JellyFish: A Fast Skip List with MVCC

Jeseong Yeon*
Line Corporation

Leeju Kim
Soongsil University

Youil Han*
Line Corporation

Hyeon Gyu Lee
Seoul National University

Eunji Lee
Soongsil University

Bryan S. Kim
Syracuse University

Abstract

Multi-version concurrency control is a widely employed concurrency control mechanism, as it allows non-blocking accesses while providing isolation among transactions. However, maintaining multiple versions increases the latency for both point lookups and ranged retrievals because of the overhead in finding the right version. In particular, the *append-only* skip list—widely used in the state-of-the-art key-value stores (KVS)—shows a significant performance degradation due to its append-only nature.

This paper presents a novel skip list implementation called *JellyFish*. JellyFish reduces the overhead of multi-version concurrency control by separating the per-key updates from the key indexing. We implement our design on top of RocksDB and compare it against a wide variety of data structures. Our evaluation with micro-benchmarks and real-world workloads show that we not only improve the throughput by up to 93%, but also reduce the latency of update operations by up to 42%.

ACM Reference Format:
Jeseong Yeon*, Leeju Kim, Youil Han*, Hyeon Gyu Lee, Eunji Lee, and Bryan S. Kim. 2020. JellyFish: A Fast Skip List with MVCC. In *Proceedings of ACM Conference (Middleware '20)*. ACM, New York, NY, USA, 15 pages.

1 Introduction

A skip list [54] is a popular data structure used in modern computing, from data store applications [3, 20, 45], to distributed applications [24, 57], and even task schedulers [58]. Evolved from the linked list, the skip list achieves a probabilistic $O(logN)$ time insertion and retrieval by additionally maintaining multiple index pointers over the data layer. As opposed to the binary search tree, the skip list does not require any rebalancing operations and thus has a much simpler design.

Although there are many implementation flavors and variants of the skip list [16, 18, 25, 29, 38], those used in data store applications (i.e., databases and key-value stores) distinctively maintain data in an *append-only* manner [20, 27, 33]. This means that the skip list never overwrites existing data: instead, all mutations are processed as insertions with a different timestamp. This design decision makes it easy to support multi-version concurrency control (MVCC) [9, 10] that enables all readers to see a point-in-time consistent view during concurrent writes. Because MVCC provides high concurrency without violating data consistency, it is adopted in most of the commercial databases and key-value stores [2, 19, 34, 44–46, 50].

However, the append-only implementation imposes non-trivial performance penalties for the skip list, especially as memory capacity grows. First and foremost, because multiple versions of the data collectively exist in the list, the performance for traversing the dataset becomes slower, even though the list is maintained in a sorted order. To get to the next key from the previous, the reader has to iterate through all of the obsolete versions in-between, performing key comparisons. This significantly degrades the performance of the scan operation that is widely used in analytical workloads and real-time databases [51, 52]. Secondly, allowing multi-versions requires a search through the bottom-most layer of the skip list because the key found through the middle layer is not guaranteed to be the latest and visible version. Lastly, each insertion requires multiple pointer updates through atomic compare-and-swap operations. The number of atomic pointer updates required scales with the total size, as nodes have more layers in larger skip lists.

These performance limitations exacerbate as the size of the skip list increases. Even though RocksDB, by default, sets the size of the skip list to only 64MB, applications that use it typically override to a much larger size [4, 22, 30]. For example, Ceph sets it to 256MB to maintain not only the metadata (a few hundreds of bytes) but also the small-sized objects (a few KB) [30]. Furthermore, with the advent of new memory technologies such as Optane [35], we expect the performance scalability of in-memory data structures to become more important.

* This work was done while at Soongsil University.

Middleware '20, December 2020, DELFT, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
Doi: 10.1145/3423211.3425672

With this in mind, this paper presents a new design of the skip list called *JellyFish*. JellyFish is a skip list implementation that supports MVCC and point-in-time consistency without the drawbacks of the append-only design. Instead of horizontally inflating the data structure, it inserts new updates into a separate per-key list that grows vertically. This design addresses the limitations of the legacy append-only design.

Coincidently, X-Engine [32], a write-optimized storage engine in Alibaba's e-commerce platform, includes a component similar to our work. To achieve high transactions per second under high concurrency, it also maintains a per-key list so that point lookups are fast regardless of data temperature. However, our work independently makes contributions as (1) our design is publically available, (2) our implementation is uniquely different from that of X-Engine, and (3) our evaluation focuses on tradeoffs among data structures without specialized hardware[1].

The main contributions of this paper are:

- We identify the performance penalties for maintaining MVCC in state-of-the-art skip lists, and present a novel skip list implementation called JellyFish that relieves this overhead while supporting multi-versioning.
- We implement JellyFish on top of RocksDB 6.8.1 with about 1K LOC modification. Our implementation is available at http://github.com/jsyeon92/jfdb.
- We evaluate our design and other production-level key-value stores under micro-benchmarks and real workloads. In particular, JellyFish improves the throughput of range query by 5.4× and the throughput of all YCSB workloads, as much as 93%.

The remainder of this paper is as follows. Section 2 presents the background for this study and Section 3 details the structure of JellyFish. Section 4 evaluates JellyFish under various workloads, and Section 5 discusses topics regarding the memory usage and write amplification of our design. Section 6 briefly highlights the related work, and Section 7 concludes the paper.

2 Background

In this section, we give a brief overview of multi-version concurrency control (MVCC), MVCC's implementation in RocksDB, and how the skip list data structure is used in modern key-value stores (KVS).

2.1 Multi-Version Concurrency Control

MVCC is a widely used concurrency control method in databases and key-value stores. In contrast to the locking-based approach, MVCC allows for simultaneous puts and gets through *snapshot isolation* [9]. This isolation level ensures that a transaction T sees the database state as produced by all the transactions that have committed before T starts, but no effects are seen from transactions that overlap with T. By so doing, snapshot isolation never suffers from inconsistent reads.

Modern KVSs use a timestamp to implement MVCC: each write is tagged with a monotonically increasing sequence number and this timestamp is used to retrieve the appropriate version of data upon a read. As an example, when the data is retrieved, the latest version of data with a timestamp younger than the requested version (timestamp) is returned. The ongoing transactions are invisible and cannot be requested; the atomicity of a transaction and the consistency of the KVS are ensured. This isolation level avoids undesired phenomena such as dirty reads and phantom reads without the performance overheads associated with full serializability [9, 60].

2.2 MVCC in RocksDB

RocksDB [20] supports MVCC with a batched commit approach [23]. Naïvely, serializing each write request in timestamp order hampers write performance significantly. On the other hand, out-of-order processing of concurrent writes may result in a younger transaction being visible while an older transaction is uncommitted, violating the data consistency of MVCC.

In a batched commit, multiple put requests are batched into a group and concurrently written. During a batched write, none of them are exposed to readers. Once all of the write requests in a group are completed, RocksDB increments the committed timestamp to the highest timestamp of transactions in a group. All readers retrieve the most recent data older than the committed timestamp. Through this, RocksDB achieves read-write concurrency while guaranteeing that readers will never see half-written or inconsistent pieces of data. This also serves to support a transaction of compound requests.

Figure 1 illustrates how RocksDB handles concurrent requests. Three put requests, (Bob, AZ), (Bob, CA), and (Bob, NY) are written concurrently as a single batch. A get request during this batched write accesses the skip list with the last committed timestamp of 3. Hence, it would return (Bob, TX) of timestamp 2 even though more up-to-date versions may have been written.

Another challenge for supporting MVCC is the timely reclamation of space allocated to old versions. The append-only update mechanism necessarily results in multiple versions of data, and the space occupied by outdated versions need to be reclaimed eventually. To this end, we need to ensure that no readers are accessing the old version marked for reclamation, otherwise read consistency can be violated [36].

RocksDB reclaims the memory space allocated to old versions when it flushes the skip list (also known as memtable) to persistent storage. The in-memory data and persistent data combined are organized as a log-structured merge-tree (LSM-tree) [49].

When the memtable becomes full, RocksDB makes it immutable and creates another memtable to service incoming requests. The immutable memtable is flushed to storage in the

[1]X-Engine is assisted by FPGA-acceleration.

