TeksDB: Weaving Data Structures for a High-Performance Key-Value Store

YOUIL HAN, Chungbuk Nat'l University, South Korea
BRYAN S. KIM, Seoul Nat'l University, South Korea
JESEONG YEON, Chungbuk Nat'l University, South Korea
SUNGJIN LEE, DGIST, South Korea
EUNJI LEE, Soongsil University, South Korea

In this paper, we examine the design tradeoffs of existing in-memory data structures of a state-of-the-art key-value store. We observe that no data structures provide both fast point-accesses and consistent ranged-retrievals, and naïve amalgamations of existing structures fail to get the best of both worlds. Furthermore, our experiments reveal a *performance anomaly* when increasing the memory size: as more key-value pairs are maintained in memory, the shortcomings of the data structures exacerbate. To address the above problems, we present TeksDB[1], a fast and consistent key-value store with a novel in-memory data structure, which efficiently handles both point- and ranged- accesses at a modest increase in memory footprint. Our evaluation demonstrates that TeksDB outperforms RocksDB by 3.6×, 9×, and 4.5× for get, scan, and range_query, respectively. The effectiveness of TeksDB extends to real-world workloads, achieving up to 3.3× speedup for YCSB.

CCS Concepts: • **Information systems** → **Data scans**; **Point lookups**; **Unidimensional range search**; **Key-value stores**; *DBMS engine architectures*; *Main memory engines*;

Additional Key Words and Phrases: Key-value Stores; NoSQL Systems; Data Structures; Scalability

ACM Reference Format:
Youil Han, Bryan S. Kim, Jeseong Yeon, Sungjin Lee, and Eunji Lee. 2019. TeksDB: Weaving Data Structures for a High-Performance Key-Value Store. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 8 (March 2019), 23 pages. https://doi.org/10.1145/3311079

1 INTRODUCTION

Key-value stores (KVS) are now an integral part of modern data-intensive systems [3, 4, 13, 17, 18, 20, 26, 28, 34] thanks to its simplicity, scalability, and efficiency over traditional database systems. The state-of-the-art KVS typically uses a log-structured merge tree (LSM) [30] that is organized as multiple increasing-in-size levels. By absorbing large batches of data updates in the smaller lower-levels and writing them sorted to the larger higher-levels, an LSM-based KVS offers high throughput for small unstructured data such as web pages [2, 9].

[1]The root word *teks-* means to weave and to make fabric. Derivatives include *text*, *architect*, and *technology*.

Authors' addresses: Youil Han, Chungbuk Nat'l University, 1 Chungdae-ro, Cheongju, South Korea, yuil@oslab.cbnu.ac.kr; Bryan S. Kim, Seoul Nat'l University, 1 Gwanak-ro, Seoul, South Korea, bryansjkim@snu.ac.kr; Jeseong Yeon, Chungbuk Nat'l University, 1 Chungdae-ro, Cheongju, South Korea, jsyeon@oslab.cbnu.ac.kr; Sungjin Lee, DGIST, 333 Techno jungang-daero, Daegu, South Korea, sungjin.lee@dgist.ac.kr; Eunji Lee, Soongsil University, 369 Sangdo-ro, Seoul, South Korea, ejlee@soongsil.ac.kr.

2476-1249/2019/3-ART8 $15.00
https://doi.org/10.1145/3311079

The popularity of KVS extends beyond the write-intensive data applications. Databases such as MySQL employ KVS (in this case, RocksDB [14]) as their backend storage instead of conventional database engines because KVS is less affected by data fragmentation [7, 16, 25, 32]. OLAP and big data analytics applications often use KVS as the backend store to make use of the efficient range_query operations of LSM-based KVS [31, 32]. While KVS is originally intended for write-oriented workloads, these cases require the handling of reads to be just as efficient as writes, as reflected in Yahoo's YCSB benchmark suite [10]. Furthermore, the increasing use of KVS as backend stores in distributed systems diversifies the workload for each local KVS, making it necessary to design a highly-concurrent data store that simultaneously processes mixed operations [9, 15, 19, 29, 39]. For instance, when KVS is used as the backend store for distributed object store, it will have its data accessed from virtual block devices or distributed file systems [22, 23].

The underlying technology trends also put immense pressure on the design of KVS. Nowadays, many core servers with hundred GBs of memory are common in production environments, and this enhanced density of hardware technologies (both in terms of the number of cores and memory capacity) leads to more data items residing in memory. This phenomenon places greater emphasis on the efficiency of the in-memory data structure in KVS, as more data accesses are serviced from its memory component, which is often called the *memtable*.

The current memtable structure of KVS, however, generally assumes a limited memory capacity that only serves as a temporary write buffer, and the use of single-threaded writes [18] or unscalable data structures [13] for the memtable fails to fulfill this external demand. To this end, we first examine the design tradeoffs of in-memory data structures using RocksDB as a case study, as its in-memory components collectively well-represent a spectrum of data structures used across KVS systems. This includes a list-based structure (skip list) and hash-based structure (cuckoo hash), which are both popular in modern KVS, and their hybrid structures (hashed skip list and hashed linked list). Our quantitative analysis reveals that there is no *one size fits all* solution; existing in-memory data structures exhibit severe performance penalties for at least one of the operations given a large memory capacity and a high access concurrency. Because there are multi-dimensionally conflicting goals (such as concurrency, scalability, fast insertion and retrieval, consistent scanning support, and efficient in-order traversals), it is virtually impossible for a single data structure to satisfy all these requirements. We also observe that naïve amalgamations of existing structures (hashed skip list and hashed linked list) fail to get the best of both worlds. Furthermore, our experiments reveal a *performance anomaly* when increasing the memory size: as more key-value pairs are maintained in memory, the shortcomings of these data structures exacerbate.

Based on our observations, we propose a novel in-memory component for key-value stores called TeksTable that exports a fast and consistent in-memory component by judiciously weaving the complementary two data structures: the hash-based structure (cuckoo hash) and list-based structure (skip list). The cuckoo hash is used for fast $O(1)$ point-accesses, while the skip list is used for ranged retrievals due to its sorted data and consistent snapshot support. Beyond simply putting together the two data structures, we optimize each structure to augment its strengths. We implement an optimistically concurrent version of the cuckoo hash, and an efficient skip list variation that selectively performs in-place updates while guaranteeing consistency. Furthermore, we connect the two data structures so that traversing the skip list benefits from the cuckoo hash's fast lookups. Despite the modest increase in memory footprint and added synchronization cost, the benefits of these optimizations outweigh the overheads of maintaining the two data structures, achieving excellent performance across a wide range of workloads particularly under the highly concurrent and scalable environments, without any compromise on consistency.

We implement TeksDB by modifying RocksDB and demonstrate that it gets the best of both worlds: it achieves 4.5× and 9× performance for range_query and scan compared to the skip list,

	SkipList	**CuckooHash**	**HashSkipList**	**HashLinkedList**	**TeksTable** (proposed)
Point Access	$O(logN)$	$O(1)$	$O(logN/B)$	$O(logN/B)$	$O(1)$
Sorted Retrieval	$O(logN) + O(K+D)$	$O(NlogN) + O(K)$	Not supported	Not supported	$O(1) + O(K)$
Read/Write Coordination	MVCC	Not supported	MVCC	MVCC	MVCC
Write/Write Coordination	Non-blocking	Blocking	Blocking	Blocking	Non-blocking

Table 1. In-Memory Component Taxonomy of KVS. *N - number of data items, B - number of buckets, K - number of data items requested in a sorted retrieval, D - number of duplicate keys.*

and achieves almost identical performance for put and get to the cuckoo hash. For real composite workloads using YCSB, our design improves the throughput by as much as 3.3× and 1.6× compared to RocksDB and Redis, respectively. Our contributions are as follows:

- We quantitatively analyze the performance of existing state-of-the-art in-memory data structures with respect to increasing concurrency and scaling memory capacity, and show that no single data technique is well-equipped to handle a wide range of operations. (§ 3)
- We design and implement [2] two complementary data structures—the cuckoo hash and the skip list—so that it efficiently handles today's diverse KVS workloads. (§ 4)
- We demonstrate the effectiveness of our design by evaluating it against existing data structures across microbenchmarks (put, get, range_query, and scan), and real workloads using YCSB suite. (§ 6)

The remainder of this paper is organized as follows. In § 2, we first review the popular in-memory data structures as well as RocksDB which is a widely used KVS. § 3 provides a quantitative analysis of the limitations of the existing KVS in-memory data structures. § 4 explains the design and implementation of TeksDB, and § 5 describes the evaluation methodology. § 6 and § 7 present the experimental results. § 8 discusses our proposed design in relation to prior work, and § 9 concludes.

