Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support

Bogdan Nicolae

Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support

Bogdan Nicolae
Argonne National Laboratory, USA
Email: bogdan.nicolae@acm.org

***Abstract*—Ordered key-value stores (or sorted maps/dictionaries) are a fundamental building block in a large variety of both sequential and parallel/distributed algorithms. However, most state-of-art approaches are either based on ephemeral in-memory representations that are difficult to persist and/or not scalable enough under concurrent access (e.g., red-black trees, skip lists), and/or not lightweight enough (e.g. database engines). Furthermore, there is an increasing need to provide versioning support, which is needed in a variety of scenarios: introspection, provenance tracking, revisiting previous intermediate results. To address these challenges, we propose a new lightweight dictionary data structure that simultaneously provides support for multi-versioning, persistency and scalability under concurrent access. We demonstrate its effectiveness through a series of experiments, in which it outperforms several state-of-art approaches, both in terms of vertical and horizontal scalability.**

***Index Terms*—key-value store; ordered dictionary; versioning control; scalable access under concurrency; persistent memory**

I. Introduction

The increasing convergence between high-performance computing (HPC), big data analytics and machine learning is leading a rapid evolution of data management requirements: we are facing an abundance of heterogeneous data, which requires rich metadata to describe it (labels, attributes, relationships, etc.). Unfortunately, data repositories have seen a relatively static evolution. For example, supercomputer storage is still dominated by parallel file systems, which are based on the aging POSIX model to represent and manipulate data as files. Despite an increasing transition towards more scalable repositories such as object stores and key-value stores [1], a majority of applications still treat data as an *external* entity that needs to be loaded into memory, transformed, and then written back to the repository. Given limited I/O bandwidth under concurrent access at scale, this pattern frequently leads to I/O bottlenecks that degrade the performance of the applications.

Once the data is loaded into memory, a fundamental data structure frequently used by applications is an ordered key-value store (also referred to as sorted map or dictionary). In a nutshell, it is a collection of key-value pairs in which each key can appear only once. Users can insert a new key-value pair, remove an existing key-value pair, find the value corresponding to a given key, or iterate over all key-value pairs in sorted order. The latter is an important property that distinguishes ordered from unordered key-value stores, and is instrumental in a variety of scenarios. For example, consider a deep learning (DL) application whose learning models are represented as a set of key-value pairs $(id, tensor)$ that define layers. Most operations that involve such models (training/inference, mutations such insertions or removal of layers, comparisons using the longest common prefix or longest common substring to facilitate transfer learning [2], etc.) need to preserve the order of the tensors.

Despite significant progress in optimizing the performance and scalability of key-value stores, both *vertically* (i.e., key-value store in shared memory accessed by multiple threads or processes concurrently), and/or *horizontally* (i.e., key-value store partitioned and distributed across multiple compute nodes), a majority of such optimizations were explored for *ephemeral* implementations that do not outlive the processes that created them, under the assumption that persistency can be achieved separately by writing the key-value store in a serialized form to an external storage repository (which is subject to I/O bottlenecks).

On top of this, an entire new level of challenges is raised by the need for *multi-versioning*, i.e., the ability to capture intermediate snapshots of the key-value store for a wide range of use cases [3]: introspection, provenance tracking, understand data evolution, revisit previous intermediate results, share intermediate results with other workflow components, roll back in case of failures, branch off in a different direction (e.g., DNN network architecture search techniques that try different model variations starting from a common ancestor). Multi-versioning is a feature typically only available in specialized external data repositories.

On the other hand, the emergence of persistent memories [4] presents an opportunity to create and manage key-value pairs directly as byte-addressable objects on the compute nodes, which reduces (if not eliminates) the need to serialize key-value pairs to external storage in order to achieve persistency, thereby greatly reducing potential I/O bottlenecks at scale. However, this opportunity has not been sufficiently explored in the context of ordered key-value stores.

In this paper, we introduce an ordered key-value store proposal that simultaneously provides native multi-versioning support and vertical/horizontal scalability under concurrent access, while leveraging persistent memories to provide durability. We summarize our contributions as follows:

- We formulate the problem of native multi-versioning and persistency for ordered key-value stores that remain scalable under concurrent access, both vertically and hor-

izontally. In this regard, we propose versioning-oriented API (Section II).
- We introduce several key design principles that underline the main idea of our proposal: compact representation of snapshots using a persistent per-key version history, hybrid indexing using ephemeral skip lists, a persistent key block chain to enable parallel reconstruction, lazy tail, distributed partitioning with multi-threaded merge support (Section IV-A).
- We discuss several algorithms and considerations that enable an efficient implementation of the design principles (Section IV-B).
- We evaluate our proposal in a series of experiments that involve both vertical (up to 64 threads) and horizontal (up to 512 nodes) scalability. We focus on a comprehensive set of scenarios that compare our proposal with five other approaches (Section V).

II. Problem statement

Consider a regular ordered key-value store (e.g. a C++ sorted std::map) that enables the following operations: insert a new key-value pair, remove an existing key-value pair, find the value of a given key, or iterate over all key-value pairs in sorted order.

TABLE I: Comparison of supported operations

Regular dictionary	*Multi-version dictionary*
insert(key, value)	insert(key, value)
remove(key)	remove(key)
find(key)	find(key, version)
for_all(key, value)	extract_snapshot(version)
	extract_history(key)
	tag(version)

Multi-versioning: we extend the ordered key-value store with support to capture an immutable *virtual* snapshot, labeled with a version number, using the tag primitive. This snapshot reflects the history of all insert and remove operations since the beginning, up to the moment when tag is called (but none of the operations issued afterwards). Tagging is similar to committing a set of changes in a revision control system. We call this snapshot virtual, because it is exposed like an independent, immutable copy of the key-value store at user-level, but the underlying implementation is free to optimize its storage (e.g., record incremental changes between snapshots). Unlike unordered key-value stores, find and iterate over all key-value pairs (denoted extract_snapshot) can refer to any past snapshot version, in addition to the current state. Furthermore, a dedicated primitive, extract_history, allows the inspection of the evolution of a specified key. It returns a list of insert or remove operations and the corresponding version when they were issued. These operations are summarized in Table I.

Persistency: in addition to multi-versioning, another important desired property is *persistency*, i.e., all virtual snapshots captured by the ordered key-value store (including the current state) need to survive after the processes that created them finished or terminated unexpectedly, without losing consistency.

Scalability under concurrency: at scale, applications consist of multiple processes and threads that are distributed across multiple nodes. Since key-value stores are often used to maintain a common state, they need to remain scalable under concurrent access, both vertically (i.e., shared memory accessed by multiple concurrent threads) and horizontally (i.e., key-value pairs partitioned across multiple nodes).

