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

Hagar Meir*
IBM Research, Israel

Dmitry Basin
Yahoo Research, Israel

Edward Bortnikov
Yahoo Research, Israel

Anastasia Braginsky
Yahoo Research, Israel

Yonatan Gottesman
Yahoo Research, Israel

Idit Keidar
Technion and Yahoo Research, Israel

Eran Meir
Yahoo Research, Israel

Gali Sheffi*
Technion, Israel

Yoav Zuriel*
Technion, Israel

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.

CCS Concepts • **Theory of computation → Data structures design and analysis**; **Concurrent algorithms**;

Keywords memory management, concurrent data structures, key-value maps

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 full paper [41].

*Work done while at Yahoo Research.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374526

We have released a Java implementation of Oak as an off-the-shelf open-source package under the Apache 2.0 License [43]. Its evaluation in §5 shows significant improvements over ConcurrentSkipListMap, e.g., 2x speed 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 90%.

We next 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 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 of contributions. All in all, Oak is the first concurrent KV-map explicitly designed to address big-data demands; the following aspects of Oak are novel:

- a data structure offering a managed programming experience with off-heap data allocation;
- a chunk-based organization that supports in-place atomic updates of variable-size keys and values;
- a concurrent algorithm supporting both conditional and unconditional updates;
- an efficient descending scan mechanism that does not require a doubly-linked list; and
- a zero-copy API.

On the practical side, this work contributes

- an open-source [43] implementation;
- an extensive synthetic evaluation showing major gains;
- a prototype of Druid's incremental index that uses Oak, achieving major reductions in memory overhead and data ingestion times.

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, 42, 45]. However, most previous 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, 42]. 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 has been used in 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 [44]. We are not aware of safety guarantees of this implementation with respect to 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.

Oak provides `OakRBuffer` and `OakWBuffer` abstractions for accessing internal readable and writable buffers, resp. These types are lightweight on-heap facades to off-heap storage, which provide the application with managed object semantics. These objects may be accessed safely for arbitrarily long. 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 drawback of the Set APIs is that they create a new ephemeral Java object for each scanned entry. To mitigate this cost in long scans, we additionally introduce a specialized *stream scan* API which re-uses the same ephemeral object to store multiple scanned entries. For instance, the set `s` returned by `keyStreamSet()` contains a single `OakRBuffer` object and `s.getNext()` changes that `OakRBuffer`'s content. Note that this semantics is non-standard in Java iterators; in particular, if the reusable object is stored in another data structure, the programmer must be aware of the fact that its contents may change.

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.

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

ZeroCopyConcurrentNavigableMap	**(Legacy) ConcurrentNavigableMap**
Queries – get and scans	
OakRBuffer get(K)	V get(K)
Set⟨OakRBuffer⟩ keySet() / keyStreamSet()	Set⟨K⟩ keySet()
Set⟨OakRBuffer⟩ valueSet() / valueStreamSet()	Set⟨V⟩ valueSet()
Set⟨OakRBuffer, OakRBuffer⟩ entrySet() / entryStreamSet()	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.

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 off-heap memory manager. Oak allows user code safe access to data in off-heap Oak buffers without worrying about concurrent access or dynamic reallocation, 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 holds a linked list of *entries*, sorted in ascending key order. The entries refer to off-heap keys and values. Oak makes sure that each key appears in at most one entry.

To allow fast access to the linked list we employ an additional *index* that maps minKeys to their respective chunks, as in [13, 15, 31, 32, 45]; see Figure 1. The index can be an arbitrary map data structure – our implementation uses a skiplist. Index updates are lazy, and so the index may be outdated, 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.

As noted above, programmers access keys and values via the `OakRBuffer` and `OakWBuffer` views. These are ephemeral on-heap Java objects created and returned by gets and scans.

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*. The pool 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 exposes methods for allocating and initializing keys and values, `allocateKey(key)` and `allocateValue(val)`, both returning references consisting of an arena id, an offset, and a length.

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

3.3 Value access and concurrency control

Oak allows atomic access to an off-heap value `v` via the methods `v.put(val)`, `v.compute(func)`, `v.remove()`, and `v.isDeleted()`. To this end, it allocates headers to all values at the beginning of their buffers. Oak's default concurrency control mechanism uses a read-write lock (in the header) to ensure that these methods execute atomically; it can be overridden, e.g., by an optimistic approach. The header also includes a bit indicating whether the value is deleted. If the value is deleted, the method calls fail (returning `false`).

There are different ways to implement memory reclamation with this approach. Oak's default mechanism (tested in this paper) simply refrains from reclaiming headers while allowing reuse of the space taken up by the deleted value. We have implemented a more elaborate solution that uses generations (epochs) in order to reclaim headers as well; this mechanism is beyond the scope of the current paper.

