys into two parts of equal size and records the boundary key K between the two parts. Note that this boundary key K serves as the splitting point. That is, the KV pairs with keys smaller than the key K form one partition P_1, while others form another partition P_2. With the splitting point, UniKV divides the valid values in the UnsortedStore into two parts and writes them to the corresponding partitions by appending them to the newly created log file of each partition. Finally, UniKV stores the value locations in the pointers which are kept together with the corresponding keys, and writes all keys and pointers back to the SortedStore in the corresponding partitions. It releases the locks and resumes to handle write requests after splitting the keys.

Second, to split the values in the SortedStore (which are stored in multiple log files separately), UniKV adopts a lazy split scheme which splits values in log files during GC with a background thread. It works as follows. The GC thread in P_1 first scans all SSTables in the SortedStore of P_1. It then reads out valid values from old log files that are shared by P_1 and P_2, and writes them back to a newly created log file belonging to P_1. Finally, it generates new pointers that are stored with corresponding keys to record the latest locations of values. The GC thread in P_2 performs the same procedure as in P_1. The main benefit of the lazy split design is to reduce the split overhead significantly by integrating it with GC operations to avoid large I/O overhead. Note that with range partitioning, the smallest key in P_2 must be larger than all keys in P_1. This range partitioning process repeats once a partition reaches its size limit. We emphasize that each split operation can be considered to have one compaction operation plus one GC operation, but they must be performed sequentially. Thus, splitting keys in a partition introduces extra I/Os. After splitting, each partition has UnsortedStore, SortedStore, and log files.

For large KV stores, the initial partition may be split multiple times and thus generates multiple partitions. To efficiently locate a certain partition when performing read and write operations, we record the partition number and the boundary keys of each partition in memory, which serves as the partition index. We also persistently store the partition

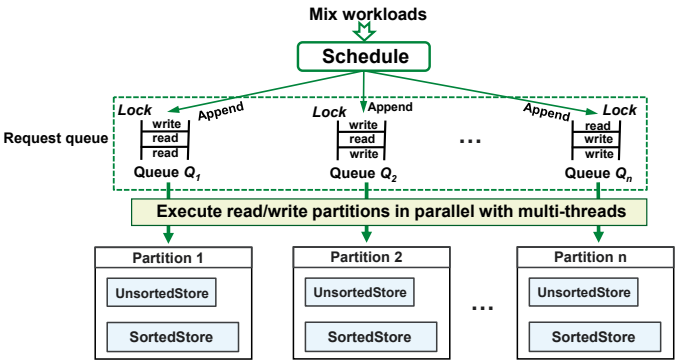


Fig. 9. Workflow of the parallel scheme.

index in the manifest file on disk. In addition, different partitions have no overlap in keys, so each key can only exist in one partition. Thus, key lookup can be performed by first locating a partition, which can be done efficiently by checking the boundary keys, followed by querying the KV pairs within only one partition. In short, the dynamic range partitioning scheme expands storage in a scale-out manner by splitting KV pairs into multiple independent partitions. Thus, the scheme can guarantee high read and write performance, as well as efficient scans, even for large KV stores so as to enable good scalability.

3.5 Parallel Optimization

Recall from §3.4 that there are multiple partitions after a series of splitting, and these partitions have no overlaps in keys. Thus, they can be managed independently. We propose a parallel optimization design to perform reads and writes across different partitions in parallel for further performance gains. Note that we should ensure that other aspects of UniKV remain unaffected.

Figure 9 depicts the workflow of reading or writing different partitions in parallel. In order to read and write different partitions in parallel, we propose a parallel multi-queue scheme (see Figure 9), which maintains a request queue for each partition. When UniKV receives reads and writes from clients, it first compares the requested keys with the boundary keys of the partitions. It then directs the requests to the corresponding request queue and appends the requests to the queue according to the request order. Note that each partition maintains a request queue and the queue size is configurable. When the request queue is full, we temporarily suspend the thread for receiving requests from clients, so that the requests are blocked on the client side until the queue has enough free space.

To handle concurrent requests from multiple clients, UniKV adopts the same lock mechanism as LevelDB. Specifically, when a read/write request is scheduled to a partition, it first acquires the lock. The request is then inserted into the in-memory request queue of the corresponding partition. After the request is appended to the request queue, it releases the lock, so the following read/write requests can acquire the lock and be inserted to the request queue. For the requests in the request queue, UniKV uses a background thread to fetch the requests from the request queue and execute them in a first-in-first-out order. If the request is a write request, UniKV writes the data to the WAL and MemTable of the corresponding partition. If the request is a read request, UniKV reads a KV pair from the corresponding partition.

In UniKV, each partition has a separate request queue, rather than a single global queue as in LevelDB. Thus, the read/write requests issued to different partitions can be fully parallelized across partitions. For example, as shown in Figure 9, each partition i has a request queue Q_i and has a dedicated background thread to process requests in the queue. Note that the read/write requests within the same partition are executed sequentially to ensure data consistency and correctness. Also, we keep the same read and write workflows within a partition as before. Thus, while the parallel multi-queue scheme ensures that the read/write requests to different partitions are executed in parallel, it remains compatible with other design parts in UniKV.

Discussion. We analyze the time cost of read/write operations when using the parallel scheme. Suppose that we write N KV pairs and read M KV pairs in total, and there are Q partitions after splitting. For UniKV without parallel optimization, we let RT_i be the time of reading KV pairs from partition i, so the total time of reading M KV pairs is $RT = \sum_{i=1}^{Q} RT_i$. Besides, we let $MemW_i$, $LogW_i$, $FlushW_i$, and $CompW_i$ be the time of writing KV pairs to MemTables, appending to the write-ahead log, flushing MemTables and performing compaction in partition i, respectively. The total time of writing N KV pairs is given by $WT = \sum_{i=1}^{Q}(MemW_i + LogW_i + FlushW_i + CompW_i)$.

