Oak: A Scalable Off-Heap Allocated Key-Value Map

Anonymous Author(s)

Abstract

Efficient ordered in-memory key-value (KV-)maps are paramount for the scalability of modern data platforms. In managed languages like Java, KV-maps face unique challenges due to the high overhead of garbage collection (GC).

We present Oak, a scalable concurrent KV-map for environments with managed memory. Oak offloads data from the managed heap, thereby reducing GC overheads and improving memory utilization. An important consideration in this context is the programming model since a standard object-based API entails moving data between the on- and off-heap spaces. In order to avoid the cost associated with such movement, we introduce a novel *zero-copy* (ZC) API. It provides atomic get, put, remove, and various conditional put operations such as *compute* (in-situ update).

We have released an open-source Java version of Oak. We further present a prototype Oak-based implementation of the internal multidimensional index in Apache Druid. Our experiments show that Oak is often 2x faster than Java's state-of-the-art concurrent skiplist.

1 Introduction

Concurrent ordered *key-value (KV-)maps* are an indispensable part of today's programming toolkits. Doug Lee's ConcurrentSkipListMap [35], for instance, has been widely used for more than a decade. Such maps are essential for building real-time data storage, retrieval, and processing platforms including in-memory [7] and persistent [1, 21, 39] KV-stores. Another notable use case is in-memory analytics, whose market is projected to grow from $1.26B in 2017 to $3.85B in 2022 [6]. For example, the Apache Druid [25] analytics engine is adopted by Airbnb, Alibaba, eBay, Netflix, Paypal, Verizon Media, and scores of others.

Today, many data platforms are implemented in managed programming languages like Java [1, 4, 25, 39]. Despite recent advances, *garbage collection (GC)* algorithms struggle to scale with the volume of managed (on-heap) memory, often leading to low utilization and unpredictable performance [12]. This shortcoming has led a number of systems to adopt home-grown *off-heap* memory allocators, e.g., Cassandra [8], Druid [26], and HBase [5, 14, 40]. Most of these use cases, however, are limited to immutable data and avoid the complexity of implementing synchronization.

In this paper, we address the demand for scalable in-memory KV-maps in Java and similar languages. We design and implement Oak – *Off-heap Allocated KV-map* – an efficient ordered concurrent KV-map that self-manages its memory off-heap. Our design emphasizes (1) performance, (2) programming convenience, and (3) correctness under concurrency.

These objectives are facilitated by Oak's novel *zero-copy* (ZC) API, which allows applications to access and manipulate data directly in Oak's internal buffers, yet in a thread-safe way. For backward compatibility, Oak also supports the (less efficient) legacy KV-map API (in JDK, ConcurrentNavigableMap [34]). Either way, Oak preserves the managed-memory programming experience through internal garbage collection. We discuss our programming model in §2.

Oak achieves high performance by (1) reduced copying, (2) efficient data organization both on- and off-heap, and (3) lightweight synchronization. Oak further features a novel approach to expedite descending scans, which are prevalent in analytics use cases. Data organization is the subject of §3, and the concurrent algorithm appears in §4; we formally prove its correctness in the supplementary material.

We have released a Java implementation of Oak as an off-the-shelf open-source package under the Apache 2.0 License [42]. Its evaluation in §5 shows significant improvements over ConcurrentSkipListMap, e.g., 2x speeding up for puts and gets. Oak's descending scans are 10x faster than the state-of-the-art thanks to the built-in support for such scans. In terms of memory utilization, Oak can ingest over 30% more data within a given DRAM size.

Finally, §6 features a case study of integrating Oak into Apache Druid [25] – a popular open-source real-time analytics database. We re-implement Druid's centerpiece incremental index component around Oak. This speeds up Druid's data ingestion by above 80% and reduces the metadata space overhead by 50%.

We now describe Oak's key features and survey prior art.

1.1 Oak's design

Off-heap allocation and GC. The principal motivation for Oak is offloading data from the managed-memory heap. Oak allocates key and value buffers within large off-heap memory pools. This alleviates the GC performance overhead, as well as the memory overhead associated with the Java object layout. The internal memory reclamation policy is customizable, with a low-overhead default that serves big data systems well. Oak also supports fast estimation of its RAM footprint – a common application requirement [38].

For simplicity, Oak manages its metadata, e.g., the search index, on-heap; note that metadata is typically small and dynamic, and Java's memory manager deals with it well.

Zero-copy API. For backward compatibility, Oak exposes the legacy JDK8 ConcurrentNavigableMap API, where input

PPoPP'20, 2020, San Diego, CA, USA
2020.

and output parameters are Java objects. With off-heap storage, however, this interface is inefficient because it requires serialization and deserialization of objects in every query or update. This is particularly costly in case keys and values are big, as is common in analytics applications. To mitigate this cost, Oak offers a novel ZC API, allowing the programmer direct access to off-heap buffers, both for reading and for updating-in-place via user-provided lambda functions. Oak's internal GC guarantees safety – buffer space that might be referenced from outside Oak is not reclaimed.

Linearizability. Oak provides atomic semantics (linearizability) for traditional point access (get, put, remove) as well as for in-situ updates (compute, put-if-absent, put-if-absent-compute-if-present). Note that consistency is ensured at the level of user data, i.e., lambda functions are executed atomically. In contrast, Java's concurrent collections do not offer atomic update-in-place (e.g., its compute method is not atomic). Supporting atomic conditional updates alongside traditional (unconditional) puts necessitated designing a new concurrent algorithm. We are not aware of any previous algorithm addressing this challenge.

Efficient metadata organization. Ordered map data structures like search trees and skiplists consist of many small objects ("nodes") with indirection among them. This induces penalties both on memory management – due to fragmentation – and on search time – because of lack of locality. Similarly to a number of recently suggested data structures [13, 16, 17], Oak's metadata is organized in contiguous *chunks*, which reduces the number of metadata objects and speeds up searches through locality of access. This is challenging in the presence of variable-sized keys and values; previous chunk-based data structures [13, 16, 17] maintain fixed-size keys and values inline, without the additional indirection level required to support variable-sized data.

Fast two-way scans. Like ConcurrentSkipListMap, and as required by many applications, Oak provides iterators to support (non-atomic) scans. The scans are not atomic in the sense that the set of keys in the scanned range may change during the scan. Supporting atomic scans would be more costly in time and space, and is rarely justified in analytics scenarios where results are inherently approximate.

Although analytics require both ascending and descending range scans, existing concurrent data structure do not have built-in support for the latter. Rather, descending scans are implemented as a sequence of gets. We leverage Oak's chunk-based organization to expedite descending scans without the complexity of managing a doubly-linked list.

Summary. All in all, Oak is the first concurrent KV-map explicitly designed to address big-data demands, including off-heap memory management, zero copy API, in-place atomic updates optimized for variable-size keys and values, index locality for fast searches, and efficient descending scans.