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

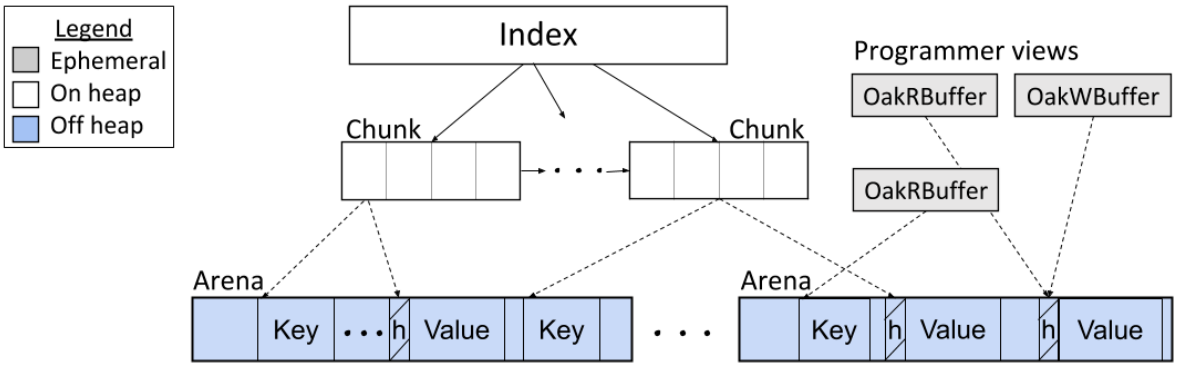


Figure 1. Oak layout: Meta-data (index and chunks) is on heap, whereas data (keys and values) is allocated in off-heap arenas. Each value is preceded by a header facilitating concurrency control and reclamation. Programmers access off-heap data via the lightweight `OakRBuffer` and `OakWBuffer` views.

Oak's queries, namely `get` and ascending and descending scans. Oak's support for both conditional and unconditional updates raises some subtle interactions that need to be handled with care. We divide our discussion of such 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 full paper [41].

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 logarithmic.

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 C holds 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.

The `allocateEntry(keyRef)` method allocates a new entry (in the chunk array) that refers to the given key; this entry does not hold a value (its value reference is $\perp$) and is not yet part of the chunk's linked list. Hardware operations like F&A ensure that the same space is not allocated twice. In case the chunk is full, `allocateEntry` triggers a rebalance and fails (returning $\perp$), in which case Oak retries the update.

`entriesLLputIfAbsent(entry)` adds an (already allocated) 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. While a chunk is being rebalanced, calls to `entriesLLputIfAbsent` fail and return $\perp$.

Updates that add or remove keys from the chunk inform the rebalancer of the operation they are about to perform by calling the `publish` method, which uses a dedicated array with an entry per thread for reporting its ongoing operation. 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`, clearing the thread's entry in the dedicated array.

Note that whereas chunk update methods that encounter a rebalance fail (return $\perp$), `lookUp` and `unpublish`, which do not modify the entries list, proceed concurrently with rebalance without aborting.

4.2 Queries – get and scans

The `get` operation is given in Algorithm 1. It returns a read-only view (`oakRBuffer`) of the value mapped to the given key. Since it is a view and not an actual copy, 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 value will be marked as deleted; reads from the `oakRBuffer` check this flag and throw an exception in case the value is deleted.

Algorithm 1 Get

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

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

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 value reference is not $\perp$ and the value 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.

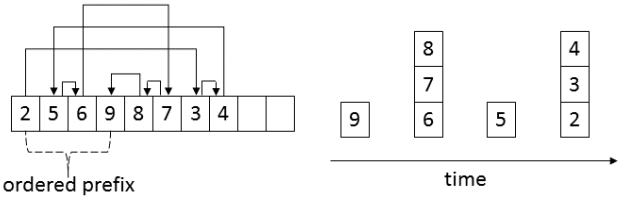


Figure 2. Example entries linked list (left) and stacks built during its traversal by a descending scan (right).

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 $8 \rightarrow 7 \rightarrow 6$. 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 $4 \rightarrow 3 \rightarrow 2$, 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.

The standard implementation of descending iterators in a skiplist calls `lookUp` anew after each key. This results in an asymptotic complexity of $O(S \log N)$ for a descending scan covering S keys in a map of N keys. With chunks of size B, and assuming the insertion order is random, Oak reduces the descending scan complexity to $O(S/B \cdot \log N + S)$.

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

1. A scan returns all keys in the scanned range 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 re-inserted until it ends.
3. A scan 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 value `v` is found (case 1: lines 19 – 26) then we say that the key is *present*. In this case, `putIfAbsent` returns `false` (line 20), `put` calls `v.put` (line 21) to associate the new value with the key, and `putIfAbsentComputeIfPresent` calls `v.compute` (line 23). These operations return `false`

if the value is deleted (due to a concurrent remove), in which case we retry (line 25).