2 BACKGROUND

In this section, we present an architectural overview of KVS, focusing on its major components. We then explain in-memory data structures that are widely used in many KVS systems.

2.1 Overview of Key-Value Store

Modern KVS consists of three key components: a *memtable*, a *persistent storage*, and a *log*. The memtable is an in-memory component that buffers updates before they are written to the persistent storage. Batches of updates are sorted and coalesced when writing to storage, but the data maintained in memory may not always be sorted. SkipList is the popular memtable structure that keeps data sorted in the main memory. It is supported by many KVS, including RocksDB [13], WiredTiger [40], and HyperLevelDB [20]. HashTable, such as CuckooHash, keeps buffered data unsorted. It is mostly used with in-memory KVS like Redis [35] and Memcached [27], where point query latency is important. RocksDB also supports CuckooHash. Our investigation focuses on the memtable because the increasing memory size makes it the key component that has a high impact on user-perceived performance.

[2]We have made TeksDB publicly available at https://github.com/yulit0738/teksdb.git

The persistent storage is an on-disk data structure where data are stored permanently. In RocksDB, data are organized using a log-structured merge tree (optimized for writes), but some other KVS such as LMDB [24], BerkeleyDB [36], and TokyoCabinet [37] use the traditional B-tree (optimized for reads). In-memory KVS such as Redis [35] and Memcached [27], on the other hand, has no specific persistent data structures, only maintaining a log.

In order to provide atomicity and durability for in-memory data that are not made persistent, most KVS, including RocksDB, use write-ahead logging (WAL). Every update is logged and written to the memtable, and the log is discarded only after the buffered updates are written into the on-disk data structure. The write-ahead logging can be performed synchronously or asynchronously, but most recent KVS opt for asynchronous logging because of the performance overhead associated with synchronous logging [21].

As we have seen above, many KVS have been proposed and they have employed complementary schemes depending on their design purposes. In this study, RocksDB is selected as a default KVS engine because it natively supports a number of in-memory data structures that are used in other KVS, allowing a comprehensive analysis and fair comparison of the in-memory components. Note that RocksDB is one of the widely used KVS across a wide range of systems from internet service providers to maintain their unstructured web-based contents [13] and distributed data service platforms (e.g., Hadoop [19], Dynamo [12], MongoDB [29], and Ceph [22]) to emerging block-chain platforms [38] and conventional database systems as their backend storage engine [14].

2.2 The Data Structures

We study the following four data structures for the in-memory component of a KVS: SkipList, CuckooHash, HashSkipList, and HashLinkedList. Table 1 summarizes performance characteristics of each data structure, and we briefly describe them below:

SkipList is a multi-layered linked list [33]. The bottom layer is maintained as same as a linked list with each node pointing to the next, but in the higher layers, nodes may point ahead, essentially *skipping* some in-between nodes. The SkipList supports a probabilistically binary search over sorted data, allowing $O(logN)$ insertion and retrieval times. The SkipList used in KVS typically operates in an append-only manner: new data with the same key are never overwritten. This append-only approach offers two advantages over its in-place update counterpart. First, maintaining all versions of the data makes it easy to support snapshot isolation [5] through multi-version concurrency control (MVCC) [6], where all readers see a point-in-time consistent view. Second, with append-only insertions, multiple writers can concurrently manipulate the data structure in a thread-safe manner through atomic operations such as compare-and-swap. Consequently, this high-concurrency allows the SkipList to scale well in today's many-core servers.

CuckooHash is a hash table that reduces the chance of collisions by using multiple hash functions. In the event of a collision (all the buckets indexed by all the hash functions are occupied), the newly inserted entry pushes the older one to a different location using a different hash function in the table, and this pushing of older entry may happen recursively. In RocksDB, four hash functions are used, and in the event of a hash collision, possible movement paths (of pushing older data to a new location) are explored to find the optimal one. If too many items need to be moved (over one hundred in RocksDB), the new item is maintained by another backup list, instead of exercising a long series of data shifts. As for the time complexity, the CuckooHash allows fast $O(1)$ point updates and lookups. However, it poorly services ranged retrievals because it does not maintain data sorted, and thus it should sort entire dataset upon a ranged query, which takes $O(NlogN)$ time. In contrast to the SkipList, the CuckooHash updates data in-place with no MVCC support.

The writes are also made by a single thread, using a write queue that serializes concurrent accesses, because CuckooHash is difficult to make thread-safe due to its long shift operations.

HashSkipList maintains multiple buckets, and the bucket is indexed by the hash of the key's prefix. Inside each bucket, entries are organized using SkipList. Prefix-hashing provides some degree of sorting, as consecutive keys fall into the same bucket. However, data are only partially sorted, without total ordering. In a HashSkipList, each bucket has a sorted skip list, yielding $O(log(N/B))$ time complexity, where B is the number of buckets.

HashLinkedList also organizes data using multiple buckets, and is indexed by the hash of the key's prefix, similar to the HashSkipList. However, within each bucket, entries are at first maintained in a linked list, but are reorganized into the SkipList once its size becomes too big (256 items in RocksDB). Similar to the HashSkipList, the HashLinkedList only provides partial ordering, by maintaining data in a sorted list within each bucket. Both the HashSkipList and the HashLinkedList do not support a globally in-ordered scan, because creating an iterator across multiple buckets causes excessive overhead.

2.3 Summary

Below is a brief summary for the four in-memory data structures in the context of the following three characteristics.

Sorted vs. Unsorted: List-based data structures keep entries sorted, trading the performance of point accesses for that of ranged retrievals. Hashing, on the other hand, provides excellent put and get performance, but in order to support ranged accesses, it needs to sort on-demand all the key-value entries in the hash table. Hybrid structures—HashSkipList and HashLinkedList—maintain data sorted in each bucket, and inherit both the weaknesses of list-based and hash-based structures. Their put and get performances are not as good as that of the CuckooHash, and they do not support global in-order scans like the SkipList.

Overwrite vs. Append: Allowing the overwrite of existing data speeds up the update operation, but it makes it challenging to support concurrent read-while-writing with consistency, because overwrites prohibit a point-in-time consistent view. CuckooHash tradeoffs data consistency for write performance, by updating data in-place. In contrast, an append-only update policy supports concurrent reads and writes without compromising the data consistency, but it comes at the cost of maintaining multiple copies for each key, which inflates the memory used per-key.

Blocking vs. Non-blocking: Non-blocking updates increase write concurrency, but they require the traversal and update algorithms to be designed with concurrency in mind. In contrast, blocking algorithms that rely on a mutex lock are easier to implement because there is only one single access at a time. In RocksDB, only the SkipList updates in a non-blocking manner, and all the other data structures serialize updates using a queue.

3 QUANTITATIVE ANALYSIS

In this section, we investigate the performance of the described in-memory data structures with respect to the number of threads (scalable concurrency) and the in-memory component size (large memory effectiveness) for various KVS operations.

We vary the number of threads from 1 to 16, and set the size of the in-memory data structure to 64MB (small memory, default configuration in RocksDB) or 16GB (large memory). In order to service user requests seamlessly when writing the memtable to persistent storage, RocksDB marks the full memtable as *immutable*, creates a fresh *mutable* memtable, and flushes the *immutable* one to storage in the background. In our experiments, we allow up to two immutable tables to be created: memory used for both mutable and immutable memtables can be up to 192MB and 48GB for the

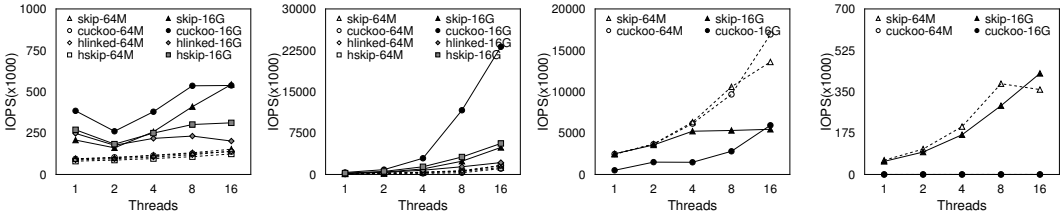


Fig. 1. Db_bench Throughput with Different Memtable Data Structures.

small and large memory setting, respectively. We exercise the KVS using four synthetic workloads of db_bench (a built-in benchmarking tool of RocksDB and LevelDB): `fillrandom` generates put requests with random keys, `readrandom` generates get requests for existing items with random keys, `readseq` emulates scan by creating an iterator and retrieving all items in sorted order, and `seekrandom` emulates `range_query` that searches a set of items within a key range through an iterator. This workload performs a seek operation, followed by ten reads.