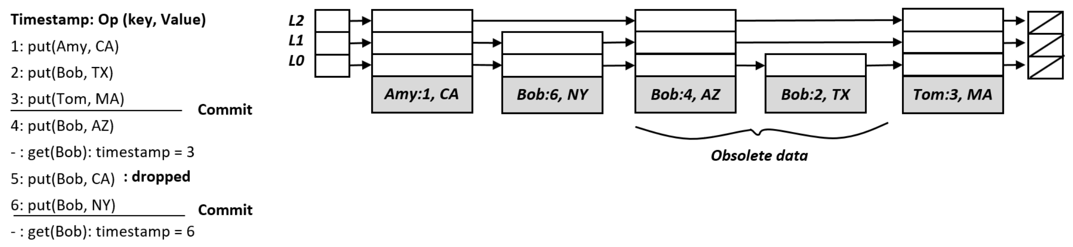


Figure 1. Skip List in RocksDB. The list of transactions on the left shows the given operations in the time order. The skip list shows the append-only and concurrent version once all operations complete, assuming that the node's maximum height is 3.

background. If multiple versions are found for the same key, RocksDB writes only the most up-to-date version to persistent storage, discarding the old versions. To avoid consistency violations during this merge operation, RocksDB allows readers to explicitly make a snapshot. The snapshotted versions are internally maintained and the data comprising them are preserved during the flush.

2.3 Skip List in KVS

A skip list [54] is widely used in modern key-value stores (KVS) to maintain a sorted order of keys in memory. For such purposes, the original skip list data structure is augmented to include the value that the key is associated with. If the skip list operates in an append-only manner for MVCC, the data structure also maintains timestamps to differentiate the distinct versions.

Table 1 describes the characteristics of some of the existing skip lists used in modern KVSs.

LevelDB [27] and RocksDB [20] both use the append-only skip list: an update is handled as an insertion with the combination of key and timestamp. Hence, both support MVCC through snapshot isolation. LevelDB uses a condition variable to serialize the write requests, while RocksDB improves this limitation by introducing the lock-free skip list. RocksDB-IN indicates RocksDB with in-place update option enabled. This setting overwrites the value upon an update if the updated value size is smaller than or equal to the existing one. Because overwriting value is not thread-safe, RocksDB-IN does not allow concurrent writes to the skip list.

KVS	UpdateMechanism	NS	CW	SI
LevelDB	Append (horizontal)	Array	✖	✓
RocksDB	Append (horizontal)	Array	✓	✓
RocksDB-IN	In-place (overwrite)	Array	✖	✖
Cassandra	In-place (swapping)	List	✓	✖
X-engine	Append (vertical)	List	✓	✓
JellyFish	Append (vertical)	Array	✓	✓

Table 1. SkipList Taxonomy. NS - Node Structure, CW - Concurrent Writes, SI - Snapshot Isolation.

Cassandra implements the in-memory data structure using ConcurrentSkipListMap class provided by OpenJDK [25, 29, 38]. Upon an update, this skip list allocates space for value and swaps the old one with a new one through the compare-and-swap operation. This behavior allows concurrent writes but does not support MVCC.

Both the proposed work and X-Engine [32] enhance the skip list data structure to reduce the overhead of MVCC under highly concurrent workloads. JellyFish handles an update as an insertion supporting MVCC, but it inserts updates into the per-key list, rather than the skip list, thereby eliminating the overhead of traversing redundant data copies. Similarly, X-Engine maintains a skip list for distinct keys, but each key-value pair is a linked list that collectively makes up the skip list[2]. Although the skip lists in X-engine and JellyFish have a similarity in that both maintain updates as a per-key vertical list, our design differs from X-engine in that it has an efficient node structure and applies a set of optimization techniques which becomes possible only when keys are distinct.

3 The Design of JellyFish

JellyFish is a fast and consistent implementation of a skip list. It allows concurrent writes while also providing a point-in-time consistent view known as snapshot isolation. In the following subsections, we explain how KVS operations work with JellyFish in contrast to that found in RocksDB. We use the term AOC to refer to the **a**ppend-**o**nly and **c**oncurrent implementation of the skip list. As we discuss the design of JellyFish, we also present simulation results of our design along with 5 other implementations from Table 1. This is to isolate the performance of the data structure from the features and compatibility issues across different production-level KVS. Detailed evaluation based on real implementation will follow after this section. We model a KVS as a single-layer skip list that provides KVS operations without any storage components. This simulator is built based on the memtable of RocksDB and modified it according to the target skip list design.

[2]The source code for X-Engine is publicly unavailable, and we interpret its internal implementation based on its publication [32].

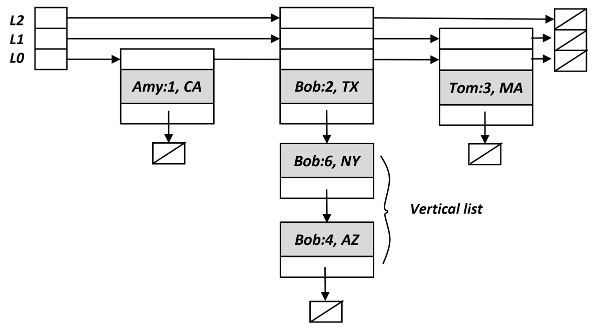


Figure 2. JellyFish Skip List. The state of JellyFish when performing the same operations in an example of Figure 1.

We measured the performance running one million of operations with 16B key and 100B value for each scenario and performed each run ten times and report the average values. We evaluate performance for two workloads generated by different distribution functions of the keys. The uniform distribution function issues distinct keys without any overlaps between operations. The Zipfian distribution function generates the skewed workloads in which a small dataset constitutes a large portion of data accesses, yielding a large number of updates. We generate the Zipfian workload using the numpy package in python3.7 [56]. In the numpy package, the skewness of popularity can be adjusted with a parameter that is larger than 1.0, and we use the parameter 1.2 for the conservative performance evaluation. Note that the greater value the parameter has, the more skewed workloads are created with more updates. Because these two workloads are each disadvantageous (uniform) and advantageous (Zipfian) for JellyFish, we can observe the downside and upside of JellyFish.

3.1 PUT

The put operation stores the key, value, and the system-internal timestamp into the skip list. To support MVCC, data are never overwritten, and new nodes are allocated for all put operations. The height of the node (or the number of links the node has) is determined at this point following a geometric progression: the bottom-most layers are always linked, and each upper layer is less likely to be connected (typically at a 25% ratio) [53].

Insertion: If the key does not exist in the skip list, JellyFish and AOC behave identically: the node is inserted at the correct sorted location with all of its links connected. Note that this involves multiple pointer updates as the node has multiple layers. RocksDB updates them starting from the bottom-most layer and upwards, using a series of atomic compare-and-swap operations for concurrency [25, 29].

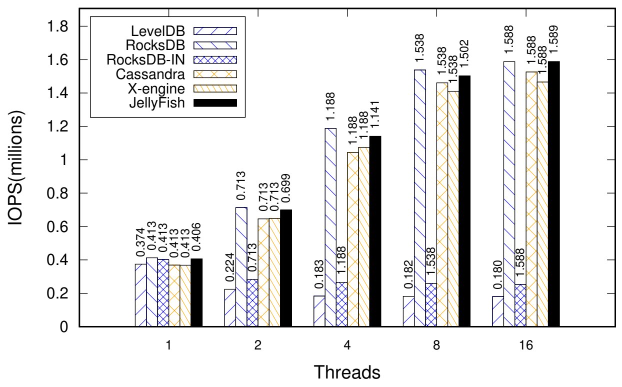
Update: On the other hand, if the key does exist in the skip list, JellyFish and AOC behave differently. AOC inserts the new node as if it is a new key. JellyFish, however, allocates a vnode and adds it to the vertically growing list (akin to jellyfish tentacles). A vnode is smaller than the nodes in the horizontal skip list as it does not need all the index pointers for the multiple layers. When adding into the vertical list, the items are inserted such that the most recent timestamp appears first; this order, however, excludes the very first version that is part of the horizontal list.