Algorithm 2 Oak's insertion operations

```
7:  procedure PUT(key, val)
8:      doPut(key, val, ⊥, PUT)
9:      return

10: procedure PUTIFABSENT(key, val)
11:     return doPut(key, val, ⊥, PUTIF)

12: procedure PUTIFABSENTCOMPUTEIFPRESENT(key, val,
    func)
13:     doPut(key, val, func, COMPUTE)
14:     return

15: procedure DOPUT(key, val, func, op)
16:     C, ei, v, newV ← ⊥; result, succ ← true
17:     C ← locateChunk(key); ei ← C.lookUp(key)
18:     if ei ≠ ⊥ then v ← C.entries[ei].valRef
19:     if v ≠ ⊥ ∧ ¬v.isDeleted() then
    ▷ Case 1: key is present
20:         if op = PUTIF then return false
21:         if op = PUT then succ ← v.put(val)
22:         if op = COMPUTE then
23:             succ ← v.compute(func)
24:         if ¬succ then              ▷ On failure, retry doPut
25:             return doPut(key, val, func, op)
26:         return true
    ▷ Case 2: key is absent
27:     if ei = ⊥ then                 ▷ No entry found
28:         ei ← C.allocateEntry(allocateKey(key))
29:         ei ←C.entriesLLputIfAbsent(ei)
30:     newV ←allocateValue(val)
31:     if ei = ⊥ ∨ newV = ⊥ then ▷ allocate or insert failed
32:         return doPut(key, val, func, op)
33:     if ¬C.publish(ei, newV, func, op) then
34:         return doPut(key, val, func, op)
35:     result ← CAS(C.entries[ei].valRef, ⊥, newV)
36:     C.unpublish(ei, newV, func, op)
37:     if ¬result then          ▷ On CAS failure, retry doPut
38:         return doPut(key, val, func, op)
39:     return true
```

In the second case, the key is absent. If we discover a removed entry that points to the same key but with `valRef` = ⊥ or a deleted value, then we reuse this entry. Otherwise, we call `allocateEntry` to allocate a new entry referring to the given key (line 28), and then try to link this new entry into the entries linked list (line 29). Either way, we allocate and write the value (line 30). These functions might fail and cause a retry (line 31).

If `entriesLLputIfAbsent(ei)` receives ⊥ as a parameter (because the allocation in line 28 fails) then it just returns ⊥ as well. If it encounters an already linked entry with the same key as `ei`, then it returns it. In this case, the `ei` passed to it, (which was allocated in line 28), remains unlinked in the entries array and other operations never reach it; the rebalancer eventually removes it from the array. Next, we allocate and write the value (off-heap, line 30).

We complete the insertion by using CAS to make the entry point to the new value (line 35). Before doing so, we publish the operation (as explained in §4.1), which can also lead to a retry (line 34). After the CAS, we unpublish the operation (line 36). If CAS fails, we retry (line 38).

To see why we retry, observe that the CAS may fail because of a concurrent non-insertion operation that sets the value reference to ⊥ (as described in §4.4 below) or because of a concurrent insertion operation that sets the value reference 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.

4.4 Non-insertion operations

The non-inserting updates, `computeIfPresent` and `remove`, use the `doIfPresent` function in Algorithm 3. It first locates the value associated with the given key in Oak, and if there is none, returns `false` (line 44).

In `computeIfPresent`, if the value exists and is not deleted (case 1), we execute the function using `v.compute`, and if it is successful, return `true`(line 46). 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 the entry's value reference to ⊥ (line 52). Since this affects the 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 51 and 54). The operation returns `false` whenever it does not find the entry, or finds the entry but with ⊥ as its value reference (line 44), or CAS to ⊥ is successful (line 55).

In `remove`, if a non-deleted value exists (case 1), it also updates the value, in this case, marking it as deleted by calling `v.remove` (line 48), 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 value as deleted, namely, marking the appropriate entry's value reference as ⊥. This serves two purposes: first, rebalance does not check whether a value is deleted, so removing the reference facilitates garbage collection; second,

Algorithm 3 Oak's non-insertion update operations

```
40: procedure doIfPresent(key, func, op)
41:     C, ei, v ← ⊥; result ← true
42:     C ← locateChunk(key); ei ← C.lookUp(key)
43:     if ei ≠ ⊥ then v ← C.entries[ei].valRef
44:     if v = ⊥ then return false                ▷ Key not found
45:     if ¬v.isDeleted() then
    ▷ Case 1: value exists and is not deleted
46:         if op = comp ∧ v.compute(func) then
47:             return true
48:         if op = rm ∧ v.remove() then
49:             return finalizeRemove(key, v)
    ▷ Case 2: value is deleted – ensure entry is removed
50:     if ¬C.publish(ei, ⊥, func, op) then
51:         return doIfPresent(key, func, op)
52:     result ← CAS(C.entries[ei].valRef, v, ⊥)
53:     C.unpublish(ei, ⊥, func, op)
54:     if ¬result then return doIfPresent(key, func, op)
55:     return false

56: procedure computeIfPresent(key, func)
57:     return doIfPresent(key, func, comp)

58: procedure remove(key)
59:     doIfPresent(key, ⊥, rm)
60:     return

61: procedure finalizeRemove(key, prev)
62:     C, ei, v ← ⊥
63:     C ← locateChunk(key); ei ← C.lookUp(key)
64:     if ei ≠ ⊥ then v ← C.entries[ei].valRef
65:     if v ≠ prev then                ▷ Key removed or replaced
66:         return true
67:     if ¬C.publish(ei, ⊥, ⊥, rm) then
68:         return finalizeRemove(key, prev)
69:     CAS(C.entries[ei].valRef, v, ⊥)
70:     C.unpublish(ei, ⊥, ⊥, rm)
71:     return true
```

updating the entry expedites other operations, which do not need to access the value in order to see that it is deleted.