1.2 Related work

Substantial efforts have been dedicated to developing efficient concurrent ordered maps [10, 11, 13, 16–20, 22–24, 27, 28, 30, 32, 35, 41, 44]. However, most of these works do not implement functionalities such as update-in-place, conditional puts, and descending iterators. Many of these are academic prototypes, which hold only an ordered key set and not key-value pairs [19, 22, 24, 27, 30, 41]. Moreover, the ones that do hold key-value pairs typically maintain fixed-size keys and values [13, 16, 17] and do not support large, variable-size keys and values as Oak does.

Java collections such as ConcurrentSkipListMap [35] do support general objects as keys and values, and also implement the full ConcurrentNavigableMap API. Nevertheless, their compute is not necessarily atomic, their organization is not chunk-based and so searches do not benefit from locality, and their descending scans are slow as we show in §5.

Chunk-based allocation is used in existing concurrent data structures [13, 16, 17] but not with variable-size entities or off-heap allocation. It is also a common design pattern in persistent (disk-resident) key-value storage. Oak, in contrast, is memory-resident.

Off-heap allocation is gaining popularity in various systems [8, 9, 14, 26, 40]. Yet the only off-the-shelf *data structure library* implementation that we are aware of is within the MapDB open source package [36], which implements Sagiv's concurrent B*-tree [43]. We are not aware of safety guarantees of this implementation wrt in-situ updates; it is also at least an order-of-magnitude slower than Oak (§5).

2 Programming Model

Oak is unique in offering a map interface for self-managed data. This affects the programming model as it allows applications to access data in Oak's buffers directly. §2.1 discusses the serialization of application objects into *Oak buffers*. §2.2 presents Oak's novel *zero-copy API*, which reduces the need for serialization and deserialization (and hence copying).

2.1 Oak buffers and serialization

Oak keys and values are variable-sized. Keys are immutable, and values may be modified. In contrast to Java data structures holding Java objects, Oak stores data in internal buffers. To convert objects (both keys and values) to and from their *serialized* forms, the user must implement a (1) serializer, (2) deserializer, and (3) serialized size calculator. To allow efficient search over buffer-resident keys, the user is further required to provide a comparator.

Oak's insertions use the size calculator to deduce the amount of space to be allocated, then allocate space for the given object, and finally invoke the serializer to write the object to the allocated space. By using the user-provided serializer, we create the binary representation of the object directly into Oak's internal memory.

ZeroCopyConcurrentNavigableMap	**(Legacy) ConcurrentNavigableMap**
Queries – get and scans	
OakRBuffer get(K)	V get(K)
Set⟨OakRBuffer⟩ keySet()	Set⟨K⟩ keySet()
Set⟨OakRBuffer⟩ valueSet()	Set⟨V⟩ valueSet()
Set⟨OakRBuffer, OakRBuffer⟩ entrySet()	Set⟨K, V⟩ entrySet()
Updates	
void put(K, V)	V put(K, V)
void remove(K)	V remove(K)
boolean putIfAbsent(K, V)	V putIfAbsent(K, V)
boolean computeIfPresent(K, Function(OakWBuffer))	*non-atomic* V computeIfPresent(K, Function(K,V))
boolean putIfAbsentComputeIfPresent(K, V, Function(OakWBuffer))	*non-atomic* V merge(K, V, Function(K,V))

Table 1. Oak's zero-copy API versus the legacy ConcurrentNavigableMap API. Key and value types are K and V, resp. Get and scans return OakRBuffers instead of objects. Updates do not return the old value in order to avoid copying.

Oak provides `OakRBuffer` and `OakWBuffer` abstractions for internal readable and writable buffers. These types are lightweight on-heap facades to off-heap storage, which provide the application with familiar managed object semantics. For example, they may exist arbitrarily long and the memory behind them will not be reclaimed. Furthermore, user code can access them without worrying about concurrent access. Since keys are immutable, they are always accessed through `OakRBuffers`, whereas values can be accessed both ways.

2.2 Zero-copy API

We introduce ZeroCopyConcurrentNavigableMap, Oak's ZC API. Table 1 compares it to the essential methods of the legacy API using a slightly simplified Java-like syntax, neglecting some technicalities (e.g., the use of Collections instead of Sets in some cases). To use the ZC API, an application creates a ConcurrentNavigableMap-compliant Oak map, and accesses it through the `zc()` method, e.g., calling `map.zc().get(key)` instead of the legacy `map.get(key)`.

The API is changed only in so far as to avoid copying. The `get()` and scans (`keySet()`, `valueSet()`, and `entrySet()`) return Oak buffers instead of Java objects, while continuing to offer the same functionality. In particular, scans offer the Set interface with its standard tools such as a stream API for mapreduce-style processing [37]. Likewise, sub-range and reverse-order views are provided by familiar `subMap()` and `descendingMap()` methods on Sets.

The update methods differ from their legacy counterparts in that they do not return the old value (in order to avoid copying it). The last two – `computeIfPresent()` and `putIfAbsentComputeIfPresent()` – atomically update values in place. Both take a user (lambda) function to apply to the `OakWBuffer` of the value mapped to the given key. Unlike the legacy map, Oak ensures that the lambda is executed atomically, exactly once, and extends the value's memory allocation if its code so requires.

While all operations are atomic, `get()` returns access to the same underlying memory buffer that other operations update in-place, while the granularity of Oak's concurrency control is at the level of individual method calls on that buffer (e.g., reading a single integer from it). Therefore, buffer access methods may encounter different values – and even value deletions[1] – when accessing a buffer multiple times. This is an inevitable consequence of avoiding copying.

3 Data Organization

Oak allocates keys and values off-heap and metadata on-heap, as described in §3.1. §3.2 presents Oak's simple internal memory manager, which controls off-heap allocation and garbage collection. To allow user code safe access to data in Oak buffers without worrying about concurrent access or dynamic reallocation, Oak employs an indirection layer called *handle*, as discussed in §3.3.

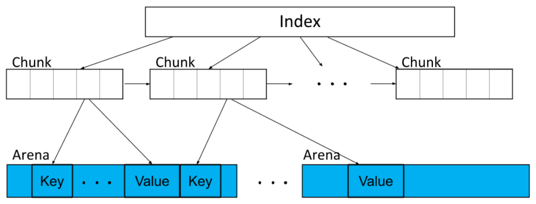
3.1 Off-heap data and on-heap metadata

Oak's on-heap metadata maps keys to values. It is organized as a linked list of *chunks* – large blocks of contiguous key ranges, as in [16]. Each chunk has a *minKey*, which is invariant throughout its lifespan. We say that key k is in the *range of chunk C* if $k \geq C.minKey$ and $k < C.next.minKey$.