For UniKV with the parallel optimization, we can read Q partitions in parallel, so the total read time is $RT' = \max_{i=1}^{Q}\{RT_i\}$ in the best case. For writes, each step on the write path, i.e., writing MemTables, appending to the log, flushing MemTables and compaction, can be executed in parallel for different partitions, so the total write time is given by $WT' = \max_{i=1}^{Q}\{MemW_i\} + \max_{i=1}^{Q}\{LogW_i\} + \max_{i=1}^{Q}\{FlushW_i\} + \max_{i=1}^{Q}\{CompW_i\}$ in the best case. In §4.3, we present a numerical study on the overhead based on the above analysis.

3.6 Scan Optimization

UniKV supports efficient scans as in the LSM-tree-based KV stores. It employs different strategies to optimize scans for the UnsortedStore and the SortedStore.

3.6.1 Size-based Merge for the UnsortedStore

Recall that all SSTables in the UnsortedStore of a partition are directly flushed from memory in an append-only manner, so they may have overlaps in key ranges with each other. Thus, scans need to read every SSTable while performing the `seek()` or `next()` operations, thereby incurring additional random reads. To avoid many random reads caused by checking every SSTable in the UnsortedStore during scans, UniKV proposes a size-based merge strategy.

The main idea of the size-based merge strategy is as follows (see Figure 10). When UniKV executes scan requests, it determines whether the number of SSTables in the UnsortedStore exceeds the threshold `scanMergeLimit`. If so, UniKV triggers a background thread to perform the size-based merge without blocking scan requests. Specifically, the size-based merge operation first reads out all SSTables from the UnsortedStore and merge-sorts all KV pairs. Any

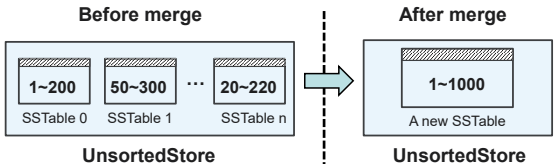


Fig. 10. Size-based merge.

invalid KV pairs will be deleted during this process. Then all valid KV pairs after the merge-sort will be written to a new SSTable. Finally, all old SSTables in the UnsortedStore will be deleted. Note that the size-based merge strategy presents a trade-off between the scan and write performance. First, the merge operation merges all SSTables in the UnsortedStore into a large one, which can significantly improve scans due to efficient sequential reads on the fully sorted KV pairs (especially when processing a large number of scans). However, the merge operation will cause extra I/Os to read, merge, and write back KV pairs, which degrades the write performance.

To tune the value of `scanMergeLimit`, we observe that a smaller `scanMergeLimit` implies that the UnsortedStore has a higher degree of ordering, but more merge operations are triggered. Thus, the write performance is worse, but the scan performance is better as fewer SSTable files are randomly read. On the other hand, the read performance is not affected by `scanMergeLimit`, as UniKV can quickly find KV pairs in the UnsortedStore by querying the in-memory hash index, regardless of the ordering of the UnsortedStore. To guide the setting of `scanMergeLimit`, we also evaluate the write, read, and scan performance under different values of `scanMergeLimit` (see §4.4).

3.6.2 AIO for the SortedStore

Recall that keys and values are stored separately in the SortedStore. This introduces random reads of values for scans. One way of accelerating scans is to fetch values with multi-threading, yet multi-threading consumes a lot of CPU resources. We observe that asynchronous non-blocking I/O (AIO) [23] has efficient I/O performance and runs in a single thread with less CPU consumption. To avoid the effect of limited CPU resources on scans, UniKV leverages AIO to accelerate scans for the SortedStore. Specifically, a scan request consists of a `Seek()` and a series of `next()` operations. UniKV first performs a `Seek()` to find the first KV pair. It then performs a series of `next()`, each of which reads a KV pair: if the value of the KV pair is a pointer, UniKV initiates an AIO to fetch its value from log files; otherwise, it returns the KV pair. Finally, UniKV waits for all AIOs to complete before performing the next scan request. With AIO, UniKV can initiate many random reads on log files concurrently when performing a series of `next()`, so the scan performance improves significantly. Also, AIO is executed with a single thread, so it achieves high scan performance even under limited CPU resources.

3.7 I/O Cost Analysis

To understand the performance trade-offs of LSM-tree-based KV stores and UniKV, we analyze their worst-case I/O costs for both writes and reads. Suppose that we have N KV pairs in total and P KV pairs in a memory component. Let T be the ratio of capacities between two adjacent levels.

LSM-tree-based KV stores. If an LSM-tree contains L levels, then level i ($i \geq 0$) contains at most $T^{i+1} \cdot P$ KV pairs, and the largest level contains approximately $N \cdot \frac{T}{T+1}$ KV pairs. Thus, the number of levels for N KV pairs can be approximated as $L = \lceil \log_T(\frac{N}{P} \cdot \frac{T}{T+1}) \rceil$. Note that the write cost measures the overall I/O cost for KV pairs being merged into the highest level. Thus, the write cost for each KV pair is $O(T \cdot L)$. For the read I/O cost, suppose that Bloom filters allocate M bits for each key. Then the false positive rate of a Bloom filter is $e^{-M \cdot \ln(2)^2}$ [12], and it can be simplified as 0.6185^M [28]. Thus, the I/O cost of each read is $O(L \cdot 0.6185^M)$, and the worst-case lookup incurs $O(L)$ I/Os by accessing all levels.

UniKV. Suppose that a partition contains R KV pairs, and there are Q partitions after splitting and key size occupies $\frac{1}{K}$ of KV pair size. Then the number of partitions for N KV pairs is $Q = \lceil \frac{N}{R} \rceil$, where $Q < L$. A component at the SortedStore will be merged $T - 1$ times until it triggers a split operation. Note that the values are not read and written during merge, and they are only moved during GC. Suppose that GC is triggered when the amount of written KV pairs is equal to $\frac{1}{H}$ of partition size, so the values will be merged $H - 1$ times before splitting and $H < \frac{T}{2}$. Also, each split operation consists of a merge operation and a GC operation. Thus, the write cost for each KV pair is $O(T \cdot \frac{Q}{K} + Q \cdot H)$. Finally, reads are very efficient for UniKV since a KV pair is either in the UnsortedStore or in the SortedStore. Each read operation incurs 1 I/O only if the KV pair exists in the UnsortedStore through querying the hash index, or 2 I/Os for accessing the key and the value from the SortedStore due to KV separation. Thus, the worst-case lookup incurs 3 I/Os, including 1 I/O in the UnsortedStore when hash collisions occur and 2 I/Os in the SortedStore.