Thus, a successful `remove` calls `finalizeRemove`, which tries to CAS the value reference to ⊥. We have to take care, however, in case the value had already changed, not to change it to ⊥. To this end, `finalizeRemove` takes a parameter `prev` – the value that `remove` marked as deleted. If the entry no longer points to it, we do nothing (line 65). Note that `prev` holds a reference to the value header, which, using our simple value access mechanism (§3.3) is not reused, avoiding potential ABA problems. When using a more sophisticated approach to reclamation, we check a monotonically increasing ABA counter at this point.

Since `remove` is linearized at the point where it marks the value 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 value (case 2), it cannot simply return, since by the time remove notices that the value is deleted, the entry might point to another one. Therefore, similarly to `computeIfPresent`, it makes sure that the key is removed by performing a successful CAS of the value reference to ⊥ (line 52). In this case (case 2) it does not perform `finalizeRemove`, but rather retries if the CAS fails (line 54). Note the difference between the two cases: in case 1, we set the value to deleted, and so changing the entry's value reference to ⊥ is merely an optimization, and should only occur if the entry still points to the deleted one. In the second case, on the other hand, `remove` does not delete any value, and so it *must* make sure that the entry's value reference is indeed ⊥ before returning.

4.5 Linearization points

In the full paper [41], 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 (line 35). Otherwise, the l.p. is when it finds a non-deleted value (line 19).

put – if it inserts a new key, the l.p. is the successful CAS (line 35). Otherwise, the l.p. is upon a successful nested call to `v.put` (line 21).

putIfAbsentComputeIfPresent – if it inserts a new key, the l.p. is the successful CAS (line 35). Otherwise, the l.p. is upon a successful nested call to `v.compute` (line 23).

computeIfPresent – if it returns `true`, the l.p. is upon a successful nested call to `v.compute` (line 46). Otherwise, the l.p. is when the entry is not found, or it is found but with ⊥ as its value reference (line 44), or, in case it is found but has been deleted, a successful CAS to ⊥ (line 52).

remove – if it is successful, the l.p. is when a successful nested call to `v.remove` sets the value to deleted (line 48). Otherwise, the l.p. is when the entry is not found, or value reference is ⊥ (line 44), or a deleted handle is found and a successful CAS to ⊥ occurs (line 52).

get – if it returns a value, then the l.p. is the read of a non-deleted value (line 4). If it returns NULL because there is no relevant entry with a non-⊥ value reference, then the l.p. is when `lookUp` (line 3) returns ⊥, or when get reads a ⊥ value reference (line 4). Otherwise, `get` reads a deleted value (line 5, second condition in disjunction). However, the l.p. cannot be the read of the deleted flag in the value header, since by that time, a new value may have been inserted to the entry. Instead, the l.p. is the later between (1) the read of the value reference by the same `get` (line 4) and (2) immediately after the deleted bit is set by some `remove` (note that since headers are not reused, exactly one `remove` sets the bit 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]. We run these experiments on an AWS instance m5d.16xlarge, utilizing 32 cores (with hyper-threading disabled) on two NUMA nodes.

Compared solutions. We mostly focus on Oak's ZC API. To quantify the benefit of zero-copying, we also run gets with the legacy API, to which we refer as Oak-Copy. For scans, we experiment with both the Set and Stream APIs.

Since our goal is to offer Oak as an alternative to Java's standard skiplist, our baseline is the JDK8 ConcurrentSkipListMap [35], which we call 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 and Oak-Copy offer 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], but it failed to scale to big datasets, performing at least ten-fold slower than Oak; we omit these results.

Methodology. The exercised key and value sizes are 100B and 1KB, 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 half of the sorted prefix. The arena size is 100MB.

5.2 Results

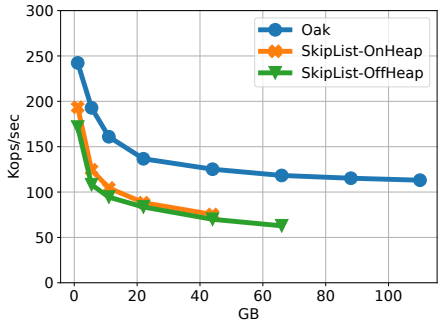
Memory efficiency. We first study how much of the available RAM can be utilized for storing raw data, and how the memory budget affects performance. Figure 3a depicts the throughput of the ingestion stage with 128GB of RAM as the number of ingested unique keys goes up from 1M (approximately 1.1GB of raw data) to 100M (110GB). In Figure 3b, we fix the dataset size to 10M KV-pairs (11GB), and vary the RAM budget from 14GB to 26GB.