Algorithm 1 PUT

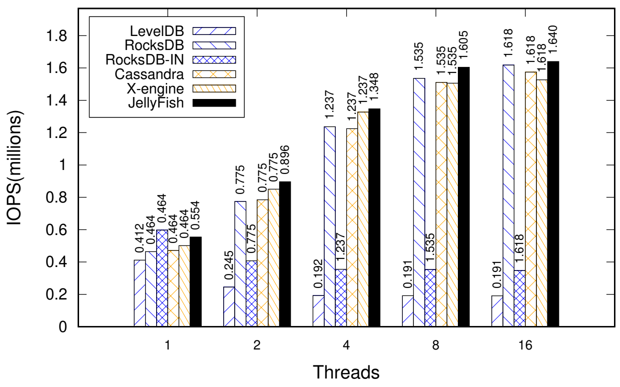
```
   Input: key K, value V, timestamp T
1: procedure PUT($K, V, T$)
       ▷ Initialize
2:     $nnode \leftarrow newNode(K, V, T)$
       ▷ Search the skip list
3:     $node = find\_location(K, prev, next)$
       ▷ Insert new key-value
4:     if $node.key \neq K$
5:         $node.init(next)$
6:         $L \leftarrow 1$
7:         if $\neg CAS(prev[L], nnode.next[L], nnode)$
8:             goto search
9:         $L \leftarrow L + 1$
10:        while $L \leq H_{nnode}$ do
11:            if $\neg CAS(prev[L], nnode.next[L], nnode)$
12:                $link\_update(prev, next, L)$
13:                continue
14:            $L \leftarrow L + 1$
15:        return
       ▷ Update existing key-value
16:    $vnode \leftarrow newVode(K, V, T)$
17:    $vl \leftarrow \&node.vl$
18:    do
19:        $vnode.next \leftarrow node.vl$
20:        if $vnode.next$ and $vnode.T < vnode.next.T$
21:            return
22:    while $\neg CAS(node.vl, vnode.next, vnode)$
23:    return
```

If the existing most recent timestamp is *younger* than the timestamp to add, that put operation is dropped as it holds a timestamp that no readers will see. This may happen if multiple put operations of the same key write concurrently. As shown in Figures 1 and 2, the put for Bob at timestamp 5 can be safely aborted without violating consistency for both AOC and JellyFish.

Algorithm 1 shows the pseudocode of the put operation in JellyFish. The subroutine in Line 3 traverses the skip list and searches for the location where the given data (a pair of key and value) should be inserted. Line 8 records the information of predecessors and successors of Lines 4 through 15 insert a new node into the skip list when the requested key does not exist in the skip list. If the pointer update fails at the bottom layer, the put procedure resumes from the search



(a) Uniform Distribution



(b) Zipfian Distribution

Figure 3. Put performance

step; otherwise, it updates the index links in an ascending order to ensure data consistency in the concurrent updates. If the link update fails, the put operation searches the changed predecessors and successors and retries the compare-and-swap operation with the updated value. Lines 16 through 23 handle the case where an existing key is updated.

Figure 3 shows the skip list performances for put operation. JellyFish performs better than RocksDB by up to 19% and 10% on average under the Zipfian workload. For the skewed workloads, JellyFish not only reduces the number of CAS operations using the vertical list, but also enhances a search process by ensuring that keys are distinct. When the workload has no update, JellyFish is slightly slower than RocksDB (within 3%) because JellyFish checks the per-key list status to determine where to put a new node: the skip list or the vertical list. However, this cost is marginal. JellyFish performs better than the other designs that also handle updates with a single CAS operation for all workloads, delivering 8.7% and 7% higher IOPS than Cassandra and X-engine on average. This difference comes from the node structure. X-engine and Cassandra use a linked list to maintain pointers for each node, while JellyFish, LevelDB, and RocksDB use a pointer array for each node. traversing a linked list requires more memory references than accessing an array. Another noticeable result for the put operations is that the non-blocking skip lists outperform the blocked skip lists (LevelDB and RocksDB-IN), providing 8.8× higher IOPS than the worst-performing skip list.

3.2 GET

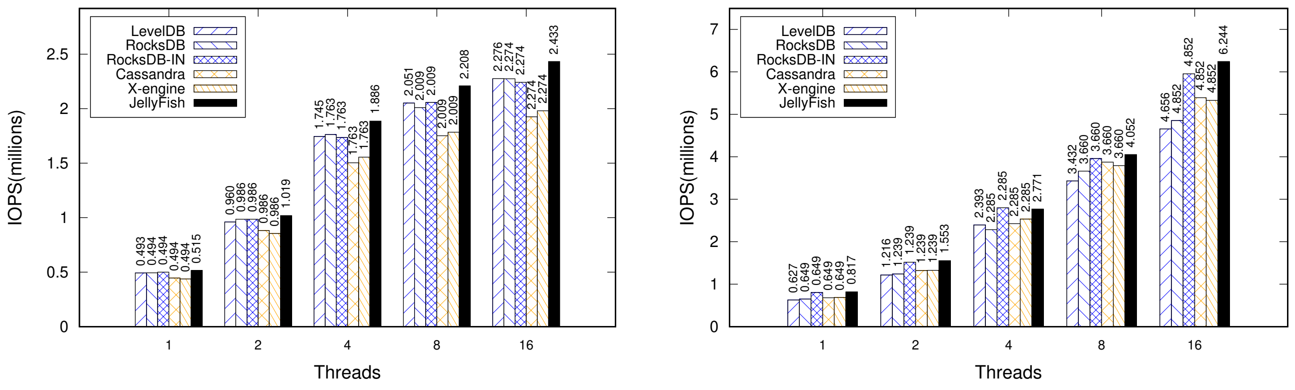
Because each key is unique in the skip list, the design of JellyFish allows for fast retrievals, stopping as soon as the key is found. Once at the correct key, the reader then investigates the vertical per-key list to retrieve the most recent (yet older than its timestamp) value. The values are sorted in recency to make retrievals fast.

For the AOC implementation, however, the readers have to investigate the key and the timestamp within the same horizontal structure. Even though the values are also sorted in recency so that newer values appear first, the height of each node is probabilistically determined. Thus, even if the key is found by traversing the faster *upper* layers, that timestamp may be stale and the reader has to track back and traverse to the bottom-most layer.

Algorithm 2 shows the pseudocode of the get operation in JellyFish. At line 5, JellyFish can stop the search if the requested key is found because it is the only one, while AOC should go through to the bottom-most layer to check for the

Algorithm 2 GET

```
    Input: key K
    Output: value V, timestamp T
 1: procedure GET($K$)
 2:     $node \leftarrow ListHead$
 3:     $L \leftarrow H_{MAX}$
 4:     while $L \geq H_{MIN}$ and $node$ do
 5:         if $node.key = K$
 6:             if $\neg node.vl$ and $node.T \leq T_{commit}$
 7:                 return $node.V, node.T$
 8:             $vnode \leftarrow node.vl$
 9:             do
10:                 if $vnode.T \leq T_{commit}$
11:                     return $vnode.V, vnode.T$
12:                 $vnode \leftarrow vnode.next$
13:             while $vnode$
14:         $node \leftarrow node.next[L]$
15:         if $node = null$ or $node.key > K$
16:             $L \leftarrow L + 1$
17:         else if $node.key < K$
18:             $node \leftarrow node.next[L]$
19:     return $null$
```



(a) Uniform Distribution **(b)** Zipfian Distribution

Figure 4. Get performance

timestamp. Line 6 to 7 retrieve the requested value from the skip list node, while line 8 to 13 searches the per-key vertical list to find the proper version.

Figure 4 compares the get performance of various skip lists. For the get operations, JellyFish outperforms other skip lists for all workloads. JellyFish (1) has no obsolete data, (2) obviates a search through the bottom-most layer, and (3) uses an array-based node structure. The other data structures fall short in at least one of the three above design aspects. It is noteworthy that the impact of node structure becomes larger in the get operations than that in the put operations. The array-structured node performs better than the list-structured node by up to 26%. For writes, the performance is affected by other factors such as the number of CAS operations and cache coherence overhead. However, in reads, the performance is more affected by memory accesses, and thus, the increase in memory reference count has more impact on the get operations.