3.8 Implementation Details

We implement a UniKV prototype on Linux based on LevelDB v1.20 [18] in C++, by adding or modifying around 8K lines of code. Since the UnsortedStore and the SortedStore build on SSTables, UniKV can leverage the mature, well-tested codes for SSTable management in LevelDB.

Crash consistency. To address crashes during writes, UniKV supports crash consistency in three aspects: (i) buffered KV pairs in MemTables; (ii) in-memory hash indexing; and (iii) GC operations in the SortedStore. For the KV pairs in MemTables, SSTables, and metadata (manifest files), UniKV adopts write-ahead logging (WAL), as in LevelDB, for crash recovery. Also, UniKV adds the specific metadata (e.g., boundary keys and partition number) to the manifest file.

For the in-memory hash index, UniKV uses the checkpointing technique. It saves the hash index in a disk file when flushing every `freq` MemTables from memory to the UnsortedStore for each partition (where `freq` is set to 30 by default). Thus, rebuilding the hash index can be done by reading the latest saved copy from the disk file and replaying the newly written SSTables since the last backup. We also study the impact of `freq` in §4.4.

Crash consistency in GC operations is different, as they may overwrite existing valid KV pairs. To protect existing valid KV pairs against crashes during GC, UniKV performs

the following steps: (i) identifying all valid KV pairs according to the keys and pointers in the SortedStore; (ii) reading all valid values from the log files and writing them back to a new log file; (iii) writing all new pointers that point to the new log file with the corresponding keys into new SSTables in the SortedStore; (iv) marking new SSTables as valid and old log files with a `GC_done` tag to allow them to be deleted. If system crashes during GC, then we can redo GC with the above steps (i)-(iv).

4 Evaluation

We compare UniKV with several state-of-the-art KV stores: LevelDB v1.20 [18], RocksDB v6.0 [17], PebblesDB [36], and Titan [35]. Note that Titan is the state-of-the-art implementation of KV separation and also widely used in production. In the experiments, we address the following questions:

- What is the performance of micro-benchmarks of UniKV and the overall performance under read-write mixed workloads? (Exp# 1-2)
- What is the performance of UniKV under the six core workloads of YCSB? (Exp# 3)
- What is the performance of the parallel optimization scheme in UniKV, the performance of scan operations with AIO, and the performance impact of hash indexing and KV separation in UniKV? (Exp# 4-7)
- What is the performance impact of different configurations (e.g., size-based merge, UnsortedStore size, workload skewness, multi-threaded operations)? (Exp# 8-12)
- What is the memory usage of the hash index and the crash recovery overhead? (Exp# 13-14)

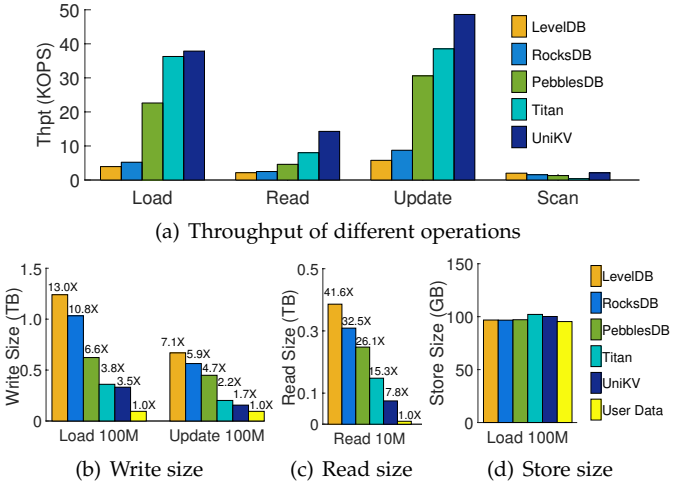
In our supplementary file, we also study the impact of other configurations (e.g., direct I/O, KV pair size, etc).

4.1 Experimental Setup

We run all experiments on a machine with a 12-core Intel Xeon E5-2650v4 2.20 GHz CPU, 16 GB RAM, and a 500 GB Samsung 860 EVO SSD. The machine runs Ubuntu 16.04 LTS, with the 64-bit Linux 4.15 kernel and ext4 file system.

We use the same setting for all stores. Specifically, we set `memtable_size` as 64 MB (same as RocksDB by default), `bloom_bits` as 10 bits, `open_files` as 1,000. For `block_cache_size`, UniKV sets it as 8 MB by default, while other KV stores set it as 170 MB to match the size of UniKV's hash index for fair comparisons. The remaining memory is used as page cache of the kernel. For the other parameters of different KV stores, we use their default values. Also, we set the number of buckets of the hash index for each partition as 4 M and use four cuckoo hash functions to ensure that the utilization of buckets exceeds 80%.

Recall that UniKV allocates the MemTable in memory for each partition. To make all systems use the same amount of memory for buffering KV pairs, we set the parameter `write_buffer_number` of RocksDB as the same number of MemTables in UniKV. We also modify the code of other KV stores to have the same total number of MemTables as in UniKV. For UniKV, by default, we set the partition size as 40 GB to balance write performance and memory cost, and the UnsortedStore size in a partition as 4 GB to limit the memory overhead of the hash index. We use YCSB [11], [37] to generate various types of workloads. By default,



(a) Throughput of different operations

(b) Write size (c) Read size (d) Store size

Fig. 11. Experiment 1 (Micro-benchmark performance).

TABLE 1
Number of I/Os (in million) of writing 100 GB of data.