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 128GB of RAM, Skiplist-OnHeap caps at 40M KV-pairs (44GB) whereas Skiplist-OffHeap and Oak can accommodate 60M pairs and 100M pairs, respectively. This is due to the overhead for storing Java objects, as well as the headroom required by the Java GC.

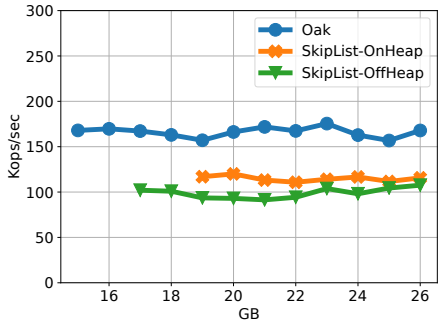
Oak utilizes the RAM most effectively because it stores much fewer metadata objects (index nodes and chunks) than the number of KV-pairs. In contrast, Skiplist-OnHeap utilizes less than 40% of the available RAM for raw data in the same setting. Neither algorithm benefits from extra memory beyond its minimum serving capacity.

The throughput of all algorithms deteriorates as the dataset scales, which is inherent because the search becomes asymptotically slower as the data structure grows. While the performance of on- and off-heap skiplists is similar, Oak is much faster, especially for large datasets. There is a tradeoff between the on- and off-heap approaches: Off-heap solutions pay an overhead for copying all ingested data to off-heap buffers, but they also eliminate much of the GC. We see that in the skiplist, these effects, by and large, cancel each other out. The advantage of Oak then stems from its ultra-low GC footprint as well as its locality-friendly data organization.

Scalability with parallelism. In the next set of experiments, shown in Figure 4, 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 32. In this setting, the raw data is less than 35% of the available memory, so the GC effect in all algorithms is minor.



(a) Fixed RAM (128GB), varying dataset



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

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

Figure 4a depicts the results of a put-only workload. Oak is markedly faster than the alternatives already with one thread and continues to outperform Skiplist-OnHeap by at least 2x with any number of threads. Moreover, whereas Skiplist-OnHeap's throughput flattens out with around 16 threads, Oak continues to improve all the way up to 32. Skiplist-OffHeap scale as well as Oak does, suggesting that GC is the limiting factor for Skiplist-OnHeap's scalability. Moreover, Oak's improvement over Skiplist-OffHeap remains steady with the number of threads, suggesting that Oak's advantage stems from factors that affect also the sequential execution speed, such as better locality, fewer Java objects (and consequently, less GC overhead), and fewer redirections.

Figure 4b evaluates incremental updates invoked using Oak's `computeIfPresent` API and (non-atomic) merge in the skiplists. Each in-place update modifies 8 bytes of the value. This workload does not increase the number of objects, and hence, the GC effect is minor. As expected, all algorithms exhibit near-linear scaling with similar performance.

Figure 4c illustrates the get-only workload. As expected in a read-only workload, all solutions scale linearly without saturating at 32 threads. Here, too, Oak is much faster than the alternatives. In particular, Oak scales 25x with 32 threads compared to its single-threaded execution, and outperforms Skiplist-OnHeap by 1.7x at peak performance. We also exercise the legacy API Oak-Copy, finding that copying induces a significant penalty and inhibits scalability.

A mix of 95% gets and 5% puts (Figure 4d) shows similar results: Oak scales 24x with 32 threads and out-performs Skiplist-OnHeap by 1.7x to 2x in all thread counts, while Skiplist-OffHeap is slower than both.