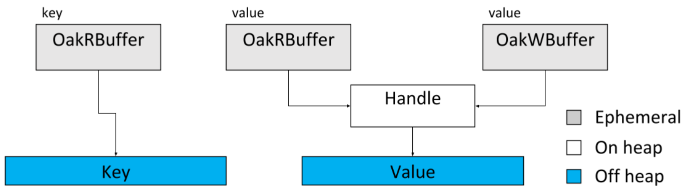
A chunk includes a linked list of *entries*, sorted in ascending key order. The entries point to off-heap keys and values. Oak makes sure that each key appears in at most one entry.

To allow fast access, we follow the approach of [13, 15, 31, 32, 44] and add an *index* that maps minKeys to their respective chunks, as illustrated in Figure 1a. The index is updated in a lazy manner, and so it may be inaccurate, in which case, locating a chunk may involve a partial traversal of the chunk linked list. A `locateChunk(k)` method returns the chunk whose range includes key k by querying the index and traversing the chunk list if needed.

[1]A `get()` method throws a `ConcurrentModificationException` in case the mapping is concurrently deleted.



(a) Index, chunks, and arenas



(b) Buffers, handles, and data

Figure 1. Oak data organization.

3.2 Memory management

Oak offers a simple default memory manager that can be overridden by applications. The default manager is suitable for real-time analytics settings, where dynamic data structures used to ingest new data exist for limited time [1, 25] and deletions are infrequent.

Oak's allocator manages a shared pool of large (100MB by default) pre-allocated off-heap *arenas*, and supports multiple Oak instances. Each arena is associated with a single Oak instance and returns to the pool when that instance is disposed. Key and value buffers are allocated from the arena's flat free list using a first-fit approach; they return to the free list upon KV-pair deletion or value resize.

The memory manager can efficiently compute the total size of an Oak instance's off-heap footprint.

3.3 Handles and concurrency control

The handle abstraction provides access to off-heap mutable memory, bridging between the on- and off-heap parts of Oak, as shown in Figure 1b. There is a single handle per value, which the `OakRBuffer` and `OakWBuffer` objects wrap. Keys are immutable, so their `OakRBuffers` reference off-heap memory directly, without handles. Since the handle is an on-heap object, it remains reachable to threads that hold Oak buffers that wrap it after the value (off-heap) is reclaimed.

The handle ensures that all methods are applied to the buffers atomically. The default implementation uses a read-write lock, but can be overridden, e.g., by an optimistic approach. The handle also interacts with the memory manager in order to dynamically resize values (requesting a new allocation and copying the buffer to it if needed), and informs it when to reclaim buffers that are no longer needed.

Once a value is removed from Oak, the handle assures that no thread will attempt to read it, since that memory may be reclaimed. To this end, the handle performs a logical remove by marking itself as deleted. A key is deemed present in Oak only if it is associated with a non-deleted handle. The handle further offers update and compute mechanisms that are used by Oak to atomically modify values.

4 Oak Algorithm

We now describe the Oak algorithm. The ZC and legacy API implementations share the most of it. We focus here on the ZC variant; supporting also the legacy API mainly entails serialization and deserialization.

We begin in §4.1 with an overview of chunk objects. We then proceed to describe Oak's operations. In §4.2 we discuss Oak's queries, namely `get` and scan. We divide our discussion of update operations into two types: insertion operations that may add a new value to Oak are discussed in §4.3, whereas operations that only take actions when the affected key is already in Oak are given in §4.4. To argue that Oak is correct, we identify in §4.5 *linearization points* for all operations, so that concurrent operations appear to execute in the order of their linearization points. A formal correctness proof is given in the supplementary material.

4.1 Chunk objects

A chunk object exposes methods for searching, allocating, and writing, as we describe in this section. In addition, the chunk object has a *rebalance* method, which splits chunks when they are over-utilized, merges chunks when they are under-used, and reorganizes chunks' internals. Our rebalance is implemented as in previous constructions [13, 17]. Since it is not novel and orthogonal to our contributions, we do not detail it, but rather outline its guarantees. Implementing the remaining chunk methods is straightforward.

When a new chunk is created (by rebalance), some prefix of the entries array is filled with data, and the suffix consists of empty entries for future allocation. The full prefix is sorted, that is, the linked list successor of each entry is the ensuing entry in the array. The sorted prefix can be searched efficiently using binary search. When a new entry is inserted, it is stored in the first free cell and connected via a bypass in the sorted linked list. If insertion order is random, inserted entries are most likely to be distributed evenly between the ordered prefix entries, keeping the search time low.

Rebalance guarantees. The rebalancer preserves the integrity of the chunks list in the following sense: Consider a `locateChunk`$(k0)$ operation that returns C_0 at some time t_0 in a run, and a traversal of the linked list using next pointers from C_0 reaching a chunk whose range ends with $k1$ at time t_1. For each traversed chunk C, choose an arbitrary time $t_0 \leq t_C \leq t_1$ and consider the sequence of keys stored in C at time t_C. Let T be the concatenation of these sequences. Then:

RB1 T includes every key $k \in [k0, k1]$ that is inserted before time t_0 and is not removed before time t_1;

RB2 T does not include any key that is either not inserted before time t_1 or is removed before time t_0 and not re-inserted before time t_1; and

RB3 T is sorted in monotonically increasing order.

Chunk methods. The chunk's `LookUp(k)` method searches for an entry corresponding to key k. This is done by first running a binary search on the entries array prefix and continuing the search by traversing the entries linked list. Note that Oak ensures that there is at most one relevant entry.

We rely on three allocation methods: `allocateHandle` allocates a new handle; `allocateEntryAndKey` allocates a new entry and also allocates and writes the given key that it points to (off-heap); and `writeValue` allocates space for the value (outside the chunk), writes the value to it, and links the handle to the value. Allocations use atomic hardware operations like F&A, so that the same space is not allocated twice. `entriesLLputIfAbsent` adds an entry to the linked list; it uses CAS in order to preserve the invariant of a key not appearing more than once. If it encounters an entry with the same key, then it returns the encountered entry.

In-chunk allocations (of handles and entries) may trigger a rebalance "under the hood". In this case, the allocate procedure fails returning ⊥ and Oak retries the update. While a chunk is being rebalanced, calls to `entriesLLputIfAbsent` fail and return ⊥.

Update operations inform the rebalancer of the action they are about to perform on the chunk by calling the `publish` method. This method, too, fails in case the chunk is being rebalanced. In principle, rebalance may help published operations complete (for lock-freedom), but for simplicity, our description herein assumes that it does not. Hence, we always retry an operation upon failure. When the update operation has finished its published action, it calls `unpublish`.

Whereas chunk update methods that encounter a rebalance fail (return ⊥), methods that do not modify the entries list proceed concurrently with rebalance without aborting. This includes `lookUp`, `writeValue`, and `unpublish`.