Our goal is to design and implement an efficient ordered key-value store that satisfies all three requirements simultaneously: multi-versioning support, persistency and scalability under concurrent access.

III. Related work

Fundamental data structures: hash tables and KV stores [5], red-black trees [6], AVL trees [7], skip lists [8], B-trees [9] are known for decades. With the rise of multi-core systems, techniques emerged to enable efficient concurrent operations on such data structures, either by fine-grain locking [10], [11], or by using lock-free approaches based on compare-and-swap [12], [13]. Despite being lightweight and highly optimized for concurrent access, such data structures lack at least one of the following requirements: ordered keys, persistency, multi-versioning support.

Database engines: General purpose databases provide rich indexing and persistency support for key-value pair collections. They are either based on SQL (e.g., traditional DB engines like POSTGRES [14]) or NoSQL (e.g. Cassandra [15]). However, such solutions are not lightweight enough to act as a replacement for fundamental data structures, nor do they provide native multi-versioning support.

Storage repositories: Traditionally, POSIX file systems are the backbone of data storage. They are used both to leverage node-local storage devices (e.g., F2F2 or ext4), as well as to mount shared repositories at large scale (e.g., parallel file systems such as Lustre [16] and GPFS [17]). Unfortunately, the aging POSIX model limits the performance and scalability under concurrency. To alleviate this issue, object storage systems such as DAOS [1] and RADOS [18] are a popular alternative. However, such repositories are not byte-addressable and require explicit read/write interactions to serialize and deserialize data, which introduces both complexity and performance overheads.

Persistent memory: A need emerged to efficiently persist data structures in a lightweight fashion using byte-addressable constructs. In this regard, persistent memories [4] are rapidly gaining traction. Most practical implementations, such as Intel's PDMK allow applications to allocate basic data structures (structs, arrays, etc.) and manipulate them through persistent pointers and primitives that enable concurrency control (transactions, compare-and-swap).

Multi-versioning: Revision control systems such as GIT [19] are widely used to keep track of changes to source code during software development. They feature native support to manipulate snapshots through operations such as commits,

roll-back, branch, merge, etc. However most optimizations revolve around compact representation of incremental changes to textual data, with little attention dedicated to performance, scalability and binary data. To address this issue, several efforts emerged to bring multi-versioning to binary large objects (e.g., BlobSeer [20]), SQL databases [21], NoSQL (e.g., Delta Lakes [22]), machine learning (e.g., DLHub [23]). However, such approaches cannot be used a as lightweight drop-in replacements of ordered key-value stores.

To our best knowledge, we are the first to consider the problem designing ordered key-value stores that simultaneously provide multi-versioning, persistency and scalability under concurrent access.

IV. Proposed Approach

A. Design principles

Our proposal is based on the following key ideas:

Compact persistent memory representation: In a majority of scenarios, a large number of key-value pairs will remain unchanged from one snapshot to another. Therefore, as noted in Section II, it is not necessary to create a full copy for every snapshot. Instead, it is possible to obtain a compact representation by associating each key with a *version history*, i.e., a list of pairs $(version, value)$ that records the versions at which the key was updated (either by inserting a new value or by removing the key, in which case a special marker is used as the value). Using this approach, insert and remove operations simply need to atomically append the new value or marker to the version history, while find operations need to perform a binary search in the version history to identify the tuple with the highest version smaller than the requested version number. Similarly, extract snapshot needs to iterate over all keys in sorted order and perform the same binary search to determine the corresponding value (or ignore the key if it was removed). The extract key history operation is trivial and simply needs to return a pointer to the version history. To persist such a compact representation without serialization overheads, we propose to store it directly in persistent memory. To this end, we only need to leverage basic constructs, such as allocating (and potentially reallocating) persistent pointers to contiguous arrays that store the version history.

Hybrid ephemeral indexing to enable high performance under concurrent access: Without efficient indexing, a compact representation performs poorly. On the other hand, a persistent index capable of maintaining a sorted key order has a high performance overhead. Therefore, we propose a hybrid approach based on a lightweight, ephemeral skip list that resides in main memory. A skip list is a probabilistic data structure that maintains a hierarchy of linked lists, organized into levels, each of which skips over more elements than the one below it. An example of how this works is illustrated in Figure 1, where we have a collection with six keys $(0, 3, 6, 7, 9, 13)$ and the corresponding persistent pointers to their version histories (A, B, C, D, E, F). The keys are organized into a skip list with five levels (left hand side). Each key has a height obtained by random coin toss: 1/2 chance to reach level 2, 1/4 to reach level 3, etc. By starting from the top level, a key can be found in logarithmic time by skipping over all lower keys in the same level, then going one level deeper. For example, to reach key 7, we start from the head H at level 5, go to level 3, skip to key 6, go to level 2, skip to key 7. At the lowest level, we obtain the version history D of key 7, which can be used to obtain the value corresponding to the desired version using binary search. A skip list can take advantage of cheap compare-and-swap operations to efficiently handle concurrency (as detailed later in Section IV-B). This not only solves the problem of scalability under concurrent access, but but it also reduces the persistent memory utilization at the same time, because no indexing information needs to be stored persistently. Instead, the compact representation is enough to reconstruct the skip list on restart. However, this inevitably introduces an overhead, which can be mitigated efficiently as discussed next.

Persistent key block chain to enable parallel reconstruction: To reconstruct an ephemeral skip list from the compact representation residing in the persistent memory, we need to re-insert all keys and pointers to the version histories. Fortunately, this can be achieved using multiple reconstruction threads efficiently. Unfortunately, it is not-trivial to organize the $(key, pointer)$ pairs in persistent memory such that: (1) they can be easily distributed in bulk among the reconstruction threads; (2) the insert performance after reconstruction remains efficient. For example, if the $(key, pointer)$ pairs are stored in an array, then the array can be easily partitioned among the reconstruction threads but this may cause high reallocation overheads when inserting new keys. Conversely, if the $(key, pointer)$ pairs are organized as a linked list, then insertions of new keys are efficient but the pairs are scattered and thus difficult to assign to the reconstruction threads. To solve this trade-off, we propose to organize the pairs using a *persistent block chain*, which is a data structure inspired by the ledgers used by crypto-currencies. Specifically, we create a linked list of blocks represented by fixed-sized arrays, which grows only when the tail block is fully filled. Thus, insertions of new keys remain efficient as a new block is only rarely allocated. On the other hand, each reconstruction thread tid can simply walk the linked list in parallel with the other threads and insert all keys of all blocks i for which $i \ mod \ tid = 0$, while skipping over the other blocks (which are claimed by the other threads). Using this approach, the pairs are evenly distributed among the reconstruction threads and can be inserted concurrently in bulk. An example is illustrated in Figure 1 (bottom-right): there are two blocks, each of which holds three pairs. Two threads coordinate as follows: the first thread claims the first block and ignores the second, the second thread ignores the first block and claims the second.