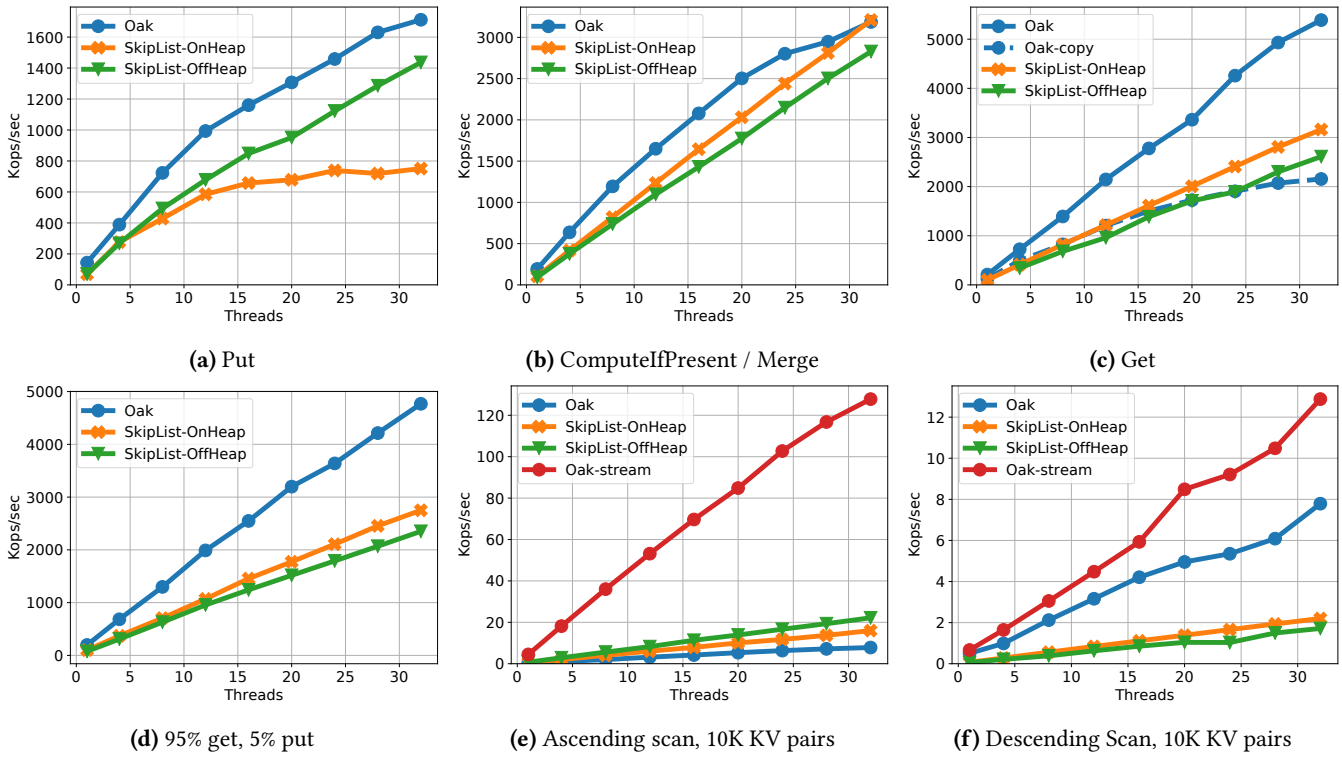
We proceed to ascending scans, shown in Figure 4e. We focus on long scans traversing 10K key-value pairs. In such scans, performance is dominated by the iteration through the entries rather than the search time for the first key, which differentiates them from gets. In this scenario, Oak's Set API is 2x slower than the alternatives. Its performance suffers from the creation of ephemeral OakRBuffer objects for all traversed keys and values, while its competitors retrieve references to existing objects. Oak's Stream API, which re-uses the same object for all the entries returned throughout the scan, eliminates this overhead. The locality of access offered by the chunk-based organization is particularly beneficial in this case, allowing Oak's Stream API to outperform Skiplist-OnHeap by nearly 8x with 32 threads.

Finally, Figure 4f depicts the performance of descending scans over 10K KV pairs. Recall that the skiplists implement such scans by issuing a new lookup for each traversed key, and thus pay the (logarithmic) search cost 10K times per scan. Oak, in contrast, issues a lookup only when a chunk (holding 2K–4K KV pairs) is exhausted. Note that Oak's descending scans are still slower than its ascending ones because it stores the traversed entries in a stack. But Oak's slower Set API achieves almost the same throughput as in ascending scans, suggesting that the iteration time is dominated by the creation of ephemeral objects, and the overhead of using a stack is substantially smaller than this cost. Even with this (slow) API, Oak outperforms Skiplist-OnHeap by more than 3.5x. With the Stream API, Oak's throughput doubles. Here, the overhead of using a stack is manifested, and so the descending scan throughput is lower than the ascending one.

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].

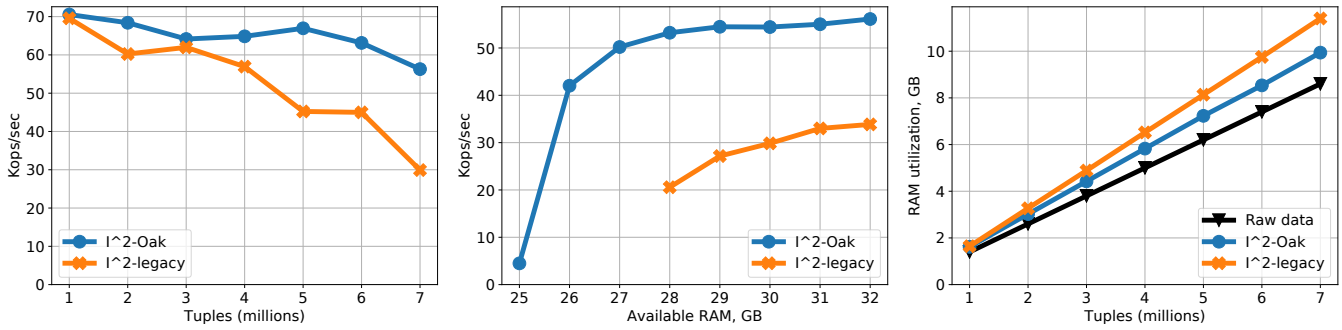
More specifically, we target Druid's *Incremental Index* (I^2) component, a data structure that absorbs new data while serving queries in parallel. Data is never removed from an I^2. Once an I^2 fills up, its data gets reorganized and persisted, and the I^2 is disposed; the data's further lifecycle is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



(a) Put **(b)** ComputeIfPresent / Merge **(c)** Get

(d) 95% get, 5% put **(e)** Ascending scan, 10K KV pairs **(f)** Descending Scan, 10K KV pairs

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 with Oak versus legacy Skiplist-OnHeap.