The data store is set up with 16B-sized keys and 256B-sized values. We initially load 100 million data items, and then run 100 million operations for each workload.

Our experiments are performed on a machine with 18 physical cores based on Intel Xeon architecture, 72 GB of memory, and a 256GB NVMe SSD, running Ubuntu 16.04 LTS with the Linux 4.4 kernel. RocksDB version 5.13 (released 2018.04) runs on top of an ext4 filesystem. We use the default configurations of RocksDB (e.g., asynchronous WAL and auto-compaction) for all other parameters.

The following subsections will describe the performance of KVS data structures using Figure 1 that shows the performance of KVS across operations while varying the number of threads for a 64MB and a 16GB memtable.

3.1 PUT

Large memory effectiveness: All data structures achieve higher performance when increasing the memtable size from 64MB to 16GB (Figure 1a). The CuckooHash improves the most, increasing its throughput by 3.6×, while the other data structures improve by 2–2.6×. With a larger memtable, more data are buffered in memory and written to storage in batch, increasing the I/O efficiency.

This also implies that a large memory buffer increases the degree that the in-memory data structure affects the overall performance. For example, with one thread exercising put using a 64MB memtable, the CuckooHash's performance is similar to that of the SkipList, but the gap increases to 18% with a 16GB memtable. While the storage performance plays a greater role with a small memtable, the time complexity of the in-memory data structure ($O(1)$ for the CuckooHash and $O(logN)$ for the SkipList) affects the overall performance more with a large memtable.

Scalable concurrency: Under a highly concurrent and ample memory setting, the CuckooHash and the SkipList outperform the HashLinkedList and HashSkipList by up to threefold, exhibiting better scalability than the other two data structures. We attribute the main culprit that limits the performances of the HashSkipList and HashLinkedList to their blocking synchronization policy. In contrast to the SkipList that supports non-blocking concurrent insertions to the memtable, the HashSkipList and HashLinkedList serialize concurrent accesses through a wait queue, having

requests processed in a single leader thread. This feature severely limits the parallelism of applications, failing to provide scalability at high concurrency. The CuckooHash also uses blocking synchronization like the HashSkipList and HashLinkedList, but its fast $O(1)$ insertion allows it to scale well with increasing memory capacity. However, at high concurrency (16 threads), the SkipList ultimately outperforms the CuckooHash. Even though the critical section for the CuckooHash is short, the blocking synchronization limits performance in a highly concurrent situation.

One counter-intuitive result is that the performances decrease when increasing the number of threads into two. This phenomenon is rooted from the RocksDB's optimization for the single-threaded operations: RocksDB executes in a different code path for single threaded execution that disregards concurrency control.

Data structure suitability: For the put operation, the CuckooHash outperforms all other data structures in general due to its low time complexity. However, its blocking synchronization is a potential performance bottleneck, as evident by the results revealed under high concurrency.

3.2 GET

Large memory effectiveness: The performance of the get operation improves by 6.3× on average across all data structures when increasing the memory size from 64MB to 16GB. While put operations cause additional writes to persistent storage regardless of memtable size (due to logging and background compaction), get only goes to storage when it misses in the memtable, and these misses may incur multiple reads to locate the correct data (read amplification). Consequently, the get performance tends to be more sensitive to memory capacity and in-memory buffer efficiency.

Scalable concurrency: Because reads do not require synchronizations, all the data structures scale with increased concurrency. Nevertheless, the degree in which the performance scale varies across the memtable structures; the CuckooHash scales well with the number of threads, while other data structures do not scale as well, increasing the gap as the concurrency increases. The cause for the performance difference is due to the time complexities of the data structures: the $O(1)$ lookup of the CuckooHash excels as concurrency increases, while the $O(logN)$ of the SkipList and $O(log(N/B))$ for the hybrid variants cause limited scalability.

Data structure suitability: Similar to our findings for put, the CuckooHash is the best-fit data structure for the get operations. Large memory places greater emphasis on the efficiency of the in-memory component, especially for get, and the CuckooHash's $O(1)$ lookup time and scalable currency make it suitable for point lookup workloads.

3.3 Scan

The scan retrieves the entire data in the KVS, and it is commonly used in data analytics applications. A typical implementation of the scan uses an iterator that sequentially accesses the entire KVS. Figure 1c shows the performance of scan operation for the SkipList and the CuckooHash while varying the number of threads and the memory buffer size. Both the HashSkipList and HashLinkedList support only the partially ordered scan, and are excluded from the analysis.

Large memory effectiveness: Interestingly, the performance of scan significantly drops when a memory size increases (Figure 1c). Both data structures exhibit performance degradation, but for different reasons. For the CuckooHash, the cost of creating an iterator becomes higher with an increase of memory capacity. Because the CuckooHash does not maintain data sorted, it sorts the data on-demand upon a scan operation, enabling an iterator to do an in-order traversal. As more data entries reside in the memory buffer, the sorting cost becomes more expensive, resulting in an

average performance loss of 70%. On the other hand, there is no cost for creating an iterator for the SkipList as the data items are maintained in sorted order; it simply needs to point to the head of the list. However, the SkipList also results in a 25% performance loss when the memory is increased from 64MB to 16GB. The performance loss of the SkipList is due to the duplicate keys in its data structure, caused by its append-only updates. When the data with the same key is updated, the SkipList appends the up-to-date data into the list, not overwriting the existing data. As a result, the SkipList maintains multiple nodes for the same key, and these duplicated keys cause superfluous accesses to data items, thereby increasing the time to get the next item.

The primary intention for the append-only update of the SkipList is to support MVCC, which provides concurrent read-during-write with consistent data. To this end, RocksDB uses a timestamp-based concurrency control; generating a unique timestamp for each transaction and associating a read request with a last committed timestamp, it allows concurrent reads to the consistent write-before data during a transaction. This property holds only when the SkipList maintains all in-use snapshots that each are pointed by a unique timestamp, by performing insertions instead of overwrites.

In contrast, the CuckooHash copies all the data items (while blocking subsequent operations) and sorts them on-demand to create a consistent snapshot.

Scalable concurrency: As shown in Figure 1c, the SkipList does not scale well as the number of threads increases. Similar to the observed performance degradation of the SkipList when increasing memtable size, this poor scalability is caused by retaining duplicated keys. With more duplicate keys under more threads, it takes longer for the SkipList to traverse the data structure that contains multiple versions of the data. Thus, at 16 threads, the in-place updating CuckooHash outperforms the append-only SkipList.

Data structure suitability: Under a scan operation, we observe that more memory does not necessarily lead to better performance, and the two data structures exhibit particular limitations. The CuckooHash needs to sort its data on-demand which increases in cost with a larger memtable. The SkipList, on the other hand, maintains duplicate keys to guarantee consistency, but this degrades the performance as memory increases. Note that duplicate keys are only removed when flushing to persistent storage in the implementation of RocksDB. These limitations need to be addressed for a high-performance KVS.

3.4 Range Query

The range_query retrieves multiple data from a given starting key, by performing a seek to the initial key and reading multiple items following it. The iterator is created for every range_query request so that the most up-to-date versions of items can be accessed at each run. Similar to scan, we do not consider both the HashSkipList and HashLinkedList, because they support range_query only for partially sorted data, and thus the fair comparison is not possible.

Figure 1d shows the performances of the SkipList and the CuckooHash for range_query operations. The immediately noticeable result is that the CuckooHash performs miserably (less than 10 IOPS) regardless of the number of threads and the memtable size. This is because of the CuckooHash sorting all data items on-demand for every range_query operations. The subsequent analysis thus shall focus on the performance of the SkipList with respect to the concurrency and memory size.

Large memory effectiveness: The range_query shows no remarkable performance change with respect to the memory capacity, but this does not mean that the operation is insensitive to the size of the memtable. With a large memtable, there are more memory hits, but it also entails traversing through duplicate keys. With a small memtable, more data need to be retrieved from persistent

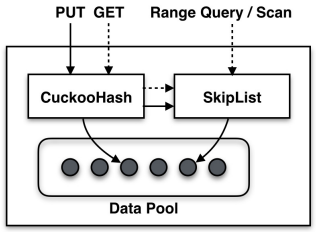


Fig. 2. TeksTable Architecture.

storage, but the data maintained in storage has duplicates removed. These costs and benefits offset each other in the SkipList and show small performance variation with respect to memory size.

Scalable concurrency: Unlike scan, range_query scales well as the number of threads increases. While the SkipList's append-only nature degrades the performance of traversing the list, because range_query is a combination of a seek and sequential retrievals, the impact of duplicate keys under range_query is less than scan. A larger range in the range_query operation will exhibit less scalability than smaller ranges.