Lazy tail to enable concurrent version history updates: Another important related aspect is how to enable efficient appends of new values (or removal markers) to the version history of each key. A straightforward solution that simply executes the append in a transaction may have a high overhead, because the transactions will be serialized.

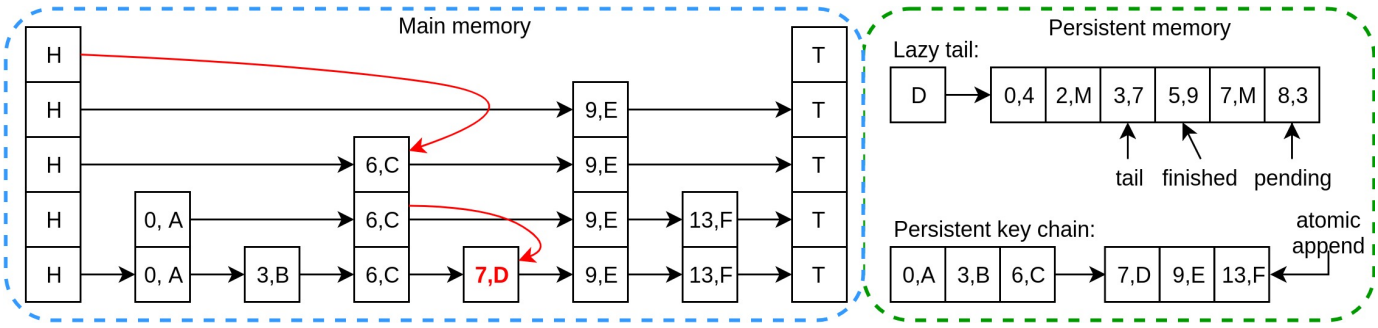


Fig. 1: Data structures in a nutshell: ephemeral skip list, version history with lazy tail, key chain organized into blocks

To alleviate this issue, we propose a new data structure we call *lazy tail*, which works as follows: each append of a new $(version, value)$ pair claims a slot from the version history by atomically incrementing a $pending$ counter. If there are multiple concurrent appends, they can proceed writing their pairs to persistent memory in parallel. When an append has finished, the corresponding slot is marked as finished. Note that the appends can finish in random order, therefore it is not guaranteed that the finished slots form a contiguous region, nor should they be exposed to any find or extract query for consistency reasons (an insert or remove is considered finished only when all inserts or removes of lower versions have finished). In fact, a find or extract query does not need to know where the tail of the version history is, unless it affects the outcome of the query. For example, consider the version history of key 7 illustrated in Figure 1 (top-right): the key was first inserted at version 0, removed at version 2 (using M as the removal marker), then re-inserted at version 3, which is the tail. If there are three concurrent appends due to inserts/removes at versions $(5, 7, 8)$, then each will claim a slot by incrementing $pending$ up to 6. If the append for version 5 has finished and is marked as finished while at the same time another concurrent find query arrives for version 2, then it is not necessary to discover the new tail and the find query can simply proceed with the binary search using the original tail. This not only avoids the extra overhead of discovering the new tail, but also makes the binary search faster. Only when a find query for a version higher than 5 arrives, then the tail needs to be extended iteratively as long as there are finished consecutive slots. Using this approach, the tail is updated only when necessary and only by the query operations, not by the insert or remove operations (hence the reason for calling the tail *lazy*).

Hierarchic multi-threaded merge to enable scalable extract snapshot: In a horizontal scalability scenario, the key-value pair collection is partitioned among K compute nodes, each of which is responsible for a different key range. In this case, most operations except extract snapshot can be implemented in an embarrassingly parallel fashion by redirecting them to the compute node responsible for their keys. However, in the case of extract snapshot, two steps are involved: (1) perform extract snapshot on each compute node in parallel; (2) gather and merge the key-value pairs. A straightforward solution to solve this problem is to simply gather the key-value pairs on a compute node and perform a K-way merge. However, this leads to high overheads. Therefore, we propose two optimizations. First, we perform a hierarchic merge using recursive doubling, which runs in $K = log(N)$ rounds, where N is the number of distributed processes (e.g., MPI ranks). In each round, the odd ranks send their key-value pairs to the even ranks (in parallel), who perform a merge (in parallel) and survive in the next round. The result is eventually collected on a single rank in the last round. Second, we introduce a multi-threaded merge approach on each rank that works as follows: assuming two arrays A and B that need to be merged, first we partition A evenly among the threads so that each thread i is assigned a partition A_i. Then each thread i uses binary search to find the position p_i in B whose key is larger than the maximum key of A_i (last element, because the keys are sorted). Since thread $i-1$ did the same in parallel, thread i knows that all keys in A_i are between p_{i-1} and p_i in B. Therefore, all threads can determine in parallel what range from B to merge with and what offset to use in the final result.

B. Algorithms

In this section, we briefly zoom on some of the high-level algorithms that illustrate the design principles introduced in Section IV-A.

First we focus on Algorithm 1, which illustrates how the lazy tail of the version history works. The insert operation simply grabs the next free slot and persists the new value (or removal marker), then marks the slot as finished by incrementing a global pending counter pc. The find operation extends the tail only if needed, i.e., the insert operation has finished, it does not come before any other non-finished insert operation from any other key (tracked by a global finished counter fc, which is incremented if possible), and the requested version is smaller than the inserted version. Using this approach, the version histories remain both consistent and resilient to failures (on restart, it is enough to count the length of all contiguous non-zero $finished$ sequences of all keys to recover fc, then prune all finished entries larger than fc and adjust $tail$ and $pending$ accordingly for each key). Another important observation is that the binary search does not need to necessarily divide