	Flush	Compaction	GC	Partition Split
LevelDB	204.4	2229.5	–	–
RocksDB	204.4	1726.2	–	–
PebblesDB	204.4	1026.3	–	–
Titan	204.4	58.7	506.3	–
UniKV	204.4	188.4	288.1	35.2

we configure the YCSB client to generate 1 KB KV pairs with 24-Byte keys and issue the requests based on the Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). The YCSB client is configured in single-threaded mode.

4.2 Performance of UniKV

Experiment 1 (Micro-benchmarks). We evaluate different aspects of UniKV, including the load, read, update and scan performance, as well as data write size, data read size, KV store size, and I/O cost of different phases of writes. We first randomly load 100 M KV pairs (about 100 GB) that will finally be split into four partitions for UniKV. We then issue 10 M reads, 100 M updates, and 1 M scans of 50 GB of data (each `seek()` has 50 `next()` operations).

(i) Throughput. Figure 11(a) shows the throughput. UniKV achieves 1.1-9.6× load throughput, 1.8-6.6× read throughput, and 1.2-8.2× update throughput compared with other KV stores. For scans, UniKV achieves nearly the same throughput as LevelDB, and achieves 1.3-5.3× throughput compared with other KV stores. Note that even compared with Titan which also integrates KV separation, UniKV still achieves 1.2× update throughput and 1.8× read throughput, as well as 5.3× scan throughput, respectively. This demonstrates the effectiveness of the new designs in UniKV.

(ii) Write amplification. We evaluate the total write size. We randomly load 100 M KV pairs, then update the loaded 100 M KV pairs to show the write amplification. Figure 11(b) shows the results. UniKV significantly reduces the write size, so it reduces the write amplification to 3.5×, which is only about half of that in PebblesDB. Compared with other KV stores, UniKV reduces 7.9-73.2% of write size in the load phase, and reduces 22.8-76.6% of write size in the update phase, respectively.

(iii) Read amplification. We evaluate the total read size when operate 10 M read requests on the loaded KV store. Figure 11(c) shows the total read size for all KV stores. UniKV significantly reduces the total read size, which decides the

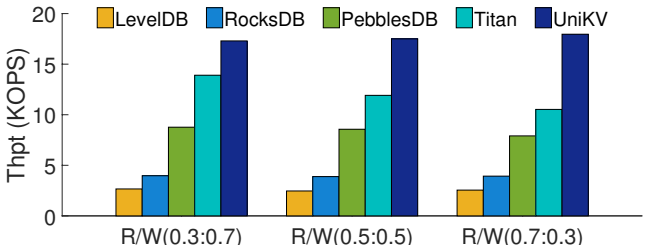


Fig. 12. Experiment 2 (Performance under mixed workloads).

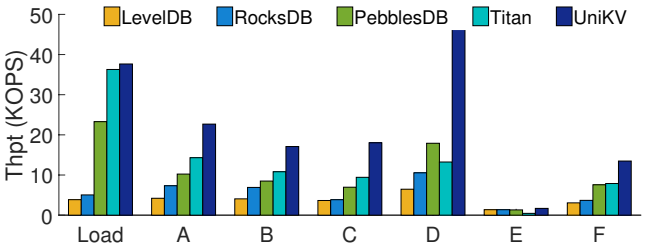
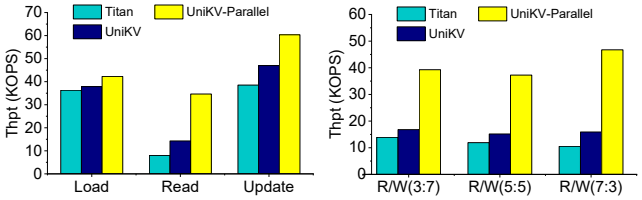


Fig. 13. Experiment 3 (YCSB performance).



(a) Micro-benchmarks (b) Mixed workloads

Fig. 14. Experiment 4 (Performance of the parallel optimization under micro-benchmarks).

read performance. The read amplification is only about 7.8×. Thus, UniKV reduces 49.1-80.6% of read size compared with other KV stores.

(iv) Space amplification. Figure 11(d) shows the KV store size of different KV stores after the load phase. All systems consume similar storage space in the load phase. UniKV and Titan incur slightly extra storage overhead, which is mainly for storing the pointers that record the locations of values.

(v) I/O costs. Finally, we evaluate the I/O cost (in number of I/Os) of different phases of writes for all KV stores, as shown in Table 1 (the I/O size is 512 bytes). For LSM-tree-based KV stores, the I/O cost is due to flushing MemTables and compacting SSTables. For UniKV, the I/O cost is due to flushing MemTables, compacting SSTables, GC in log files, and splitting partitions. Compared to other KV stores, UniKV reduces 7.1-70.6% of I/O cost in total, since the partial KV separation scheme avoids the unnecessary movement of values during compaction, and the dynamic range partitioning scheme manages each partition independently and allows UniKV to perform more fine-grained GC operations.

Experiment 2 (Performance under mixed workloads). We evaluate the overall performance of all KV stores under read-write mixed workloads. We first randomly load 100 M KV pairs, and run a workload of 100 M operations mixed with both reads and writes with different read-write ratios (30:70, 50:50, and 70:30). Figure 12 shows the overall throughput under different read-write mixed workloads. The overall throughput of UniKV is 6.5-7.1×, 4.4-4.6×, 2.0-2.3×, and 1.2-1.7×, compared with LevelDB, RocksDB, PebblesDB, and Titan, respectively. The reason is that UniKV maintains a two-layered storage architecture with in-memory hash indexing for hot data to improve read performance. It also adopts partial KV separation and dynamic range partitioning to reduce merge overhead. Thus, it improves both read and write performance simultaneously.

Experiment 3 (YCSB performance). We evaluate the performance under YCSB [11], which is an industry standard to evaluate KV stores and contains a standard set of six core workloads (A-F). We first load 100 M KV pairs before running each YCSB workload. Each workload consists of 50 M operations, except for Workload E, which contains 10 M operations with each scan involving 100 `next()`.