Data structure suitability: The CuckooHash that requires on-demand sorting for each operation is unusable in practice. Although the range_query operation (10 sequential reads from a key) show that the SkipList is a suitable data structure, a much larger range will reveal the limitations of the SkipList caused by its append-only updates. For example, a scan operation can be considered as an extreme version of the range_query, and we observed that it does not scale well.

4 TEKSDB

Through extensive analyses, we observe that there is no one-size-fits-all memtable structure and the best-performing data structure depends on the workload. The CuckooHash excels in point lookups and insertions, but it performs miserably for range_query. The SkipList well-handles sorted data retrievals such as scan and range_query, but its lookup and insertion are not as efficient as the CuckooHash. Furthermore, the performance of the SkipList degrades as more key-value pairs exist in memory.

Motivated by this, we design a new KVS called TeksDB that not only seamlessly integrates the two complementary data structures, CuckooHash and SkipList, but also augments the strength of each data structure. Figure 2 illustrates the conceptual view of TeksDB. TeksDB utilizes the CuckooHash for servicing point lookup and updates, and the Skiplist for handling sorted data retrievals. The key-value pairs are maintained in a shared data pool, and the data structures in the CuckooHash and the SkipList only maintain pointers to the data pool. Thus, the CuckooHash and SkipList are essentially indices to the data, simply organized in a different way. With this design, TeksDB aims to provide the optimal performance across a wide range of workloads.

While the concept of TeksDB is straightforward, there are several challenges that need to be addressed to achieve both high performance and data consistency. Listed below briefly outlines the inherent issues and our approaches to resolve them.

- **Blocking updates of the CuckooHash.** Although the CuckooHash handles put in $O(1)$ time, it serializes writes with a single-threaded work queue, limiting the performance under highly concurrent workloads. We resolve this issue by implementing an *optimistically concurrent* version of the CuckooHash; updates proceed concurrently in most cases, only using a mutex lock in the event of a collision (§ 4.1).

- **Failing to provide point-in-time consistent views.** The CuckooHash overwrites data even while it is being read. This property causes a half-written or timestamp-inconsistent data to be read, making TeksDB fail to provide data consistency. We resolve this issue by exploiting SkipList that supports MVCC; we service the data in CuckooHash if it is consistent, otherwise, retrieving the consistent version of data in SkipList (§ 4.2).
- **List size inflation due to duplicate keys.** The SkipList updates data in an append-only manner, maintaining all obsolete values. This makes it easy to support multi-version concurrency control, but inflates the size of the list, reducing the efficiency of list traversals especially in large memory settings. To resolve this issue, we propose a variant of the SkipList that *selectively allows in-place updates* to eliminate the number of duplicate keys without violating data consistency (§ 4.3).
- **Threats and opportunities for combining two data structures.** Integrating two data structures into an in-memory component presents both threats and opportunities. On the upside, we can accelerate the seek operation for locating the starting key in the SkipList by using a *shortcut pointer* from the CuckooHash. This allows $O(1)$ seek time complexity (§ 4.4). On the downside, this imposes a data synchronization overhead. The two data structures—the CuckooHash and SkipList—must be maintained in a consistent manner, but if the two are updated synchronously, the performance is bound to the slower one. TeksDB resolves this challenge by deferring the update of the SkipList until ranged retrieval is needed, but processing it fast through multi-threaded updates (§ 4.5).

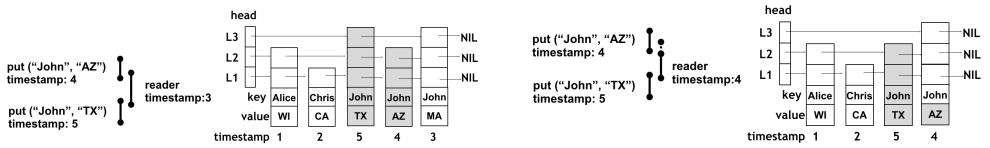
4.1 Optimistically Concurrent CuckooHash

TeksDB uses a CuckooHash to service fast point updates and lookups. However, as observed in § 3, the CuckooHash does not allow concurrent updates, and instead serializes I/O requests through a single wait queue. When multiple updates occur in the CuckooHash, they will be batched and processed sequentially by a leader thread that works on behalf of the other blocked threads. This limits the scalability of KVS under high concurrency, as observed in Figure 1a.

However, building a non-blocking and concurrent CuckooHash is challenging due to its complex update protocol upon hash collision. In the event of a hash collision, the newly inserted entry pushes the older one to a different location, and this may repeatedly push others in a recursive manner. Under such a scenario, all the possible movement paths are explored with a breadth-first graph traversal, and the number of critical sections involved in this process makes it challenging to mutate the CuckooHash in a non-blocking and concurrent fashion.

We design and implement an optimistically concurrent version of the CuckooHash, exploiting the fact that the hash functions are designed to reduce the collision rate. This allows concurrent and lock-free accesses when any vacant slot is available with the use of atomic instructions (i.e. CompareAndSwap). In the event of a collision, the CuckooHash uses a mutex lock to provide thread-safety and migrates a series of data items. It is noteworthy that RocksDB keeps the collision rate of CuckooHash low by making the in-use memtable immutable when the ratio of filled buckets goes over a configured threshold, and creating a new hash table. TeksDB also inherits this policy.

The mutex lock serializes multiple collisons, but our design allows a concurrent execution of a single collision handling routine and other operations such as put and get. The collision handling routine is made up a series of atomic compare-and-swap (CAS) instructions and implements a careful *check-before-update* logic: as the collision handler performs a series of updates found from the breadth-first search to resolve the hash collision, it checks and ensures that a concurrent update in the middle of the path is not lost.



(a) Append-only SkipList (b) In-place update SkipList

Fig. 3. Two Versions of SkipList.

4.2 Point-In-Time Consistency Support

A modern KVS typically supports a point-in-time consistent view through timestamp-based concurrency control, allowing concurrent reads and writes without blocking. More specifically, the KVS including RocksDB use a group-based update protocol to readily provide this facility: that is, each write operation receives the unique timestamp upon a request, but they proceed in groups. The writers arrived during updates of a preceding group make up of another group, and they are blocked until the preceding group is completed. When all writes of the active group are done, the KVS update the *last committed timestamp* with a latest timestamp within the group, launching writers in the next group. The last committed timestamp represents the version of the most recent snapshot, which is used to service the read requests until the next consistent version is created.

This property holds on the SkipList that supports data versioning, but not on the CuckooHash that overwrites data even during reads. In order to ensure point-in-time consistency, the CuckooHash checks the timestamp of the target read request with the system-wide last-committed timestamp. If the data maintained by the CuckooHash is timestamp-consistent(i.e., the timestamp is before the last committed timestamp), it is serviced immediately; if not, it falls back to the SkipList to retrieve the latest consistent data.

4.3 Selective In-Place Updates in SkipList

The SkipList is used to provide fast and consistent ranged data retrievals, and its index structure must be updated consistently with the CuckooHash. However, unlike the CuckooHash that updates in-place, the append-only approach for the SkipList—designed for snapshot isolation—inflates the data structure and increases the overhead for list traversals when updates are frequent (as is the case in real workloads such as YCSB's session store and photo tagging).

TeksDB resolves this problem by selectively allowing in-place updates. The rules are simple; when the multi-version maintenance is essential for snapshot isolation, i.e., under the concurrent reads and writes, the SkipList performs the native append-only updates. However, when the multi-versions are not necessarily needed, i.e., when only writers exist, the SkipList performs in-place updates, eliminating duplicates.

Figure 3a and Figure 3b illustrate the inner workings of the original append-only SkipList and the proposed SkipList, respectively. This example assumes two put operations occur for the key *John* sequentially, interleaved with a read request (e.g, get, range query, or scan). In the original SkipList (Figure 3a), both updates are inserted into the list, such that a reader concurrently retrieves a consistent data (i.e., *MA* with timestamp 3, the last committed timestamp), even during updates (i.e., *AZ* and *TX* with timestamp 4 and 5). As such, the original SkipList in Figure 3a has three copies of the same key, increasing a traversal cost of the dataset. On the contrary, our design selectively updates data in-place, as shown in Figure 3b: the first put request overwrites existing data because

there is no reader when it starts, while the second put operation appends a new data, as there exists at least one on-going read operation.

Now let us discuss data consistency of the read operations. The data consistency cannot be guaranteed if a reader accesses SkipList in an in-place update mode; the pending writes issued before the read can overwrite the requested version of data. Hence, the readers arriving when SkipList runs in an in-place mode wait until all updates in the active group are completed (i.e., put(*John*, *AZ*) in this case). Once they all are completed, TeksDB switches the SkipList into the append-only mode, allowing readers to access it with the last committed timestamp. In the example in Figure 3b, the reader will access the snapshot associated with the timestamp 4, which is the system-wide last committed timestamp.