3.3 Scan

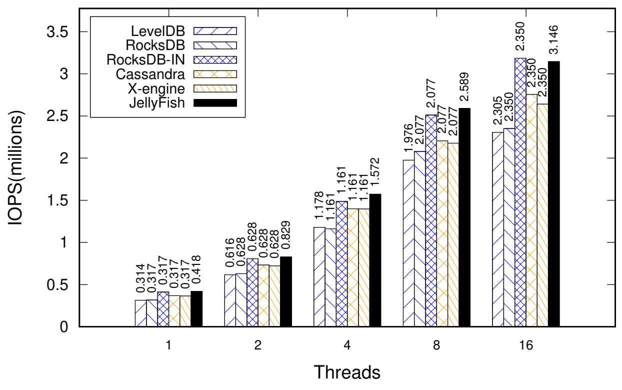
The design advantages of JellyFish are further amplified when handling a scan operation, which is generally triggered by range query. A scan retrieves *n* key-values starting from the start key. In the AOC design, the scan operation exhibits a very poor performance as it needs to traverse through the bottom-most linked list layer from the start key. In the example of Figure 1 and 2, an AOC implementation would need to go through all 5 nodes for querying 3 key-value pairs, starting from the key *Amy*. JellyFish, on the other hand, nearly halves this as all keys are unique in the skip list.

Algorithm 3 shows the pseudocode of the scan operation in JellyFish. It first seeks to the starting key by searching the skip list. The scan returns the next *n* distinct key-value pairs from the starting key. Visiting each node, JellyFish retrieves the appropriate value either from the skip list or from the vertical list (lines 4 through 14). In the AOC skip list, every node requires key comparison because there are redundant data. JellyFish does not need it because nodes in the skip list have distinct keys.

Algorithm 3 SCAN

```
    Input: key K,   items N
    Output: value V,  timestamp T
 1: procedure SCAN(K,N)
 2:     $vset \leftarrow \{\}$ :
 3:     $node \leftarrow SEEK(K)$ :
 4:     while $node$ and $N > 0$ :
 5:         if $!node.vl$ and $node.T <= T_{commit}$ :
 6:             $vset.add(node.key, node.value)$
 7:             continue :
 8:         $vnode \leftarrow node.vl$
 9:         do :
10:             if $vnode.T <= T_{commit}$ :
11:                 $vset.add(vnode.key, vnode.value)$
12:             $vnode \leftarrow vnode.next$
13:         while($vnode$) :
14:         $N \leftarrow N - 1$
15:     return $vset$
```

Figure 5 graphs the scan performance of the simulator for uniform and Zipfian workloads. The scan operation seeks to the start key and reads ten next distinct keys. The scan operation under Zipfian distribution workloads achieves the most significant performance improvement in JellyFish, compared to RocksDB. The scan of JellyFish is 170x faster than that of RocksDB. JellyFish also outperforms X-engine by 16% and 15% in uniform and Zipfian workloads. This gain is achieved by the optimized seek: array-based node structure and fast return in point look-ups.


(a) Uniform Distribution

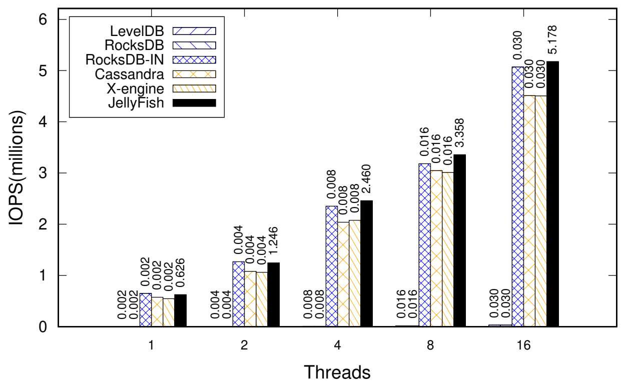

(b) Zipfian Distribution

Figure 5. Scan performance

4 Evaluation

We implement JellyFish on top of RocksDB 6.8.1, replacing the default append-only and concurrent skip list—about 1K lines of code have been modified. We call this resulting implementation JFDB. We compare JFDB against four production-level key-value stores that support MVCC: LevelDB [27], WiredTiger [47], LMDB [40], and RocksDB [20]. For a fair comparison, we do not evaluate MVCC-noncompliant systems such as Redis and Cassandra: Redis uses a hash table and allows only a single writer, and even though Cassandra uses a skip list, its hybrid amalgam with B-tree ultimately does not support snapshot isolation. However, we do include an additional version of RocksDB that performs in-place updates (denoted as RocksDB-IN). Although RocksDB-IN does not support MVCC nor implement concurrent writes, it lacks performance drawbacks associated with inflated skip lists.

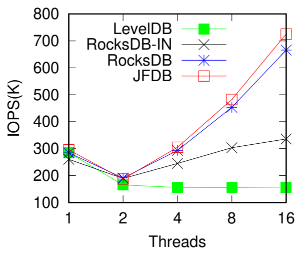
For evaluation, we use both micro-benchmarks and real-world workloads. The performance evaluation has been conducted on the same configuration described in the previous section: 18 physical cores, 72GB of memory, and 256GB of NVMe storage (see § 3).

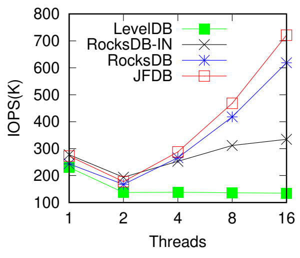
4.1 db_bench Results

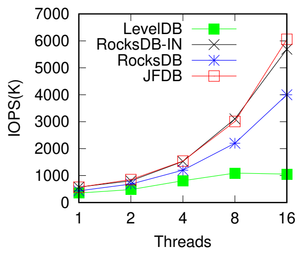
We measure the performance of JFDB using the db_bench benchmark (native to LevelDB and RocksDB). We use the following four microbenchmarks in db_bench: `fillrandom` that inserts items with random keys for each thread; `overwrite` that updates existing values in random order; `readrandom` that retrieves items randomly; and `seekrandom` that performs random seeks and queries the next ten items.

In order to highlight the performance of the in-memory data structure, we avoid the activation of flush (persisting data to storage) and compaction (merging persistent data) by using a large memtable size of 10GB. We first load the key-value store with one million entries of 16B-sized keys and 100B-sized values, and then execute the following workload, one million operations each in the following order: (1) put using `fillrandom` for insertions, (2) put using `overwrite` for updates, (3) get using `readrandom`, and (4) range query using `seekrandom`.

Figure 6 compares the performance of JFDB to that of LevelDB and two versions of RocksDB. The code path for handling put with one thread is different from that with two


(a) Put (fillrandom)


(b) Put (overwrite)


(c) Get (readrandom)

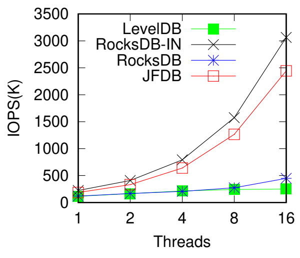
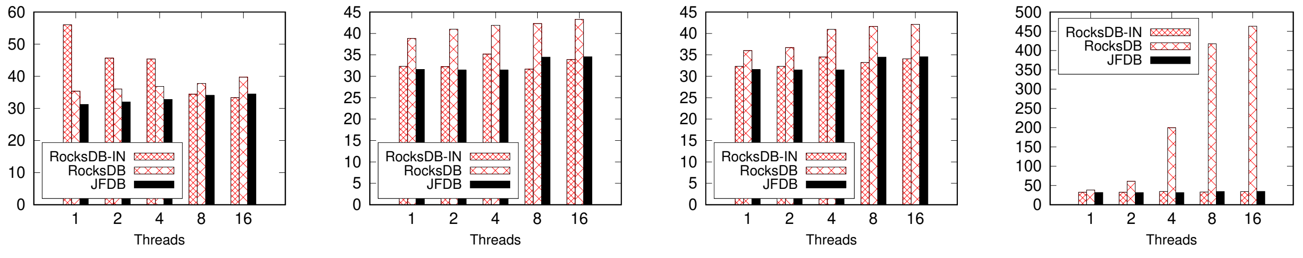
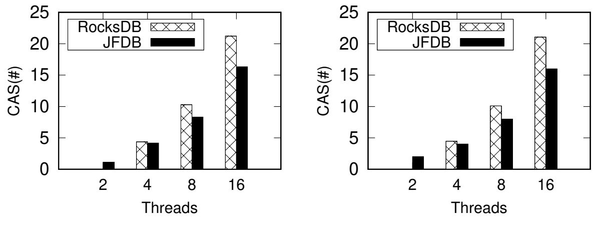

(d) Range Query (seekrandom)

Figure 6. Micro-benchmark throughput with 100B value size.