4.2 Queries – get and scans

The `get` operation is given in Algorithm 1. It returns a read-only view (`oakRBuffer`) of the handle that holds the value that is mapped to the given key. Since it is only a view and not a copy of the value, if the value is then updated by a different operation, the view will refer to the updated value. Furthermore, a concurrent operation can remove the key from Oak, in which case the handle will be marked as deleted; reads from the `oakRBuffer` check this flag and throw an exception in case the value is deleted.

The algorithm first locates the relevant chunk and calls lookUp (line 3) to search for an entry with the given key. Then, it finds the handle and checks if it is deleted. If an entry holding a valid and non-deleted handle is found, it creates a new `oakRBuffer` that points to the handle and returns it. Otherwise, `get` returns NULL.

Algorithm 1 Get

```
1: procedure GET(key)
2:     C, ei, hi, handle ← ⊥
3:     C ← locateChunk(key) ; ei ← C.lookUp(key)
4:     if ei ≠ ⊥ then hi ← C.entries[ei].hi
5:     if hi ≠ ⊥ then handle ←C.handles[hi]
6:     if handle = ⊥ ∨ handle.deleted then return NULL
7:     else return new OakRBuffer(handle)
```

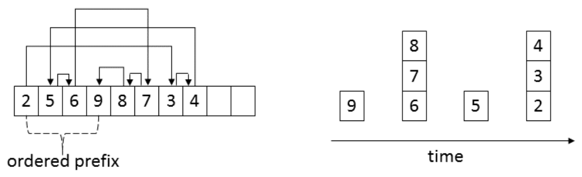


Figure 2. Example entries linked list (left) and stacks built during its traversal by a descending scan (right).

The ascending scan begins by locating the first chunk with a relevant key in the scanned range using `locateChunk`. It then traverses the entries within each relevant chunk using the intra-chunk entries linked list, and continues to the next chunk in the chunks linked list. The iterator returns an entry it encounters only if its handle index is not ⊥ and the handle is not deleted. Otherwise, it continues to the next entry.

The descending iterator begins by locating the *last* relevant chunk. Within each relevant chunk, it first locates the last relevant entry in the sorted prefix, and then scans the (ascending) linked list from that entry until the last relevant entry in the chunk, while saving the entries it traverses in a stack. After returning the last entry, it pops and returns the stacked entries. Upon exhausting the stack and reaching an entry in the sorted prefix, the iterator simply proceeds to the previous prefix entry (one cell back in the array) and rebuilds the stack with the linked list entries in the next bypass.

Figure 2 shows an example of an entries linked list and the stacks constructed during its traversal. In this example, the ordered prefix ends with 9, which does not have a next entry, so we can return it. Next, we move one entry back in the prefix, to entry 6, and traverse the linked list until returning to an already seen entry within the prefix (9 in this case), while creating the stack $8 \to 7 \to 6$. We then pop and return each stack entry. Now, when the stack is empty, we again go one entry back in the prefix and traverse the linked list. Since after 5 we reach 6, which is also in the prefix, we can return 5. Finally, we reach 2 and create the stack with entries $4 \to 3 \to 2$, which we pop and return. When exhausting a chunk, the descending scan queries the index again, but now for a the chunk with the greatest minKey that is strictly smaller than the current chunk's minKey.

By RB1-3 it is easy to see that the scan algorithm described above guarantees the following:

1. A scan returns all relevant keys that were inserted to Oak before the start of the scan and not removed until its end.
2. A scan does not return keys that were never present or were removed from Oak before the start of the scan and not inserted until it ends.
3. A scans does not return the same key more than once.

Note that relevant keys inserted or removed concurrently with a scan may be either included or excluded.

4.3 Insertion operations

The three insertion operations – `put`, `putIfAbsent`, and `putIfAbsentComputeIfPresent` – use the `doPut` function in Algorithm 2. `DoPut` first locates the relevant chunk and searches for an entry. We then distinguish between two cases: if a non-deleted handle is found (case 1: lines 21 – 28) then we say that the key is *present*. In this case, `putIfAbsent` returns `false` (line 22), `put` calls handle.put (line 23) to associate the new value with the key, and `putIfAbsentComputeIfPresent` calls handle.compute (line 25). The handle operations return `false` if the handle is deleted (due to a concurrent remove), in which case we retry (line 27).

In the second case, the key is absent. If we discover a removed entry that points to the same key but with `hi` $= \perp$ or a deleted handle, then we reuse this entry. Otherwise, we call `allocateEntryAndKey` to allocate a new entry as well as allocate and write the key that it points to (line 30), and then try to link this new entry into the entries linked list (line 31). Either way, we allocate a new handle (line 32). These functions might fail and cause a retry (line 33).

If `entriesLLputIfAbsent` receives $\perp$ as a parameter (because the allocation in line 30 fails) then it just returns $\perp$ as well. Otherwise, if it encounters an already linked entry then it returns it. In this case, the entry allocated in line 30 remains unlinked in the entries array and other operations never reach it; the rebalancer eventually removes it from the array. After allocations of the entry, key, and handle, we allocate and write the value (outside the chunk), and have the new handle point to it (line 35).

We complete the insertion by using CAS to make the entry point to the new handle index (line 38). Before doing so, we publish the operation (as explained in §4.1), which can also lead to a retry (line 37). After the CAS, we unpublish the operation, as it is no longer pending (line 39). If CAS fails, we retry the operation (line 41).

To see why we retry, observe that the CAS may fail because of a concurrent non-insertion operation that sets the handle index to $\perp$ or because of a concurrent insertion operation that sets the handle index to a different value. In the latter case, we cannot order (linearize) the current operation before the concurrent insertion, because the concurrent insertion operation might be a `putIfAbsent`, and would have returned `false` had the current operation preceded it.

Algorithm 2 Oak's insertion operations

```
8:  procedure PUT(key, val)
9:      doPut(key, val, ⊥, PUT)
10:     return
11: procedure PUTIFABSENT(key, val)
12:     return doPut(key, val, ⊥, PUTIF)
13: procedure PUTIFABSENTCOMPUTEIFPRESENT(key, val, func)
14:     doPut(key, val, func, COMPUTE)
15:     return

16: procedure DOPUT(key, val, func, operation)
17:     C, ei, hi, newHi, handle ← ⊥; result, succ ← true
18:     C ← locateChunk(key); ei ← C.lookUp(key)
19:     if ei ≠ ⊥ then hi ← C.entries[ei].hi
20:     if hi ≠ ⊥ then handle ←C.handles[hi]
21:     if handle ≠ ⊥ ∧ ¬handle.deleted then
    ▹ Case 1: key is present
22:         if operation = PUTIF then return false
23:         if operation = PUT then succ ← handle.put(val)
24:         if operation = COMPUTE then
25:             succ ← handle.compute(func)
26:         if ¬succ then
27:             return doPut(key, val, func, operation)
28:         return true
    ▹ Case 2: key is absent
29:     if ei = ⊥ then
30:         ei ← C.allocateEntryAndKey(key)
31:         ei ←C.entriesLLputIfAbsent(ei)
32:     newHi ←C.allocateHandle()
33:     if ei = ⊥ ∨ newHi = ⊥ then ▹ allocation or insertion failed
34:         return doPut(key, val, func, operation)
35:     C.writeValue(newHi, val)
36:     if ¬C.publish(ei, hi, newHi, func, operation) then
37:         return doPut(key, val, func, operation)
38:     result ← CAS(C.entries[ei].hi, hi, newHi)
39:     C.unpublish(ei, hi, newHi, func, operation)
40:     if ¬result then
41:         return doPut(key, val, func, operation)
42:     return true
```