4.4 Bridging Structures for Fast Seek

TeksDB maintains two complementary data structures simultaneously to service different requests in an optimized way. CuckooHash serves lookups and retrievals with $O(1)$, while the SkipList supports sorted data scans immediately, without reorganizing data on-demand. The unioning of these structures, though, enables further optimization that accelerates the random ranged retrievals. When the range_query occurs, it begins with seeking a starting key location from which a sequence of data are retrieved. SkipList seeks data with $O(logN)$ time complexity doing a binary search over the sorted keys. This latency is longer than optimal ($O(1)$), and it becomes longer as the number of data items increases, which is the case in a large memory. We reduce this seek latency by introducing a *shortcut pointer* in CuckooHash, which points to the node with a given key in SkipList. The shortcut pointer allows to locate the starting node for the range_query in SkipList with $O(1)$ time complexity.

4.5 Weaving the Two Data Structures Together

We maintain two data structures, the CuckooHash and the SkipList, for indexing the location of the data in the data pool. If the CuckooHash and SkipList are updated synchronously, the user-perceived latency for insertions will inevitably increase, which is undesirable. TeksDB resolves this challenge by deferring the update of the SkipList, while synchronously updating the CuckooHash. Upon a put request, TeksDB inserts a new data into the CuckooHash in $O(1)$ time, and puts the request to a work queue that is serviced by a background worker thread. The background worker processes the outstanding requests in the queue by updating the SkipList in accordance with them. This asynchronous update protocol decouples the backend SkipList and the frontend CuckooHash, hiding the double update cost from the users. Another critical design issue when weaving the two data structures together is ensuring that the wait queue is not the performance bottleneck. To avoid this pitfall, we adopt the concurrent queue implemented in a lock-free manner by internally making use of pre-allocated contiguous blocks instead of linked lists [8].

Although the SkipList is updated asynchronously, frequent sorted data retrieval operations can make these background updates the performance bottleneck. If the update in the SkipList data structure lags behind that of the CuckooHash, applications may experience a long-tail latency when performing range_query. TeksDB uses a single background worker to manage the cost of maintaining a large thread pool, but it receives help from user threads in processing concurrent operations. More specifically, foreground threads for the range_query or scan are blocked and queued in a wait queue while the SkipList becomes up-to-date. If needed, TeksDB wakes these foreground threads and have them execute the update routine on the SkipList, thereby allowing concurrent updates without managing multiple background worker threads.

5 METHODOLOGY

We implement a TeksDB based on RocksDB 5.13 with 6.7K LOC. Our implementation includes the techniques described in § 4, from augmenting the CuckooHash for concurrency and the SkipList for fast traversals, to cross-sectional optimizations that weave the two data structures together. As opposed to the analysis section that experimented both 64MB and 16GB memtable, we demonstrate the effectiveness of TeksDB with a 16GB in-memory component, which is more comparable to today's high-performance computing environments.

We evaluate TeksDB under two settings: when the entire dataset fits in the memtable (§ 6), and when it does not (§ 7). Although large datasets that spill out of the memtable is more realistic, we first demonstrate the effectiveness of TeksDB by isolating the in-memory component and removing I/O activities caused by flush and compaction. We run both db_bench micro-benchmark (§ 6.1) and YCSB real-world benchmark (§ 6.2) for evaluation. We compare TeksDB against four production-level KVS: Redis, WiredTiger, LMDB, and RocksDB. For the micro-benchmark, however, we only present results compared against RocksDB: the other three does not natively support db_bench.

6 EVALUATION WITH FIT-IN MEMORY DATASET

We evaluate TeksDB with a fit-in-memory dataset to isolate and demonstrate the effectiveness of our proposed in-memory component, using both micro- and real-world benchmarks. We also measure and discuss the overheads of TeksDB: the increased memory footprint due to the double indexing structure (§ 6.3.1), and the delayed handling of a read request when blocked by an in-place update (§ 6.3.2).

6.1 Microbenchmark

We measure the performance of TeksDB using db_bench benchmark suite. For analysis, we implement several intermediate versions of TeksDB. Teks-s updates both the CuckooHash and the SkipList synchronously, and Teks-a asynchronously updates the SkipList with no other enhancement techniques. Teks-ac implements the concurrent CuckooHash on top of Teks-a, and Teks-acsp connects the two data structures with a shortcut pointer for fast indexing. Finally, Teks-acsp-in builds on top of Teks-acsp and enhances the SkipList for fast traversals using selective in-place updates. We evaluate TeksDB on the same machine described in § 3, with physical 18 Intel Xeon cores, 72 GB of memory, and a 256GB NVMe SSD.

We compare five versions of TeksDB with the original RocksDB running with SkipList and CuckooHash, respectively, because they provide upper- and lower-bound performance across workloads. We load the KVS with 1 million data items of 16B keys and 256B values, and then execute the following workload, 1 million operations each, in the following order: `readseq` (scan), `fillrandom` (put), `readrandom` (get), and `seekrandom` (range_query).

The goals of running the microbenchmarks are twofold. First, we expect to verify that TeksDB performs at least as good as the best-fit data structure. That is, we expect that TeksDB performs as good as the CuckooHash for put and get, and as good as the SkipList for scan and range_query. Second, we wish to discern the different enhancement techniques implemented in TeksDB, and observe the performance gains over the best-fit data structure.

6.1.1 PUT. Figure 4a compares the performance of TeksDB with RocksDB using CuckooHash and SkipList for put operations. The synchronous TeksDB-s performs worse than the inferior data structure, SkipList, because the double update cost is reflected to the user-facing latency. However, all versions of asynchronous TeksDB (all other than Teks-s) provide almost same performance as the best-performing data structure, CuckooHash, hiding well the costly update penalty from the

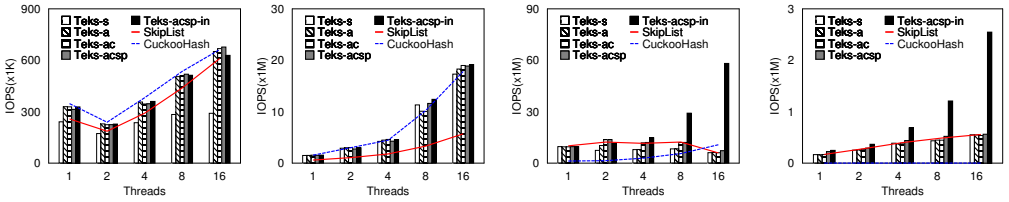


Fig. 4. Db_bench Throughput.

users. The baseline asynchronous version Teks-a, a performance loss compared to CuckooHash is within 7%, and the performance is better than SkipList by 6–27%, even maintaining two data structures.

The comparison of Teks-a and Teks-ac shows the effectiveness of optimistic concurrency of CuckooHash. The result, however, shows that optimistic concurrency (Teks-ac) makes a little or negative impact on the performance when the number of threads are less than 16. This counter-intuitive result is due to the fact that the cost of waking up and executing threads for parallel updates outweighs the benefit from concurrency, particularly when a critical section is very short and contention is not high. However, Teks-ac exhibits better scalability with many threads, showing that optimistic concurrency is effective under a highly concurrent circumstance, which is a target environment of TeksDB. Teks-acsp that uses a shortcut pointer additionally has little impact on put operations because it is originally designed to improve the seek performance.

Now let us discuss Teks-acsp-in with a selective in-place update policy. Teks-acsp-in decreases the performance slightly compared to the CuckooHash. This is because the data-reference pointer in Teks-acsp-in is updated synchronously, affecting the user-facing performance in put. Indeed, this pointer is originally subject to the asynchronous update that is performed by a background thread into the SkipList. However, a single pointer update costs much less to do synchronously, than deferring it later with queueing and dequeueing. Thus, we perform it immediately in foreground, at a marginal performance loss. It can be longer when many threads content to it because CAS operation can fail and it should retry several times. Nevertheless, the performance loss is only less than 6%, which is a reasonable trade-off for a significant gain in another operation.

6.1.2 GET. Figure 4b shows the performance of TeksDB family compared to RocksDB using SkipList and CuckooHash in get workload. All versions of TeksDB show excellent performance equivalent to a best-performing CuckooHash, and provide 2.5–3.6× higher throughputs than SkipList. This result is obvious because TeksDB uses CuckooHash to process point lookup operations, which has $O(1)$ time complexity.