Algorithm 1: Version history

```
1  Function insert(version, value):
2      slot ← atomic_inc(pending)
3      if slot = size(history) then
4          extend(history)
5      history[slot] ← (version, value)
6      finished[slot] ← atomic_inc(pc)
7  Function remove(version):
8      insert(version, marker)
9  Function find(version):
10     t ← tail
11     while 0 < finished[t] ≤ fc + 1 and
         history[t].version < version do
12         t ← t + 1
13         if finished[t] = fc + 1 then
14             fc.compare_exchange(fc, fc + 1)
15     if t > tail then
16         tail.compare_exchange(tail, t)
17     left ← 0
18     right ← t − 1
19     while left <= right do
20         middle ← (left + right)/2
21         if t < history[middle].version then
22             right getsmiddle − 1
23         else if t > history[middle].version then
24             left ← middle + 1
25         else
26             return history[middle].value;
27     return marker if right < 0 else history[right].value;
```

Algorithm 2: FindSkip

```
   Input  : key k
   Output: predecessors preds, successors succs
1  level ← max_level(head)
2  pred ← head
3  curr ← pred.next[level]
4  while true do
5      succ ← curr.next[level]
6      if curr.key < k then
7          pred ← curr
8          curr ← succ
9      else
10         preds[level] ← pred
11         succs[level] ← curr
12         if level = 0 then
13             break
14         curr ← pred.next[level]
15         level ← level − 1
16 return (preds, succs, curr if curr = k else NULL)
```

the history in the middle. Instead, it can be biased towards older or newer versions, depending on what type of queries are expected to be dominant.

Next, we focus on our ephemeral skip list, which is organized as a multi-level linked list holding the keys and their persistent pointers to the version histories. Each pair has multiple successors, one for each level. We use a helper function FindSkip, listed in Algorithm 2, that returns the predecessors and successors at each level for a given key, and, if the key exists, a pointer to the pair. Using this helper function, find operations simply need to obtain a pointer to the version history pointer, then use binary search to find the value in the version history. Similarly, extract snapshot simply needs to walk the last level of the skip list and associate each key with the corresponding value at the given version.

The predecessor and successors at each level are needed in order to implement the insert operation, which is detailed by Algorithm 3. We base our design on compare-and-swap, which enables efficient access under concurrency [12]. Two important persistency-related observations need to be made in this context. First, the insert operation needs to be able to either create a new version history when a new key is inserted or to use an existing persistent pointer ptr if the insert is performed due to restart (in which case, no new value is added to the history). Secondly, it can happen that concurrent insert operations compete to set the next pointer of the same predecessors. In this case, a thread needs to check if it is still the successor of its predecessor, and, if it is not, it needs to retry. Furthermore, it can also happen that two threads compete to allocate the data structures for the same key. In this case, the slower thread needs to detect this situation and clean up accordingly, then reuse the pointer of the faster thread.

Complexity analysis: Assuming the total number of inserted keys is N_k, the complexity of extract key history is $O(log(N_k)$. Find operations on the version history are logarithmic in the number of changes between snapshots N_c, but also linear in the number of inserts N_i by which the tail can be extended. Therefore, given a total of N_k keys, the overall complexity of find operations is $O(log(N_k) \cdot (log(N_c) + N_i))$. However, in practice, N_i is not likely to dominate. Similarly, the complexity of extract snapshot is $O(N_k \cdot (log(N_c) + N_i))$. Note that N_k continuously grows as more snapshots are added, which is a limitation of our proposal if some keys are only valid in certain versions that are rarely accessed. In this case, we can imagine garbage collection and/or aging mechanisms that move keys between multiple skip lists. However, such optimizations are outside the scope of this work.

V. Experimental evaluation

A. Experimental Setup

Our experiments were performed on Argonne's *Theta* supercomputer, a 11.69 petaflops pre-Exascale Cray XC40 system based on the second-generation KNL Intel Xeon Phi 7230 SKU. The system has 4392 nodes, each equipped with 64 core processors (256 hardware threads), 16 GB of high-bandwidth MCDRAM (300-450 GB/s), 192 GB of main memory (DDR4 RAM, 20 GB/s), and a 128 GB SSD (700 MB/s). The interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec.

The system was configured to run in caching mode (MCDRAM caches DRAM at hardware level). We emulate per-

Algorithm 3: Insert operation

```
Input : key k, value v, persistent ptr to version history
while true do
    node ← NULL
    found, preds, succs ← FindSkip(k)
    if found ≠ NULL then
        if node ≠ NULL then
            // somebody inserted the same key
               faster, clean up
            if node.history ≠ ptr then
                delete_persistent node.history
            delete node
        node ← found
    else if node = NULL then
        // height determined using fist
           significant bit of random number
        node ←new node(key, fsb(random))
    if ptr = NULL then
        if node.history = NULL then
            new_persistent node.history
        node.history.insert(curr_version, value)
    else
        node.history ← ptr
    succ ← succs[0];
    if succ = node then
        return
    for level ← 0..max_level(node.next) do
        node.next[level] ← succs[level]
    pred ← preds[0]
    if pred.next[0].compare_exchange(succ, node) then
        if ptr = NULL then
            key_chain.append(key, node.history)
        break
level ← 1
while level < max_level(node.next) do
    pred ← preds[level]
    succ ← succs[level]
    if ¬pred.next[level].compare_exchange(succ, node)
      then
        (_, preds, succs) ← FindSkip(k)
        continue
    level ← level + 1
return;
```

sistent memory by using the /dev/shm/ mountpoint, which provides an in-memory, POSIX-compatible temporary file system that we configure as a backing store for the *Intel PDMK* v.1.10 toolkit, which provides the transactional object store used by our approach. To facilitate easier integration of our code with PDMK, we use the *libpmemobj-cpp* v.1.12 library, which provides a set of convenient C++ abstractions.

B. Compared Approaches

Throughout our evaluations, we compare five approaches:

SQLiteReg: This approach implements the API introduced in Section II using a traditional database engine. To this end, we have chosen *SQLite* v.3.28, which provides a lightweight SQL engine that is available on most Linux distributions out-of-the box and can be directly embedded into an application by linking with a library. The collection of key-value pairs is represented by table, whose rows correspond to insertions and removals. Each row is composed of version number, key and value. In the case of removals, the value is a special marker outside of the allowable range of valid values. Find and extract queries are implemented by using select statements over this table. These statements are highly optimized by combining several best practices: (1) multi-column indexing over both version number and key; (2) prepared statements that allow us to bind the variables directly into a pre-compiled binary form; (3) write-ahead logging, which allows performance improvements under concurrency while maintaining ACID transactional properties. Any thread is allowed to issue any kind of query. The concurrency control is managed internally by the DB engine using fine-grain locking. The database backing file is stored in /dev/shm, same as for our approach. We use this approach as a reference baseline that provides both multi-versioning and persistency.