Figure 13 shows the results. UniKV always outperforms other KV stores under both read-dominated and write-dominated workloads. Compared with other KV stores, the throughput of UniKV is 1.6-5.4× under Workload A, 1.6-4.2× under Workload B, 1.9-4.4× under Workload C, 2.7-7.3× under Workload D, and 1.7-4.4× under Workload F. For the scan-dominated Workload E, UniKV achieves 3.7× throughput compared with Titan, and slightly better performance compared with other KV stores.

TABLE 2
Experiment 4 (Read latency breakdown) (in seconds).

	Partition	MemTable	Unsorted Store	Sorted Store	SUM
UniKV	1	0.004	9.9	166.5	176.4
	2	0.013	2.1	152.3	154.4
	3	0.001	12.7	160.9	173.6
	4	0.001	6.7	159.0	165.7
UniKV-Parallel	1	0.001	4.4	252.9	257.3
	2	0.004	39.2	184.8	224.0
	3	0.012	4.2	229.1	233.3
	4	0.004	38.5	176.6	215.1

TABLE 3
Experiment 4 (Write latency breakdown) (in seconds).

	Partition	Log	MemTable	Flush	Compaction
UniKV	1	196.6	103.2	481.4	707.3
	2	82.0	48.9	232.9	362.9
	3	26.5	17.4	85.2	69.7
	4	17.8	11.8	55.6	68.6
	SUM	322.9	181.3	855.1	1208.5
UniKV-Parallel	1	198.4	107.9	634.0	724.9
	2	62.4	54.6	350.8	348.8
	3	19.1	17.8	111.2	81.7
	4	11.6	11.4	65.6	56.0
	SUM	291.5	191.7	1161.6	1211.4

4.3 Impact of Optimization Techniques

The previous experiments do not include the parallel and scan optimizations. We evaluate them in this subsection. We also study the impact of hash indexing and KV separation.

Experiment 4 (Performance of the parallel optimization under micro-benchmarks). We compare the performance of UniKV with and without the parallel optimization. We first randomly load 100 M KV pairs, and then read 10 M KV pairs and update 100 M KV pairs. We also run a workload of 100 M operations mixed with both reads and writes with different read-write ratios. Figure 14(a) shows the throughput of loads, reads, and updates, and Figure 14(b) shows the overall throughput under read-write mixed workloads. Compared with Titan and UniKV without parallel optimization, UniKV with parallel optimization further improves both the read and write performance, e.g., it achieves 1.15-1.18×, 2.42-4.32×, and 1.20-1.56× throughput for loads, reads and updates, respectively. The parallel optimization also improves the overall throughput under mixed workloads to 2.33-4.45× by accessing KV pairs in different partitions in parallel.

We provide a breakdown on the read and write latency

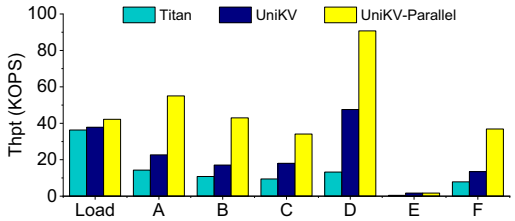


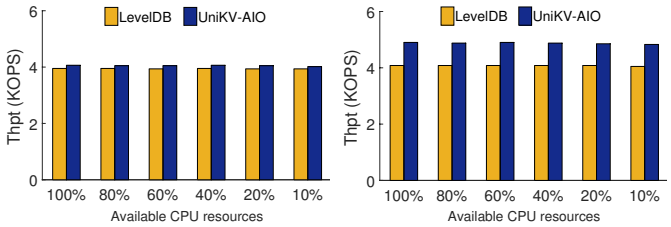
Fig. 15. Experiment 5 (Performance of the parallel optimization under YCSB workloads).

for different partitions. Table 2 shows the read latency breakdown, and the read latency in a partition consists of reading the MemTable, UnsortedStore and SortedStore. For UniKV without parallel optimization, the total time of reading 10 M KV pairs is 713.0s, which is approximately equal to the sum of the read time of each partition; that is, the reads are performed serially across partitions. For UniKV with parallel optimization (UniKV-Parallel), the total time of reading 10 M KV pairs is 288.6s, which is approximately equal to the maximum read time in four partitions (e.g., 257.3s in partition 1). It implies that the reads are performed in parallel across partitions.

Table 3 shows the write latency breakdown results. The write latency in a partition consists of appending Log, inserting MemTable, flushing MemTable to disk and compaction between the UnsortedStore and the SortedStore. For UniKV without parallel optimization, the total time of writing 100 M KV pairs is 2781.1s, which is approximately equal to the sum of the write time of each partition, as writes are performed serially across partitions. For UniKV with parallel optimization (UniKV-Parallel), the total time of writing 100 M KV pairs is 2406.0s, which approximately consists of the following parts: $2406.0 \approx \max(\text{Log}) + \max(\text{MemTable}) + \frac{3}{4}\sum(\text{Flush}) + \sum(\text{Compaction})$. That is, operations including appending Log, inserting MemTable and part of the operation of flushing are performed in parallel across partitions, while compaction operations are performed serially across partitions as the compaction in different partitions may not be triggered at the same time.

Experiment 5 (Performance of the parallel optimization under YCSB workloads). We also compare UniKV with and without parallel optimization under YCSB core workloads (Workloads A-F). We first randomly load 100 M KV pairs before running each YCSB workload. Each workload contains 50 M operations, except for Workload E, which contains 10 M operations with each scan involving 100 `next()`. Figure 15 shows the results. Compared with Titan and UniKV without parallel optimization, the parallel optimization scheme improves performance under both read-dominated and write-dominated workloads. Specifically, UniKV with parallel optimization increases the throughput to 2.42-3.85× under Workload A, 2.51-3.97× under Workload B, 1.89-3.61× under Workload C, 1.90-6.85× under Workload D and 2.73-4.67× under Workload F, respectively.