6.1.3 Scan. Figure 4c shows the performance of scan for five versions of TeksDB and for the CuckooHash and the SkipList. At high-level of concurrency, the advantages of reducing effects of duplicate keys become evident, improving the performance of scan by 9× for Teks-acsp-in compared to the SkipList. In db_bench run, scan operation is performed over the initial dataset that are inserted by multiple threads with a same range of keys respectively. Thus, there are duplicated keys as many as the number of threads. Teks-acsp-in, which performs in-place update selectively, increases the performance by 9×, compared to the original SkipList. Eliminating duplicated keys in SkipList, TeksDB is able to reduce the average number of memory references to get a next item in an in-order traversal, overcoming the drawback of SkipList.

Workload	*Description*	*Ratio*
A	Session store recording actions	50% Read, 50% Update
B	Photo tagging and tag reads	95% Read, 5% Update
C	User profile cache	100% Read
D	User status update	95% Read, 5% Insert
E	Recent post scan in conversation	95% RangeScan, 5% Insert
F	Database	50% Read, 50% Read-Modify-Write

Table 2. YCSB Workload Characteristics.

Workload	*Operations*	*put*	*get*	*range_query*
A	1,000,000	1,000,000	1,000,000	500,000
B	1,000,000	100,000	1,000,000	950,000
C	1,000,000	0	1,000,000	1,000,000
D	1,000,000	600,000	1,100,000	950,000
E	1,000,000	600,000	49,000,000	49,000,000
F	1,000,000	1,000,000	15,000,000	1,000,000

Table 3. YCSB Workload Operations.

6.1.4 Range Query. Figure 4d shows the performance of TeksDB variants in range_query, comparing it with those of CuckooHash and SkipList. In this scenario, we perform one million of seek operations followed by 10 subsequent reads, over the initial dataset (4.4GB). The range_query yields the similar performance trend to the scan operation, increasing performance by 4.5× in Teks-acsp-in. The shortcut pointer (denoted sp), which is introduced to accelerate seek operations in range_query, achieves performance gain by 15% on average and up to 35%, which can be measured by comparing the performances of Teks-ac and Teks-acsp. Although this seems relatively small in this workload because the range_query has 10× more reads than seek, and the performance benefit by key deduplication in the SkipList dominates, the effectiveness of the shortcut pointer is more well-demonstrated in YCSB workloads that include a plenty number of seeks followed by unique key traversals. The result is presented in the next section.

6.2 Real-World Benchmark

We evaluate TeksDB using YCSB benchmark to demonstrate its effectiveness under real-world workloads. We generate YCSB workloads using Ardb [1], a Redis-protocol compatible NoSQL system, with TeksDB configured as its backend KVS. To investigate the performance of in-memory component of KVS, we initially load 5 millions of data items, and perform 1 million of operations for each workload (total 7.8 GB dataset). The workloads are generated with 16 threads. We perform the experiments on a machine with a 24-core Intel Xeon processor, 192GB memory, and a 256GB NVMe SSD.

Table 2 summarizes details of YCSB benchmark suite. It has six types of workloads and each are a composite of operations such as read, update, insert, and range-scan, well-representing the scenario. For the sake of clarity, we report the number of KVS operations actually sent to KVS by Ardb in Table 3, when the total number of operations in YCSB workloads is set to 1 million. Ardb, a NoSQL system used in our experiments, translates YCSB requests into the KVS operations, conforming to the Redis protocol. For example, a read request involves a `get` operation for referencing the metadata for data access, and a `range_query` for retrieving the actual data. YCSB benchmark handles web objects, and each item consists of a set of fields (e.g., 11 fields and 3 metadata fields). The read

operation of YCSB retrieves all these data at all using a `range_query`. As another example, an update operation in YCSB workload *A* means updating a field of the data item, followed by metadata update, and thus it causes two `put` operations. These operations are generated concurrently so as to well-represent real-world circumstances. The workload Load A creates a large dataset by inserting data, and it is used in workload A, B, C, D, and F, while the dataset created by workload Load E is used for workload E. The workload Load A and Load E are all writes, and the workload C is all reads. The workload A, B, and D are read-intensive scenarios with small writes, and the workload E generates a large number of range queries with small inserts. The workload F generates read and read-modify-write operations equally.

Four production-level KVS—Redis [35], WiredTiger [40], LMDB [24], and RocksDB [13]—are also configured as backend stores for Ardb for fair comparison. For the sake of brevity, we evaluate the final version of TeksDB only, Teks-acsp-in (denoted by TeksDB). Below is a brief overview of the compared KVS.

Redis is the most popular in-memory NoSQL system these days that maintains all data in memory using a hash table. It allows nested data structures where entries of the KVS's hash table can be, in turn, another data structure such as lists, sorted sets, or even another hash table. Redis has no persistent store component; for persistence, it only supports simply logging of data, or periodic dump of the in-memory data. It should be noted that Redis does not support MVCC; it is single-threaded and overwrites the existing data on an update in the hash table.

WiredTiger is a persistent KVS system, and maintains data in memory using a skip list, with a choice between a B+tree and an LSM tree as the persistent data structure. For our experiments, we set WiredTiger to use the LSM tree. Because WiredTiger uses an append-only SkipList for the in-memory data structure, it supports MVCC (i.e., consistent reads concurrently with writes). Like RocksDB, WiredTiger also performs write-ahead logging, guaranteeing the eventual consistency, rather than immediate consistency.

LMDB (Lightning Memory-Mapped Database) is a persistent KVS system that relies on the system-level page cache for buffering and accesses the persistent files through the mmap interface. This design is intended to reduce the context-switch and IO stack overhead, which arises from maintaining double write buffers in user and kernel spaces. For the persistent data structure, LMDB uses a B+tree, which fits well with block-based caching in the page cache. LMDB supports data consistency by means of a copy-on-write technique instead of logging. This technique reduces double-write overhead of a logging, but imposes significant write amplification for small writes.

RocksDB: The detailed description of RocksDB is presented in § 2.1. We only present the results for RocksDB using the SkipList as its in-memory data structure as the performance of CuckooHash is prohibitively slow for range_query which are often invoked by YCSB.

6.2.1 Throughput. Figure 5 shows the throughput of KVS, and the numbers above bars represent the normalized throughput of each KVS with respect to RocksDB. TeksDB outperforms RocksDB for all workloads by 1.8× to 3.3×, and outperforms all compared KVS for most workloads. The Load workloads are the only cases when Redis performs better. TeksDB's high performance is attributed to the $O(1)$ time complexity for all types of requests by judiciously weaving the complementary data structures. On the other hand, KVS that use list-based data structures achieves at most $O(logN)$ for point lookup and seek operations. This operation-invariant excellence of TeksDB makes great achievements in YCSB workloads where various types of operations are concurrently mixed.

Furthermore, TeksDB's selective in-place update that slims down the list also contributes to the performance improvement by reducing the traversal cost. Figure 6 quantifies this factor. Figure 6a shows the breakdown for all write operations into three classes: insert, update(o), and update(a). insert represents put operations with a new key, while both updates, existing keys. Among updates,

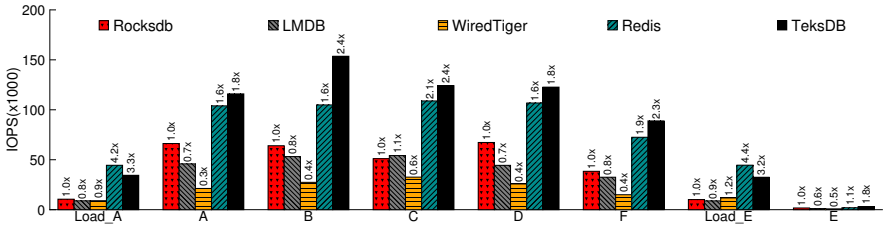
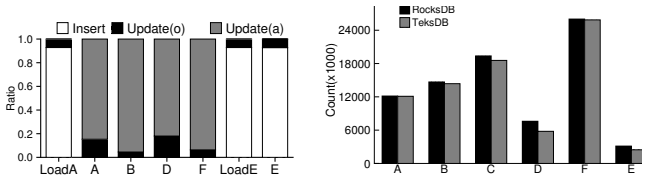


Fig. 5. YCSB Throughput with Fit-in-memory Dataset.



(a) Write breakdown (b) Skip count

Fig. 6. Effectiveness of In-place update SkipList.

update(o) represents the portion of overwrites through selective in-place updates, and update(a) is the put operations of an existing key that are appended. This analysis shows that YCSB workloads' updates account for a large portion in total writes. Of those updates, however, most are appended instead of overwritten, particularly in workload A, B, D, and F. This is because under these workloads, readers and writers are running concurrently with 16 threads, preventing existing data to be updated in-place for consistency guarantees. Nevertheless, on average 9.3%, and up to 18% of updates are overwritten across the workloads, improving the performance of traversal over the sorted dataset needed in range_query operations.