SQLiteMem: This approach is similar to SQLiteReg, except that the database is created in-memory and does not have a backing file, which means there is no persistency support and I/O overheads are limited to memory page management only. Furthermore, we configure SQLite to use a shared page cache across all threads, which further reduces overheads by eliminating extra copies. SQLiteMem is important complement to SQLiteReg, because it is representative of the best performance that can be achieved by using a database engine to provide multi-versioning, at the expense of removing persistency support.

LockedMap: This approach extends a C++ ordered std::map (whose underlying implementation is typically a red-black tree) with support for multi-versioning (but no persistency). To achieve this, each key is associated with a version history, implemented using a lock-free ephemeral vector with binary search support, as introduced in Section IV-A. The overall concurrency control is enforced by means of locking. We are including this approach in the comparison for two reasons: (1) it provides an alternative to database engines that is easy to implement on top of standard data structures; (2) it enables us to study the impact of our concurrency control strategy for key indexing vs. naive lock-based approaches.

ESkipList: This approach improves on the previous implementation by using a lock-free skip list instead of std::map. Since there is no need to support removal from the skip list itself, the implementation can be simplified to use raw pointers in compare-and-exchange operations. Overall, this approach combines all optimizations proposed by our work except that it uses ephemeral instead of persistent pointers and data structures, thereby lacking persistency support. We use it as an upper bound in our comparisons to study how much performance degradation is due to persistency support.

PSkipList: This approach implements our core proposal, combining all design principles introduced in Section IV-A. Unlike ESkipList, it provides full persistency support thanks

to our hybrid solution that combines a compact persistent representation with ephemeral indexing based on a lock-free skip list. PSkipList fulfills all requirements mentioned in Section II and aims to offer performance levels as close as possible to ESkipList.

C. Methodology

We designed a set of benchmarks to compare the approaches introduced in Section V-B. These benchmarks measure the performance and scalability of all supported operations (insert, remove, find, extract snapshot, evolution of key) under concurrency, both at the level of a single node (in which case we vary the number of concurrent threads) and multiple nodes (in which case we keep the number of threads per node fixed).

Our goal is to stress the compared approaches as much as possible under concurrency. To this end, we use a large number of tiny key-value pairs, where each key and value are represented by integers. For the purpose of this work, we generate the content of the keys and values randomly. However, to facilitate a fair comparison, we fix the random seeds used by each thread on each node, which allows us to construct reproducible scenarios. Furthermore, we tag after each insert and remove operation, which means each operation generates its own snapshots, thereby resulting in a large number of snapshots.

The benchmarks were implemented using a combination of MPI (one MPI rank per node) and OpenMP (multiple threads per MPI rank). The key-value pairs to be inserted and removed are pre-generated using a pseudo-random number generator (Mersenne Twister) and cached, which minimizes the impact of accessing the input data during our experiments.

D. Single Node: Concurrent Insert and Remove

Our first series of experiment studies the scalability of the insert and remove operations on a single node, starting from an empty state. We study both insert and remove because there is an important difference between them: insert operations may encounter new keys, in which case it is necessary to instantiate a new history and add a corresponding key-value pair, in addition to appending a new value to the version history (which is the only step relevant for remove operations). To emphasize this difference, we ensure the pre-generated key-value pairs have unique keys, thereby forcing the insert operations to exhibit a worst-case scenario. In the case when an insert operation is used to update an existing key, its behavior is equivalent to a remove operation and therefore we do not need to study it separately.

The experiment runs in two phases. First, we pre-generate $N = 1000000$ key-value pairs, we evenly distribute them to T threads, then we let the T threads insert them concurrently. Second, we create a random shuffling of the keys that are evenly distributed to the T threads, then we let the T threads remove them concurrently. We record the total time taken by all threads to finish. We repeat this experiment for each of the five compared approaches using a variable $T = 1..64$. Note that N remains fixed, therefore this is a strong scalability study.

The results are depicted in Figure 2. In the case of a single thread, LockedMap is the fastest, thanks to its optimized red-black tree implementation. It is followed closely by ESkipList, whose lock-free skip list introduces higher overheads. Surprisingly, SQLiteMem performs almost 10x slower than LockedMap and 5x slower than ESkipList, which indicates that database engines have too many additional overheads to be competitive versus specialized data structures. Moreover, these overheads are not due to I/O operations: we can observe that adding persistency has significant performance impact: SQLiteReg is 4x slower than SQLiteMem, while PSkipList is 12x slower than ESkipList. However, PSkipList is ultimately still 35% faster than SQLiteReg, therefore it is the fastest persistent approach.

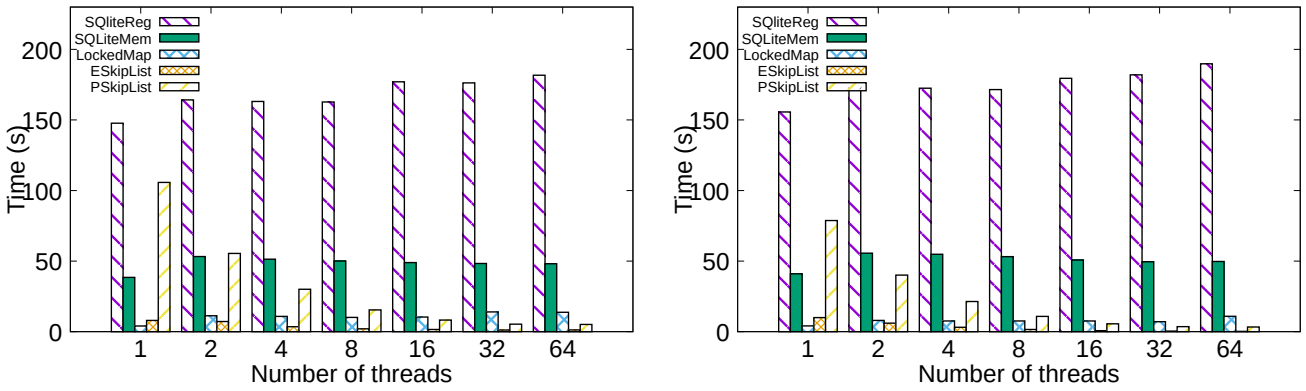
When increasing the number of threads, we can observe clear scalability trends. SQLiteMem and SQLiteReg are not scalable and actually exhibit lower performance for an increasing number of threads. The same is true for LockedMap, whose performance degradation is significant: 64 threads take 3x longer to finish than a single thread. All three approaches are locking based, which ultimately is the cause of this slowdown. On the other hand, ESkipList and PSkipList are highly scalable thanks to their lock-free design: 64 threads are 6.6x and, respectively, 20x faster than a single thread. This gives PSkipList a significant edge against the other approaches: it is 30x faster than SQLiteReg, 10x faster than SQLiteMem, 2.6x faster than LockedMap and only 4x slower than ESkipList, all while delivering full persistency.

