UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing

Qiang Zhang[1], Yongkun Li[1], Patrick P. C. Lee[2], Yinlong Xu[1], Qiu Cui[3], Liu Tang[3]
[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong* [3]*PingCAP*
zhgqiang@mail.ustc.edu.cn, {ykli, ylxu}@ustc.edu.cn, pclee@cse.cuhk.edu.hk, {cuiqiu, tl}@pingcap.com

***Abstract*—Persistent key-value (KV) stores are mainly designed based on the Log-Structured Merge-tree (LSM-tree), which 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 to tackle the issues caused by unifying the indexing techniques, so as to simultaneously improve the performance in reads, writes, and scans. Experiments show that UniKV significantly outperforms several state-of-the-art KV stores (e.g., LevelDB, RocksDB, HyperLevelDB, and PebblesDB) in overall throughput under read-write mixed workloads.**

I. 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 [1], [2], e-commerce [3], social networking [4], [5], data deduplication [6], [7], and photo stores [8]. 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 [9]; the read-write ratio of low-latency workloads at Yahoo! has shifted from 8:2 to 1:1 in recent years [10]; Baidu's cloud workload also shows a read-write ratio of 2.78:1 [11]. 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) [12] is a popular indexing design in modern persistent KV stores, including local KV stores (e.g., LevelDB [2] and RocksDB [5]) and large-scale distributed KV stores (e.g., BigTable [1], HBase [13], and Cassandra [14]). 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 multiple 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× [15]), and hence severely hurt the I/O performance and the endurance of storage systems backed by solid-state drives (SSDs) [16]. 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 §V). Examples include new LSM-tree organizations (e.g., a trie-like structure in LSM-trie [17] or a fragmented LSM-tree in PebblesDB [18]) and KV separation [15], [19]. 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 [17] trades the scan support for improved read and write performance; PebblesDB [18] relaxes the fully sorted requirement in each level and sacrifices read performance; KV separation [15], [19] 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 [19]. Thus, the full performance potentials of KV stores are still constrained by the inherent multi-level-based LSM-tree design.

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-off. For example, 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. On the other hand, 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 [9], [20], [21], 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

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00034

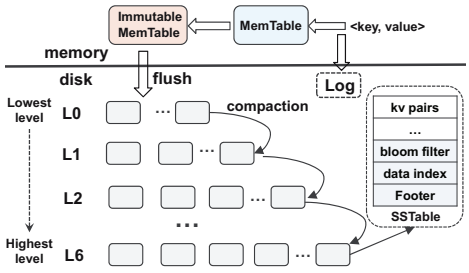
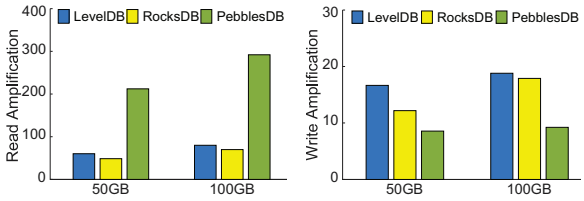


Fig. 1: Architecture of an LSM-tree-based KV store (LevelDB).



(a) Read amplification (b) Write amplification

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

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 three 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. Based on the unified indexing with multiple carefully designed techniques, UniKV simultaneously achieves high read, write, scan performance for large KV stores.

We implement a UniKV prototype atop LevelDB [2] and evaluate its performance via both micro-benchmarks and YCSB [22]. For micro-benchmarks, compared to LevelDB [2], RocksDB [5], HyperLevelDB [23], and PebblesDB [18], UniKV achieves up to 9.6×, 7.2×, 3.2×, and 1.7× load throughput, respectively. It also achieves significant throughput gains in both updates and reads. For the six core YCSB workloads except for scan-dominated workload E (in which UniKV achieves slightly better performance), UniKV achieves 4.2-7.3×, 2.5-4.7×, 3.2-4.8×, and 1.8-2.7× overall throughput, respectively.

II. Background and Motivation

A. Background: LSM-tree

To support large-scale storage, many KV stores are designed based on the LSM-tree (e.g., [2], [5], [15], [18], [19]). Figure 1 depicts the design of an LSM-tree-based KV store, using LevelDB as an example. Specifically, it keeps two 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 [24] 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*.

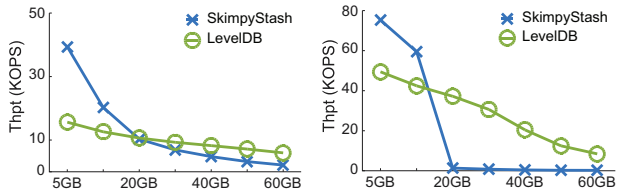
Limitations. The main limitations of LSM-tree-based KV stores are the large write and read amplifications. Prior studies show that the write amplification factor of LevelDB could reach up to 50× [15]. For example, when merging a SSTable from L_i to L_{i+1}, LevelDB may read up to 10 SSTables from L_{i+1} 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× [15].