To verify the effectiveness of in-place update SkipList, Figure 6b shows the number of skips occured in iterating sorted data in the SkipList for servicing the range_query in YCSB. The skip count decreases by up to 24% in the workload D where the ratio of overwritten data is high.

Even though Redis uses a hash table, TeksDB outperforms it for most workloads. One of the shortcoming of Redis is that, it is a single-threaded program that serializes all reads and writes without concurrency. Thus, Redis performance decreases as the number of threads increases. Another shortcoming is that Redis's seek performance is $O(logN)$[3], as opposed to TeksDB's $O(1)$ (thanks to the shortcut pointer).

The cases when Redis outperforms TeksDB are for the Load workloads. This overhead is due to translating the Redis-protocol based requests into RocksDB interface. YCSB maintain multiple fields for a single object, reflecting web-contents workloads of distributed NoSQL systems such as BigTable or HBase. Redis supports complex data structures for each item, such as list, sorted set, and hash table, and thus Redis-client stores each object using a small hash table that has multiple different fields, issuing one put operation at the end. This request, however, is translated into multiple individual put operations in other KVS, intrinsically increasing the latency.

[3] Normally, hash tables need to be dumped and sorted to service seek, costing $O(NlogN)$. However, Redis can achieve a faster seek performance by maintaining a sorted data structure nested within its hash table.

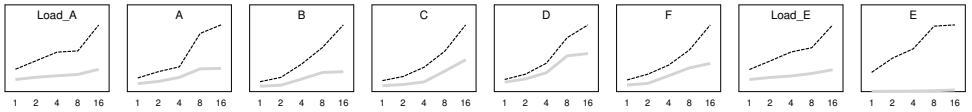


Fig. 7. Scalability in YCSB (Dotted: TeksDB, Solid: RocksDB, x-axis: threads, y-axis: throughput).

6.2.2 Scalability. To demonstrate the scalability of TeksDB, we plot the YCSB throughput as a function of the number of threads in Figure 7. This figure compares the performance of TeksDB (dotted line) to that of RocksDB (solid gray line) when increasing the number of threads 1 to 16. The results show that TeksDB scales better than RocksDB in all range of workloads. Through the various optimization techniques, TeksDB increases data access concurrency and reduces the negative effect of redundant updates caused by multiple threads, accomplishing great scalability at high concurrency.

6.3 Discussion

We have seen far the effectiveness of TeksDB in terms of performance. In this section, we discuss shortcomings of the TeksDB: an increase of memory footprint and a delay of a read request.

6.3.1 Memory Footprint. TeksDB incurs a modest overhead in memory footprint because it maintains both hash-based and list-based index structures. It also includes the space for data-reference pointer needed to connect separated key and value in TeksDB. However, this overhead can be offset by the reduced execution time because the much higher throughput comes with more memory, leading to a saving in the total cost. Moreover, the memory space overhead is reduced with proactive memory reclamations in TeksDB, which recycles memory occupied by obsolete data after in-place updates, whenever possible.

Figure 8a shows the memory usage of RocksDB and TeksDB with and without proactive memory reclamation under db_bench and YCSB workloads, respectively. The db_bench result is obtained by performing 100 millions of random put opeartions with 16 threads, and YCSB result is obtained by loading 5 millions of items and running workload A with 1 million of operations. With proactive reclamation, obsolete data are coalesced and removed online, reducing the memory footprint at a small cost in performance.

In db_bench, comparing RocksDB (solid grey) and TeksDB without reclamation (solid black), TeksDB uses 12% more memory for maintaining an additional hash-based index structure, but performs 18% better. The dotted black line represents TeksDB with proactive reclamation. It consumes much less memory, even 34% less than the baseline RocksDB, but performs 10% better than it. As a result, db_bench reduces the total memory consumption (calculated by multiplying average memory consumption with the elapsed time) by 38% with reclamation, and 3% even without reclamation, compared to RocksDB.

In YCSB, as shown in Figure 8b, TeksDB uses more memory space than RocksDB by 54–59% due to its intrinsic memory overhead. The interesting observation is that there is very little saving in memory with reclamation. The primary reason for this, as shown in Figure 6a, lies in that not only the ratio of in-place updates is not high, but also the overwritten items are supposedly metadata and thus small in sizes than other requests. However, TeksDB reduces the execution time in half, increasing IOPS by 1.89–1.91× than RocksDB, thereby leading to 16–20% saving in total cost.

6.3.2 Read Tail Latency. TeksDB eliminates duplicated keys in the SkipList using a selectively in-place update technique, which performs in-place update if there is no outstanding readers, thereby supporting MVCC. However, this technique can delay a read request because if the SkipList

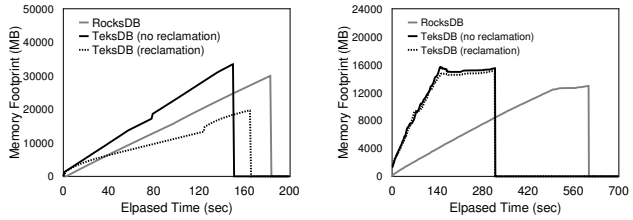


Fig. 8. Memory Footprint.

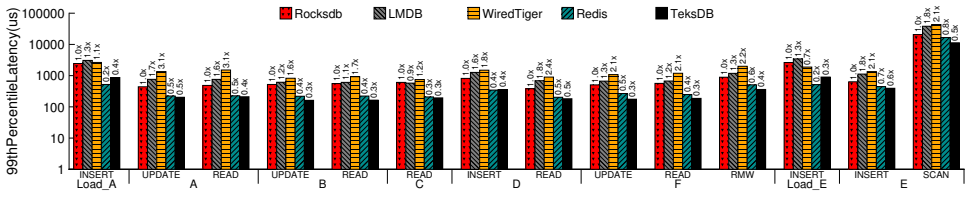


Fig. 9. YCSB Operation 99th Latency with Fit-in-memory Dataset.

is in an in-place update mode when a read request arrives, the read request should wait until the in-progress update is completed and a SkipList turns into the append-only mode. To examine an effect of this weakness, we plot the tail latency of KVS for all workloads in YCSB. Figure 9 shows the 99th percentile latency of each operation across the workloads in a log scale.

Our experiments however reveal that TeksDB provides a excellent read tail latency, even shorter than other key-value stores. This positive result is due to the fact that the proposed selectively in-place update SkipList resorts back to the native append-only mode in the presence of concurrent reads and writes. With this property, only the first read arriving in the in-place update mode is subject to blocking, the number of which is small. Furthermore, we optimize the SkipList update mechanism through concurrent updates using waiting threads, fast seeking with a shortcut pointer, and lock-free implementation of a wait queue, subsequently achieving the cost effectiveness of TeksDB.

7 EVALUATION WITH STORAGE COMPONENTS

To understand the effectiveness of TeksDB in environments where the underlying storage components are involved, we carry out experiments with a dataset larger than the memtable size (16GB). Once the memtable becomes full, the in-memory component flushes its data to persistent storage, so that the memtable can accommodate incoming writes. This flush operation not only requires costly I/O operations, but causes expensive compaction operations in the future, thereby having a high impact on the overall performance.

Figure 10 shows the performance of db_bench in TeksDB and RocksDB with large datasets. We initially load 100 millions of insertions, and perform 100 millions of operations in total for each scenario (56GB write). Except for the number of operations performed, all configurations are the same as those used in § 6. Because of a large number of the possible combinations to evaluate

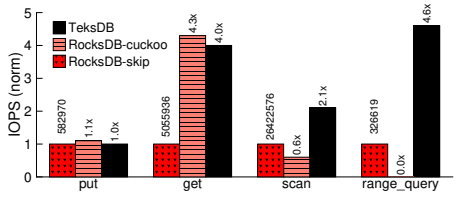


Fig. 10. Db_bench with Large Dataset.

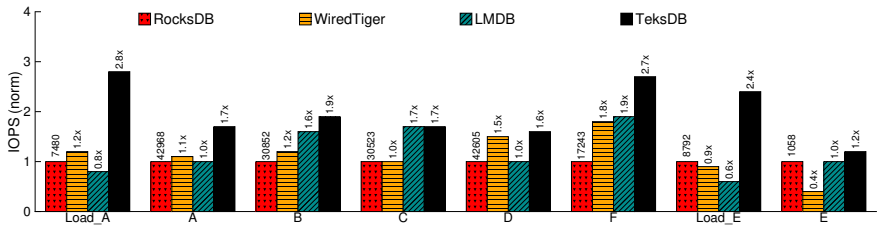


Fig. 11. YCSB Performance with Large Dataset.

and very long execution time, we compare TeksDB with SkipList and CuckooHash only. It is noteworthy that db_bench is not available in WiredTiger and LMDB KVS.

The degree of performance improvement becomes smaller in comparison to the results with fit-in-memory dataset, but TeksDB still performs well. It exhibits the performance comparable to the best-suited structure (CuckooHash) in the put and get scenarios, despite the background SkipList update cost. For the scan and range_query workloads, TeksDB achieves 2.1× and 4.6× higher IOPS than the best-performing SkipList.

Figure 11 shows the performance of YCSB workloads with a larger dataset than a memtable. In this setting, we compare the performance of TeksDB with three popular KVS implementations, WiredTiger [40], LMDB [24], and RocksDB, which all can be used as a backend storage engine for Ardb. We do not consider Redis for this configuration because it is essentially an in-memory KVS and it has no storage-side components. For the fair comparison, RocksDB, WiredTiger, and LMDB are set to use the same amount of memory as TeksDB for their memtable. We initially perform 30 millions of insertions (i.e., Load A and Load E) and then run 1 million of operations. Because each insertion stores an item with a 16B key and 11 100B fields with a few bytes of metadata, each dataset has 46GB data footprint.

TeksDB outperforms all other KVSs across all workloads. It improves IOPS by 1.2–2.8× than RocksDB, and performs better than WiredTiger and LMDB by 1.96× and 1.91× respectively. LMDB provides decent performance in read-intensive workloads, but it performs poorly under write-intensive workloads (i.e., Load A and Load E). LMDB uses B+tree, so the frequent split and merge operations deteriorate the write performance. In contrast, WiredTiger uses an LSM-tree as a persistent data structure, which is optimized for writes, and it shows better performance than LMDB under Load workloads.

In summary, the fast in-memory data structure of TeksDB still achieves performance benefits in more realistic circumstances where underlying storage components are involved, albeit at a lesser degree.

8 RELATED WORK

This paper builds upon several prior work done on key-value stores (KVS). In this section, we briefly outline them and discuss our work in relation to them.

Scaling Key-Value Store: cLSM [17] is a log-structured merge tree (LSM)-based design that focuses on increasing concurrency. It achieves scalable concurrency by reducing contention between in-memory updates (PUT) and persistent storage updates (*compaction*) with the use of reader-writer locks. In our work, we inherit cLSM's approach of removing the global mutex lock, but extend it further by completely redesigning the in-memory component. We achieve this by quantitatively evaluating the performance of existing data structures for various workloads, and integrating complementary data structures for optimal performance.

Similar to our observations, FloDB [4] identifies that the in-memory component of LSM-based KVS can potentially be the performance bottleneck in large memory environments. It addresses this challenge by adding a small in-memory buffer layer on top of the sorted memory component. By hierarchically organizing the in-memory data structures, however, updates are essentially written to both components, and in-memory lookups can be probabilistically serviced from the slower in-memory data structure. Our approach, on the other hand, organizes the two complementary data structures side-by-side, using the fast cuckoo hash table for point lookups and updates, and the sorted skip list for ranged data retrievals.

Nibble [28] investigates the concurrency of KVS in many-core environments, evaluating it under a 240-core and 12TB memory setting. Its design allows high concurrency by allowing optimistic lookups, retrying if data is updated while reading, and by maintaining a multi-head log to reduce locking contention. However, Nibble is an in-memory data store that requires all data to be resident in memory.

Optimizing KVS Operations: Monkey [11] shows that a LSM-based KVS exhibits tradeoffs in lookups, updates, and memory footprint. It explores the pareto-optimal curve by tuning the amount of memory allocated to Bloom filters across all levels of the KVS. While Monkey studies the inherent tradeoffs of leveling and tiering LSM data structures, we focus on combining two complementary in-memory data structures to provide high performance across a wide range of operations.

TellStore [32] is a distributed KVS with similar motivations to our work: to well-support ranged data retrievals in analytical queries along with simple put and get workloads. TellStore exploits the fact that analytical queries require compact representation of data, and stores data in different data structures (row-major, column-major, log-structured) to take advantage of data locality.

PebblesDB [34] organizes its data using fragmented LSM-trees that allow data in the same level of the persistent storage to overlap. This approach reduces write amplification as it avoids costly data rewrites during compaction. PebblesDB shows excellent random write performance, but it suffers from sorted data retrievals.

9 CONCLUSION

Despite a high growth in memory capacity and data access concurrency, existing in-memory components of KVS are not well-equipped to adapt to these external changes. Our exploratory analysis reveals that there is no one-size-fits-all solution: the existing in-memory components have severe performance slowdown for at least one of the operations in a highly-concurrent and ample memory setting. We present a new KVS called TeksDB that resolves this challenge by seamlessly integrating two complementary data structures—the skip list and the cuckoo hash—so that it performs well across a wide range of operations. We demonstrate that TeksDB achieves comparable or better performance than the state-of-the-art KVS solutions using microbenchmarks and YCSB workloads.

ACKNOWLEDGMENTS

The authors would like to thank our shepherd, Bhuvan Urgaonkar, and the anonymous reviewers for their insightful comments. We also thank Sang Lyul Min for reviewing the early stages of this work. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF-2017R1D1A1B03031494, NRF-2018R1A5A1060031, and NRF-2017R1E1A1A01077410).

REFERENCES

[1] Ardb. 2013. Ardb. https://github.com/yinqiwen/ardb.
[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*. ACM, NewYork, NY, USA, 53–64.
[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. USENIX, Berkely, CA, USA, 363–375.
[4] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. ACM, NewYork, NY, USA, 80–94.
[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, Vol. 24. ACM, NewYork, NY, USA, 1–10.
[6] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.
[7] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. 2015. Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database. In *Proceedings of the 2015 ACM International Conference on Management of Data, SIGMOD Conference 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, NewYork, NY, USA, 251–264.
[8] cameron314. 2014. Concurrent Queue. https://github.com/cameron314/concurrentqueue.git.
[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26.
[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. ACM, NewYork, NY, USA, 143–154.
[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, NewYork, NY, USA, 79–94.
[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. ACM, NewYork, NY, USA, 205–220.
[13] Facebook. 2012. RocksDB. https://github.com/facebook/rocksdb.
[14] Facebook. 2018. MyRocks. https://myrocks.io.
[15] The Apache Software Foundation. 2008. Cassandra. https://github.com/apache/cassandra.
[16] FoundationDB. 2013. FoundationDB. https://www.foundationdb.org/.
[17] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. ACM, NewYork, NY, USA, 32:1–32:14.
[18] Google. 2011. LevelDB. https://github.com/google/leveldb.
[19] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S. Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS under HBase: a facebook messages case study. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*. USENIX, Berkely, CA, USA, 199–212.
[20] HyperDex. 2011. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX, Berkely, CA, USA, 993–1005.

[22] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. 2017. Understanding write behaviors of storage backends in Ceph object store. In *Proceedings of the 2017 International Conference on Massive Storage Systems and Technology*. Santa Clara University, Santa Clara, CA, USA.

[23] Eunji Lee, Youil Han, Suli Yang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. How to Teach an Old File System Dog New Object Store Tricks. In *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX, Berkely, CA, USA.

[24] LMDB. 2011. LMDB. https://symas.com/lmdb/.

[25] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the Design and Scalability of Distributed Shared-Data Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, NewYork, NY, USA, 663–676.

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.

[27] Memcached. 2003. Memcached. https://memcached.org.

[28] Alexander Merritt, Ada Gavrilovska, Yuan Chen, and Dejan Milojicic. 2017. Concurrent log-structured memory for many-core key-value stores. *Proceedings of the VLDB Endowment* 11, 4 (2017), 458–471.

[29] Inc. MongoDB. 2018. MongoDB. https://www.mongodb.com.

[30] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385.

[31] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX, Berkely, CA, USA, 251–264.

[32] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast Scans on Key-Value Stores. *PVLDB* 10, 11 (2017), 1526–1537.

[33] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.

[34] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, NewYork, NY, USA, 497–514.

[35] Redis. 2009. Redis. https://redis.io.

[36] Margo Seltzer and Keith Bostic. 1994. Berkeley DB. http://https://www.oracle.com/database/berkeley-db/index.html.

[37] TokyoCabinet. 2009. TokyoCabinet. http://fallabs.com/tokyocabinet/.

[38] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. 2018. ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications. *PVLDB* 11, 10 (2018), 1137–1150.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. USENIX, Berkely, CA, USA, 307–320.

[40] WiredTiger. 2016. WiredTiger. http://www.wiredtiger.com/.

Received October 2018; revised January 2019; accepted February 2019