Experiment 6 (Performance impact of AIO on scans). We study how AIO improves the scan performance. From Experiments 1, LevelDB has the best scan performance among existing KV stores, so we use it as our baseline. We first randomly load 100 M KV pairs, and issue 1 M scan operations with each scan involving 50 `next()`. We vary the available CPU resources of UniKV and LevelDB processes



(a) Scan group size = 8 (b) Scan group size = 16

Fig. 16. Experiment 6 (Performance impact of AIO on scans).

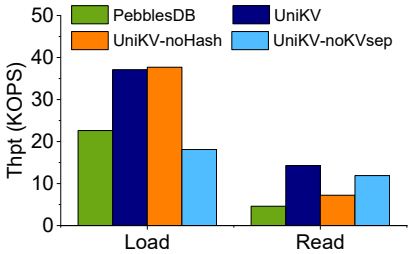


Fig. 17. Experiment 7 (Impact of hash indexing and KV separation).

TABLE 4
Experiment 8 (Impact of `scanMergeLimit`) (KOPS).

`scanMergeLimit`	1	10	20	40	*unlimited*
Write throughput	4.7	20.1	26.4	31.5	33.3
Merge cost (GB)	5534	553	274	131	0
Scan throughput	1.35	1.14	1.0	0.87	0.7
Accessed SSTables/scan	2.0	6.5	11.4	21.3	34.0
Read	15.1	14.8	14.9	14.9	14.8

from 100% to 10% of the server by using the tool `cpulimit`. Figure 16 shows the results when varying the available CPU resources. AIO always guarantee a high scan performance even though the CPU resource is very limited. The reason is that UniKV with AIO can issue a series of asynchronous I/Os to fetch values from log files concurrently, which can achieve high scan performance. Also, UniKV performs AIOs in a single thread, so the CPU overhead is limited.

We also evaluate the impact of AIO on the scan latency. When the scan group size is 8, the scan latencies of LevelDB, UniKV with AIO, UniKV without AIO are 247.4μs, 241.2μs, and 310.0μs, respectively. Thus, AIO also reduces the scan latency for UniKV.

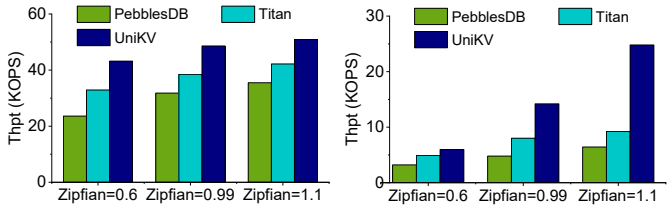
Experiment 7 (Impact of hash indexing and KV separation). We study the impact of hash index and KV separation on UniKV. We randomly load 100 M KV pairs and issue 10 M reads. Figure 17 shows the performance of UniKV and compare it with the cases without hash indexing and without KV separation. Compared to UniKV without hash indexing (UniKV-noHashIndex), UniKV further improves reads by 2.0×, while write performance keeps nearly the same. Besides, Compared to UniKV without KV separation (UniKV-noKVSep), UniKV further improves writes by 2.1× due to fewer compactions after KV separation, and improves reads by 1.2× as the SortedStore layer of UniKV becomes smaller after KV separation and almost all metadata of SSTables can be cached. We point out that in this experiment, the parallel optimization is not integrated in UniKV.

4.4 Impact of Different Configurations

Experiment 8 (Impact of `scanMergeLimit`). We study the impact of `scanMergeLimit` on the size-based merge strategy of UniKV (see §3.6.1). We randomly load 100 M KV pairs, run 50 M operations mixed with both scans and

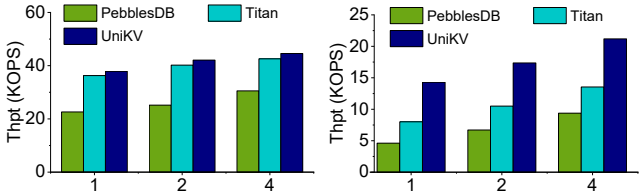
TABLE 5
Experiment 9 (Impact of `UnsortedLimit`).

UnsortedStore size	1 GB	4 GB	8 GB	16 GB
Write (KOPS)	22.8	37.5	40.6	43.1
Read (KOPS)	14.3	14.3	14.4	14.4
Index overhead (MB)	46	172	349	718



(a) Update throughput (b) Read throughput

Fig. 18. Experiment 10 (Impact of workload skewness).



(a) Load throughput (b) Read throughput

Fig. 19. Experiment 11 (Impact of concurrent KV operations).

writes (1:9), and finally issue 10 M reads. Table 4 shows the impact of different values of `scanMergeLimit`. As `scanMergeLimit` increases, the write throughput increases significantly, the read throughput keeps unchanged, but the scan throughput greatly decreases. The reason is that a larger `scanMergeLimit` implies fewer merge operations, i.e., fewer merge cost as also shown Table 4, but it also incurs more random reads on SSTables during scans, so it needs to access more SSTables on average for each scan, which is also shown in Table 4. Note that UniKV always reads KV pairs from the UnsortedStore by querying the hash index regardless of the number of SSTables.

Experiment 9 (Impact of `UnsortedLimit`). We study the impact of `UnsortedLimit` on the UnsortedStore of UniKV (see §3.3). We randomly load 100 M KV pairs and issue 10 M reads. Figure 5(a) shows the results by varying `UnsortedLimit` from 1 GB to 16 GB, while the partition size is fixed as 40 GB. As `UnsortedLimit` increases, the write throughput increases, while the read performance remains nearly the same. However, the memory overhead used for the hash index increases. Thus, we should limit the UnsortedStore size to balance between performance and memory overhead.

Experiment 10 (Impact of workload skewness). We study the impact of workload skewness on UniKV. We first randomly load 100 M KV pairs. We then issue 100 M updates and 10 M reads under a Zipf distribution. Figure 18 shows the results under different Zipfian constants (i.e., 0.6, 0.99, and 1.1). Compared with PebblesDB and Titan, UniKV consistently improves the read and write performance under different Zipfian constants, since UniKV adopts the two-layer architecture with hash indexing and KV separation with range partitioning to mitigate I/O amplification.