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

B. Motivation

In-memory hash indexing. Given the high write and read amplifications in LSM-tree-based KV stores, we can leverage



(a) Write throughput (b) Read throughput

Fig. 3: Performance comparison of hash-index-based and LSM-tree-based KV stores (SkimpyStash [7] and LevelDB [2]).

in-memory hash indexing for efficient KV pair lookups. Hash indexing has been widely studied in KV stores [6], [7], [25]. 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. However, 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 operations, 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. To illustrate, we compare the read and write performance of SkimpyStash [7], an open-source KV store using hash indexing, and LeveLDB, using the default settings and evaluation setup in §IV. 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).

On the other hand, the LSM-tree supports large data stores without extra in-memory indexing overhead and outperforms hash indexing in large KV stores. However, it suffers from serious read and write amplifications (see §II-A). Some efforts are made to optimize the LSM-tree, but unavoidably make a trade-off between read and write performance. For example, PebblesDB [18] improves the write performance by relaxing the fully sorted feature in each level of the LSM-tree, but sacrifices the read performance (see Figure 2).

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 [9]–[11] (see §I), but also have high data access skewness [9], [20], [21], 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

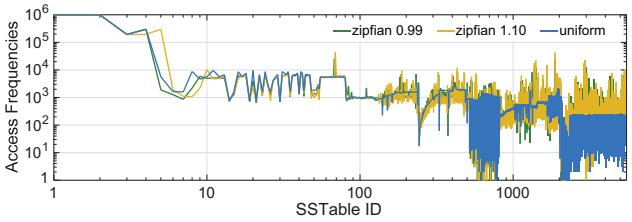


Fig. 4: Access frequency of SSTables in LevelDB.

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. By unifying hash indexing and the LSM-tree in a single system with dynamic range partitioning, we can achieve high performance in reads, writes, and scans in large KV stores.

III. UniKV Design

A. Architectural Overview

UniKV adopts a two-layer architecture as shown in Figure 5. The first layer is called the *UnsortedStore*, which keeps the SSTables[1] that are recently flushed from memory in an unsorted manner. 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 §II-B) that the 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 directly with in-memory hash indexing for fast reads and writes. Meanwhile, we keep the remaining 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, respectively. Also, UniKV designs light-weight two-level hash indexing to balance memory usage and hash collisions (see §III-B).
- **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 §III-C).

[1]As in LevelDB, we organize data in memory as Memtables, and refer to them as SSTables when they are flushed from memory to disk.

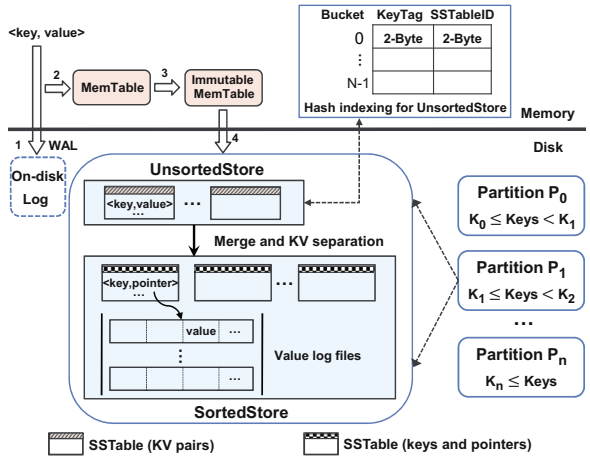


Fig. 5: UniKV architecture.

- **Dynamic range partitioning.** To achieve high read and write performance in large KV stores, UniKV proposes a range partitioning scheme that dynamically splits KV pairs into multiple partitions that are independently managed according to the key ranges, so as to expand a KV store in a scale-out manner (see §III-D).
- **Scan optimizations and consistency.** UniKV optimizes its implementation for high scan performance (see §III-F). It also supports crash recovery for consistency (see §III-G).

B. Differentiated Indexing

Data management. Recall from §III-A that UniKV adopts a layered architecture with two index structures. To elaborate, 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 is based on the current LSM-tree design, in which each SSTable has a fixed size limit and contains a certain amount of KV pairs.

However, we *remove* Bloom filters from all SSTables to save memory space and reduce computation overhead. The rationale is as follows. 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 may contain the KV pair via binary search which compares the key with boundary keys of SSTables that are kept in memory, as all keys in the SortedStore are fully sorted across SSTables. 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 inspections to SSTables and incur 2.3 I/Os on average for a key lookup due to false positives of Bloom filters as well as multi-level searching [26].

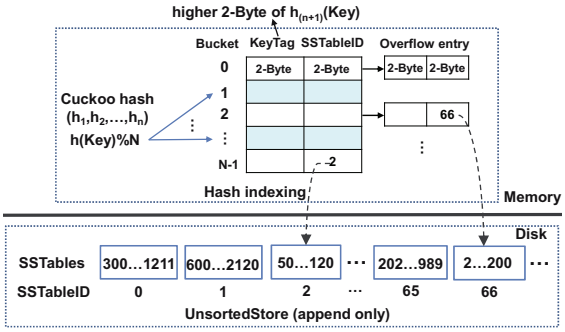


Fig. 6: Structure of the UnsortedStore and in-memory hash indexing.