4.4 Non-insertion operations

The non-removing updates, `computeIfPresent` and `remove`, use the `doIfPresent` function in Algorithm 3. It first locates the handle, and if there is none, returns `false` (line 47).

In `computeIfPresent`, if the handle exists and is not deleted (case 1), we run handle.compute and return `true` if it is successful (line 50). Otherwise (case 2), a subtle race may arise: it is possible for another operation to insert the key after we observe it as deleted and before this point. In this case, to ensure correctness, `computeIfPresent` must assure that the key is in fact removed. To this end, it performs a CAS to change handle index to $\perp$ (line 56). Since this affects the

Algorithm 3 Oak's non-insertion update operations

```
43: procedure doIfPresent(key, func, op)
44:     C, ei, hi, handle ← ⊥; res ← true
45:     C ← locateChunk(key); ei ← C.lookUp(key)
46:     if ei ≠ ⊥ then hi ← C.entries[ei].hi
47:     if hi = ⊥ then return false
48:     handle ←C.handles[hi]
49:     if ¬handle.deleted then
      ▷ Case 1: handle exists and not deleted
50:         if op = comp ∧ handle.compute(func) then
51:             return true
52:         if op = rm ∧ handle.remove() then
53:             return finalizeRemove(handle)
      ▷ Case 2: handle is deleted – ensure key is removed
54:     if ¬C.publish(ei, hi, ⊥, func, op) then
55:         return doIfPresent(key, func, op)
56:     res ← CAS(C.entries[ei].hi, hi, ⊥)
57:     C.unpublish(ei, hi, ⊥, func, op)
58:     if ¬res then return doIfPresent(key, func, op)
59:     return false

60: procedure computeIfPresent(key, func)
61:     return doIfPresent(key, func, comp)

62: procedure remove(key)
63:     doIfPresent(key, ⊥, rm)
64:     return

65: procedure finalizeRemove(prev)
66:     C, ei, hi, handle ← ⊥
67:     C ← locateChunk(key); ei ← C.lookUp(key)
68:     if ei ≠ ⊥ then hi ← C.entries[ei].hi
69:     if hi = ⊥ then return true
70:     handle ←C.handles[hi]
71:     if handle ≠ prev then return true
72:     if ¬C.publish(ei, hi, ⊥, ⊥, rm) then
73:         return finalizeRemove(prev)
74:     CAS(C.entries[ei].hi, hi, ⊥)
75:     C.unpublish(ei, hi, ⊥, ⊥, rm)
76:     return true
```

chunk's entries, we need to synchronize with a possibly ongoing rebalance, so we publish before the CAS and unpublish when done. If publish or CAS fails then we retry (lines 55 and 58). The operation returns `false` whenever it does not find the entry, or finds the entry but with ⊥ as its handle index (line 47), or CAS to ⊥ is successful (line 59).

In `remove`, if a non-deleted handle exists (case 1), it also updates the handle, in this case, marking it as deleted by calling handle.remove (line 52), and we say that the remove is *successful*. This makes other threads aware of the fact that the key has been removed, which suffices for correctness. However, as an optimization, `remove` also performs a second task after marking the handle as deleted, namely, marking the appropriate entry's handle index as ⊥. Updating the entry serves two purposes: first, rebalance does not check whether a handle is deleted, so changing the handle index to ⊥ is needed to allow garbage collection; second, updating the entry expedites other operations, which do not need to read the handle in order to see that it is deleted.

Thus, a successful `remove` calls `finalizeRemove`, which tries to CAS the handle index to ⊥. We have to take care, however, in case the handle index had already changed, not to change it to ⊥. To this end, `finalizeRemove` takes a parameter `prev` – the handle that `remove` marked as deleted. If the entry no longer points to it, we do nothing (line 71). We save in `prev` the handle itself and not the handle index, to avoid an ABA problem, since after a rebalance, the handle index might remain the same but reference a different handle. Since `remove` is linearized at the point where it marks the handle as deleted, it does not have to succeed in performing the CAS in `finalizeRemove`. If CAS fails, this means that either some insertion operation reused this entry or another non-insertion operation set the index to ⊥.

If `remove` finds an already deleted handle (case 2), it cannot simply return, since by the time remove notices that the handle is deleted, the entry might point to another handle. Therefore, similarly to `computeIfPresent`, it makes sure that the key is removed by performing a successful CAS of the handle index to ⊥ (line 56). In this case (case 2) it does not perform `finalizeRemove`, but rather retries if the CAS fails (line 58). Note the difference between the two cases: in case 1, we set the handle to deleted, and so changing the entry's handle index to ⊥ is merely an optimization, and should only occur if the entry still points to the deleted handle. In the second case, on the other hand, `remove` does not delete any handle, and so it *must* make sure that the entry's handle index is indeed ⊥ before returning.

4.5 Linearization points

In the supplementary material we show that Oak's operations (except for scans) are *linearizable* [33]; that is, every operation appears to take place atomically at some point (the linearization point, abbreviated *l.p.*) between its invocation and response. We now list the linearization points.

putIfAbsent – if it returns `true`, the l.p. is the successful CAS of handle index (line 38). Otherwise, the l.p. is when it finds a non-deleted handle (line 21).

put – if it inserts a new key, the l.p. is the successful CAS of handle index (line 38). Otherwise, the l.p. is upon a successful nested call to handle.put (line 23).

putIfAbsentComputeIfPresent – if it inserts a new key, the l.p. is the successful CAS of handle index (line 38). Otherwise, the l.p. is upon a successful nested call to handle.compute (line 25).