(a) Put (fillrandom) **(b)** Put (overwrite) **(c)** Get (readrandom) **(d)** Range Query (seekrandom)

Figure 7. Number of key comparisons. The number of key comparisons (in millions) is measured for RocksDB and JFDB using the statistics profiler. LevelDB does not report this.



(a) Put (fillrandom) **(b)** Put (overwrite)

Figure 8. Number of CAS operations. The number of CAS operations (in millions) are measured for RocksDB and JFDB while varying the number of threads. CAS operations are not used when running with a single thread.

Workload	Ref(R)	Miss(R)	Ref(J)	Miss(J)
fillrandom	29,831	4,125	21,986	2,731
overwrite	38,272	5,810	20,949	2,727
readrandom	9,238	4,368	4,844	1,938
seekrandom	38,732	29,529	8,389	4,092

Table 2. Number of cache references and misses. Ref(R) is the number of cache references when running RocksDB; **Miss(R)**, the number of cache misses with RocksDB; **Ref(J)**, references with JFDB; and **Miss(J)**, misses with JFDB.

threads: this causes the performance drop when increasing the number of threads from one to two.

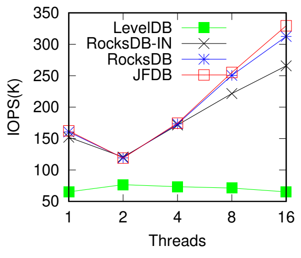
The micro-benchmark results reveal that, at a high degree of concurrency (16 threads), JFDB outperforms RocksDB by up to 8%, 16%, 51%, and 545% for insertion, update, get, and range query, respectively. These performance benefits are thanks to fewer pointer updates for put operations and faster search for the key. To better understand the results, we measure the number of key comparisons (Figure 7), the number of compare-and-swap (CAS) operations (Figure 8), and the number of cache misses (Table 2).

For the number of key comparisons (Figure 7), JFDB has 13%, 20%, 18%, and 90% fewer comparisons than RocksDB for the four micro-benchmarks, respectively. The large reduction in key comparisons for retrievals highlights the design advantages of JellyFish. As expected, JFDB has a similar number of key comparisons to that of RocksDB-IN for most of the workloads. However, RocksDB-IN records an unusually high number of comparisons for Figure 7a (`fillrandom`). In its implementation, RocksDB-IN first searches for the random key, and if not found, then searches again to insert the key. This quirk mitigates as the number of threads increase, as keys become more likely to be found on the first search.

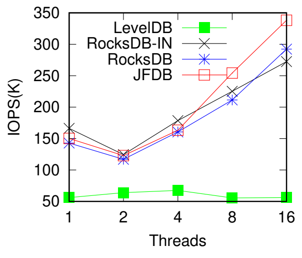
The number of CAS operations is measured for only RocksDB and JFDB because RocksDB-IN and LevelDB perform blocking put and do not use CAS operations (also evident by its lower IOPS in Figure 6). As shown in Figure 8, JFDB reduces the number of CAS operations by up to 23% (for `fillrandom`) and 24% (for `overwrite`) when there are 16 threads. Even though `fillrandom` inserts random keys, there is a higher chance of a key already existing in the skip list as the number of threads increases.

To understand the cache behavior of our design, we measure the number of cache references and misses across all four stages of the micro-benchmark through `perf` [39]. As shown in Table 2, JFDB not only reduces the number of cache references, but also the overall miss rate. RocksDB's large working set size causes a large number of cache misses. JFDB, on the other hand, only accesses unique keys during traversals, resulting in a 57% reduction in cache misses on average. The advantage of a small working set size especially improves the performance of the get operations.

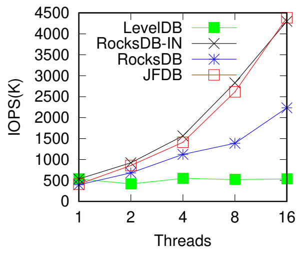
Lastly, we ran the same experiment but with a larger value size of 1KB, as presented in Figure 9. Although modern key-value store workloads are dominated by small values [5, 14], observing the behavior of our design under uncommon workloads may help the development of future applications. There are two main observations when compared to the experiment with a small value size (Figure 6). First, as expected, the overall IOPS drops as the value size increased. Second, while



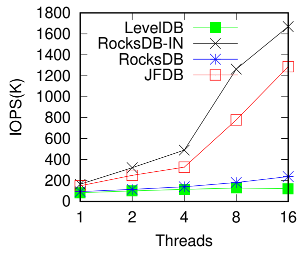
(a) Put (fillrandom)



(b) Put (overwrite)



(c) Get (readrandom)



(d) Range Query (seekrandom)

Figure 9. Micro-benchmark throughput with 1KB value size.

the overall shape for the get operations (`readrandom` and `seekrandom`) are similar, the performance gap between JFDB, RocksDB, and RocksDB-IN has become smaller for the put operations (`fillrandom` and `overwrite`). The limited performance of the put operation for RocksDB-IN under small value sizes was due to locking contention as it performs a blocking put. The scalability limit for RocksDB-IN still exists, but the larger value sizes seemingly hide this as more time is spent transferring data.

4.2 YCSB Results

We investigate the performance of JFDB using YCSB benchmarks [15]. We generate YCSB workloads using Ardb [4]—a Redis-protocol compatible NoSQL system—with JFDB (JellyFishDB) configured as its backend KVS. LevelDB [27], LMDB [40], WiredTiger [47], and RocksDB [20] are also configured as backend stores for comparison. All YCSB experiments are performed with 16 threads. Unlike the micro-benchmark experiments, we configure the memtable to a more realistic size of 256MB for LevelDB, WiredTiger, RocksDB, and JFDB. Thus, our performance results include the overhead of flush (writing from memory to storage) and compaction (re-organizing persistent data). Up to six memtables are allowed before they are flushed, which is the default configuration in Ardb. LMDB, on the other hand, relies on the system-level page cache for buffering and accesses persistent files through the `mmap` interface. For a fair comparison, we limit the memory usage of LMDB to 1.2GB through `cgroup`.

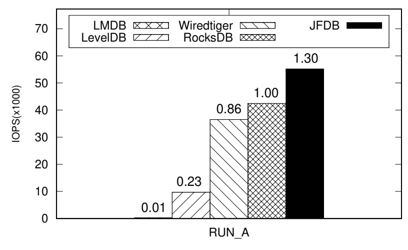
The YCSB benchmark suite has six different workloads and Table 3 summarizes the characteristics of each workload. We initially load one million data items and perform one million operations for each workload. These operations at the Ardb-level translate into a series of put, get, and range query for the KVS backends. Ardb internally maintains two types of key-value pairs: the actual data with ten fields and the system-wide metadata. Thus, the user-level accesses involve both metadata and data accesses, increasing the number of KVS operations.

Figure 10 shows the average IOPS and latency of the workloads. Latency results are normalized to that of RocksDB (with the absolute latency values displayed), and the multiplier for JFDB is relative to the performance of RocksDB, for both throughput and latency. LMDB performs poorly and its latency measurements are not presented.