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 flushed to the UnsortedStore on disk in an append-only way by a background process.

Hash indexing. SSTables in the UnsortedStore are written in an append-only way and indexed with an in-memory hash index. 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 [27], 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 [28]), 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 now 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. Note that for very small KV pairs, hash indexing may incur large memory overhead. One solution is to differentiate the management for KV pairs of different sizes; for example, we use the conventional LSM-tree to manage small KV pairs and use UniKV to manage large KV pairs. We pose this optimization as future work.

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% [29]), as also shown in our experiments. Even if hash collisions happen, we can still resolve them by comparing the keys stored on disk.

C. Partial KV separation

Recall that KV pairs in the UnsortedStore are indexed with an in-memory hash index, which incurs extra memory usage. Also, the key ranges of SSTables in the UnsortedStore are also overlapped with each other due to the append-only write policy without sorting, so every SSTable in the UnsortedStore needs to be examined when performing a scan operation. To limit the memory overhead and guarantee scan performance, UniKV limits the size of the UnsortedStore. When the size reaches a predefined threshold UnsortedLimit, UniKV triggers a merge operation to merge KV pairs from the UnsortedStore into the SortedStore. Note that the parameter UnsortedLimit is configurable according to the available memory.

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 data in the SortedStore so as to reduce the merge overhead.

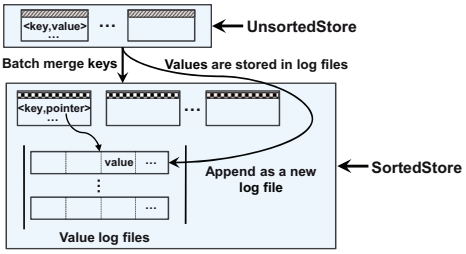


Fig. 7: Partial KV separation.

Figure 7 depicts the partial KV separation design. When merging KV pairs from UnsortedStore to 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 of all, 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 the compaction process, 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., [15]) 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 just 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.

D. 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

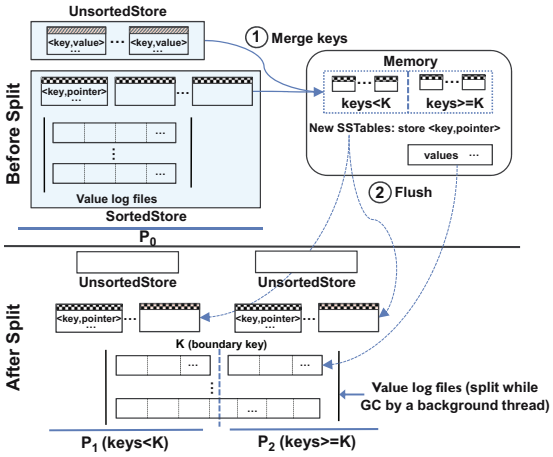


Fig. 8: Dynamic range partitioning.

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 the two new 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 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. Note that UniKV releases the locks and resumes to handle write requests after splitting 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 its own 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 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.

E. 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 $e^{-M \cdot \ln(2)^2}$ [30], and it can be simply represented as 0.6185^M [26]. 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.

F. Scan Optimization

UniKV aims to support efficient scans as in LSM-tree-based KV stores. Scans are efficiently supported by the LSM-tree as the KV pairs are stored in a sorted manner. However, for UniKV, keys and values are stored separately for KV pairs in the SortedStore. This makes scan operations issue random reads of values. Also, all SSTables in the UnsortedStore of a partition are directly flushed from memory in an append-only manner, so they may overlap with each other. Thus, scans may read every SSTable while performing `seek()` or `next()` operations, thereby incurring additional random reads.

To optimize scans, UniKV first locates the corresponding partitions quickly by querying the boundary keys of partitions according to the scanned key range, which can significantly reduce the amount of data to be scanned. Also, UniKV employs multiple strategies to optimize scans. For the UnsortedStore, to avoid many random reads caused by checking every SSTable during scans, UniKV proposes a size-based merge strategy. The main idea is to merge all SSTables in the UnsortedStore into a big one which keeps all KV pairs fully sorted with a background thread during scans when the number of SSTables in the UnsortedStore exceeds the predefined threshold `scanMergeLimit`. Note that even though this operation causes extra I/O overhead, the overhead is small if the size of UnsortedStore is limited. This improves scans significantly due to efficient sequential reads, especially under a large amount of scans.

For the SortedStore, UniKV leverages I/O parallelism of SSDs by fetching values from log files concurrently with multiple threads during scans. Also, UniKV leverages the `read-ahead` mechanism to prefetch values into the page cache. The read-ahead mechanism works as follows. Initially, UniKV obtains keys and pointers within the scan range from SSTables in the SortedStore one by one. It then issues a read-ahead request to the log files starting with the value of the first key (via `posix_fadvise`). Finally, it reads the values according to the pointers and returns the KV pairs. On the other hand, we point out that UniKV does not perform in-memory merging and sorting with the scan results from the UnsortedStore and SortedStore. This is because a scan operation is implemented with the following three steps: (i) `seek()`, which locates the beginning key from each of the SSTables that need to check in both the UnsortedStore and the SortedStore, and return the KV pair of the beginning key once it is found; (ii) `next()`, which finds the next smallest key that is bigger than the last returned one from each of the SSTables that need to check, and return the minimum key with its value; (iii) repeat step (ii) until the number of returned KV pairs equals the scan length. With the above optimizations, our experiments show that UniKV achieves similar scan performance as LevelDB, which always keeps keys fully sorted in each level.