- **computeIfPresent** – if it returns `true`, the l.p. is upon a successful nested call to handle.compute (line 50). Otherwise, the l.p. is when the entry is not found, or it is found but with ⊥ as its handle index (line 47), or a successful CAS of handle index to ⊥ (line 56).
- **remove** – if it is successful, the l.p. is when a successful nested call to handle remove sets the handle to deleted (line 52). Otherwise, the l.p. is when the entry is not found, or handle index is ⊥ (line 47), or a successful CAS of handle index to ⊥ occurs (line 56).
- **get** – if it returns a handle, then the l.p. is the read of a non-deleted handle (line 6). If it returns NULL and there is no relevant entry then the l.p. is when `lookUp` (line 3) returns ⊥, or when get reads that the handle index is ⊥ (line 4). Otherwise, `get` reads a deleted handle (line 6). However, the l.p. cannot be the read of the `deleted` flag in the handle, since by that time, a new handle may have been inserted. Instead, the l.p. is the later between (1) the read of handle index by the same `get` (line 4) and (2) immediately after the set of `deleted = true` by some `remove` (note that exactly one `remove` set deleted to true).

5 Evaluation

We now evaluate Oak's Java implementation using synthetic benchmarks. In §6 below, we discuss a real-world use case.

5.1 Experiment setup

We generate a variety of workloads using the popular synchrobench tool [29]. Our hardware testbed features an industry-standard 12-core Xeon E5-4650 CPU with 192 GB RAM.

Compared solutions. Our ultimate goal is to offer Oak as an alternative to Java's standard skiplist. We therefore compare it to the JDK8 ConcurrentSkipListMap [35], which we refer to as Skiplist-OnHeap. To isolate the impact of off-heap allocation from other algorithmic aspects, we also implement an off-heap variant of the Java skiplist, which we call Skiplist-OffHeap. Note that whereas Skiplist-OnHeap offers an object-based API, Skiplist-OffHeap also exposes Oak's ZC API. Internally, Skiplist-OffHeap maintains a concurrent skiplist over an intermediate *cell* object. Each cell references a key buffer and a value buffer allocated in off-heap arenas through Oak's memory manager. This solution is inspired by off-heap support in production systems, e.g., HBase [5].

We also experimented with the open source concurrent off-heap B-tree implementation from MapDB [36], however it failed to scale to big datasets, peforming at least ten-fold slower than Oak; we omit these results for brevity.

Methodology. The exercised keys and values are 100B- and 1KB-big, respectively. In each experiment, a specific *range* of keys is accessed. Accessed keys are sampled uniformly at random from that range. The range is used to control the dataset size: Every experiment starts with an *ingestion* stage, which runs in a single thread and populates the KV-map with 50% of the unique keys in the range using `putIfAbsent` operations. It is followed by the *sustained-rate* stage, which runs the target workload for 30 seconds through one or more symmetric worker threads. Every data point is the median of 3 runs; the deviations among runs were all within 10%.

In each experiment, all algorithms run with the same RAM budget. Oak and Skiplist-OffHeap split the available memory between the off-heap pool and the heap, allocating the former with just enough resources to host the raw data. Skiplist-OnHeap allocates all the available memory to heap.

We configure Oak to use 4K entries per chunk, and invoke rebalance whenever the unsorted linked list exceeds 0.5 of the sorted prefix. The arena size is 100MB.

5.2 Results

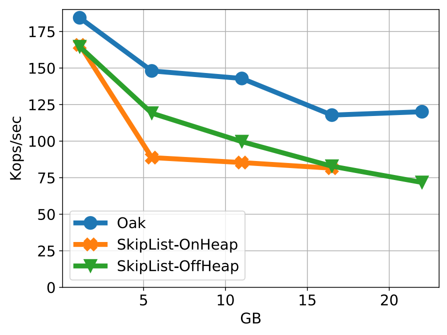
Memory efficiency. We first study how much of the available RAM can be utilized for the raw data, and how fast algorithms can perform within a given memory budget. Figure 3a depicts the throughput of the ingestion stage with 32GB of RAM as the number of ingested unique keys goes up from 1M (approximately 1.1GB of raw data) to 20M (22GB). In Figure 3b, we fix the dataset size to 10M KV-pairs (11GB), and vary the RAM budget from 15GB to 26GB.

First, we observe that the off-heap solutions can accommodate bigger datasets within the same RAM, and conversely, require less RAM to accommodate the same amount of data. For example, with 32GB of RAM, Skiplist-OnHeap caps at 15M KV-pairs while the off-heap solutions can accommodate 20M pairs. This is due to the memory overhead for storing Java objects. Note that all algorithms deteriorate as the number of keys rises. This happens because (1) the search time becomes slower as the data structure grows, and (2) the Java GC overhead increases.

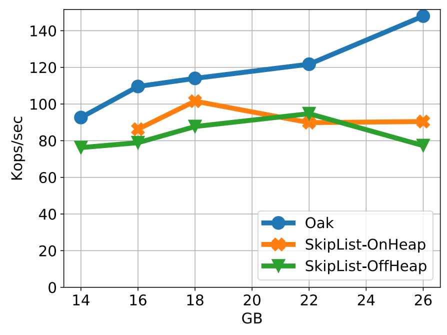
Second, while the performance of on- and off-heap skiplists is similar, Oak is significantly faster, especially for large datasets. There are multiple factors at play here. While off-heap solutions pay an overhead for copying all ingested data to off-heap buffers, they also eliminate much of the GC overhead. These effects cancel each other out. The advantage of Oak then stems from its locality-friendly data organization.

Scalability with parallelism. In the next set of experiments, we fix the available memory to 32GB and the ingested dataset size to 10M KV-pairs. We measure the sustained-rate stage throughput for multiple workloads, while scaling the number of worker threads from 1 to 12. In this setting, the raw data is less than 35% of the available memory, i.e., the GC effect in all algorithms is minor.

Figure 4a depicts the results of a put-only workload. Oak scales 7.7x with 12 threads, while maintaining a 1.5x–1.9x throughput gap over Skiplist-OnHeap. Skiplist-OffHeap is faster than Skiplist-OnHeap but slower than Oak.



(a) Fixed RAM (32GB), varying dataset



(b) Fixed dataset (11GB), varying RAM

Figure 3. Ingestion throughput scaling with RAM, single-threaded execution.

Figure 4b evaluates Oak's incremental update API, where each in-place update modifies 8 bytes of the value. In Skiplist-OnHeap, we invoke the (non-atomic) merge API. In this experiment, Oak scales 8x with 12 threads, and has 1.4x–1.6x higher throughput than Skiplist-OnHeap.

Figure 4c illustrates the get-only workload. Note that after calling get, an application can choose how much of the 1KB value to deserialize. To provide an upper bound on the impact of copying retrieved data, we also exercise the legacy API for Oak, which copies the entire value. We see that copying slows down Oak's performance by 30%, although, perhaps surprisingly, it is still faster than Skiplist-OnHeap. When refraining from copying, Oak scales 10x with 12 threads, and consistently outperforms Skiplist-OnHeap by 1.5x–1.6x. A mix of 95% gets and 5% puts (Figure 4d) shows similar results: Oak outperforms the competition by up to 1.75x.