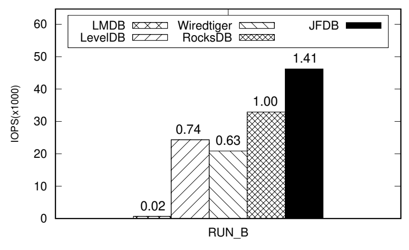
Overall, JFDB outperforms RocksDB by 30%, 41%, and 93% for workloads A, B, and F, respectively, while showing similar performance for the other workloads. The performance benefits of JFDB are threefold: (1) faster range query as the skip list next iteration results in the immediate next key; (2) faster get as the skip list itself is slimmer; and (3) faster put because updates only require a single pointer change. In particular, we observe a large reduction in read latency, as high as 41% for reads in workload F. This is because the user-level RMW(Read-Modify-Write) creates a large number of updates, and the user-level read translates into a metadata get, followed by a range query. The design of JFDB allows for a fast range query under a high number of updates.

Workload	Description	Ratio	PUT	GET	SCAN
A	Session store recording actions	50% Read, 50% Update	1M	1M	0.5M
B	Photo tagging and tag reads	95% Read, 5% Update	0.1M	1M	0.95M
C	User profile cache	100% Read	0	1M	1M
D	User status update	95% Read, 5% Insert	0.6M	1.1M	0.95M
E	Recent post scan in conversation	95% RangeScan, 5% Insert	0.6M	4.9M	4.9M
F	Database	50% Read, 25% RMW, 25% Update	1M	1.5M	1M

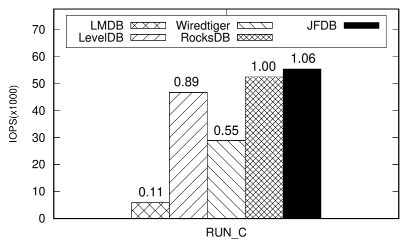
Table 3. YCSB workload characteristics.



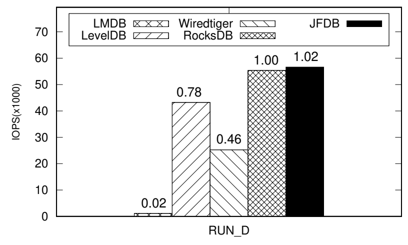
(a) YCSB-A IOPS



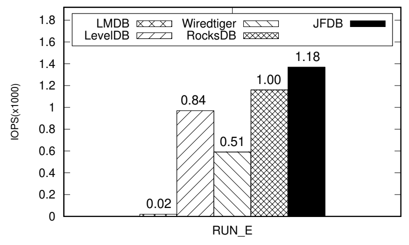
(b) YCSB-B IOPS



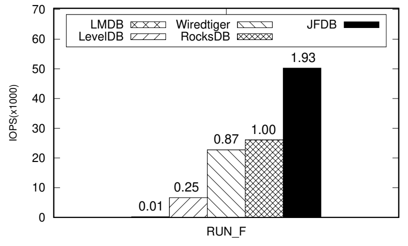
(c) YCSB-C IOPS



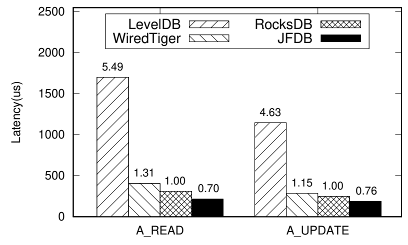
(d) YCSB-D IOPS



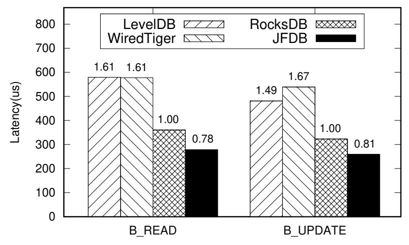
(e) YCSB-E IOPS



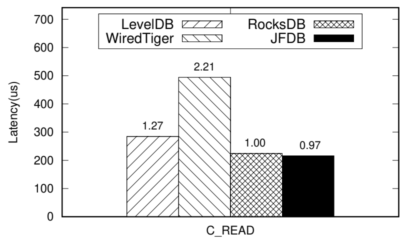
(f) YCSB-F IOPS



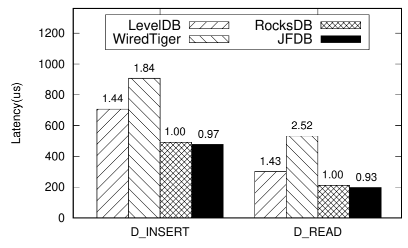
(g) YCSB-A latency



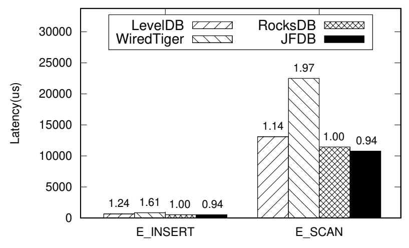
(h) YCSB-B latency



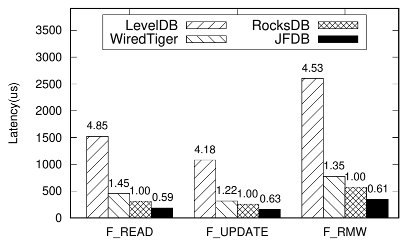
(i) YCSB-C latency



(j) YCSB-D latency



(k) YCSB-E latency



(l) YCSB-F latency

Figure 10. YCSB performance.

For workload C, D, and E, JFDB performs marginally better or similarly to RocksDB. Workload C only reads over data without multi-versions, and workload D and E mostly reads with only 5% writes, all of which are insertions, not updates. Furthermore, these user-level insertions translate into only metadata updates at the backend KVS, limiting the performance improvement of JFDB.

4.3 LinkBench Results

With the popularity of KVS, databases also employ KVS as backend stores in place of conventional storage engines. A representative example is MyRocks, a database engine that provides MySQL features with RocksDB implementations [22]. To evaluate the performance of JFDB in these scenarios, we run LinkBench, a database benchmark for social graph [21], while using MariaDB [42] with JFDB as the backend store. For comparison, we similarly configure MariaDB with RocksDB as the backend. We run 16 threads that each perform 30 million operations over a social graph initially loaded with 10K nodes. LinkBench randomly generates operations based on the distribution table shown in Table 4.

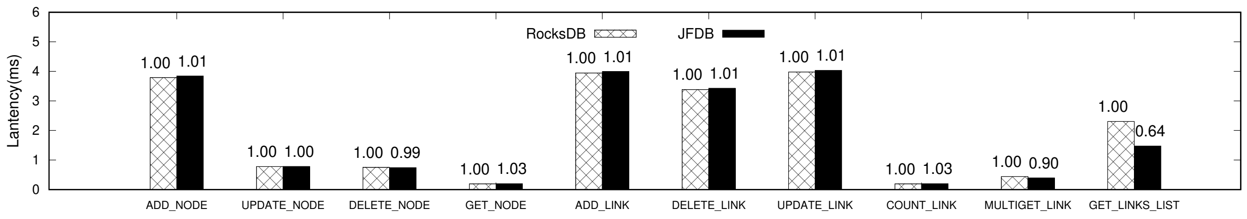


Figure 11. LinkBench latency

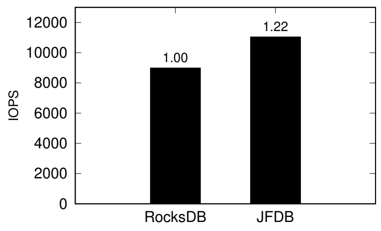


Figure 12. LinkBench IOPS

Operation	Ratio
ADD_NODE	2.6%
UPDATE_NODE	7.4%
DELETE_NODE	1.0%
GET_NODE	12.9%
ADD_LINK	9.0%
DELETE_LINK	3.0%
UPDATE_LINK	8.0%
COUNT_LINK	4.9%
MULTIGET_LINK	0.5%
GET_LINKS_LIST	50.7%

Table 4. LinkBench operations.

Figures 11 and 12 respectively show the mean latency and average IOPS under LinkBench. For latency (Figure 11), JFDB shows similar performance to RocksDB for operations invoking the point lookup or updates. However, it achieves a large performance improvement for MULTIGET_LINK and GET_LINKS_LIST, reducing the latency by 10% and 36%, respectively. Because these operations involve scanning the KVS while the items are updated concurrently (UPDATE_NODE and UPDATE_LINK account for 7.4% and 8.0% in total operations), JFDB's lean data structures allow quick data retrieval compared to that of RocksDB. Because GET_LINKS_LIST accounts for 50.7% of the operations, JFDB also improves the throughput by 22% compared to RocksDB (Figure 12).