G. Crash Consistency

Crashes may occur during writes. UniKV addresses 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) for crash recovery. WAL is also used in LevelDB. 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 half of the `UnsortedLimit` SSTables from memory to the UnsortedStore. 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.

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).

We have implemented the above mechanisms in UniKV to provide crash consistency guarantees, and our experiments show that UniKV can recover inserted data, hash indexing, and GC-related state correctly and the recovery overhead is small.

H. Implementation Issues

We implement a UniKV prototype on Linux based on LevelDB v1.20 [2] in C++, by adding or modifying around 8K lines of code. Most changes are to introduce hash indexing, dynamic range partitioning, partial KV separation, and GC operations. Since the UnsortedStore and the SortedStore build on SSTables, UniKV can leverage the mature, well-tested codes for SSTable management in LevelDB.

For scans, UniKV leverages multi-threading to fetch values concurrently. However, the number of threads is limited by the

memory space of a process, and using too many threads can trigger frequent context switches. UniKV maintains a thread pool of 32 threads and allocates threads for fetching values from the pool in parallel. During scans, UniKV inserts a fixed number of value addresses to the worker queue, and then wakes up all the sleeping threads to read values from log files in parallel.

IV. Evaluation

We present evaluation results on UniKV. We compare via testbed experiments UniKV with several state-of-the-art KV stores: LevelDB v1.20 [2], RocksDB v6.0 [5], HyperLevelDB [23], and PebblesDB [18]. For fair comparisons, we enable write-ahead logging to guarantee crash consistency for all KV stores. We also issue writes in asynchronous mode, in which all written KV pairs are first buffered in the page cache of kernel and flushed when the cache is full. We further disable compression of all systems to remove the impact of compression overhead. We address the following questions:

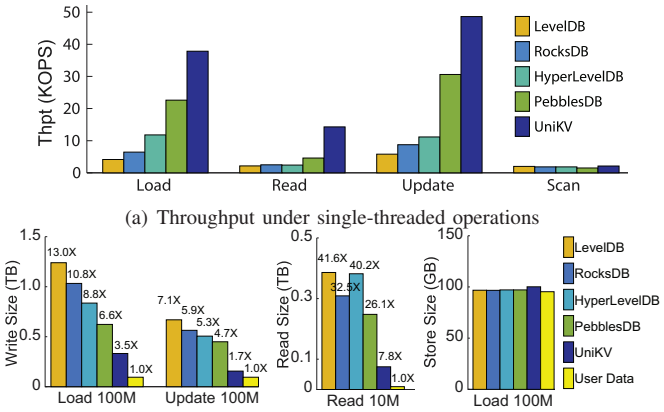
- What is the load/read/update/scan performance of UniKV under single-threaded environment and the overall performance under read-write mixed workloads? (Experiments 1-2)
- What is the performance of UniKV under six core workloads of YCSB and large KV stores? (Experiments 3-4)
- What is the performance impact of different configurations (e.g., KV pair size, UnsortedStore size, partition size, multi-threading on scans, and direct I/O)? (Experiments 5-9)
- What are the memory usage of hash indexing and the crash recovery overhead? (Experiments 10-11)

A. 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 the ext4 file system.

The performance of KV stores is mainly affected by four parameters: (i) `memtable_size`, (ii) `bloom_bits`, (iii) `open_files` and (iv) `block_cache_size`. 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, so that the total number of MemTables is the same as in UniKV. For other parameters of 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



(a) Throughput under single-threaded operations

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

Fig. 9: Experiment 1 (Micro-benchmark performance).

	Flush	Compaction	GC	Partition Split
LevelDB	204.4	2229.5	–	–
RocksDB	204.4	1726.2	–	–
HyperLevelDB	204.4	1402.2	–	–
PebblesDB	204.4	1026.3	–	–
UniKV	204.4	188.4	288.1	35.2

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

4 GB to limit the memory overhead of the hash index. For GC operations, UniKV uses a single GC thread; for scans, UniKV maintains a background thread pool with 32 threads (the impact of multi-threading on scans is evaluated in Experiment 8). Finally, we use YCSB [22], [31] to generate various types of workloads. By default, we focus on 1 KB KV pairs with 24-Byte keys (the impact of KV pair size is studied in Experiment 5) and issue the requests based on the Zipf distribution with a Zipfian constant of 0.99 (the default in YCSB).