E. Single Node: Concurrent Key History and Find

Our next series of experiments focuses on key history and find queries. Starting from the state obtained after running the previous set of experiments (i.e., $N = 1000000$ inserts followed by N removes), we run a third step that inserts another N different pre-generated key-value pairs. Therefore, we have a total of $P = 2 \cdot N$ keys, each of which is holding a version history reflecting one insert or an insert followed by a remove.

Next, each of the T threads chooses N/T random keys out P and extracts the version history / runs a find query (using a random version) for each key. We measure the total time needed by all threads to finish. Similarly to the previous set of experiments, we perform a strong scalability study by keeping N fixed and varying T in the range $1..64$.

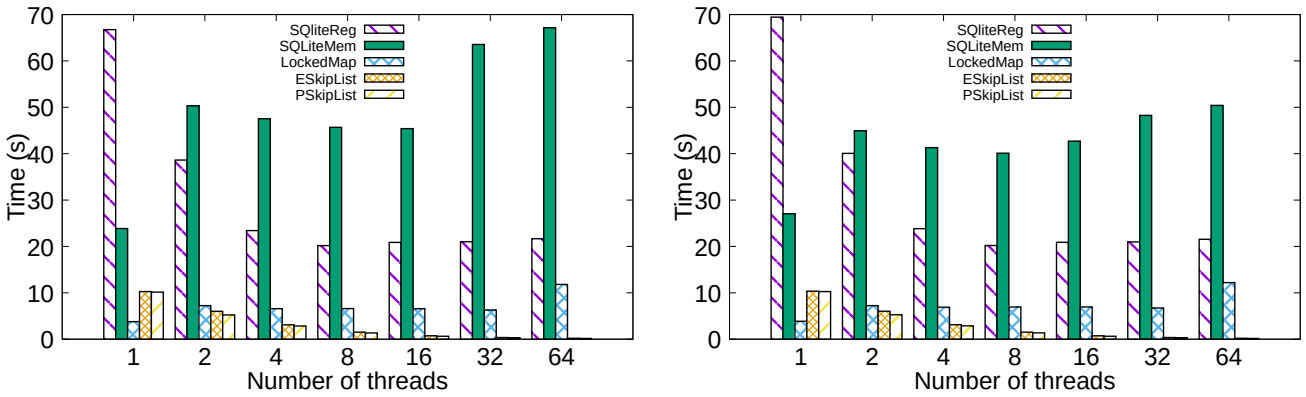
The results are reported in Figure 3. As expected, key history and find queries have similar behavior for all five approaches, with one notable exception for SQLiteMem, which exhibits performance degradation with increasing number of threads, especially visible in the case of key history. To explain this effect, remember that SQLiteMem shares the memory pages between all threads. Therefore, an increasing number of threads causes higher access competition, which ultimately translates to a bottleneck. Key history queries further amplify this competition because they involve multiple rows in the



(a) Insert operations (unique keys). Lower is better.

(b) Remove operations. Lower is better.

Fig. 2: Strong scalability of insert and remove operations on a single node using a variable number of threads.



(a) Extract key history. Lower is better.

(b) Find queries. Lower is better.

Fig. 3: Strong scalability of extract key history and find queries on a single node using a variable number of threads.

table (unlike the case of find queries where single rows are involved).

As expected, LockedMap is much faster than the other approaches for a single thread but experiences increasingly larger performance degradation under concurrency, which becomes especially visible for 64 threads. On the other hand, the other approaches see performance improvement for an increasing number of threads, including SQLiteReg (which unlike SQLiteMem keeps a separate page cache for each thread). However, the performance of SQLiteReg quickly flattens starting with 8 threads, while ESkipList and PSkipList continue to scale all the way up to 64 threads. Interesting to note is that PSkipList has no performance penalty compared with ESkipList, despite storing the version histories in persistent memory. At the maximum of 64 threads, both ESkipList and PSkipList are 130x, 420x, and 75x faster than SQLiteReg, SQLiteMem and LockedMap.

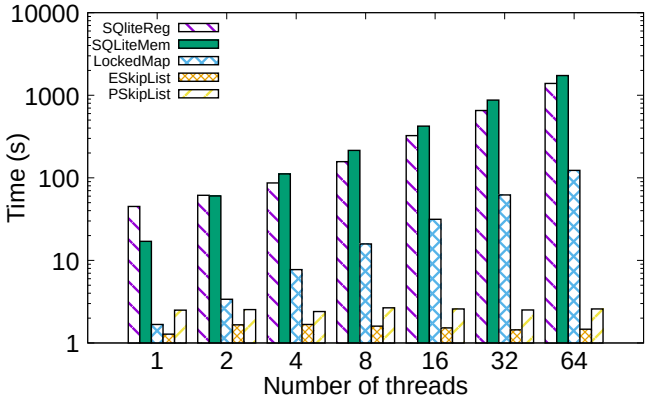


Fig. 4: Weak scalability of extract snapshot on a single node using a variable number of threads. Lower is better. Note the Y axis is represented using logscale.

F. Single Node: Concurrent Extract Snapshot

Next we focus on extract snapshot queries. Recall that these are complex operations that need to extract a sorted set of all key-value pairs if a given snapshot version number. To evaluate this scenario, we start from the previous state ($P =$ 2000000 distinct keys). Then, we run T concurrent extract snapshot queries, one per thread and measure the time taken by all threads to finish. The version number of the snapshot is randomly chosen by each thread. Unlike the previous set of experiments, this is a weak scalability study: the amount of

work per thread is similar but the overall work increases for an increasing number of threads.

The results are reported in Figure 4. As can be observed, SQLiteReg and SQLiteMem are lagging behind even for a single thread. ESkipList is almost 2x faster than LockedMap, which can be explained by the fact that walking over a red-black tree is not as efficient as iterating over the last level of a skip list. PSkipList is close but has slightly higher overheads due to reading from persistent memory.

For an increasing number of threads, only ESkipList and PSkipList maintain perfect weak scalability, while the other approaches are struggling. At 64 threads, the differences are so high that it was necessary to use a log scale for the Y axis. Specifically, ESkipList is 2.4x faster than PSkipList, 1260x faster than SQLiteReg, 1575x faster than SQLiteMem and 110x faster than LockedMap.