5 Discussion

Below we briefly discuss concerns regarding the design of JFDB: the space efficiency, the write traffic, and the memory allocation and reclamation policies.

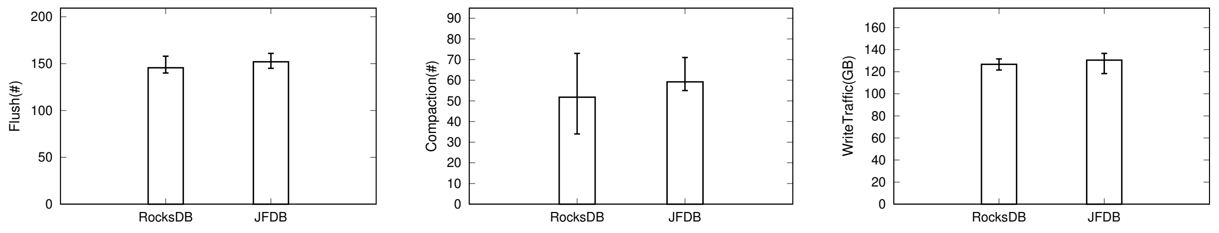
5.1 Memory Usage

The space efficiency of JFDB relative to RocksDB depends on the workloads. For insertion, JFDB uses more space than RocksDB because each node has one more pointer to the vertical per-key list. In contrast, for updates, JFDB can save more memory by applying the updates into the vertical per-key list which is connected by a single link instead of multiple links. However, this is not the case in our JFDB implementation. We implement JFDB by augmenting RocksDB to use JellyFish; in RocksDB, the node height is determined and its pointer space is allocated before figuring out whether the requested key resides in the skip list or not. If the key is found in the skip list, JFDB actually does not need the index links of the new node. Nevertheless, it is challenging to shrink the node size at the timepoint because it incurs space fragmentation. To search the skip list before determining the node height, however, we need to modify a large number of lines of codes, hampering the abstraction of software layers. As a result, the current JFDB implementation uses slightly more space than RocksDB.

Table 5 compares the memory usage of JFDB and RocksDB when key-value pairs of 16B keys and 100B values are inserted. These keys are uniformly and randomly distributed: without any skew in distribution, this represents the case where JFDB is least space-efficient. In a skip list data structure, each node contains metadata associated with the node. On average, 10.7 bytes are used for linking nodes in the skip list, because on average, the average node height is 1.33 [53]. Accounting for all these metadata, the overhead of maintaining the additional pointer for the vertical list is only about 5%.

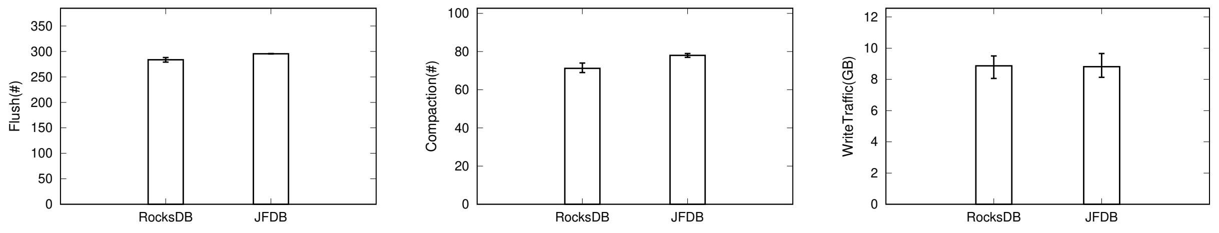
Type	RocksDB	JFDB
Key	16 bytes	16 bytes
Value	100 bytes	100 bytes
Index Links	10.7 bytes	10.7 bytes
Timestamp	7 bytes	7 bytes
Value Type	1 byte	1 byte
Vertical List Ptr	-	8 bytes
Total Usage	1x	1.05x
Space Utilization	100%	94.6%

Table 5. Memory usage.



(a) Flush Count **(b)** Compaction Count **(c)** Write Traffic

Figure 13. The number of merge operations and write traffic (insertion only).



(a) Flush Count **(b)** Compaction Count **(c)** Write Traffic

Figure 14. The number of merge operations and write traffic (with updates).

5.2 Write Amplification

An LSM tree-based KVS buffers changes in the memtable and flushes them to persistent storage when the memtable is full. Because JFDB uses slightly more memory space for indexing, the memtable becomes full faster than RocksDB, and thus it flushes more often. We investigate how this property affects the write amplification of KVS through an empirical study. We measure the number of merge operations and the write traffic to storage caused by them. We run two workloads using db_bench. First, we run `fillseq` where each thread inserts the keys in a certain range sequentially and thus causing the same key to be updated multiple times (equal to the number of threads). Second, we run a modified version of `fillseq` where each thread inserts keys in disjoint ranges. This causes no updates and thus there is no buffering effect. For the two workloads above, we execute 25 millions of put operations (16B key and 100B value) per thread using 20 threads, writing 55GB in total.

Figures 13 and 14 compare the number of merge operations and the write traffic of JFDB and RocksDB for two workloads. For detailed analysis, we classify the merge operations into *flush* and *compaction*. RocksDB and JFDB maintain data in a format known as Sorted-String Table (SST) files in storage. For fast lookups, they divide SST into levels: the smaller level files contain more recent data. The *flush* refers to writing a memtable into the level-0 file, and the *compaction* refers to merging SST-files in level N-1 into level N.

As the figures show, JFDB incurs 4-5% and 9-14% more flushes and compaction than RocksDB across workloads. However, the total amount of writes to storage remains similar. For the case when all put operations are insertions (Figure 13), JFDB's total amount of write to persistent storage is similar to that of RocksDB despite triggering flush and compaction more frequently. This is because all the writes are for unique keys and all data are written to storage eventually. Interestingly, we observe a similar trend for the opposite workload dominated by updates (Figure 14), but for a different reason. In the case where keys overlap, frequent flushing impairs the buffering effect of memtable, resulting in more write traffic and slower performance. However, LSM-tree essentially performs multi-level buffering as each layer serves as a buffer for the one-level lower layer. Thus, even though the topmost buffering is slightly less effective, the lower layers effectively complement the drawback. As such, even though the absolute number of flushes and compactions increase for the update-heavy case, the total amount of writes to persistent storage becomes similar between RocksDB and JFDB.

5.3 Allocation and Reclamation

For allocation memory, RocksDB maintains a pre-allocated pool of memory called the *arena*. This filters the frequent call to `malloc`, allowing for a scalable allocation managed by RocksDB. When a new node is created, RocksDB allocates the required size of consecutive memory space by increasing the in-use offset of the arena. This requires only an atomic increment of the integer and thus it is fast and scalable. JFDB follows the same model, allocating both skip list nodes and vnodes from the arena.

Because MVCC inherently improves concurrency by maintaining multiple versions of data, the efficient reclamation of obsolete data is an integral component for MVCC-compliant data structures. Prior work such as MV-RLU [36] show that reclaiming space occupied by obsolete data is cumbersome; it is challenging to ensure that no readers access the data to be reclaimed. Existing KVSs such as LevelDB and RocksDB resolves this issue by exposing an explicit interface GetSnap to readers. Through this, if the data is in-used by a snapshot, it is preserved and its reclamation is deferred.

6 Related Work

Our work investigates the performance overhead of multi-version concurrency control (MVCC) in key-value stores (KVS). As such, it builds upon a number of prior work in MVCC, skip lists, and KVS.

6.1 Multi-Version Concurrency Control

MVCC is a widely used concurrency control method in databases and key-value stores, and it provides a fairly strong isolation level of what is known as *snapshot*. Snapshot isolation, however, is weaker than the strongest form of isolation (i.e., serializability), and consequently, a number of studies [11, 13, 37, 48] aim to provide a stronger isolation level using MVCC. Interestingly, Neumann et al. overcome the performance overhead of serialization by maintaining the version ranges so that a specific version of data can be efficiently retrieved [48]. JellyFish does not maintain such version information, but we share a similar motivation for improving range query operations.