B. Performance Comparisons

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

(i) Throughput. Figure 9(a) shows the throughput. Compared to other KV stores, UniKV achieves 1.7-9.6× load throughput, 3.1-6.6× read throughput, and 1.6-8.2× update throughput. Note that UniKV is now implemented on LevelDB, yet it still outperforms other advanced KV stores with specific optimization features. For scans, UniKV achieves nearly the same throughput as LevelDB and 1.3-1.4× throughput compared to RocksDB, HyperLevelDB and PebblesDB. The reason is that UniKV adopts optimizations, such as multi-threading for fetching values concurrently and the read-ahead mechanism for prefetching values into cache (see §III-F).

(ii) Write amplification. We evaluate the total write size. We randomly load 100 M KV pairs, then update the loaded 100 M

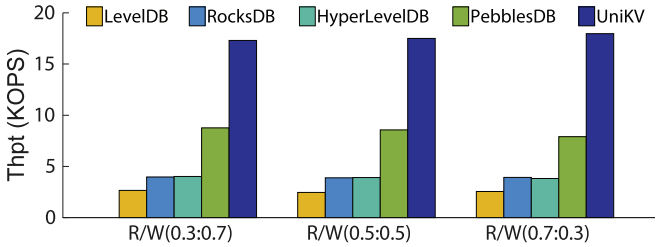


Fig. 10: Experiment 2 (Performance under mixed workloads).

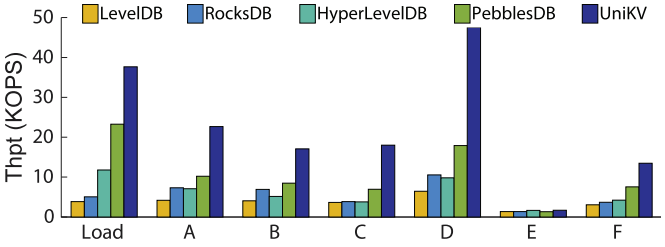


Fig. 11: Experiment 3 (YCSB performance).

KV pairs to show the write amplification. Figure 9(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. In particular, it reduces 46.7-73.2% of write size in the load phase, and reduces 65.2-76.6% of write size in the update phase, compared to other KV stores.

(iii) Read amplification. We evaluate the total read size when operate 10 M read requests on the loaded KV store. Figure 9(c) shows the total read size for all KV stores. UniKV significantly reduces the total read size, which decides the read performance. The read amplification is only about 7.8×. Thus, UniKV reduces 69.7-80.6% of read size in the read phase compared to other KV stores.

(iv) Space amplification. Figure 9(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 incurs 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 I (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 41.9-70.6% of I/O cost in total, mainly because 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 read and write with different read-write ratios (30:70, 50:50, and 70:30). Figure 10 shows the overall throughput under different read-write mixed workloads. UniKV outperforms all other KV stores. Specifically, the overall throughput of UniKV is 6.5-7.1×, 4.4-4.6×, 4.3-4.7× and 2.0-2.3×, compared to LevelDB, RocksDB, HyperlevelDB, and PebblesDB, respectively. The reason is that UniKV maintains a two-layered storage architecture with in-memory hash indexing for hot data to improve read performance, and adopts partial KV separation and dynamic range partitioning to reduce merge overhead and support good scalability to improve write performance. Thus, it improves both read and write performance simultaneously.

C. Yahoo Cloud Serving Benchmark

Experiment 3 (YCSB performance). Now we evaluate the performance under YCSB [22], which is an industry standard to evaluate KV stores and contains a standard set of six core workloads (Workloads A-F). Each workload represents a specific real-world application. Specifically, Workloads A and B are read-write mixed with 50% and 95% reads, respectively. Workload C is a read-only workload with 100% reads. Workload D also contains 95% reads, but the reads query the latest values. Workload E is a scan-dominated workload which contains 95% scans and 5% writes. Finally, Workload F contains 50% reads and 50% read-modify-writes which require a `get()` before every `insert()` operation. We first randomly 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()`.

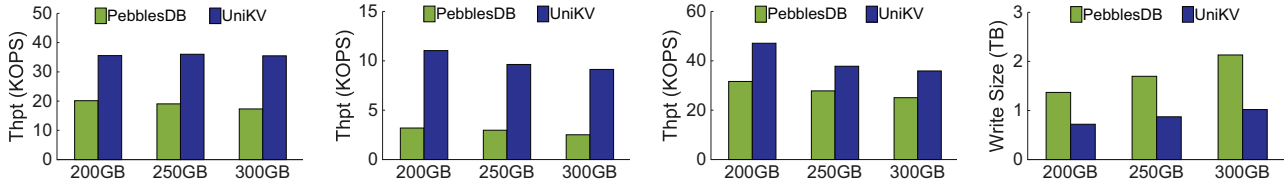
Figure 11 shows the results. UniKV always outperforms other KV stores under both read-dominated and write-dominated workloads. Compared to other KV stores, the throughput of UniKV is 2.2-5.4× under Workload A, 2.1-4.2× under Workload B, 2.6-4.4× under Workload C, 2.7-7.3× under Workload D, and 1.8-4.4× under Workload F. For the scan-dominated Workload E, UniKV achieves slightly better performance, as its scan throughput is 1.1-1.3× compared to other KV stores.