We proceed to ascending scans. Short scans, like gets, are dominated by the search time for the first KV-pair, and so we do not depict them separately. Long scans, on the other hand, are dominated by the iteration through the retrieved pairs. In such scans, all three solutions perform similarly, as depicted in Figure 4e for scans traversing 10K values.

Finally, Figure 4f depicts the performance of descending scans over 10K values. We see that Oak's efficient chunk-based descending scans are almost as fast as its ascending scans. Oak exhibits a huge advantage over the competition, outperforming both skiplists more than tenfold.

6 Case Study: Druid

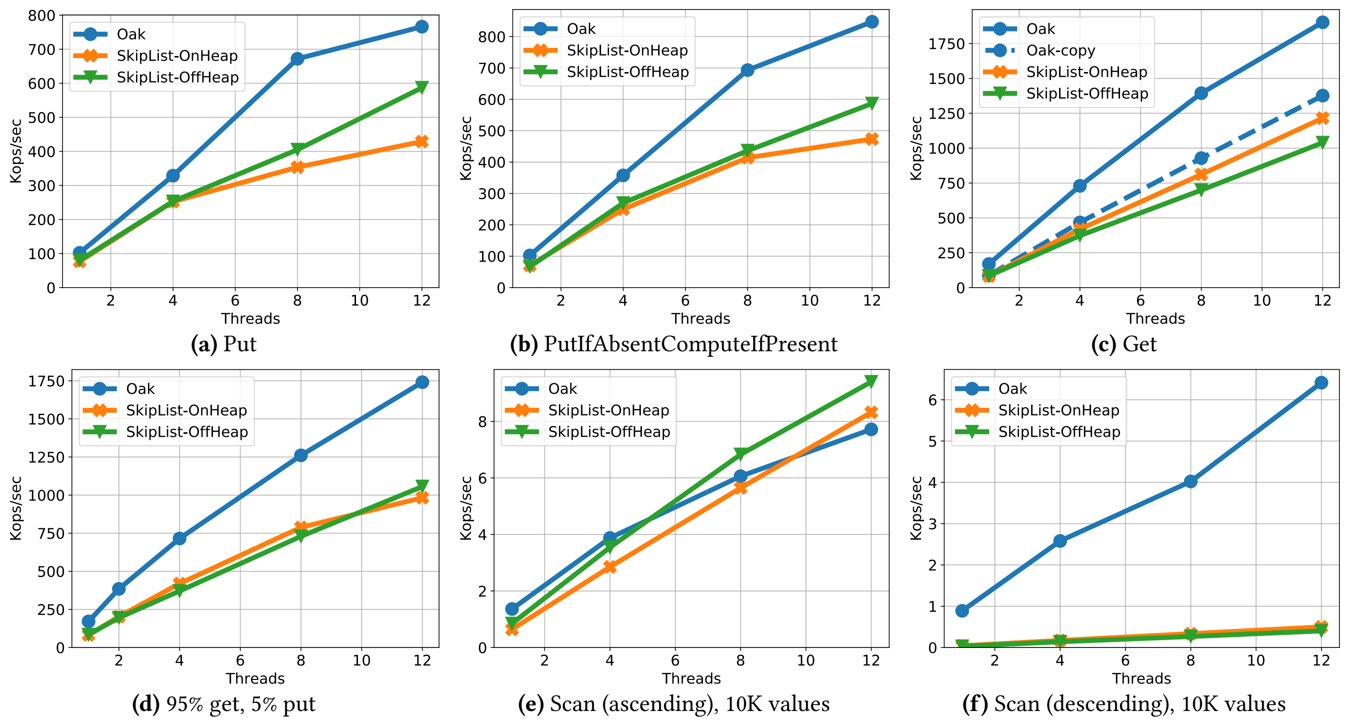
This section presents a case study of Oak's applicability for real-time analytics platforms. We build a prototype integration of Oak into Apache Druid [25] – a popular open-source distributed analytics database in Java. Our goal is to enable faster ingestion and improve RAM utilization, which, in turn, can lead to I/O reduction. The code, which might be further productized, is under community review [3].

More specifically, we target Druid's *Incremental Index* (I^2) component, a data structure that absorbs new data while serving queries in parallel. Data is never removed from an I^2. Once an I^2 fills up, its data gets reorganized and persisted, and the I^2 is disposed; the data's further lifecycle is beyond the scope of this discussion.

I^2 keys and values are multi-dimensional. In *plain* I^2's, the values are raw data, whereas in *rollup* I^2's they are materialized aggregate functions. Complex aggregates (e.g., unique count and quantiles) are embodied through *sketches* [2] – compact data structures for approximate statistical queries; the rest are numeric counters. In order to save space, variable-size (e.g., string) dimensions are mapped to numeric code-words, through auxiliary dynamic dictionaries. A key maps to a flat array of integers; time is always the primary dimension. Keys are typically up to a few hundreds of bytes long. Values are usually up to a few KBs long in rollups, and may vary widely in plain indexes. For every incoming data tuple, I^2 updates its internal KV-map, creating a new pair if the tuple's key is absent, or updating in-situ otherwise.

We re-implement I^2 by switching the internal map from the JDK ConcurrentSkiplistMap to Oak; the auxiliary data structures remain on-heap. We implement an adaption layer that controls the internal data layout and provides Oak with the appropriate lamdba functions for serialization, deserialization, and in-situ compute. The write path exploits Oak's `putIfAbsentComputeIfPresent()` API for atomic update of multiple aggregates within a single lambda. The read path adapts the I^2 tuple abstraction to Oak's ZC API. Namely, the new tuple implementation is a lightweight facade to off-heap memory, operating atop Oak buffers.

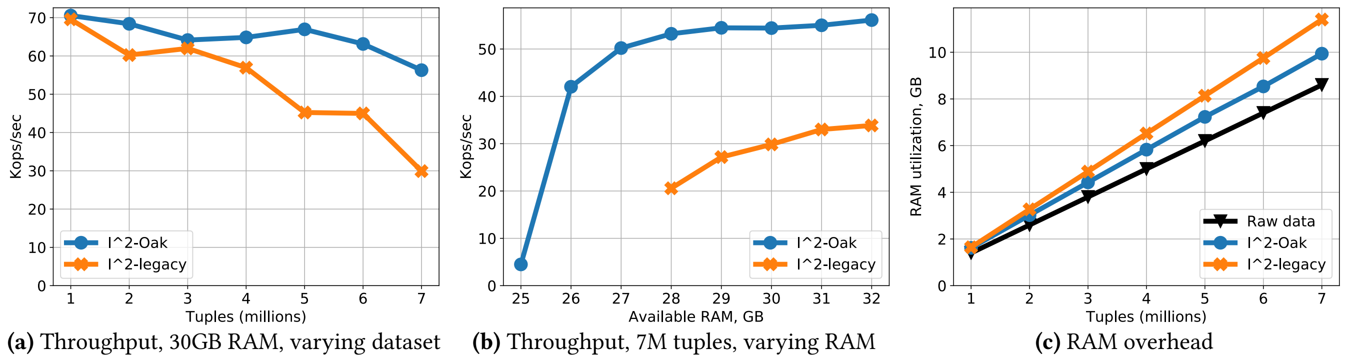
Evaluation. We evaluate the speed and memory utilization of data ingestion with the new solution, I^2-Oak, versus the legacy implementation, I^2-legacy. The first experiment generates 1M to 7M unique tuples of size 1.25K and feeds them into the index, in a single thread. The primary dimension is the current timestamp (in ms), (i.e., the workload is spatially-local). In order to measure ingestion performance in isolation, all the input is generated in advance.