6.2 Scalability of Skip List

For skip lists, a number of recent studies focus on improving its concurrency, through lock-less approaches [25, 29, 38] and optimistic locking [31]. Other variations include the work by Crain et al. [16] that lazily links upper-layer pointers, NUMASK [17] that is optimized for non-uniform memory architectures, and the design by Alam et al. [1] that parallelizes range queries on distributed clusters. S3 is a scalable skip list for KVS that uses the highly optimized binary tree atop a skip list for fast search and relaxes the sorted property to increase write performance [61]. Our approach instead explores how to improve performance when multiple versions of data exist.

6.3 KVS Optimization

As the KVS is ubiquitously used in a wide range of applications, many studies have been performed to optimize it over the last years [6, 26, 41, 43, 55, 59]. Among them, we briefly highlight several studies that focus on in-memory component optimization, which are related to our work but not covered in the previous section.

The authors of Accordion [12] argue that persistent KVSs will gravitate toward using larger memory, thus becoming the performance bottleneck. Accordion specifically addresses the problem of memory fragmentation caused by the dynamic allocation of in-memory objects, and proposes a novel memory management algorithm that re-organizes in-memory data. Our work, on the other hand, addresses this fragmentation problem by employing preallocated memory pools, and instead focus on the performance overhead of traversing the in-memory data structure.

FloDB [7] and TeksDB [28] both augment the in-memory skip list by integrating it with a hash data structure. More specifically, FloDB places a small hash table in front of the sorted skip list, and TeksDB uses both the hash table and the skip list at the same level that both point to the same value. In doing so, they achieve fast response time for point accesses, while enjoying the benefit of sorted data retrievals. However, FloDB [7] unfortunately does not support MVCC, while TeksDB [28] incurs overhead in synchronizing the two complementary data structures.

KiWi [8] presents a design that provides wait-free scans and lock-free updates without the loss of consistency. Its core intuition is that multi-versioning is essentially intended for consistent synchronization between readers and writers, and thus only makes a snapshot when there are on-going scans; otherwise, updates are overwritten in-place. In this paper, we also address the overheads associated with multi-versioning, but reduce its overhead by maintaining a separate per-key list of values.

7 Conclusion

In this paper, we present the design, implementation, and evaluation of JellyFish, a skip list that reduces the performance overheads for MVCC. JellyFish separates per-key updates from the key indexing, allowing fast list traversals, efficient point queries, and fewer pointer changes for updates. The benefits of our design are amplified as the memory size increases, and our evaluation results show that JellyFish not only improves the throughput, but also the latency significantly.

8 Acknowledgments

We thank Fernando Pedone (our shepherd) and the anonymous reviewers for their insightful and constructive comments, and also thank Sungjin Lee for reviewing the early stages of this work. This work was supported by the National Research Foundation of Korea (NRF) (NRF-2019R1A2C1090337). Eunji Lee is the corresponding author.

References

[1] Sarwar Alam, Humaira Kamal, and Alan Wagner. 2014. A Scalable Distributed Skip List for Range Queries. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 315–318. https://doi.org/10.1145/2600212.2600712

[2] Apache. [n.d.]. Apache CouchDB. https://en.wikipedia.org/wiki/Apache_CouchDB.

[3] Apache. 2019. Cassandra. http://cassandra.apache.org/.

[4] Ardb. 2013. Ardb. https://github.com/yinqiwen/ardb.

[5] 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*, Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale (Eds.). ACM, 53–64. https://doi.org/10.1145/2254756.2254766

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[7] 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* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3064176.3064193

[8] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

[9] 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.

[10] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[11] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. 2011. One-copy serializability with snapshot isolation under the hood. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE. https://doi.org/10.1109/icde.2011.5767897

[12] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1863–1875. https://doi.org/10.14778/3229863.3229873

[13] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

[14] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[15] 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* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[16] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No hot spot non-blocking skip list. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 196–205.

[17] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *32nd International Symposium on Distributed Computing (DISC 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 121)*, Ulrich Schmid and Josef Widder (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:19. https://doi.org/10.4230/LIPIcs.DISC.2018.18

[18] Ian Dick, Alan Fekete, and Vincent Gramoli. 2016. A skip list for multicore. *Concurrency and Computation: Practice and Experience* 29, 4 (May 2016), e3876. https://doi.org/10.1002/cpe.3876

[19] Facebook. [n.d.]. Under the Hood: Building and open-sourcing RocksDB. https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/.

[20] Facebook. 2012. RocksDB. https://github.com/facebook/rocksdb.

[21] Facebook. 2019. Linkbench. https://github.com/facebookarchive/linkbench.

[22] Facebook. 2019. MyRocks. https://github.com/facebook/mysql-5.6/wiki.

[23] Facebook. 2019. RocksDB - MemTable. https://github.com/facebook/rocksdb/wiki/MemTable.

[24] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*. 50–59. https://doi.org/10.1145/1011767.1011776.

[25] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[26] 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.

[27] Google. 2011. LevelDB. https://github.com/google/leveldb.

[28] 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/10.1145/3322205.3311079

[29] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists.. In *DISC (Lecture Notes in Computer Science, Vol. 2180)*, Jennifer L. Welch (Ed.). Springer, 300–314.

[30] Red Hat. 2020. Ceph. https://github.com/ceph/ceph/blob/master/src/common/options.cc#l4385.

[31] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A Provably Correct Scalable Concurrent Skip List. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*.

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/3299869.3314041

[33] HyperDex. 2011. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[34] IBM. [n.d.]. DB2: Currently committed semantics improve concurrency. https://docs.oracle.com/cd/E17076_02/html/programmer_reference/transapp_read.html.
[35] Intel. 2019. Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[36] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 779–792. https://doi.org/10.1145/3297858.3304040
[37] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 298–309. https://doi.org/10.14778/2095686.2095689
[38] Doug Lea. [n.d.]. Doug Lea's Home Page. https://gee.cs.oswego.edu.
[39] Linux. 2015. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
[40] LMDB. 2011. LMDB. https://symas.com/lmdb/.
[41] 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.
[42] MariaDB. 2019. MyRocks for MariaDB. https://mariadb.com/kb/en/myrocks/.
[43] 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.
[44] Microsoft. [n.d.]. Snapshot Isolation in SQL Server. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server?redirectedfrom=MSDN.
[45] MIT. 2012. MemSQL. https://github.com/facebook/rocksdb.
[46] MongoDB. [n.d.]. MongoDB CTO: How our new WiredTiger storage engine will earn its stripes. https://www.zdnet.com/article/mongodb-cto-how-our-new-wiredtiger-storage-engine-will-earn-its-stripes/.
[47] MongoDB. 2016. WiredTiger. http://www.wiredtiger.com/.
[48] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372.2749436
[49] 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. https://doi.org/10.1007/s002360050048
[50] Oracle. [n.d.]. Berkeley DB Transactional Data Store Applications. https://docs.oracle.com/cd/E17076_02/html/programmer_reference/transapp_read.html.
[51] 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*. 251–264. http://www.usenix.org/events/osdi10/tech/full_papers/Peng.pdf.
[52] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast Scans on Key-Value Stores. *PVLDB* 10, 11 (2017), 1526–1537. http://www.vldb.org/pvldb/vol10/p1526-bocksrocker.pdf.
[53] William Pugh. 1990. *A Skip List Cookbook*. Technical Report.
[54] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676.
[55] 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.
[56] SciPy. [n.d.]. numpy.random.zipf. https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.zipf.html.
[57] Nir Shavit and Itay Lotan. 2000. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*. 263–268. https://doi.org/10.1109/IPDPS.2000.845994.
[58] Andrew Shewmaker. 2013. A kernel skiplist implementation. https://lwn.net/Articles/551896/.
[59] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: Parallel Augmented Maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 290–304. https://doi.org/10.1145/3178487.3178509
[60] Wikipedia. [n.d.]. Snapshot Isolation. https://en.wikipedia.org/wiki/Snapshot_isolation.
[61] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. 2019. S3: A Scalable in-Memory Skip-List Index for Key-Value Store. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2183–2194. https://doi.org/10.14778/3352063.3352134