Experiment 11 (Impact of concurrent KV operations). We vary the number of threads in the YCSB client to study the impact of concurrent KV operations on UniKV. We randomly load 100 M KV pairs and issue 10 M reads. Figure 19 shows the results versus the number of threads in the YCSB client. Compared with PebblesDB and Titan, UniKV consistently improves the read and write performance with different number of threads due to several optimizations (e.g., hash indexing, KV separation, and dynamic partitioning). Thus, UniKV can scale well under concurrent KV operations.

TABLE 6
Experiment 12 (Impact of checkpointing frequency).

Checkpoint Interval	10 MemT	20 MemT	30 MemT	∞
Write (KOPS)	33.9	36.2	36.6	37.9
Read (KOPS)	14.5	14.6	14.6	14.4
Rebuild index (s)	9.8	13.4	16.7	18.1

TABLE 7
Experiment 13 (Memory usage) (in MB).

	LevelDB	PebblesDB	UniKV
Writes (100 M)	442	796	622
Reads (10 M)	318	748	142
Scans (1 M)	112	516	96

Experiment 12 (Impact of checkpointing frequency). We study the impact of checkpointing frequency on UniKV (see §3.8). We randomly load 100 M KV pairs and issue 10 M reads. We then remove the hash index and rebuild it. Table 6 shows the read/write throughput and the time of rebuilding the hash index under different checkpointing frequencies, in which we save the hash index when flushing every 10, 20, and 30 MemTables in a partition. We also include the case without checkpointing. A higher checkpointing frequency implies lower write performance as the hash index is saved more frequently during writes, yet the time to rebuild the hash index is smaller as fewer SSTables need to be scanned. Note that the read performance is not affected by the checkpointing frequency.

4.5 Overhead Analysis

Experiment 13 (Memory usage). We evaluate the memory usage of UniKV. UniKV builds a light-weight in-memory hash index for the UnsortedStore of each partition, and the number of buckets is set as 4 M and each bucket costs 8-Byte. Also, considering the utilization of buckets is about 80%, it means that the remaining 20% of index entries overflow. Thus, UniKV incurs extra (4 M×8-Byte+4 M×20%×8-Byte)×4=154 MB index memory in total when loading 100 M KV pairs that are split into 4 partitions. Recall that we set `block_cache_size` as 170 MB for all KV stores, while UniKV uses 8 MB by default. Thus, UniKV costs nearly the same memory as LevelDB but much less than PebblesDB, since PebblesDB builds extra `SSTable_File_Level` Bloom filters for each SSTable in memory to improve reads; it consumes 300 MB for 100 M KV pairs [36]. On the other hand, UniKV removes the Bloom filters of SSTables and does not need to read and cache them during read phase, thereby saving about 100 M×10 bits/key=120 MB memory used by Bloom filters for 100 M KV pairs.

We also compare the memory usage by experiments. We keep the setting of randomly loading 100 M KV pairs and issuing 10 M reads and 1 M scans. We record the largest memory usage during load, read and scan phase by monitoring the memory usage of KV store process in real time (via the `top` command), and treat it as the actual memory consumption of KV stores. Table 7 shows that

TABLE 8
Experiment 14 (Crash recovery time).

	MemTable (s)	Metadata (s)	Index (s)
LevelDB	0.92	0.05	0
UniKV	0.96	0.06	11.28

UniKV consumes 180 MB more memory than LevelDB for writes as it needs to build the hash index in memory; however, UniKV costs 176 MB and 16 MB less memory than LevelDB for reads and scans, respectively, as it keeps a smaller block cache and does not need to cache Bloom Filters in memory. Compared with PebblesDB, UniKV always costs less memory, especially for reads and scans.

Experiment 14 (Crash recovery). UniKV guarantees crash consistency (§3.8). We measure the recovery time after a crash when randomly loading 100 M KV pairs. Table 8 shows the results. UniKV requires more time for crash recovery compared with LevelDB. In particular, UniKV takes nearly the same time to recover the metadata from the manifest file and the KV pairs in MemTable from log file after a crash, but it costs another 11.28 seconds to recover the hash index from disk files.

5 Related Work

Hash indexing. Several KV stores are based on hash indexing. FlashStore [13] and SkimpyStash [14] maintain an in-memory hash index that stores a key signature and a pointer for referencing KV pairs on disk. Various optimizations on hash indexing include: page-oriented access in LLAMA [26], log-based storage and multi-version indexing in LogBase [41], and lossy and lossless hash indexing for parallel accesses in MICA [31]. In-memory hash indexing is also explored in data caching, such as concurrent in-memory hash tables for caching recently written KV pairs in FloDB [6]. However, hash indexing has poor scalability and cannot efficiently support scans.

LSM-tree. Persistent KV stores mainly build on the LSM-tree [33]. Many prior studies focus on improving the write performance of the LSM-tree design, including: fine-grained locking in HyperLevelDB [16], concurrency and parallelism optimization [17], [42], compaction overhead reduction [36], [38], optimized LSM-tree-like structures [45], hotness awareness [4], I/O scheduling for an LSM-tree-based KV store to reduce tail latencies [5], spatial data partitioning with LSM-tree-like batch I/Os to improve writes in EvenDB [19], as well as KV separation to reduce write amplification [9], [32], [35], etc. Some studies focus on improving the read performance of the LSM-tree, including differentiated Bloom filters for multiple levels in Monkey [12], elastic memory allocation for Bloom filters in ElasticBF [28], adaptive caching in AC-Key [44] to improve read performance, a succinct trie structure in SuRF [47], a space-efficient KV index to improve range queries in RemixDB [49], and differentiated management by different value sizes in DiffKV [27]. Some studies also consider mixed workloads, such as leveraging a compaction buffer on disks to minimize the cache invalidation on buffer cache caused by compaction in LSbM-tree [40]. In contrast, UniKV aims to improve the read and write performance simultaneously via unified indexing.