D. Performance on Large KV Stores

Experiment 4 (Performance on large KV stores). We evaluate the performance on large KV stores to show the scalability of UniKV. As shown in Experiments 1-3, PebblesDB has the best performance among existing KV stores, so we focus on PebblesDB in the following experiments. We still show the throughput of load, read, and update, as well as the total write size during load using the same setting as before, except now we vary the KV store size from 200 GB to 300 GB. Figure 12 shows that UniKV consistently outperforms PebblesDB in all aspects. It achieves 1.8-2.0× write throughput, 3.2-3.6× read throughput, and 1.4-1.5× update throughput. Also, the performance gain of UniKV increases for larger KV stores. For the write size, we show the total write size when loading different KV stores. Compared to PebblesDB, UniKV reduces 47.4-52.1% of write size during the load phase.

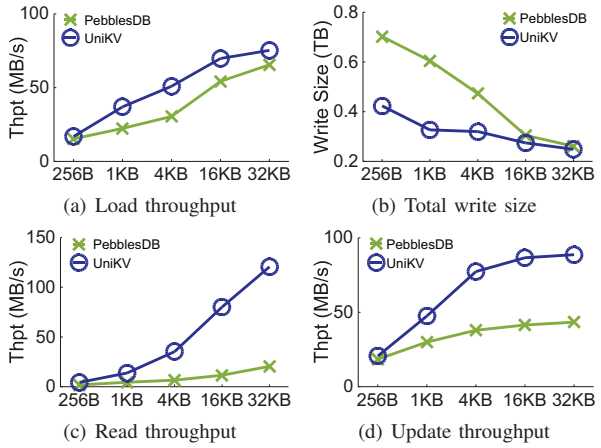
E. Performance Impact and Overhead

Experiment 5 (Impact of KV pair size). We study the impact of KV pair size varied from 256 B to 32 KB, and keep other parameter settings as before. Figure 13 shows the



(a) Load different KV stores (b) Read on different KV stores (c) Update on different KV stores (d) Total write size

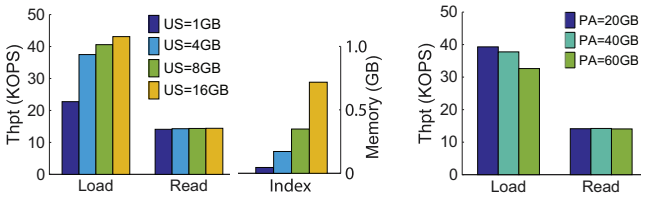
Fig. 12: Experiment 4 (Performance on large KV stores).



(a) Load throughput (b) Total write size

(c) Read throughput (d) Update throughput

Fig. 13: Experiment 5 (Impact of KV pair size).



(a) Impact of the US size (b) Impact of partition size

Fig. 14: Experiment 6 (Impact of the UnsortedStore (US) size) and Experiment 7 (Impact of partition size).

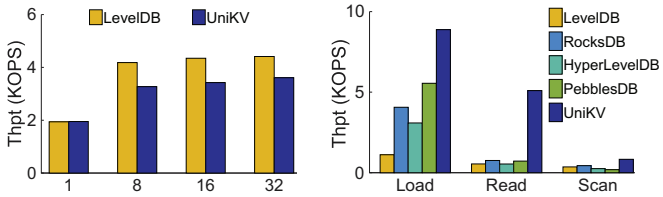
throughput and total write size of randomly loading 100 GB KV pairs, as well as the throughput of reading 10 GB and updating 100 GB KV pairs; note that the figure reports the throughput of KV stores in terms of MB/s instead of KOPS to better illustrate the performance trend with respect to the amount of data being accessed. As the KV pair size increases, both UniKV and PebblesDB have higher throughput due to the efficient sequential I/Os. For load, read and update performance, UniKV always outperforms PebblesDB. When KV pairs become larger, the improvement of UniKV decreases for the throughput and write size of loading a KV store, and the improvement increases for the throughput of reads and updates. The reason is that PebblesDB maintains more SSTables in the first level as KV pair size increases. This reduces the compaction overhead, but causes reads to check these SSTables one by one, leading to degraded read performance. Also, more SSTables in the first level will be merged to next level in the update phase, which leads to degraded update performance. In contrast, UniKV always maintains fixed UnsortedStore and partition sizes. Thus, all the throughput of load, read and update in UniKV grows steadily as the KV pair size increases.

Experiment 6 (Impact of the UnsortedStore size). We study the impact of UnsortedStore size on UniKV. We randomly load 100 M KV pairs and issue 10 M read operations. Figure 14(a) shows the results by varying the UnsortedStore size from 1 GB to 16 GB when fixing the partition size as 40 GB. As the UnsortedStore size increases, the load throughput increases, while the read performance remains nearly the same. However, the memory overhead used for the hash index for UnsortedStore increases. Thus, the UnsortedStore size should be limited to trade between performance and memory overhead.