(a) Put **(b)** PutIfAbsentComputeIfPresent **(c)** Get

(d) 95% get, 5% put **(e)** Scan (ascending), 10K values **(f)** Scan (descending), 10K values

Figure 4. Sustained-rate throughput scaling with the the number of threads for uniform workloads, 11GB raw data.



(a) Throughput, 30GB RAM, varying dataset **(b)** Throughput, 7M tuples, varying RAM **(c)** RAM overhead

Figure 5. Single-thread ingestion performance, Druid incremental index: (a,b) throughput, (c) memory efficiency.

Figure 5a depicts the throughput scaling with dataset size, for a fixed RAM budget of 30GB. I^2-Oak is up to 1.8x faster, e.g., its ingestion speed for 7M tuples (8.6GB raw data) is 56K tuples/sec, whereas I^2-legacy's is 30K. Figure 5b studies the 7M-tuple dataset under varying memory budget. We see that I^2-legacy with 32GB RAM is more than 25% slower than I^2-Oak with 26GB RAM, and cannot run with less than 28GB.

Finally, Figure 5c underscores, I^2-Oak's memory efficiency. We see that I^2-Oak induces at most 15.5% metadata overhead (including Oak's index and the on-heap auxiliary data structures), whereas I^2-legacy's space overhead is as high as 32.5%.

7 Conclusion

We presented Oak – a concurrent ordered KV-map for memory-managed programming platforms like Java. Oak features off-heap memory allocation and GC, in-situ atomic data processing, zero copy API, and internal organization for high speed data access. It implements an intricate efficient concurrent algorithm. Multiple benchmarks demonstrate Oak's advantages in scalability and resource efficiency versus the state-of-the-art. Oak's code is production quality and open sourced. Its prototype integration with Apache Druid (incubating) demonstrates decisive performance gains.

References

[1] 2014. Apache HBase, a distributed, scalable, big data store. http://hbase.apache.org/. (April 2014).
[2] 2018. Druid DataSketches extension. https://druid.apache.org/docs/latest/development/extensions-core/datasketches-extension.html.
[3] 2018. Druid Integration with Oak. Anonymized for blind review.
[4] 2018. Elasticsearch: Open Source Search and Analytics. https://elastic.co/.
[5] 2018. HBase Offheap write path. https://hbase.apache.org/book.html#regionserver.offheap.writepath.
[6] 2018. In-Memory Analytics Market worth 3.85 Billion USD by 2022 (retrieved October 2018). https://www.marketsandmarkets.com/PressReleases/in-memory-analytics.asp.
[7] 2018. Memcached, an open source, high-performance, distributed memory object caching system. https://memcached.org/.
[8] 2018. Off-heap memtables in Cassandra 2.1. https://www.datastax.com/dev/blog/off-heap-memtables-in-cassandra-2-1.
[9] 2018. Offheap read-path in production the Alibaba story. https://blog.cloudera.com/blog/2017/03/.
[10] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. 2012. CBTree: A Practical Concurrent Self-adjusting Search Tree. In *Proceedings of the 26th International Conference on Distributed Computing (DISC'12)*. Springer-Verlag, Berlin, Heidelberg, 1–15. https://doi.org/10.1007/978-3-642-33651-5_1
[11] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/2611462.2611471
[12] Avoiding Full GC 2011. https://www.slideshare.net/cloudera/hbase-hug-presentation.
[13] 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 *PPoPP'17*. 13. https://doi.org/10.1145/3018743.3018761
[14] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB* 11, 12 (2018), 1863–1875. https://doi.org/10.14778/3229863.3229873
[15] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. 2016. CBPQ: High Performance Lock-Free Priority Queue. In *Euro-Par*.
[16] Anastasia Braginsky and Erez Petrank. 2011. Locality-conscious Lock-free Linked Lists. In *ICDCN'11*. 107–118.
[17] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *SPAA '12*. 58–67. https://doi.org/10.1145/2312005.2312016
[18] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488
[19] Trevor Brown and Hillel Avni. 2012. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*. Springer, 31–45.
[20] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 329–342. https://doi.org/10.1145/2555243.2555267
[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (June 2008), 4:1–4:26.
[22] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A Contention-friendly Binary Search Tree. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 229–240. https://doi.org/10.1007/978-3-642-40047-6_25
[23] 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*. 196–205. https://doi.org/10.1109/ICDCS.2013.42
[24] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 343–356. https://doi.org/10.1145/2555243.2555269
[25] Druid [n. d.]. (retrieved August 2018). http://druid.io/.
[26] Druid off-heap [n. d.]. (retrieved August 2018). http://druid.io/docs/latest/operations/performance-faq.html.
[27] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736
[28] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.
[29] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501
[30] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer.
[31] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *SIROCCO'07*. 15.
[32] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers.
[33] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972
[34] Java Concurrent Navigable Map 2018. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html.
[35] Java Concurrent Skip List Map 1993. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.
[36] Java Maps, Sets, Lists, Queues and other collections backed by off-heap or on-disk storage 2019. http://www.mapdb.org/.
[37] Java Stream Package 2018. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.htmll.
[38] Anoop Sam John. 2017. Track memstore data size and heap overhead separately. https://issues.apache.org/jira/browse/HBASE-16747.
[39] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.
[40] Yu Li, Yu Sun, Anoop Sam John, and Ramkrishna S Vasudevan. 2017. Offheap Read-Path in Production - The Alibaba story. https://blogs.apache.org/hbase/entry/offheap-read-path-in-production.
[41] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256
[42] Oak Repository 2018. Oak Open-Source Repository (anonymized for blind review). https://github.com/xxx/Oak.
[43] Yehoshua Sagiv. 1985. Concurrent Operations on B-trees with Overtaking. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '85)*. ACM, New York, NY, USA, 28–37. https://doi.org/10.1145/325405.325409

[44] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *PLDI '16*. 682–696. https://doi.org/10.1145/2908080.2908112