Hybrid indexing. Some KV stores also combine different indexing techniques. For example, SILT [30] uses hybrid index to balance memory efficiency and high performance for flash-based KV stores. Precisely, it combines hash indexing with a trie structure to chain three basic KV stores, using a partial-key Cuckoo hashing to index the on-flash LogStore and a trie structure to index the on-flash SortedStore. HiKV [46] adopts hybrid indexing, with a hash index persisted in NVM for fast search and a B+-tree kept in DRAM for fast updates and scans. SLM-DB [22] uses persistent memory to maintain a B+-tree index and adopts a single-level LSM-tree on disk. Data Calculator [21] and Continuums [20] unify major distinct data structures for self-designing KV stores. Different from them, UniKV adopts hybrid storage design by combining an in-memory hash indexing with the on-disk LSM-tree. Specifically, it leverages an in-memory hash index to index hot data and uses the LSM-tree to organize cold data on secondary storage. It is also applicable to commodity storage devices without relying on sophisticated hardware like NVM nor complicated computations.

6 Conclusion

We propose UniKV, which unifies hash indexing and the LSM-tree in a single system to enable high-performance and scalable KV storage. By leveraging data locality with a layered design and dynamic range partitioning, UniKV supports efficient reads and writes via hash indexing, as well as efficient scans and scalability via the LSM-tree. Experiments on our UniKV prototype justify its performance gains in various settings over state-of-the-art KV stores.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc of USENIX ATC*, 2008.

[2] Apache. HBase. https://hbase.apache.org/.

[3] 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.

[4] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proc. of USENIX ATC*, 2017.

[5] 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.

[6] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking memory in persistent key-value stores. In *Proc. of ACM EuroSys*, 2017.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *Proc. of USENIX OSDI*, 2010.

[8] K. Bogart and C. Stein. Hash collision probabilities. https://www.math.dartmouth.edu/archive/m19w03/public_html/Section6-5.pdf.

[9] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. of USENIX ATC*, 2018.

[10] 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 Trans. Comput. Syst.*, 26:4:1–4:2, June 2008.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[12] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proc. of ACM SIGMOD*, 2017.

[13] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *Proc. of VLDB Endow.*, 3:1414–1425, Sept. 2010.

[14] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proc. of ACM SIGMOD*, 2011.

[15] 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 ACM SOSP*, 2007.

[16] R. Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB/.

[17] Facebook. RocksDB. https://rocksdb.org.

[18] S. Ghemawat and J. Dean. LevelDB. https://leveldb.org.

[19] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovici, and R. Shahout. EvenDB: Optimizing key-value storage for spatial locality. In *Proc. of ACM EuroSys*, 2020.

[20] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, Ross, Jain, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *Proc. of CIDR*, 2019.

[21] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In *Proc. of ACM SIGMOD*, 2018.

[22] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi. SLM-DB: Single-level key-value store with persistent memory. In *Proc. of USENIX FAST*, 2019.

[23] L. kernel 2.6. Asynchronous non-blocking I/O (AIO). http://lse.sourceforge.net/io/aio.html.

[24] 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.

[25] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, Apr. 2010.

[26] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A cache/storage subsystem for modern hardware. *Proc. of VLDB Endow.*, 6:877–888, Aug. 2013.

[27] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui. Differentiated key-value storage management for balanced I/O performance. In *Proc. of USENIX ATC*, 2021.

[28] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proc. of USENIX ATC*, 2019.

[29] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In *Proc. of USENIX ATC*, 2018.

[30] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. of ACM SOSP*, 2011.

[31] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. of USENIX NSDI*, 2014.

[32] 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.

[33] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. *Acta Informatica*, 33(4):351–385, 1996.

[34] A. Partow. General Hash Functions Library. https://www.partow.net/programming/hashfunctions/.

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

[36] 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.

[37] J. Ren. C++ version of YCSB. https://github.com/basicthinker/YCSB-C.

[38] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proc. of ACM SIGMOD*, 2012.

[39] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proc. of USENIX FAST*, 2012.

[40] D. Teng, L. Guo, R. Lee, F. Chen, Y. Zhang, S. Ma, and X. Zhang. A low-cost disk solution enabling LSM-tree to achieve high performance for mixed read/write workloads. *ACM Trans. on Storage*, 14:26, Apr. 2018.

[41] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A scalable log-structured database system in the cloud. *Proc. of VLDB Endow.*, 5:1004–1015, June 2012.

[42] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proc. of ACM EuroSys*, 2014.

[43] Wikipedia. Bloom Filter. https://en.wikipedia.org/wiki/Bloom_filter.

[44] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *Proc. of USENIX ATC*, 2020.

[45] 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.

[46] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proc. of USENIX ATC*, 2017.

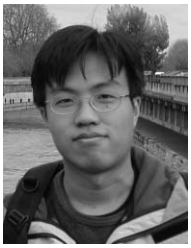
[47] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. of ACM SIGMOD*, 2018.

[48] Q. Zhang, Y. Li, P. P. C. 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.

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

Qiang Zhang is now a Ph.D. student in the school of Computer Science and Technology, University of Science and Technology of China. He received his Bachelor degree in computer science from Chongqing University. His research mainly focuses on storage systems, including key-value systems, as well as SSD RAID systems.

Yongkun Li is currently an associate professor in School of Computer Science and Technology, University of Science and Technology of China. He received the B.Eng. degree in Computer Science from USTC in 2008, and the Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong in 2012. After that, he worked as a postdoctoral fellow in Institute of Network Coding at The Chinese University of Hong Kong. His research mainly focuses on memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O optimization for virtualized systems and cloud computing.



Patrick P. C. Lee received the B.Eng. and the M.Phil. degrees from the Chinese University of Hong Kong in 2001 and 2003, respectively, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now a Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in storage systems, distributed systems, and networks.

Yinlong Xu received the B.S. degree in mathematics from Peking University in 1983, and the M.S. and Ph.D. degrees in computer science from University of Science and Technology of China (USTC) in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology at USTC, and is leading a research group in doing some networking and high performance computing research. His research interests include network coding, storage systems, combinatorial optimization, design and analysis of parallel algorithms, parallel programming tools, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences in 2006.