G. Single Node: Concurrent Find after Restart

Next, we evaluate the performance of PSkipList when restarting from a persisted snapshot. Since we adopt a hybrid approach that requires the reconstruction of the skip list on restart, the first set of experiments focuses on this aspect. To this end, we save the state of the dictionary after inserting $P = 2000000$ distinct keys (same state used in the previous experiments). Then, we measure the time needed to reconstruct the skip list for an increasing number of threads T.

As can be observed in Figure 5a, the reconstruction process is highly scalable, dropping from 17s for one thread to a little over 2s for 64 threads. Given that restart is an infrequent operation, such a low overhead is more than justified considering the performance gain and storage space reduction it provides during normal operation.

Note that after the skip list was reconstructed, only the persistent pointer to the history of each key is cached in-memory. Therefore, find queries will have a slight performance penalty to access the version history. To evaluate this effect, repeat the same find experiment presented in Section V-E (each of the T threads chooses N/T random keys out P). We compare our approach with SQLiteReg, which persists both the table and indices after shutdown, therefore it has all required information readily available on restart. The results are depicted in Figure 5b.

As expected, SQLiteReg remains efficient after restart, showing no significant difference compared with its counterpart in Figure 3b. Our approach shows a similar behavior: compared with the case when the cache is warm, the performance overhead is less than 9% even at the maximum of 64 threads. This allows our approach to maintain a growing advantage over SQLiteReg for an increasing number of threads, reaching a speed-up of up to 150x speedup at 64 threads. In fact, even if we add the rebuild overhead to the duration of the find queries, our approach is still 10x faster than SQLiteReg.

H. Multiple Nodes: Distributed Find/Extract Snapshot

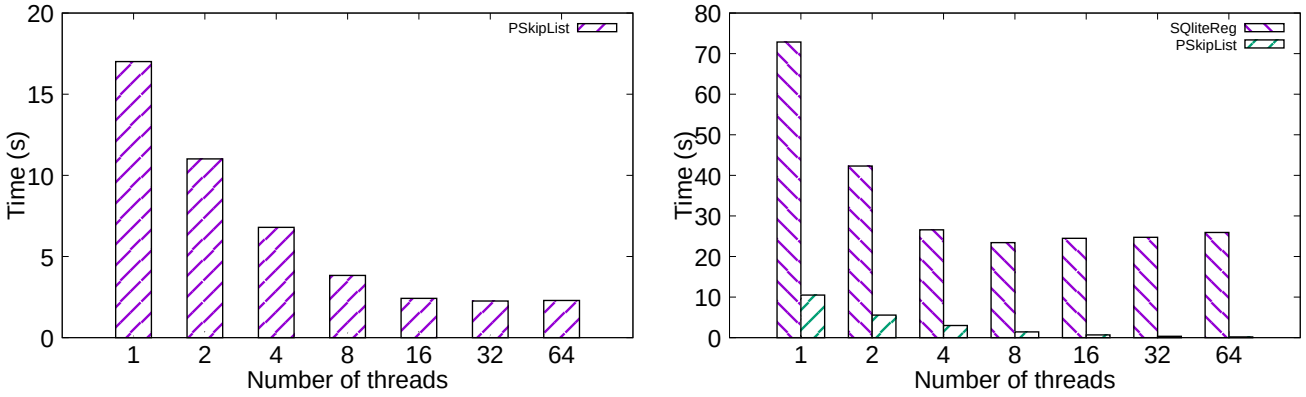
Our final series of experiments studies the behavior of our proposal at large scale. To this end, we deploy a set of K processes (MPI ranks), each of which runs on a separate compute node and is responsible to manage a distinct partition of N key-value pairs. Each partition was pre-generated and its entries were inserted in a local key-value store. We are interested in the horizontal scalability of find and extract snapshot queries across all MPI ranks. To this end, rank 0 acts as the initiator and broadcasts the query to all other ranks, then collects the results.

We fix $N = 100000$, an order of magnitude lower than in the previous experiments, which is necessary in order to compensate for the fact that we are using a large number of nodes. To emphasize the horizontal scalability, we issue only one query at a time, therefore it is enough to use only one thread per MPI rank. However, it is important to note that queries can also run in bulk mode (multiple queries in a single broadcast) or in parallel by different ranks (by using different communicators). In this case, the queries can take advantage of the multi-threaded vertical scalability sustained by our approach, as shown in the previous sections. Since this aspect is complementary, we limit our study to a single thread.

First, we focus on find queries, which are implemented in two steps using MPI collectives: broadcast the query to all nodes, then reduce the replies of all ranks to obtain the final answer. Specifically, we let rank 0 execute N queries and measure the throughput. The key and version number of each query is randomly chosen. The results are depicted in Figure 6. As can be observed, with an increasing number of nodes, the throughput drops from up to 20K queries/second to 7K queries/second. This effect is more noticeable for a small number of nodes but quickly stabilizes. It can be explained by the fact that a large part of the overhead can be attributed to the MPI collectives, whose scalability bounds the throughput of the queries. Nevertheless, the overhead of the individual find queries is still significant, despite the fact that their execution on the MPI ranks is embarrassingly parallel. This is visible when comparing SQLiteReg and PSkipList: the latter consistently achieves as 25% better throughput regardless of the number of nodes.

Next, we focus on extract snapshot queries, which are implemented using two approaches: (1) a naive approach (NaiveMerge) that first gathers the results of executing the extract snapshot queries on all MPI ranks individually, then performs a K-way merge on rank 0; (2) an optimized approach (OptMerge) that combines recursive doubling with multi-threaded merge (as described in Section IV-A). We use the highest version number for the queries, which means each rank contributes its whole partition, thus a total of $N \cdot K$ pairs need to be gathered and merged.

First, we study the gather performance, which represents the lowest possible overhead of accessing the whole snapshot without preserving a globally sorted key order. As can be observed in Figure 7, PSkipList maintains a speedup over SQLiteReg ranging from 2x (512 nodes) to 5x (8 nodes), which shows that the performance of extract snapshot queries on individual nodes plays a crucial role at scale, despite an increasing communication overhead needed to collect all query



(a) Skip list reconstruction overhead after restart. Lower is better.

(b) Find queries with cold cache after restart. Lower is better.

Fig. 5: Scalability of skip list reconstruction and find queries on a single node after restart using a variable number of threads.