Experiment 7 (Impact of partition size). We study the impact of partition size on UniKV. We again randomly load 100 M KV pairs and issue 10 M reads. Figure 14(b) shows the results by varying the partition size from 20 GB to 60 GB, while fixing the UnsortedStore size as 4 GB. The partition size only has a small impact on write performance and almost no impact on read performance. The reason is that GC is operated within each partition independently. Thus, the smaller the partition, the more efficient the GC operations for finer granularity of GC. However, the partition size influences the memory cost as UniKV needs to allocate a MemTable for each partition. Thus, a smaller partition size may incur more memory usage, so the partition size should be limited.

Experiment 8 (Impact of multi-threading on scans). We study the impact of multi-threading on scans. As shown in Experiments 1, LevelDB has the best scan performance among existing KV stores, so we focus on comparing UniKV with LevelDB. We first randomly load 100 M KV pairs, and issue 1 M scan operations with each scan involving 50 `next()`. Figure 15(a) shows the results by varying the number of threads for scan requests from 1 to 32. As the number of threads increases, both UniKV and LevelDB have higher throughput. However, UniKV has slightly lower throughput than LevelDB when there are multiple scan threads (e.g., by 10-20% for 8 to 32 threads). The reason is UniKV maintains a thread pool that fetches values from log files concurrently during scans, which incur more CPU and I/O usage. The scan efficiency degrades as the CPU and I/O usage is more demanding under multi-threaded requests.

Experiment 9 (Performance under direct I/O). We study the impact of direct I/O on performance. Since all KV stores except RocksDB do not support direct I/O, we modify their source code to include the `O_DIRECT` attribute in the `open()` calls and turn off write-caching of disk to enable direct I/O. We first randomly load 100 M KV pairs, and issue 10 M reads and 1 M scans under direct I/O. Figure 15(b) shows the results.



(a) Multi-threaded scan (b) Performance under direct I/O

Fig. 15: Experiment 8 (Impact of multi-threading on scans) and Experiment 9 (Performance under direct I/O).

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

TABLE II: Experiment 10 (Memory usage) (in MB).

The throughput of writes, reads, and scans for all KV stores drops significantly under direct I/O since all operations involve disk I/O access. Nevertheless, UniKV outperforms other KV stores under direct I/O. It achieves 1.6-8.0× load throughput, 6.7-9.3× read throughput, 1.9-4.4× scan throughput compared to other KV stores. The improvements of UniKV for reads and scans increase under direct I/O, since the multi-level access in other KV stores reads more SSTables through disk and UniKV can take full advantage of the I/O parallelism of SSDs.

Experiment 10 (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 [18]. 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 biggest 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. As shown in Table II, 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 to PebblesDB, UniKV always costs less memory, especially for reads and scans.

	MemTable (s)	Metadata (s)	Index (s)
LevelDB	0.92	0.05	0
UniKV	0.96	0.06	3.18

TABLE III: Experiment 11 (Crash recovery time).

Experiment 11 (Crash recovery). Note that UniKV guarantees crash consistency by implementing the mechanism in §III-G, which can recover inserted data, hash indexing, and GC-related state correctly. We measure the recovery time after a crash when randomly loading 100 M KV pairs. Table III 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 3.18 seconds to recover the hash index from disk files. Note that the larger UnsortedStore also implies the larger hash indexing, so the recovery time is also longer.

V. Related Work

Hash indexing. Several KV stores are based on hash indexing. The baseline design [6], [7], [25] maintains an in-memory hash index that stores a key signature and a pointer for referencing KV pairs that are stored on disk. Various optimizations on hash indexing include: page-oriented access in LLAMA [32], log-based storage and multi-version indexing in LogBase [33], and lossy and lossless hash indexing for parallel accesses in MICA [34]. 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 [35]. However, hash indexing has poor scalability and cannot efficiently support scans.

LSM-tree. Many persistent KV stores build on the LSM-tree [12] to address scans and scalability issues. Prior studies focus on improving the write performance of the LSM-tree design, including: fine-grained locking in HyperLevelDB [23], concurrency and parallelism optimization [5], [36], [37], compaction overhead reduction [10], [38], optimized LSM-tree-like structures [17], [18], hotness awareness [20], and KV separation [15], [19], [39], etc. Some studies focus on improving the read performance of the LSM-tree, including differentiated Bloom filters for multiple levels in Monkey [30], elastic memory allocation for Bloom filters based on data locality in ElasticBF [26], and a succinct trie structure in SuRF [40].

Hybrid indexing. Some KV stores combine different indexing techniques. For example, SILT [41] combines log-structured, hash-based, and sorted KV stores and uses compact hash tables to reduce per-key memory usage. NV-tree [42], HiKV [43], and NoveLSM [44] design new index structures for non-volatile memory (NVM). Data Calculator [45] and Continuums [46] focus on unifying major distinct data structures for enabling self-designing KV stores.

UniKV also takes a hybrid indexing approach. Unlike the above hybrid indexing techniques, UniKV aims to simultaneously achieve high performance in reads, writes, and scans, while supporting scalability. It is also deployable in commodity storage devices (e.g., SSDs) without relying on sophisticated hardware like NVM nor complicated computations.

VI. 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. Testbed experiments on our UniKV prototype justify its performance gains in various settings over state-of-the-art KV stores.