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 codewords, 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 replacing the JDK ConcurrentSkiplistMap in the internal map with Oak; the auxiliary data structures remain on-heap. We implement an adaptation 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. Our hardware testbed features an industry-standard 12-core Xeon E5-4650 CPU with 192 GB RAM. 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), so the workload is spatially-local. In order to measure ingestion performance in isolation, all the input is generated in advance.

Figure 5a depicts the throughput scaling with dataset size for a fixed RAM budget of 30GB. When the ingested set is small (1M tuples), the two solutions have similar performance. But as the dataset grows, the GC overhead burdens the legacy solution while I^2-Oak continues to thrive. For example, it ingests 7M tuples (8.6GB raw data) twice as fast as I^2-legacy.

Figure 5b studies the 7M-tuple dataset under a varying memory budget. We see that I^2-legacy cannot run with less than 29GB. Finally, Figure 5c underscores I^2-Oak's memory efficiency. We see that I^2-Oak induces a negligible metadata overhead of less than 5% (including Oak's index and the on-heap auxiliary data structures), whereas I^2-legacy's space overhead is as high as 35%.

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. https://github.com/apache/incubator-druid/issues/5698.

[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] Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Idit Keidar, and Gali Sheffi. 2018. Oak – A Key-Value Map for Big Data Analytics. (May 2018). https://hal.archives-ouvertes.fr/hal-01789846 working paper or preprint.

[42] 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

[43] Oak Repository 2018. Oak Open-Source Repository. https://github.com/yahoo/Oak.

[44] 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

[45] 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

A Artifact Evaluation Appendix

A.1 Abstract

Our artifact refers to the GitHub repository, which contains all source files, scripts and benchmarks to reproduce the results presented in the paper. We created a special branch to keep the presented state of the library without further enhancements that may come.

All compared solutions together with variety of workloads presented in the paper are integrated in the provided version of the synchrobench tool. The scripts running the synchrobench and creating the plots are also part of the repository.

The hardware required is any industry standard multi-core machine with enough cores and RAM. We run the experiments on an AWS instance m5d.16xlarge, utilizing 32 cores (with hyper-threading disabled) on two NUMA nodes.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Java off-heap memory, skiplist search, binary search, concurrency control
- **Program:** Java code
- **Compilation:** Maven compilation
- **Binary:** Binary not included
- **Data set:** The keys and values for the workloads are run-time generated by internally provided synchrobench tool
- **Run-time environment:** Our code has been developed and tested on Linux and iOS environments. However, it is not restricted. The main software dependency is Java 8, Git and Maven.
- **Hardware:** For experiments presented in the paper we used an AWS instance m5d.16xlarge, utilizing 32 cores (with hyper-threading disabled) on two NUMA nodes and up to 256 GB RAM. Similar hardware should give comparable results. Main memory usage is about 128GB.
- **Metrics:** Throughput
- **Output:** Throughputs and other benchmark parameters are printed out as a table
- **Experiments workflow:** Git clone project; update and run benchmark scripts; observe the results
- **How much disk space required (approximately)?:** None. All in-memory.
- **How much time is needed to prepare workflow (approximately)?:** Time it takes to git clone the project
- **How much time is needed to complete experiments (approximately)?:** Depends on number of threads and iterations, up to 24 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License 2.0

A.3 Description

A.3.1 How delivered

Our benchmarks, source code (including all compared solutions), and scripts are available on Github: https://github.com/yahoo/oak

A.3.2 Hardware dependencies

Our experiment workflows use 1 to 32 threads to compare with competitors and hence we suggest a machine with at least 32 CPU cores with hyper-threading disabled and with at least 128GB main memory.

A.3.3 Software dependencies

The Oak open-source library is expected to run correctly under variety of Linux *x*86_64 distributions. Building the JAR files requires JDK 8. To use the automated scripts, we require git and python3 to be installed. Python3 is needed just to build the plots, which might be skipped.

A.4 Installation

First, clone the repository then compile using maven:

```
$ git clone https://github.com/yahoo/oak
$ cd oak
$ git checkout PPOPP20_AE
$ mvn clean package -DskipTests=true
```

A.5 Experiment workflow

The script `run.sh` runs the experiment. It can be found in directory: `oak/benchmarks/synchrobench/`. The script is mostly prepared to run the experiments presented in Fig 4 . However `run.sh` might need to be updated if a different evaluation is done. For example for evaluation of Fig 3. In Sec A.7 we will go through the script structure.