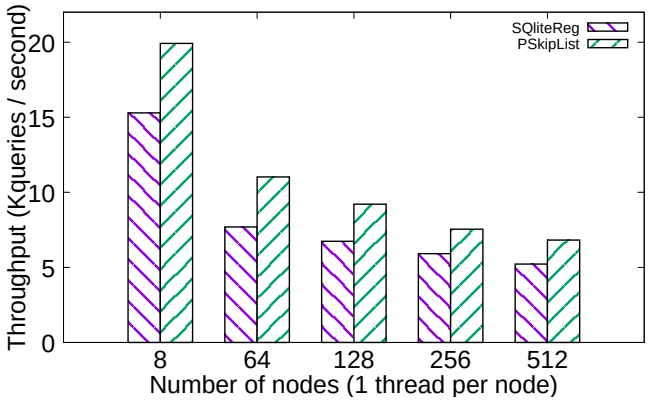


Fig. 6: Find throughput for an increasing number of nodes using one thread. Higher is better.

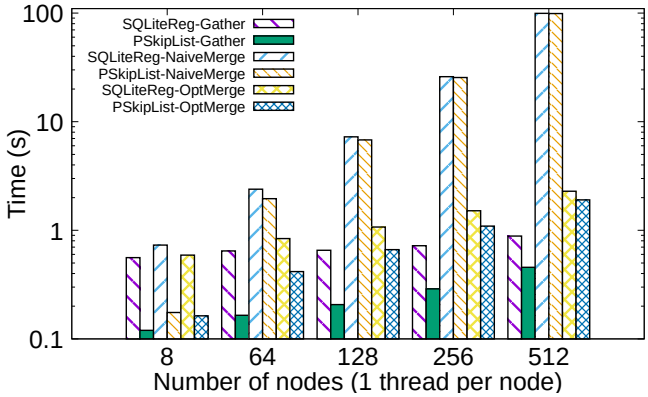


Fig. 7: Scalability of extract snapshot for an increasing number of nodes using NaiveMerge and OptMerge, with MPI_Gather as reference. Lower is better. Note the Y axis is represented using logscale.

results (implemented using MPI_Gather collective operations). Attempting to merge the results on a single node after gathering them leads to poor scalability, as can be observed in the trend of SQLiteReg–NaiveMerge and PSkipList–NaiveMerge: both approaches become at least two orders of magnitude slower for 512 nodes. On the other hand, our optimized merge approach exhibits much better scalability: for 512 nodes, it is 50x faster than the naive approach. Furthermore, this overhead is small enough to emphasize the performance benefit of PSkipList over SQLiteReg, which reaches 20% for 512 nodes.

VI. Conclusions

In this paper we focus on the problem of designing ordered key-value stores with multi-versioning support that are scalable under concurrent access (vertically and horizontally) and leverage persistent memories to provide durability. To this end, we propose a hybrid data structure and associated algorithms that combine ephemeral indexing based on lock-free skip lists with a compact incremental representation optimized for persistent memories.

We conducted extensive vertical scalability experiments that compare our proposal with several other implementations based on SQL database engines, lock-based red-black trees and lock-free skip lists. Under concurrency, our proposal is several orders of magnitude faster (30x for insert, 130x for find, 520x for extract snapshot) than SQL database engines. Furthermore, it is also faster than ephemeral lock-based red-black trees (2.6x for insert, 75x for find, 45x for extract snapshot). Additionally, we studied the horizontal scalability of our proposal by comparing it with a naive merge approach, in which case we obtained a 50x speedup, which is enough to maintain a lead over SQL databases even when using a single thread per node.

Encouraged by these promising results, in future work we plan to explore how to leverage the proposed data structures as a building block for data management systems that emphasize versioning and intermediate representations of datasets, such as *DataStates* [3]. Furthermore, we are also planning to address one of the limitations of our work regarding the need to traverse the whole set of keys even if they are not pertinent to the requested version. To this end, we are considering extensions to the ephemeral index based on a lock-free skip list.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

REFERENCES

[1] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: A proposal for an exascale storage system," in *SC '16: The 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, Utah, 2016, pp. 50:1–50:12.

[2] H. Liu, B. Nicolae, S. Di, F. Cappello, and A. Jog, "Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer," in *CLUSTER'21: The 2021 IEEE International Conference on Cluster Computing*, Portland, USA, 2021.

[3] B. Nicolae, "DataStates: Towards Lightweight Data Models for Deep Learning," in *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference*, Nashville, United States, 2020.

[4] A. Baldassin, J. a. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Comput. Surv.*, vol. 54, no. 7, 2021.

[5] K. Ouaknine, O. Agra, and Z. Guz, "Optimization of rocksdb for redis on flash," in *ICCDA '17: The 2017 International Conference on Compute and Data Analysis*. Lakeland, USA: Association for Computing Machinery, 2017, p. 155–161.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., 2001.

[7] R. Sedgewick and K. Wayne, *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016.

[8] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, p. 668–676, 1990.

[9] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *SIGMOD '70: The 1970 ACM SIGFIDET (SIGMOD) Workshop on Data Description, Access and Control*. Houston, USA: Association for Computing Machinery, 1970, p. 107–141.

[10] H. Park and K. Park, "Parallel algorithms for red–black trees," *Theoretical Computer Science*, vol. 262, no. 1, pp. 415–435, 2001.

[11] G. Graefe, "A survey of b-tree locking techniques," *ACM Trans. Database Syst.*, vol. 35, no. 3, 2010.

[12] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[13] A. Natarajan, L. H. Savoie, and N. Mittal, "Concurrent wait-free red black trees," in *Stabilization, Safety, and Security of Distributed Systems*. Cham: Springer International Publishing, 2013, pp. 45–60.

[14] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Commun. ACM*, vol. 34, no. 10, p. 78–92, 1991.

[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[16] "Lustre: A parallel file system that supports many requirements of leadership class hpc simulation environments," https://www.lustre.org/.

[17] M.-A. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann, "Challenges and solutions for tracing storage systems: A case study with spectrum scale," *ACM Trans. Storage*, vol. 14, no. 2, 2018.

[18] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: A scalable, reliable storage service for petabyte-scale storage clusters," in *PDSW '07: The 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, Reno, Nevada, 2007, p. 35–44.

[19] S. Chacon and B. Straub, *Pro Git*, 2nd ed. Berkely, CA, USA: Apress, 2014.

[20] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, pp. 169–184, 2011.

[21] "Dolt: A sql database that supports clone, branch, and merge," https://www.dolthub.com/.

[22] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. undefineduszczak, M. undefinedwitakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta lake: High-performance acid table storage over cloud object stores," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3411–3424, 2020.

[23] R. Chard, L. Ward, Z. Li, Y. Babuji, A. Woodard, S. Tuecke, K. Chard, B. Blaiszik, and I. Foster, "Publishing and serving machine learning models with dlhub," in *PEARC '19: Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, Chicago, USA, 2019.