Acknowledgment

This work is supported in part by the National Key Research and Development Program of China under Grant No.2018YFB1800203, NSFC 61772484, 61772486, Youth Innovation Promotion Association CAS, and PingCAP. Yongkun Li is USTC Tang Scholar, and he is the corresponding author.

References

[1] 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.*, vol. 26, pp. 4:1–4:2, Jun. 2008.

[2] S. Ghemawat and J. Dean, "LevelDB," https://leveldb.org.

[3] 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. ACM SOSP'07*, Washington, USA, Oct. 2007, pp. 205–220.

[4] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proc. USENIX FAST'12*, San Jose, CA, Feb. 2012, pp. 18–18.

[5] Facebook, "RocksDB," https://rocksdb.org.

[6] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, pp. 1414–1425, Sep. 2010.

[7] ——, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proc. ACM SIGMOD'11*, Athens, Greece, Jun. 2011, pp. 25–36.

[8] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage," in *Proc. USENIX OSDI'10*, Vancouver, Canada, Oct. 2010, pp. 47–60.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS'12*, London, England, UK, Jun. 2012, pp. 53–64.

[10] R. Sears and R. Ramakrishnan, "blsm: A general purpose log structured merge tree," in *Proc. ACM SIGMOD'12*, Scottsdale, Arizona, USA, May 2012, pp. 217–228.

[11] 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. MSST'15*, Santa Clara, CA, USA, May 2015, pp. 1–14.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[13] Apache, "HBase," https://hbase.apache.org/.

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

[15] 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. USENIX FAST'16*, Santa Clara, CA, Feb. 2016, pp. 133–148.

[16] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *Proc. USENIX ATC'08*, Boston, Massachusetts, Jun. 2008, pp. 57–70.

[17] 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. USENIX ATC'15*, Santa Clara, CA, Jul. 2015, pp. 71–82.

[18] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proc. ACM SOSP'17*, Shanghai, China, Oct. 2017, pp. 497–514.

[19] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in KV storage via hashing," in *Proc. USENIX ATC'18*, Boston, MA, USA, Jul. 2018, pp. 1007–1019.

[20] 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. USENIX ATC'17*, Santa Clara, CA, USA, Jul. 2017, pp. 363–375.

[21] 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. USENIX ATC'18*, Boston, MA, USA, Jul. 2018, pp. 981–992.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. ACM SoCC'10*, Indianapolis, Indiana, USA, Jun. 2010, pp. 143–154.

[23] R. Escriva, "HyperLevelDB," https://github.com/rescrv/HyperLevelDB/.

[24] Wikipedia, "Bloom Filter," https://en.wikipedia.org/wiki/Bloom_filter.

[25] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proc. ACM SOSP'09*, Big Sky, Montana, USA, Oct. 2009, pp. 1–14.

[26] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores," in *Proc. USENIX ATC'19*, Renton, WA, Jul. 2019, pp. 739–752.

[27] Wikipedia, "Cuckoo hash," https://en.wikipedia.org/wiki/Cuckoo_hashing.

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

[29] Dartmouth, "Hash Collision Probabilities," https://www.math.dartmouth.edu/archive/m19w03/public_html/Section6-5.pdf.

[30] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM SIGMOD'17*, Chicago, USA, May 2017, pp. 79–94.

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

[32] J. Levandoski, D. Lomet, and S. Sengupta, "Llama: A cache/storage subsystem for modern hardware," *Proc. VLDB Endow.*, vol. 6, pp. 877–888, Aug. 2013.

[33] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "Logbase: A scalable log-structured database system in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 1004–1015, Jun. 2012.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proc. USENIX NSDI'14*, Seattle, WA, Apr. 2014, pp. 429–444.

[35] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "Flodb: Unlocking memory in persistent key-value stores," in *Proc. ACM EuroSys'17*, Belgrade, Serbia, Apr. 2017, pp. 80–94.

[36] 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. ACM EuroSys'14*, Amsterdam, The Netherlands, Apr. 2014, pp. 16:1–16:14.

[37] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proc. ACM EuroSys'15*, Bordeaux, France, Apr. 2015, pp. 32:1–32:14.

[38] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proc. USENIX FAST'13*, San Jose, CA, Feb. 2013, pp. 17–30.

[39] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," in *Proc. USENIX FAST'16*, Santa Clara, CA, Feb. 2016, pp. 167–180.

[40] 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. ACM SIGMOD'18*, Houston, USA, Jun. 2018, pp. 323–336.

[41] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proc. ACM SOSP'11*, Cascais, Portugal, Oct. 2011, pp. 1–13.

[42] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proc. USENIX FAST'15*, Santa Clara, CA, Feb. 2015, pp. 167–181.

[43] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *Proc. USENIX ATC'17*, Santa Clara, CA, USA, Jul. 2017, pp. 349–362.

[44] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *Proc. USENIX ATC'18*, Boston, MA, USA, Jul. 2018, pp. 993–1005.

[45] 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. ACM SIGMOD'18*, Houston, USA, Jun. 2018, pp. 535–550.

[46] 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. CIDR'19*, Asilomar, USA, Jan. 2019.