A.6 Evaluation and expected result

To run the experiments do:

```
$ cd oak/benchmarks/synchrobench/
$ ./run.sh
```

The results are built in `oak/benchmarks/synchrobench/output` directory. The log files per each experiments are less interesting. Total results are summarized in `summary.csv` file, that can be later directly used to build the plots. Hereby find an example of `summary.csv` file layout:

```
Scenario Bench Heap size Direct Mem #Threads Final Size Throughput
4c-zc-get-only OakMap   12g     20g       1    1.00E+07 0.186242003
4c-zc-get-only OakMap   12g     20g       4    1.00E+07 0.7185844075
4c-zc-get-only OakMap   12g     20g       8    1.00E+07 1.510101392
4c-zc-get-only OakMap   12g     20g       12   1.00E+07 2.000772053

4c-get-only JavaSkipListMap 20g 20g 1 1.00E+07 0.09323596854
4c-get-only JavaSkipListMap 20g 20g 4 1.00E+07 0.4210559611
4c-get-only JavaSkipListMap 20g 20g 8 1.00E+07 0.8588087848
4c-get-only JavaSkipListMap 20g 20g 12 1.00E+07 1.394160355
```

Each block describes an experiment executing on different number of threads. The name of experiment (that may vary) is on the left, the first two characters are the related paper figure. The second column `Bench` represents the competitors that are: `'OakMap'`, `'JavaSkipListMap'` and `'OffHeapList'` as explained in the paper. The fourth `Direct Mem` column shows amount of off-heap allocated memory, for `'JavaSkipListMap'` this value is irrelevant. The last `Throughput` column shows millions of operations in seconds. This column is used to present the results of Fig 4.

A.7 Experiment customization

Hereby we will go through the `run.sh` script structure, explaining how it can be altered. The default parameters in the script may vary from what define in the paper due to unintentional mistakes and multiple usages. This is how script's header looks like:

```
thread="01 04 08 12 16 20 24 28 32"
size="10000000"
keysize="100"
valuesize="1000"
writes="0"
warmup="0"
iterations="3"
duration="30000"
```

Only fields that can be altered are explained. Change `thread` to limit number of threads to 12 or to use different number of threads. Change `size` if different warm-up size of the map is requested (currently 10M pairs). Change `keysize` or `valuesize` if you want different size of the input in bytes. Change `iterations` to calculate the average results over less or more iterations. Change `duration` to make each experiment to run less or more milliseconds (currently 30 seconds). We continue to memory sizes:

```
declare -A heap_limit=(["OakMap"]="10g"
                       ["OffHeapList"]="10g"
                       ["JavaSkipListMap"]="32g"
                       )

declare -A direct_limit=(["OakMap"]="22g"
                         ["OffHeapList"]="22g"
                         ["JavaSkipListMap"]="0g"
                         )
```

Change `heap_limit` (or `direct_limit`) per competitor to change the on-heap (or off-heap) heap size requirements. For fairness, `heap_limit` plus `direct_limit` needs to be the same for each competitor. JavaSkipListMap disregards `direct_limit` even if set. We continue to tested scenarios:

```
declare -A scenarios=(
  ["4a-put"]="-a 0 -u 100"
  ["4b-putIfAbsentComputeIfPresent"]="--buffer -u 0 -s 100 -c"
  ["4c-get-zc"]="--buffer"
  ["4c-get-copy"]=""
  ["4d-95Get5Put"]="--buffer -a 0 -u 5"
  ["4e-entrySet-ascend"]="--buffer -c"
  ["4e-entryStreamSet-ascend"]="--buffer -c --stream-iteration"
  ["4f-entrySet-descend"]="--buffer -c -a 100"
)
```

The first two characters of the labels are the related paper figure. The labels of the scenarios are self-explaining, e.g. `4a-put` is put only scenario to be run for all competitors.Scenario `4c-get-zc` vs `4c-get-copy` present average throughput of get operations with zero-copy API vs legacy API, with copying and creating the objects for each get. For JavaSkipListMap that doesn't have the zero-copy API regular get operation will be invoked both for `4c-get-zc` and for `4c-get-copy`. Scenario `4d-95Get5Put` scenario runs 95% get operations and 5% puts. Scenario `4e-entrySet-ascend` (or `4f-entrySet-descend`) runs ascending (or descending) scan of 10K pairs, returning both key and value either via zero-copy scan iterator API (for OakMap and OffHeapList) or via conventional scan iterator (for JavaSkipListMap). `4e-entryStreamSet-ascend` is applicable only for OakMap and performs stream scan as explained in

the paper. Each experiment is running after 10M pairs are inserted as a warm up phase.

A.8 Creating the plots

Given the `summary.csv` file, in order to create the plots use script: `oak/benchmarks/synchrobench/generate.py`. Both `summary.csv` file and `generate.py` script need to be in the same directory. To invoke the script do:

```
$ cd oak/benchmarks/synchrobench/
$ mv output/summary.csv .
$ python3 generate.py
```

The output PDFs files with the plots are going to be created in the same directory.

A.9 Notes

To know more about our library, send feedback, or file issues, please visit our Github page https://github.com/yahoo/oak.

A.10 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging