Oak: Off-Heap Allocated Keys for Big Data Analytics

Hagar Meir

Oak: Off-Heap Allocated Keys for Big Data Analytics

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

Hagar Meir

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tevet 5779 Haifa December 2018

The research thesis was done under the supervision of Prof. Idit Keidar in the Electrical Engineering Department.

The results of this thesis have been published as a paper by the author and research collaborators in the HAL archives: https://hal.archives-ouvertes.fr/hal-01789846.

In addition the author and other research collaborators published a paper in a conference: *N. Shalev, E. Harpaz, H. Porat, I. Keidar, and Y. Weinsberg, Csr: Core surprise removal in commodity operating systems, in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2016, pp. 773787.* This work is not included in this thesis.

The generous financial support of the Technion is gratefully acknowledged.

Acknowledgments

First, I would like to thank my advisor, Prof. Idit Keidar. Thank you for taking this journey with me, walking me through every step of the way. For believing in me since day one, showing me what I can achieve. Your guidance and support were priceless. It was my great pleasure to work with you and mostly learn from you.

I wish to thank the researchers with whom I collaborated: Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Eran Meir, and Gali Sheffi. Thank you for a very productive and enjoyable collaboration.

I would like to thank members of Idits research group: Alon Berger, Alexander (Sasha) Spiegelman, Noam Shalev, Naama Kraus, Kfir Lev-Ari, and Dani Shaket. I enjoyed our coffee breaks, the lunch time discussions, the group meetings, and especially the BBQs. In particular, I thank Alon Berger and Itay Tsabary for being the best office mates and friends I could ask for.

Last but not least, I would like to thank my family. For all the love, for always being supportive, showing interest in my work, and encouraging me to continue with my studies. Especially, my amazing husband and best friend Oz, I am very fortunate to have you by my side.

Contents

List of Figures

Abstract

We present Oak (Off-heap Allocated Keys), a scalable concurrent key-value map designed for real-time big data analytics. Oak offloads the data from the virtual machine heap in managed-memory languages like Java, thereby reducing garbage collection overheads.

Oak is optimized for big keys and values and for frequent incremental maintenance of existing values, as prevalent in streaming analytics use cases. To this end, it adopts a zero-copy approach to data update and retrieval, e.g., through concurrent update-in-place. Oak's API is similar to that of Java's ConcurrentNavigableMap with adjustments for efficient zero-copy implementation. It provides strong (atomic) semantics for read, write, and various read-modify-write operations, such as compute (in-situ update) and put-if-absent, as well as (non-atomic) ascending and descending iterators.

We provide proof of Oak's correctness, by identifying linearization points for all operations, so that concurrent operations appear to execute in the order of their linearization points. We further report on our experiments which show that Oak is faster by 1.3-4.8x than the currently standard concurrent KV-map, the Java ConcurrentSkipListMap. In addition, our results demonstrate that off-heap allocation is beneficial in scenarios with conditional updates of large values.

Our industrial partners are integrating Oak as the core data index in the popular Apache Druid in-memory analytics platform. This integration is beyond the scope of this thesis.

Abbreviations and Notations

Abbreviations

RAM	—	Random Access Memory
DRAM	—	Dynamic Random Access Memory
KV	—	Key-Value
CPU	—	Central Processing Unit
API	—	Application Programming Interface
GC	—	Garbage Collection
CAS	—	Compare and Swap
F&I	—	Fetch and Increment
F&A	—	Fetch and Add

Chapter 1

Introduction

Fueled by the steady decline in DRAM prices, the in-memory analytics market is on the rise. It is projected to grow from $1.26B in 2017 to $3.85B in 2022 [1]. Storing ever-growing amounts of data in main memory enables new processing paradigms, e.g., advanced real-time analytics over high-rate event feeds. Modern decision support systems continuously ingest large volumes of data, while providing up-to-date insights with minimum delay. For example, Apache Druid (incubating) [17] is a powerful platform for multi-dimensional exploration of event data, which is adopted by Airbnb, Alibaba, eBay, Netflix, Paypal, and Verizon (Oath), to name only a few. A prominent Druid application is Flurry Analytics [21] – a service for mobile developers that enables exploration of user characteristics (age, gender, location, app context, etc.) and behavior (e.g., which code paths they follow and how they churn). As of late 2017, Flurry infrastructure monitored 1M+ mobile apps on 2.6B devices [22].

In-memory analytics engines often implement complex data layouts and query semantics atop a simple dynamic *key-value (KV)-map* storage abstraction. A KV-map is an ordered collection of key-value (KV-)pairs that provides simple random write (*put*), random read (*get*), and range query (*scan*) API. In many cases, both keys and values are composites of application-level data. For example, consider a typical Druid table for Flurry that summarizes mobile traffic statistics like counts of page views, clicks, and unique visitors, grouped by date and user features (dimensions). Druid implements this table as a KV-map, in which (1) the keys are induced by unique combinations of dimension codes, and (2) the value for each key is a collection of aggregation objects that accumulate summaries, some of which are scalars while others are composite – e.g., *data sketchs* [13] (compact structures that maintain approximate stream statistics). Both keys and values are therefore big (hundreds of bytes to kilobytes).

Scaling the KV-map implementations on multi-CPU hardware is crucial for the overall system performance. Analytics engines achieve simultaneous high-speed data ingestion and reporting through *concurrent* read and write access to data from multiple threads. Efficient harnessing of 8 to 16 CPU cores per host is expected with middle-tier server hardware. Scaling with the growth of available RAM is usually no less important. For example, a typical multi-dimensional Druid table can easily exceed a million KV-pairs and have a multi-gigabyte footprint. Modern production servers often feature 192 to 384 GB of RAM, thereby potentially accommodating hundreds of such tables.

Existing in-memory KV-maps, in particular implementations in managed-memory languages like Java, are ill-suited for scaling to very big RAM sizes. Despite recent

advances, *garbage collection (GC)* algorithms struggle to scale with the volumes of memory in big data platforms, capping at heap sizes of about tens of gigabytes. For example, the Elasticsearch administrator guide recommends using a heap no bigger than 32GB [19]. This limitation has led developers of big data platforms to consider ad hoc *off-heap* memory allocators, as reported, e.g., by Druid [18], HBase [33], and others.

We address the demand for large in-memory concurrent KV-maps for analytics platforms, treating memory allocation as a first-class citizen. We design and implement Oak (*Off-heap Allocated Keys*), a scalable ordered concurrent KV-map for real-time analytics, which self-manages its memory off-heap. To the best of our knowledge, Oak is the first data structure to combine read-write parallelism with zero-copy access to internally managed data. A key design consideration in this context is the programming model and API, which ought to allow efficient access to internally stored keys and values.

We formally prove Oak's correctness and benchmark it under a variety of workloads. The evaluation results show decisive scalability and performance benefits over the de-facto standard Doug Lee's JDK8 ConcurrentSkipListMap [30].

We now describe in detail Oak's key features and design decisions, and then survey prior art.

1.1 Design principles

Off-heap allocation. One of the principal motivations for using Oak in Druid and similar analytics engines is offloading the KV-map from the managed-memory heap. This allows working with an order-of-magnitude more memory (hundreds of gigabytes instead of tens) and serializing data to avoid Java's memory overhead for object headers. It further counters undesirable phenomena introduced by GC, like unpredictable timing of GC cycles, which aggravates tail latencies and may even render the system unresponsive [4].

Zero-copy API. Oak provides functionality similar to Java's ConcurrentNavigableMap (implemented, e.g., by ConcurrentSkipListMap). However, direct support of this API is not suited as-is for self-managed memory, where data is stored in internal buffers rather than in first-class objects. For example, the traditional get interface returns a value object. Supporting this API would entail deserializing and copying the entire object from the internal buffer. Instead of doing so, Oak returns a lightweight façade object (*buffer view*) that allows deferred access to the value's components. Similarly, the put method omits the return of the old value, with the same rationale.

The *update-in-place* principle is another manifestation of the zero-copy approach. Oak allows in-situ update of objects in internal buffers through the use of lambda functions. This technique allows efficient incremental maintenance of big values (e.g., aggregate sketches). To allow zero-copy safe concurrent access to such values, Oak introduces the abstraction of *handles* – an indirection that frees application programmers from the need to deal with concurrency control.

Correctness guarantees. Oak is inherently thread-safe. It provides strong (atomic) semantics for get, put, remove, and various read-modify-write operations, such as compute (update-in-place) and conditional insertion (put-if-absent). Note that in contrast,

in the current Java ConcurrentSkipListMap implementation, if compute performs in-place updates, its operations are not guaranteed to be atomic. Supporting atomic conditional updates alongside traditional (unconditional) put operations necessitated designing a new concurrent algorithm. We are not aware of any previous algorithm addressing this challenge.

Memory organization for efficient lookup. Similarly to some recently suggested data structures (e.g., [5,7,8]), Oak stores its keys in contiguous *chunks*, which speeds up queries through locality of access. This is challenging in the presence of dynamic-sized keys and values. In contrast, existing chunk-based data structures [5,7,8] maintain fixed-size serialized keys and values inline, without the additional indirection level.

Expedited descending scans. Oak's range scans are not atomic in the sense that the set of keys in the scanned range may change during the scan; supporting atomic range queries would be more costly [1], which is not justified for most analytics scenarios. Although analytics queries require both ascending and descending scans, no previous concurrent data structure we are familiar with has built-in support for the latter. Rather, descending scans are implemented by invoking a query (get) for a smaller key after each scanned key. In contrast, Oak's chunk-based organization is amenable to expediting descending scans without the complexity of managing a doubly-linked list. In our experiments, Oak's descending scans are 4.8x faster than ones using ConcurrentSkipListMap.

Summary. All in all, Oak is the first KV-map designed to address the needs of big-data real-time analytics engines, including off-heap memory allocation, incremental in-place maintenance of large, variable-size values and keys, index locality for fast queries, and efficient descending scans. However, its API is not fully compatible with ConcurrentNavigableMap for efficiency reasons, and hence, porting applications to use Oak requires extra effort.

1.2 Related work

Substantial efforts have been dedicated to developing efficient concurrent data structures [2, 3, 5, 7–11, 14–16, 20, 23, 26, 28, 30, 34, 35]. Each of [2, 3, 8–11, 14, 16, 20, 34] presents a different technique for creating efficient concurrent search trees; other work focus on scalable concurrent algorithms for priority queues [6], linked lists [7,25], and skip lists [15,26,27].

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 [10,14,16,20,26,34]. Moreover, the ones that do hold key-value pairs typically maintain fixed-size keys and values [5,7,8] and do not support large, variable-size keys and values as Oak does.

The only exception we are aware of is JDK's ConcurrentSkipListMap [30], which does support general objects as keys and values, and also implements the full ConcurrentNavigableMap API. Nevertheless, its compute is not necessarily atomic, its organization is not chunk-based and so searches do not benefit from locality, and its

[1] We also experimented with atomic scans and they were ∼25% slower.

descending iterators are inefficient, as we show in this paper. Note further that unlike Oak, ConcurrentSkipListMap does not deal with memory allocation (it stores pre-allocated key and value objects), and is therefore subject to GC scalability limits. Finally, ConcurrentSkipListMap does not manage concurrent access to the objects it stores, which complicates application development in comparison with Oak.

Chunk-based structures were first introduced by [7] to allow cache-conscious lock-free linked lists. Later, in [8], the chunk mechanism was used to create a lock-free B^+ tree. KiWi [5] is a KV-map that supports atomic scans and its data is organized in a collection of chunks.

Chunk-based allocation was used in previous research [5,7,8], but not with variable-size entities or off-heap allocation. Custom off-heap memory management is employed by multiple data management systems (e.g., Druid [18] and HBase [33]), predominantly for storing immutable data. Updates in-place were recently considered also in the context of persistent key-value storage [12] but not in-memory KV-maps. To the best of our knowledge, Oak is the first general-purpose *data structure* library that provides a dynamic ordered map API with built-in concurrency control and zero-copy updates, allocated completely off-heap. Using it both simplifies the development and improves the performance of big data platforms, as our Druid proof-of-concept demonstrates.

Chapter 2

Programming model

Oak is unique in supporting a map interface for self-managed data, which it stores in internal buffers, as discussed in Section 2.1. In order to handle concurrent access to Oak-resident values as well as the dynamic memory usage of such values, Oak uses the abstraction of handles with pluggable concurrency control, as described in Section 2.2. We detail Oak's API in Section 2.3.

2.1 Oak buffers and serialization

A key consideration in the design of Oak is allowing keys and values to be kept in self-managed (off-heap) memory. Thus, in contrast to Java data structures, which store Java objects, Oak stores data in internal buffers. To convert objects (both keys and values) to their *serialized* buffer forms, the user must implement the `OakSerializer` interface given in Algorithm 1. This interface consists of a (1) serializer, (2) deserializer, and (3) serialized size calculator (for variable-sized keys and values).

Oak's insertion operations use the size calculator to deduce the amount of space to be allocated, then allocate space for the given key or value, and finally use the serializer to write the key or value directly to the allocated space. By using the user-provided serializer, we create the byte representation of the object directly into Oak's in memory, saving the need to copy it.

Algorithm 1 Interface for user-provided Oak serializer and comparator.

```
public interface OakSerializer<T> {
  // serializes the object
  void serialize(T source, ByteBuffer targetBuffer);
  // deserializes the given byte buffer
  T deserialize(ByteBuffer byteBuffer);
  // returns num of bytes needed for serializing the object
  int calculateSize(T object);
}

public interface OakComparator<K> {
  int compareKeys(K key1, K key2);
  int compareSerializedKeys(ByteBuffer serializedKey1, ByteBuffer serializedKey2);
  int compareSerializedKeyAndKey(ByteBuffer serializedKey, K key);
}
```

To allow efficient search over buffer-resident keys, the user is required to provide the

`OakComparator` interface for keys, which is also given in Algorithm 1. The comparator compares two keys, each of which may be provided either as a deserialized object or as a serialized buffer. It determines whether they are equal, and if not, which is bigger.

After key-value pairs are ingested, internally kept keys and values may be accessed as memory buffers of two types, `OakRBuffer` (read-only) and `OakWBuffer` (read and write), supporting the standard API for read-only Java ByteBuffers and writable Java ByteBuffers, respectively. In addition to the standard ByteBuffer API, Oak buffers manage concurrency control and dynamic memory allocation for their users. Thus, user functions can run computational steps on Oak-resident values with no concern of concurrency control or memory overflow and release. To this end, Oak buffers use the abstraction of handles, as described in Section 2.2 below.

`OakRBuffer`s are accessed via Oak's `getBufferView` API, which returns an `OakBufferView` object, which, in turn, supports `get(key)` and iterators returning `OakRBuffers`. `OakWBuffer`s are accessed by user-provided lambda functions passed to Oak's various compute methods. Oak buffer objects are created on demand by operations that need access to the key or value; they are ephemeral, and cease to exist once the operation is completed.

The buffer-based direct access to serialized key-value pairs reduces copying and deserialization of the underlying mappings. Furthermore, it relieves programmers of the need to implement concurrency control for update operations. Note however, that Oak's `get` returns access to the same underlying memory buffer that compute operations update in-place, and the granularity of Oak's concurrency control is at the level of individual operations (such as `getInt`) on that buffer. Therefore, the reader may encounter different values – and even value deletions[1] – when accessing a buffer returned from the same `get` multiple times. This is of course normal behavior for a concurrent map that avoids copying.

As an aside, we note that Oak supports two additional access views to objects in addition to `OakRBuffer`s. First, for backward compatibility with legacy code, we support "standard" `get(key)` operations and iterators returning objects rather than buffers, but this requires deserializing the keys and values and is therefore less efficient. Second, the *transformed view* API allows users to provide lambda functions for extracting partial information out of the serialized buffer, and the operations return this extracted information rather than the full `OakRBuffer`; in contrast to the `OakRBuffer`, the returned value is copied and hence remains unchanged after the `get` returns it. We do not discuss these two views further in the paper.

2.2 Handles, concurrency control, and dynamic memory use

To facilitate programming with Oak buffers, Oak allows user code to access buffers without worrying about concurrent access or dynamic memory consumption. The user function provided in a compute operation expects an `OakWBuffer`, and can run computational steps on the value with no concern of concurrency control. It can also increase the size of the buffer without worrying about its reallocation.

To this end, Oak's value buffers employ an additional indirection called *handle*, as shown in Figure 2.1. Each value has its own handle and threads are directed by Oak

[1] An `OakRBuffer` method throws a `ConcurrentModificationException` in case the mapping is concurrently deleted.

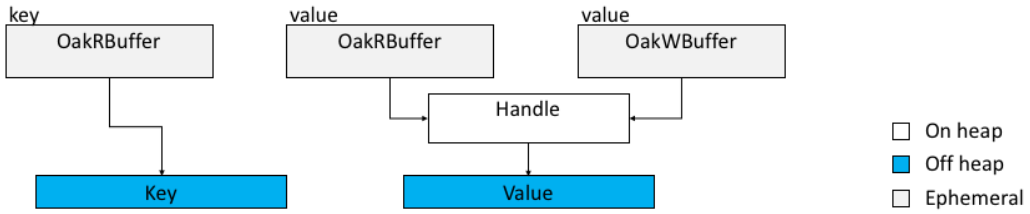

Figure 2.1: Oak handle and buffers.

to this handle for each read or write of the value. `OakWBuffer` extends the ByteBuffer interface and wraps the handle.

The handle implementation is pluggable, and may use different types of concurrency control (optimistic or pessimistic). We provide a simple handle implementation using a read-write lock. In addition, the handle interacts with the memory allocator in order to dynamically increase value sizes (requesting a new allocation and copying the buffer to it if needed), and informs it when to reclaim buffers that are no longer needed. The memory manager is a separate module, and is also pluggable in Oak.

Once a value is removed from Oak, the handle assures that no thread will attempt to read this value, since that memory may be reclaimed. To this end, the handle has a `remove` method that performs a logical remove by marking the handle as deleted. A key is deemed present in Oak only if it is associated with a non-deleted handle.

Since the handle is an on-heap object it remains reachable to all threads that hold `OakBuffer`s that warp it, even though the value's memory (off-heap) may have been reclaimed. In this sense, the handle serves as a bridge between the on-heap and the off-heap memory parts of Oak.

The handle further offers `put` and `compute` methods that are used by Oak to replace and update values, respectively. The handle's `put` method directs the handle to point to the given value, and its `copmute` method executes a user-provided lambda, ensuring that the update occurs *atomically*.

2.3 API

Oak's API is given in Algorithm 2. For data retrieval, `OakBufferView` offers – `get(key)` and iterators (iterating over keys, values, or entries containing both). The `subMap` and `descendingMap` methods are used for range and descending iterators (resp.). As noted above, Oak provides memory allocation and (epoch-based) reclamation for its internally kept keys and values. To this end, we need an indication of the end of each operation. Unlike other operations, the end of the iterator operation is outside Oak's control. To allow its discovery, Oak's iterator implements the Java AutoCloseable Interface, which adds a `close` method to the iterator; more details are given in Section 3.3.

The data ingestion API supports five conditional and unconditional updates: `put`, `putIfAbsent`, `remove`, `computeIfPresent`, and `putIfAbsentComputeIfPresent`. The latter two take a user-provided `computer` function to apply to the value mapped to the given key, which they do atomically.

Because the ConcurrentNavigableMap API (like all Java APIs) was designed with managed memory in mind, it required several adaptations in order to become more off-heap-friendly. First, as explained above, data retrieval methods replace Java ob-

Algorithm 2 Oak API.

```
//  OakBufferView methods for data retrieval:
OakRBuffer get(K key)
CloseableIterator<OakRBuffer> keysIterator()
CloseableIterator<OakRBuffer> valuesIterator()
CloseableIterator<Map.Entry<OakRBuffer, OakRBuffer>> entriesIterator()

OakMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
OakMap<K, V> descendingMap()

// Ingestion methods:
void put(K key, V value)
boolean putIfAbsent(K key, V value)
void remove(K key)
boolean computeIfPresent(K key, Consumer<OakWBuffer> computer)
void putIfAbsentComputeIfPresent(K key, V value, Consumer<OakWBuffer> computer)
```

jects with `OakRBuffer`s and the `computer` functions passed in the update methods manipulate `OakWBuffer`s. Second, the `put` API differs from the one in ConcurrentNavigableMap in that it does not return the old value. This is because returning the old value inevitably requires copying that value, which violates Oak's zero-copy design principle. For similar reasons, Oak does not require the user-provided function to return the computed value (as ConcurrentNavigableMap does); note that since the computation steps are performed in-place, the new computed value is already accessible.

Oak increases the value's memory allocation if the updating function requires it. (As explained above, this is managed by the handle). In the Druid integration, `computer` functions update different types of aggregates, ranging from simple ones (like counters and sums) to more complex data sketches.

Chapter 3

Oak algorithm

We now describe the Oak algorithm, which implements a concurrent key-value map supporting various atomic (linearizable) read and update operations, and non-atomic ascending/descending iterators over the map and sub-maps. Keys and values are variable sized. Oak makes use of commodity atomic hardware operations like CAS, F&I, and F&A.

Section 3.1 explains Oak's chunk-based data organization. Section 3.2 details how Oak's operations are implemented. Section 3.3 discusses epoch-based memory reclamation.

3.1 Data organization

Chunks and index. Oak's structure is chunk-based; it is organized as a linked list of large blocks of contiguous key ranges, as suggested in [7]. 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 $k < C.next.minKey$. The chunk object has a dedicated *rebalance* procedure, which splits chunks when they are over-utilized, merges chunks when they are under-utilized, and reorganizes the chunks' internals. The rebalancer is implemented as in previously suggested chunk-based constructions [5, 8]. Since it is not novel and orthogonal to our contributions, we do not detail it here.

To allow fast access, we follow the approach of [5, 6, 27, 28, 35] and add an *index* that maps keys to chunks, as illustrated in Figure 3.1. Each chunk is indexed according to its minKey. 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 (as in [5]).

The index supports standard lookup, insert, and remove operations. It further supports a flavor of lookup that returns the chunk with the greatest minKey that is strictly lower than the search key, which is used by the descending iterator. Insert and remove are exclusively used by rebalance. In addition, Oak provides the `locateChunk(k)` method, which returns the chunk whose range includes key `k`, by querying the index and possibly a chunks linked list traversal. This method is used in all of Oak's operations.

Intra-chunk organization. As shown in Figure 3.2, chunks hold three types of objects: entries, keys, and handles. *Entries* reside on the managed memory heap in an array-based linked list, sorted in ascending key order. Each entry holds a pointer

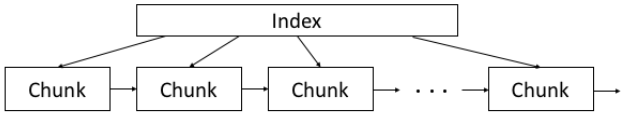

Figure 3.1: Oak chunk list and index.

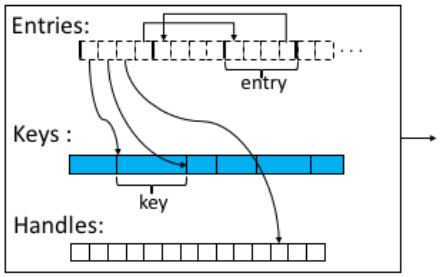

Figure 3.2: Oak intra-chunk organization.

to a key, a pointer to a handle, and the index of the entry that holds the next key in the linked list. Oak makes sure that a key does not appear in more than one entry. Keys are variable size so each entry holds a pointer to the beginning of a key and its size. Keys are stored in a large off-heap ByteBuffer that is considered part of the chunk, whereas values reside outside the chunk. Handles are stored within the chunk, on-heap. There is a single handle per value, and once a key-value pair is removed from Oak its handle is deleted and never reused (subjected to GC).

As in previous work [5], when the rebalancer creates a new chunk, some prefix of the entries array is filled with data, and the suffix consists of empty entries for future allocation. This prefix is initialized 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. In case the insertion order is random, inserted entries are most likely to be distributed evenly between the ordered prefix entries, thus creating fairly short bypasses.

Chunk objects and rebalancing. A chunk object exposes an API for searching, allocating, and writing, as we now detail. `LookUp` searches for an entry corresponding to the given key. 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 there is at most one relevant entry. `AllocateEntryAndKey` allocates a new entry in the entries array and also allocates and writes the given key that it points to; `AllocateHandle` allocates a new handle in the handles array. Allocations are done using atomic hardware operations like F&A and F&I, so that the same space is not allocated twice. After allocating a new entry, Oak tries to link this new entry into the entries linked list by calling `entriesLLputIfAbsent`, which uses CAS for safe insertion to the linked list, so that the invariant of a key not appearing more than once in Oak is preserved. If it encounters a linked entry with the same key (added by a concurrent insertion operation), then it returns the encountered entry. `WriteValue` allocates space for the value (outside the chunk) and writes the value to it.

The allocation procedures (`allocateEntryAndKey` and `allocateHandle`) may trigger a rebalance "under the hood". In this case, the allocate procedure fails returning $\perp$ and Oak retries the update operation. In case the chunk is being rebalanced, chunk update methods (`entriesLLputIfAbsent`) fail and return $\perp$.

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, the rebalancer may help published operations complete (in order to ensure lock-freedom), but for simplicity, our description in Section 3.2 below assumes that the rebalancer does not help published operations. 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 encounters a rebalance fail (return $\perp$), chunk methods that read the chunk (`lookUp`), modify existing handles (`writeValue`), or `unpublish` an operation proceed with no need to abort. Beyond rebalancing, which is borrowed from earlier work [5], the implementation of the chunk's operations is straightforward.

The rebalancer preserves the integrity of the chunks linked list, as we now specify. A *path* is a sequence of chunks $C_1, C_2, \ldots, C_k$ reached by traversing chunks' `next` pointers in a run (until reaching a `null` pointer). *Traversals*(C_0, r) is the sequence of keys in all paths starting from chunk C_0 in all extensions of run r.

The rebalancer implementation guarantees the following: If `locateChunk(k)` returns C at time t in run r, then for every traversal $T \in$ *Traversals*(C, r) :

RB1 all keys $\geq$ `k` that are inserted before time t and not removed after time t are reachable in T;

RB2 all keys $\geq$ `k` that are removed before time t and not inserted after time t are not reachable in T; and

RB3 keys are encountered in T in monotonically increasing order.

3.2 Oak operations

In Section 3.2 we discuss Oak's queries, namely `get` and iterators. Oak's support for various conditional and unconditional updates raises some subtle interactions that need to be handled with care. We divide our discussion of these operations into two types: insertion operations, which may add a new value to Oak, are discussed in Section 3.2, whereas operations that only take actions when the affected key is in Oak are given in Section 3.2. To argue that Oak is correct, we identify in Section 3.2 *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 Section 5.

Queries

Get. The `get` operation is given in Algorithm 3. `get` returns a read-only view (`oakRBuffer`) of the handle that holds the value that is mapped to the given key, in accordance with our zero-copy policy. 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,

Algorithm 3 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)
```

in which case the handle will be marked as deleted; reads from the `oakRBuffer` view check this deleted 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 checks if the handle is deleted (line 6). 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.

Ascending iterator. The ascending iterator 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 $\perp$ and the handle is not deleted. Otherwise, it continues to the next entry.

Descending iterator. 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 3.3 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 the iterator 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 iterator continues by querying the index again, but now for the chunk with the greatest minKey that is strictly smaller than the current chunk's minKey. From the chunk returned by the index, we again traverse the chunks linked list until the last chunk with a smaller minKey than the last key the iterator returned.

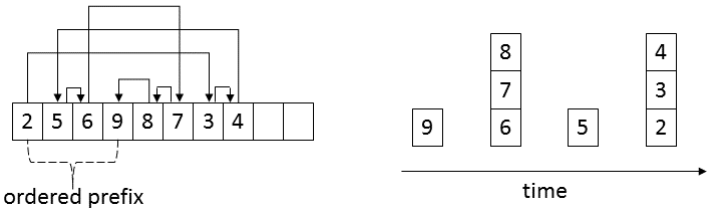

Figure 3.3: Example entries linked list (left) and stacks built during its traversal by a descending iterator (right).

Iterator correctness. By RB1-3 it is easy to see that the iterators algorithm described above guarantees the following:

1. An iterator returns all relevant keys that were inserted to Oak before the start of the iteration and not removed until the end of the iteration.
2. An iterator does not return keys that were removed from Oak before the start of the iteration and not inserted until the end of the iteration.
3. Iterators do not return the same key more than once.

Note that relevant keys inserted or removed concurrently with an iteration may be either included or excluded.

Insertion operations

The insertion operations – `put`, `putIfAbsent`, and `putIfAbsentComputeIfPresent` – try to associate the given key with a new value using the `doPut` function in Algorithm 4.

`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 – 26) 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 24). These atomic handle operations return `false` if the handle is deleted (due to a concurrent remove), in which case we retry (line 25).

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 28), and then try to link this new entry into the entries linked list (line 29). Either way, we allocate a new handle (line 30). These functions might fail and cause a retry (line 31).

If `entriesLLputIfAbsent` receives $\perp$ as a parameter (because the allocation in line 28 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 28 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 33).

We complete the insertion by using CAS to make the entry point to the new handle index (line 36). Before doing so, we publish the operation (as explained in Section 3.1),

Algorithm 4 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 succ ← handle.compute(func)
25:        if ¬succ then return doPut(key, val, func, operation)
26:        return true
  ▷ Case 2: key is absent
27:    if ei = ⊥ then
28:        ei ← C.allocateEntryAndKey(key)
29:        ei ←C.entriesLLputIfAbsent(ei)
30:    newHi ←C.allocateHandle()
31:    if ei = ⊥ ∨ newHi = ⊥ then                    ▷ allocation or insertion failed
32:        return doPut(key, val, func, operation)
33:    C.writeValue(newHi, val)
34:    if ¬C.publish(ei, hi, newHi, func, operation) then
35:        return doPut(key, val, func, operation)
36:    result ← CAS(C.entries[ei].hi, hi, newHi)
37:    C.unpublish(ei, hi, newHi, func, operation)
38:    if ¬result then return doPut(key, val, func, operation)
39:    return true
```

which can also lead to a retry (line 35). After the CAS, we unpublish the operation, as it is no longer pending (line 37). If CAS fails, we retry the operation (line 38).

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 5 Oak's non-insertion update operations

```
40: procedure computeIfPresent(key, func)
41:     return doIfPresent(key, func, comp)

42: procedure remove(key)
43:     doIfPresent(key, ⊥, rm)
44:     return

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

59: procedure finalizeRemove(prev)
60:     C, ei, hi, handle ← ⊥
61:     C ← locateChunk(key);
         ei ← C.lookUp(key)
62:     if ei ≠ ⊥ then hi ← C.entries[ei].hi
63:     if hi = ⊥ then return true
64:     handle ←C.handles[hi]
65:     if handle ≠ prev then return true
66:     if ¬C.publish(ei, hi, ⊥, ⊥, rm) then return finalizeRemove(prev)
67:     CAS(C.entries[ei].hi, hi, ⊥)
68:     C.unpublish(ei, hi, ⊥, ⊥, rm)
69:     return true
```

Non-insertion operations

The second type of updates – `computeIfPresent` and `remove` do not insert new entries. Both invoke the `doIfPresent` function given in Algorithm 5. It first locates the handle, and if there is no such handle, returns `false` (line 49).

In `computeIfPresent`, if the handle exists and is not deleted (case 1), we run handle.compute and return `true` if it is successful (line 52). 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 55). Since this affects the chunk's entries, we need to synchronize with a possibly ongoing rebalance, and so here too, we publish before the CAS and unpublish when done. If publish or CAS fails then we retry (lines 54 and 57). The operation returns `false` whenever it does not find the entry, or finds the entry but with $\perp$ as its handle index (line 49), or CAS to $\perp$ is successful (line 58).

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 53), and we say that the remove is *successful*. This makes all other threads aware of the fact that the key has been removed, so there will be no further attempts to read its value, 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 $\perp$. Updating the entry serves two purposes: first, rebalance does not check whether a handle is deleted, so changing the handle index to $\perp$ 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 the `finalizeRemove` function, which tries to CAS the handle index to $\perp$. We have to take care, however, in case the handle index had already changed, not to change it to $\perp$. 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 65). 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. We note that `remove` is linearized at the point where it marks the handle as deleted, and therefore 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 already set the handle index to $\perp$.

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 $\perp$ (line 55). In this case (case 2) it does not perform `finalizeRemove`, but rather retries if the CAS fails (line 57). 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 $\perp$ 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 $\perp$ before returning.

Linearization points

In Section 5 we show that Oak's operations (except for iterators) are *linearizable* [29]; that is, every operation appears to take place atomically at some point (the linearization point) between its invocation and response. Here, we list the linearization points, abbreviated *l.p.*.

putIfAbsent – if it returns `true`, the l.p. is the successful CAS of handle index (line 36). 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 36). 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 36). Otherwise, the l.p. is upon a successful nested call to handle.compute (line 24).

computeIfPresent – if it returns `true`, the l.p. is upon a successful nested call to handle.compute (line 52). Otherwise, the l.p. is when the entry is not found, or it is found but with $\perp$ as its handle index (line 49), or a successful CAS of handle index to $\perp$ (line 55).

remove – if it is successful, the l.p. is when a successful nested call to handle remove occurs, setting the handle to deleted (line 53). Otherwise, the l.p. is when the entry is not found, or handle index is $\perp$ (line 49), or a deleted handle is found and a successful CAS of handle index to $\perp$ occurs (line 55).

get – if it returns a handle, then the l.p. is the read of a non-deleted handle (line 6). If, it returns NULL there are two cases. If there is no relevant entry then the l.p. is when `lookUp` (line 3) returns $\perp$, or when get reads that the handle index is $\perp$ (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. Therefore, if `get` finds `deleted` = `true`, then 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).

3.3 Off heap support - epoch-based reclamation

We use epoch-based reclamation (based on [25]) to support off-heap keys and values. We implement a global timestamp that is incremented at the beginning of each operation. Each thread maintains an active flag and a local timestamp. When a thread performs an operation (query, insertion, or non-insertion), it first sets its active flag, increments the global timestamp, and updates its local timestamp to match the global one. At the end of an operation, the thread unsets its active flag. When a thread calls `remove`, it attaches the removed value to a release list with the current global timestamp. The keys are released in the same manner during rebalance. An entry in the release list whose timestamp is smaller than the minimum current local timestamp of any active thread can be reclaimed.

One deviation we make from the basic protocol is to expand the active flag from one bit to several bits (we use one byte), to allow nesting of operations, since we consider an iteration as a single continuous operation. Therefore, a thread starts an operation by incrementing its active counter, and decrements it when the operation ends. A thread is considered active when its active counter is positive. When using an iterator, we decrement the active counter by calling the `close` method of the iterator. For this reason we use closeableIterators.

Chapter 4

Evaluation

We implement Oak in two ways: off-heap and on-heap. The two implementations differ in the memory used for the allocation of keys and values that reside in Oak and the management of that memory. Both implementations are in Java, the handles are implemented using a Java ReentrantReadWriteLock [31], and the index is based on Java ConcurrentSkipListMap [30]. In order to unify the on- and off-heap versions, we use Oak to allocate space for keys and values in both; for fairness of the comparison, we also copy new values before inserting them into the Java skiplist, to simulate the serialization done in Oak. The experiments are run on a hardware platform with four Intel Xeon E5-4650 processors, each with 8 cores. We use the Synchrobench framework [24]. Each experiment consists of 10 iterations, a few seconds each, and we report the average result. Unless stated otherwise, the experiment is preceded by a warm-up period where randomly selected keys are inserted into the map. We first compare Oak's on-heap implementation to the (on-heap) Java skiplist, and then compare the two versions of Oak.

4.1 Oak vs. Java skiplist

Oak is a scalable map that achieves high throughput. We show this by comparing Oak to Java ConcurrentSkipListMap (skiplist) [30]. Java skiplist holds arbitrary objects as its keys and values, including ByteBuffers as in Oak. Similarly to Oak, Java skiplist supports put, remove, get, and ascending and descending iterators. We first run experiments consisting of these operations. The map is initially filled up with 1M randomly selected keys out of a range of 2M, keys are 4 bytes and values are 100B (in Figure 4.2 we further report on experiments with 1KB values, where the results are similar). Figure 4.1 depicts the throughput scalability with the number of threads.

The first experiment is a read-only workload (Figure 4.1a). Oak's chunk-based structure has better locality than the skiplist, and indeed we see that Oak's get operation outperforms Java skiplist's by 3.3x. The second (Figure 4.1b) is a write-only workload with 50% puts and 50% removes. Here, Oak outperforms Java by 1.3x, again, thanks to speeding up the search, which is the first part of an update operation. Next we run ascending (Figure 4.1c) and descending iterators (Figure 4.1d), scanning ranges of 100 keys. The ascending iterator is twice as fast as Java's, again, thanks to fast first key search. Oak has built-in support for descending iterators while Java skiplist does not, and so in this workload, Oak outperforms Java significantly – by 4.8x.

Next we evaluate the compute operation. Java supports an operation called com-

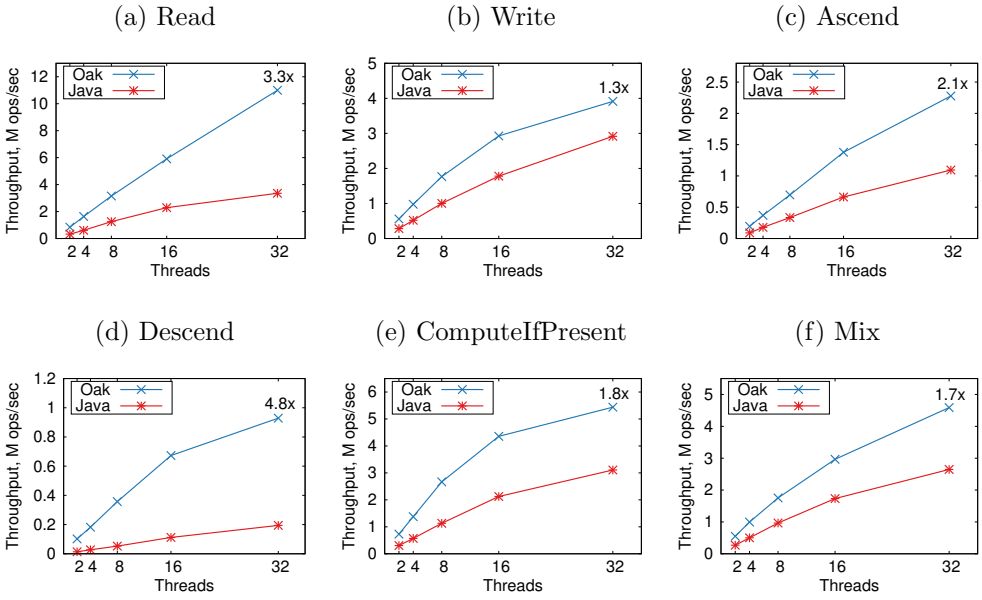

Figure 4.1: Oak vs. Java, 100B values.

pute, which also assigns a new value computed by a function received as a parameter, however the returned value from the function is used as the new value, therefore this compute is not an atomic update-in-place as in Oak. If the function updates the received value in-place, there are no atomicity guarantees. To allow for a fair comparison, we create a new object *LockableByteBuffer* consisting of a ByteBuffer and a Java ReentrantReadWriteLock [31] (as used in the handle implementation), and store LockableByteBuffers in the Java skiplist. We implement a function that first locks the given value using the write lock, then runs the computation steps on that value, and finally unlocks the lock. The function returns a pointer to the same value. In our workload, the computation reads a random byte of the given ByteBuffer and writes it to a random byte of the same ByteBuffer. We run two workloads, one executing only computeIfPresent, and one mixed workload with get, compute (putIfAbsentComputeIfPresent), put, and remove, 25% each. In both workloads, Oak outperforms Java skiplist by 70-80%, as shown in Figures 4.1e and 4.1f. The speedup is due to the fast search in Oak and the extra CAS in Java's compute, which is used to replace the current value with the returned value computed by the user's function.

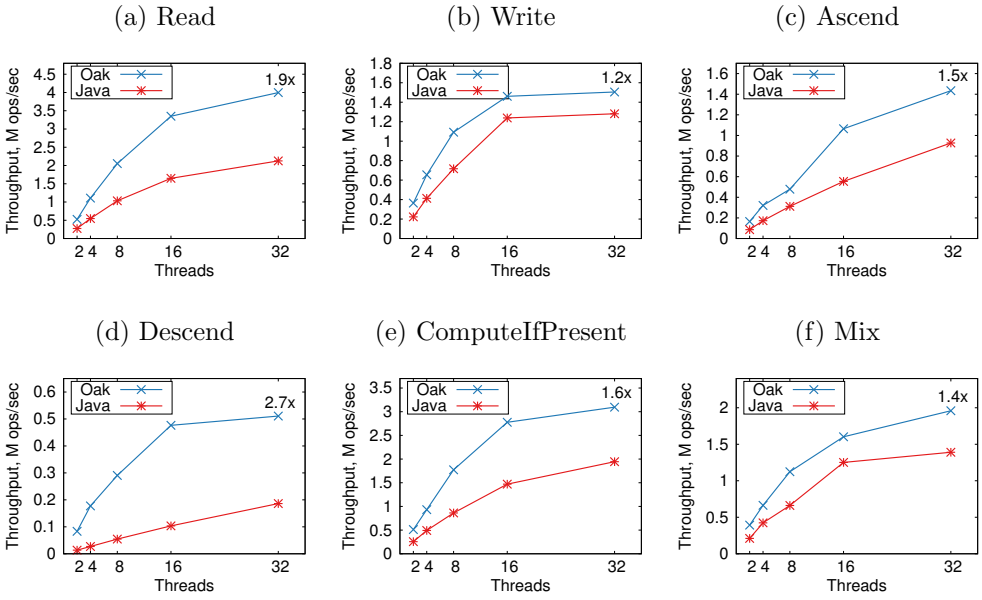

Figure 4.2: Oak vs. Java, 1KB values.

4.2 Oak Off-heap vs. On-heap

We show that there are use cases where one can benefit from off-heap allocation. We configure JVM to use the same amount of memory for both implementations. Since our off-heap implementation currently supports only 2GB of off-heap memory, we configure the off-heap heap size to 500MB, and the on-heap heap size to 2.5GB (= 2GB + 500MB). In the following experiments we populate Oak with ∼100K keys, 4 bytes each, with 1KB or 5KB values; the memory used to hold these keys and values nearly reaches the 2GB capacity of Oak's off-heap memory.

We implement and supply Oak with simple off-heap allocation and deallocation methods. At the initialization, one continuous 2GB off-heap ByteBuffer is allocated, and the allocation method uses an atomically incremented index over the ByteBuffer to manage Oak's requests. Deallocation appends an entry to a list of reclaimed memory locations, and the allocator scans it linearly. Note that in contrast, Oak's on-heap implementation uses Java's highly optimized GC to manage the memory for keys and values, since they all reside in the Java heap.

The main scenarios in Druid, the platform that Oak is designed to support, are ones with conditional updates on large values. When Druid runs out of memory it usually archives it in persistent storage. Therefore, we run experiments demonstrating these scenarios, and show that in these cases even our simple (unoptimized) allocation/deallocation is sufficient, and Oak off-heap prevails, since Java's GC works hard on trying to free space for future allocations.

But before running these Druid-oriented experiments, we first study the impact of off-heap allocation on a classical mixed workload, as depicted in Figure 4.3a. We run a mixed workload with equal shares of get, put, remove, and compute (putIfAbsent-ComputeIfPresent) operations with 1KB values. In this experiment Oak is initially filled with 50K randomly selected keys out of a range of 100K. In this case the off-heap

implementation is 20% slower, since the Java GC is better optimized for this case compared to Oak's simple GC implementation.

Next we run the Druid scenarios, with bigger values (5KB) and conditional updates. In these experiments, shown in Figures 4.3b, 4.3c, and 4.3d, off-heap outperforms on-heap by 40-80%. In the first experiment we use putIfAbsent to fill up the data structure (with 300K keys), which is initially empty. In the second experiment we run a mixed workload of putIfAbsent and putIfAbsentComputeIfPresent, after the data structure is initially filled with 150K keys out of a range of 300K. In the last experiment we run putIfAbsentComputeIfPresent with a data structure initially filled with 300K keys (out of a range of 300K). As expected, when monitoring the GC using Java's JConsole [32], we observe that Java's GC wastes time by trying to free space while Oak's simple deallocation method is almost idle.

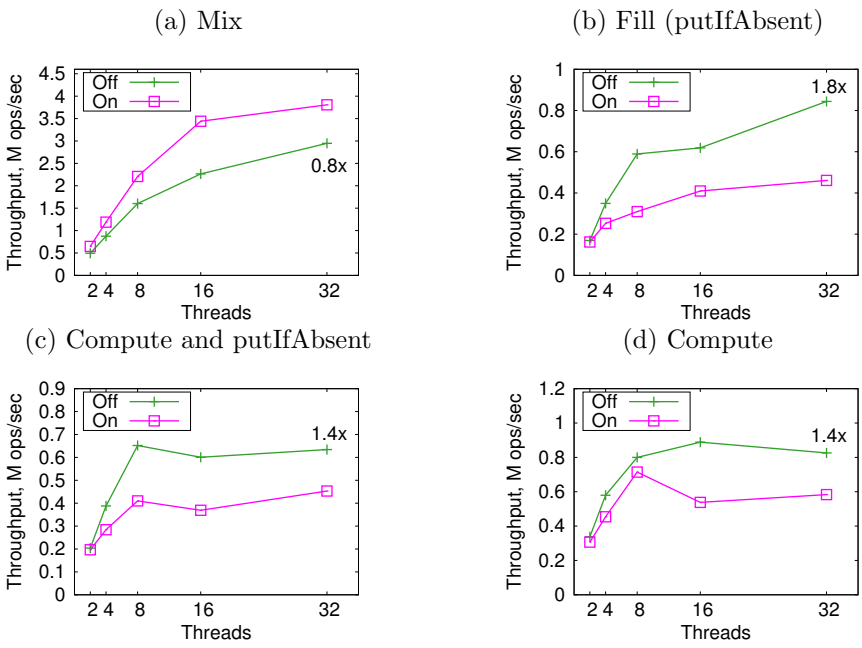

Figure 4.3: Oak off-heap vs. on-heap, 1-5KB values.

Chapter 5

Correctness

In this section, we prove Oak's correctness. Since the rebalancer is orthogonal to our contribution, we omit it from the discussion of Oak's correctness. We only assume that RB1-3 hold. We note that a similar rebalance was fully proven in [8].

5.1 Preliminaries

We consider a shared memory system consisting of a collection of shared variables accessed by threads, which also have local variables. An *algorithm* defines the behaviors of threads as deterministic state machines, where state transitions are associated with either an instance of a shared variable primitive (read, write, CAS, etc.) or a local step affecting the thread's local variables. A *configuration* describes the current state of all local and shared variables. An *initial configuration* is one where all variables hold an initial value. A data structure implementation provides a set of operations, each with possible parameters. We say that operations are *invoked* and *return* or *respond*. The invocation of an operation leads to the execution of an algorithm by a thread. Both the invocation and the return are local steps of a thread. A *run* of algorithm $\mathcal{A}$ is an alternating sequence of configurations and steps, beginning with some initial configuration, such that configuration transitions occur according to $\mathcal{A}$. We say that two operations are *concurrent* in a run r if both are invoked in r before either returns. We use the notion of time t during a run r to refer to the configuration reached after the t^{th} step in r. An *interval* of a run r is a sub-sequence that starts with a step and ends with a configuration. The *interval of an operation op* starts with the invocation step of *op* and ends with the configuration following the return from *op* or the end of r, if there is no such return.

An implementation of concurrent data structure is *linearizable* [29] (a correctness condition for concurrent objects) if it provides the illusion that each invoked operation takes effect instantaneously at some point, called the *linearization point* (l.p.), inside its interval. A *linearization* of a run r ($lin(r)$) is the sequential run constructed by serially executing each operation at its l.p.

5.2 Linearizability proof

Definition 1. If there is an entry e in Oak that points to key k and handle h, (i.e., lookUp(k) returns e s.t. $h =$ `handles[entries[e].hi]`) and h`.deleted = false`, we say that h is *associated with* k.

Claim 2. If an Oak operation searches for key k and finds a non-deleted handle h (`h.deleted` = `false`), then h is associated with k.

Proof. If an operation searches for k and finds h, then there is an entry e that points to k, since Oak ensures that there is at most one entry that points to k, and k is found only if there is such entry. This also means that e points to handle h (by handle index hi). Assume that e does not point to handle h, then the handle index is now $hi' \neq hi$. If $hi' = \perp$ then the handle index can be set only by a non-insertion operation using a CAS. According to Algorithm 5 this is only possible when h in `handles[`hi`]` is already deleted, but h is not deleted. Otherwise, $hi' \neq hi$ and $hi' \neq \perp$, then the handle index can be set only by an insertion operation using a CAS. According to Algorithm 4 this is only possible when h in `handles[`hi`]` is already deleted, which is not the case. Therefore, there is an entry e that points to k and h and `h.deleted` = `false`, so by Definition 1 h is associated with k. □

Claim 3. Assume handle h is associated with key k at time t in a run r. Then, h is associated with k at time $(t+1)$ in r if and only if the $(t+1)^{st}$ step in r is not the l.p. of a successful remove(k) operation.

Proof. Assume that h is not associated with k at time $(t+1)$.

If there is no handle associated with k at time $t+1$, then by Definition 1 either `h.deleted = true` or the entry's handle index (hi) is $\perp$. In the first case, the only possible step that marks a handle as deleted is the l.p. of a successful remove(k). In the second case, only non-insertion operations turn hi to $\perp$ by using CAS (lines 55 and 67), and according to Algorithm 5 this is only possible when the handle is deleted. However, at time t, h is still associated with k. Therefore, the entry's handle index (hi) is not $\perp$.

Otherwise, there is a different handle $h' \neq h$ that is associated with k at time $t+1$ ($h' \neq \perp$). This change can only be done by an insertion operation using CAS (line 36). According to Algorithm 4 an insertion operation reaches that CAS only if the handle (h) is already deleted (line 21). However, at time t, h is still associated with k, and so there is no different handle that is associated with k.

Therefore, as long as the $(t+1)^{st}$ step is not the l.p. of a successful remove(k), then h is still associated with k at time $t+1$ in r, and there is no handle associated with k at time $t+1$ if the $(t+1)^{st}$ step is a l.p. of a successful remove(k), as required. □

Claim 4. Assume no handle is associated with key k at time t in a run r. Then, no handle is associated with k at time $t+1$ in r if and only if the $(t+1)^{st}$ step in r is not the l.p. of a successful insertion operation of k.

Proof. If no handle is associated at time t, and at time $t+1$ there is an associated handle, then according to Definition 1 either a handle's deleted flag turned from false to true, or the entry's handle index turned from $\perp$ to a valid one. The former is not possible because the handles are initialized as not deleted and only become deleted by a remove; no operation turns a deleted handle to a non-deleted one. In the second case, this can only be done by a successful insertion operation, at its l.p. (line 36), as required. □

Look at the linearization $lin(r)$ of run r using l.p.s defined in Section 3.2. From Claims 3 and 4, by induction on the steps of a run, we get:

Corollary 5. At any point in a concurrent run r, the set of keys associated with handles is exactly the same as the set of inserted keys and not removed keys, associated with the same handles, in $lin(r)$ up to that point.

Claim 6 (Get)**.** In run r, if get(k) returns h then the corresponding get(k) in $lin(r)$ returns h, and if get(k) returns `null` then the corresponding get(k) in $lin(r)$ returns `null`.

Proof. There are three cases for get's l.p.:

1. Get(k) finds a non-deleted handle h (line 6), then get(k) returns h and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of get then the corresponding get(k) in $lin(r)$ returns h as well.

2. LookUp(k) by get(k) (line 3) returns $\perp$ or if get(k) reads that the handle index is $\perp$ (line 4), then there is no handle associated with key k, and get(k) returns `null`. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of get then the corresponding get(k) in $lin(r)$ returns `null` as well.

3. Get(k) finds a deleted handle h at time t_2 (line 6) and returns `null`. Then its l.p. is the later between the read of handle index hi by get(k) at time $t_1 < t_2$ (line 4) and immediately after the set of `deleted` = `true` by remove(k) at some time $t < t_2$. Again there are two cases:

   (a) If $t > t_1$ then the l.p. is immediately after the set of `deleted` = `true` then there is no handle associated with key k, and by Corollary 5, in $lin(r)$ the map does not hold k, and the corresponding get(k) in $lin(r)$ returns `null` as well.

   (b) If $t_1 > t$ then the l.p. is the read of handle index hi by get(k) (line 4) at time t_1, after the set of `deleted` = `true` at time t. We need to show that at no time between t and t_1 the handle index changed to $hi' \neq hi$ and now it does not point to a deleted handle. Notice that only an insertion operation l.p. can change hi to hi'. Assume by contradiction that the l.p. of such an operation occurs between t and t_1. Then when get sees hi at time t_1, it is already hi' and not hi. A contradiction. Hence, at the l.p. of get(k), there is no handle associated with key k, and by Corollary 5, in $lin(r)$ the map does not hold k, so the corresponding get(k) in $lin(r)$ returns `null` as required.

□

Claim 7 (PutIfAbsent)**.** In run r, if putIfAbsent(k) returns `true` then the corresponding putIfAbsent(k) in $lin(r)$ returns `true`, and if putIfAbsent(k) returns `false` then in $lin(r)$ the corresponding putIfAbsent(k) returns `false`.

Proof. If putIfAbsent(k) finds a non-deleted handle h (line 21), then putIfAbsent(k) returns `false` and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of putIfAbsent then the corresponding putIfAbsent(k) in $lin(r)$ returns `false` as well.

Otherwise, if putIfAbsent(k) performs a successful CAS of handle index from $\perp$ (line 36), then putIfAbsent(k) returns `true` and by Definition 1 there was no handle

associated with k just before the CAS. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of putIfAbsent then the corresponding putIfAbsent(k) in $lin(r)$ returns `true` as required. □

Claim 8 (ComputeIfPresent)**.** In run r, if computeIfPresent(k) returns `true` then in $lin(r)$ the corresponding computeIfPresent(k) returns `true`, and if computeIfPresent(k) returns `false` then the corresponding computeIfPresent(k) in $lin(r)$ returns `false`.

Proof. If computeIfPresent(k) finds a non-deleted handle h and there is a successful nested call to handle compute (line 52), then computeIfPresent(k) returns `true` and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ k is inserted and not removed (the map holds k) and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `true` as well.

If lookUp(k) by computeIfPresent(k) returns $\perp$, or if computeIfPresent(k) reads that the handle index is $\perp$ (line 49), then there is no handle associated with key k, and computeIfPresent(k) returns `false`. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `false` as required.

Otherwise, a successful CAS of handle index to $\perp$ is performed by computeIfPresent(k) (line 55), from a handle index pointing to a deleted handle (line 51). Then computeIfPresent(k) returns `false` and by Definition 1 there is no handle associated with k just before the CAS and right after it. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of computeIfPresent then the corresponding computeIfPresent(k) in $lin(r)$ returns `false`. □

Claim 9 (Put)**.** In run r, if put(k) inserts k and returns then in $lin(r)$ the corresponding put(k) inserts k and returns, and if put(k) replaces k's value and returns then in $lin(r)$ the corresponding put(k) replaces k's value and returns.

Proof. If put(k) finds a non-deleted handle h and there is a successful nested call to handle put (line 23), then put(k) replaces k's value and returns, and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ the map holds k and since this is the l.p. of put then the corresponding put(k) in $lin(r)$ replaces k's value and returns as well.

Otherwise, put(k) performs a successful CAS of handle index (line 36) from $\perp$, and inserts k and returns. By Definition 1 there is no handle associated with k just before the CAS, and there is one right after the CAS (the handle is initialized as non-deleted). Since this is the l.p. of put, and by Corollary 5 in $lin(r)$ the map does not hold k before the l.p. and does after. Therefore, the corresponding put(k) in $lin(r)$ inserts k and returns as required. □

Claim 10 (PutIfAbsentComputeIfPresent)**.** In run r, if putIfAbsentComputeIfPresent(k) inserts k and returns then in $lin(r)$ the corresponding putIfAbsentComputeIfPresent(k) inserts k and returns, and if putIfAbsentComputeIfPresent(k) updates k's value and returns then in $lin(r)$ the corresponding putIfAbsentComputeIfPresent(k) updates k's value and returns.

Proof. If putIfAbsentComputeIfPresent(k) performs a successful CAS of handle index (line 36) from $\perp$, then it inserts k and returns. By Definition 1 there is no handle associated with k just before the CAS, and there is one right after the CAS (the handle is initialized as non-deleted). Since this is the l.p. of putIfAbsentComputeIfPresent,

and by Corollary 5 in $lin(r)$ the map does not hold k before the l.p. and does after. Therefore, the corresponding putIfAbsentComputeIfPresent(k) in $lin(r)$ inserts k and returns as required.

Otherwise, putIfAbsentComputeIfPresent(k) finds a non-deleted handle h and there is a successful nested call to handle compute (line 24), then putIfAbsentComputeIfPresent(k) updates k's value and returns, and by Claim 2 h is associated with k. By Corollary 5, in $lin(r)$ the map holds k and since this is the l.p. of putIfAbsentComputeIfPresent then the corresponding putIfAbsentComputeIfPresent(k) in $lin(r)$ updates k's value and returns as well. □

Claim 11 (Remove)**.** In run r, if remove(k) removes k and returns then in $lin(r)$ the corresponding remove(k) removes k and returns, and if remove(k) returns unsuccessfully (without removing any key) then in $lin(r)$ the corresponding remove(k) returns unsuccessfully.

Proof. If remove(k) finds a non-deleted handle h and a successful nested call to handle remove occurs, setting the handle to deleted (line 53), then remove(k) removes k and returns. By Claim 2 h is associated with k before and there is no handle associated with k right after (by Definition 1). Since this is the l.p. of remove, and by Corollary 5 in $lin(r)$ the map does hold k before the l.p. and does not after. Therefore, the corresponding remove(k) in $lin(r)$ removes k and returns as required.

If lookUp(k) by remove(k) returns $\perp$, or if remove(k) reads that the handle index is $\perp$ (line 49), then there is no handle associated with key k, and remove(k) returns unsuccessfully. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of remove then the corresponding remove(k) in $lin(r)$ returns unsuccessfully as required.

Otherwise, a successful CAS of handle index to $\perp$ is performed by remove(k) (line 55), from a handle index pointing to a deleted handle (line 51). Then remove(k) returns and by Definition 1 there is no handle associated with k just before the CAS and right after it. By Corollary 5, in $lin(r)$ the map does not hold k, and since this is the l.p. of remove then the corresponding remove(k) in $lin(r)$ returns unsuccessfully. □

Having shown that all of Oaks operations behave the same way in a run r and its linearization $lin(r)$, we can conclude the following theorem:

Theorem 12. Oak is linearizable with the l.p.s defined in Section 3.2.

Chapter 6

Conclusion

We presented Oak, a scalable concurrent KV-map for big data analytics. Two main requirements guided us when designing Oak. The first is supporting large keys and values. To this end, Oak enforces a zero-copy policy, which allows updates and reads to occur concurrently and atomically on the same memory location. It further supports off-heap allocation (and reclamation) of these keys and values, which is a recent trend in systems like HBase [33] and Druid [18]. The second requirement is supporting an analytics API. In addition to the standard get, put, and remove, Oak provides compute methods for performing an update of the value in-place. Oak also has built-in support for ascending and descending scans.

Our experiments have shown that Oak is faster by 1.3-4.8x than the currently standard concurrent KV-map, the Java ConcurrentSkipListMap. In addition, our results demonstrated that off-heap allocation is beneficial in scenarios with conditional updates of large values.

Bibliography

[1] In-memory analytics market worth 3.85 billion usd by 2022 (retrieved october 2018). https://www.marketsandmarkets.com/PressReleases/in-memory-analytics.asp.

[2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: A practical concurrent self-adjusting search tree. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Maya Arbel and Hagit Attiya. Concurrent updates with rcu: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 196–205, New York, NY, USA, 2014. ACM.

[4] https://www.slideshare.net/cloudera/hbase-hug-presentation, 2011.

[5] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *PPoPP'17*, 2017.

[6] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *Euro-Par*, 2016.

[7] Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *ICDCN'11*, pages 107–118, 2011.

[8] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *SPAA '12*, pages 58–67, 2012.

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[10] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *SIGMOD'18*, Houston, TX, USA, June 2018.

[13] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 4:1–294, January 2012.

[14] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag.

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

[16] Dana Drachsler, Martin Vechev, and Eran Yahav. 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, pages 343–356, New York, NY, USA, 2014. ACM.

[17] (retrieved august 2018). `http://druid.io/`.

[18] (retrieved august 2018). `http://druid.io/docs/latest/operations/performance-faq.html`.

[19] Heap: Sizing and swapping (retrieved september 2018). `https://www.elastic.co/guide/en/elasticsearch/guide/current/heap-sizing.html#compressed_oops`.

[20] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[21] (retrieved august 2018). `https://developer.yahoo.com/flurry/docs/analytics/`.

[22] `http://flurrymobile.tumblr.com/post/169545749110/state-of-mobile-2017-mobile-stagnates`, 2017.

[23] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[24] Vincent Gramoli. 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, pages 1–10, New York, NY, USA, 2015. ACM.

[25] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01*, pages 300–314, 2001.

[26] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[27] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *SIROCCO'07*, 2007.

[28] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[29] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[30] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`, 1993.

[31] `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html`, 1993.

[32] `://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html`, 1993.

[33] Yu Li, Yu Sun, Anoop Sam John, and Ramkrishna S Vasudevan. Offheap read-path in production - the alibaba story. `https://blogs.apache.org/hbase/entry/offheap-read-path-in-production`, March 2017.

[34] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[35] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *PLDI '16*, pages 682–696, 2016.

לאיסוף זבל.

המפה מותאמת עבור שמירה של מפתחות וערכים גדולים ועבור תחזוקה שוטפת של ערכים הקיימים במפה, פעילות השכיחה במקרי שימוש של מערכות ביג דאטה המטפלות בשטף של נתונים. למטרה זו, המפה מסגלת גישת אי־העתקה עבור עדכון של מידע ושליפתו, לדוגמא, המפה מאפשרת פעולות מקביליות של עדכון של ערך במקום.

ממשק התכנות של המפה דומה לממשק של המפה הסטנדרטית (שניתן לנווט בה) המקבילית של ג'אווה, עם התאמות למימוש יעיל של גישת האי־העתקה. המפה מספקת פעולות של קריאה וכתיבה עם סמנטיקות חזקות (אטומיות), כמו גם פעולות אטומיות נוספות המשלבות קריאה וכתיבה כמו עדכון של ערך במקום או הכנסת מפתח חדש רק אם המפתח נעדר מהמפה. בנוסף, המפה מספקת יכולות ניווט בצורת איטרטורים עולים ויורדים (לא אטומיים).

כמו כן, בעבודה זו אנו נותנים הוכחה לנכונות של המפה, ע"י מציאת נקודות לינאריזציה עבור כל אחת מהפעולות של המפה, כך שנראה כי הפעולות המקביליות קורות בסדר הנקבע לפי נקודות הלינאריזציה שלהן. בנוסף לכך, אנו מביאים תוצאות של ניסויים שערכנו שמראים כי המפה מהירה פי 1.3־4.8X מהמפה המקבילית הסטנדרטית היום, המפה המקבילית של ג'אווה. יתרה מזאת, התוצאות מראות כי שימוש בזיכרון שאינו מנוהל אכן יעיל יותר במקרי שימוש שבהם מתבצעים עדכונים מותנים של ערכים גדולים.

השותפים שלנו מהתעשייה לעבודה זו ממזגים את המפה לתוך פלטפורמה פופולרית של אנליטיקות של ביג דאטה בזמן אמת. המיזוג הזה לא נכלל בתזה זו.

תקציר

בעקבות ירידת מחירים מתמדת של זיכרון גישה אקראית, השוק של אנליטיקות של ביג דאטה (נתוני עתק) המתבצעות בזיכרון נמצא בעלייה. אחסון של כמויות הולכות וגדלות של מידע בזיכרון הראשי מאפשרות פרדיגמות חדשות של עיבוד מידע, למשל, אנליטיקות מתקדמות בזמן אמת על שטף בקצב גבוה של אירועים. מערכות מודרניות שתומכות בקבלת החלטות מטפלות כל הזמן במידע שמגיע תוך כדי שהן מספקות ניתוחים עם עיכוב מינימלי.

מערכות המספקות אנליטיקות מעל הזיכרון לעיתים קרובות מממשות סידור מורכב של הנתונים ושאילתות מורכבות מעל אבסטרקציית אחסון של מפה פשוטה ודינאמית של מפתחות וערכים. מפה הינה אוסף סדור של זוגות של מפתחות וערכים המספקת פעולות פשוטות של קריאה (גישה) רנדומית, כתיבה (הכנסה) רנדומית, ושאילתת טווח (סריקה). קיימים הרבה מצבים בהם המפתחות והערכים הם קומפוזיציה של מידע ברמת האפליקציה, לכן המפתחות והערכים יכולים להיות גדולים (מאות עד אלפי בתים).

סקילבילית של מימושים של המפה על חומרה בעלת יחידות חישוב רבות הינה קריטית לביצועים של כלל המערכת. מערכות של אנליטיקות משיגות טיפול מהיר במידע שמגיע ומספקות ניתוחים מהירים באמצעות ביצוע מקבילי של קריאה וכתיבה של המידע ע"י מספר חוטים. כמו כן, השגת עלייה בביצועים עם הגדלת כמות זיכרון הגישה האקראית חשוב באותה המידה.

מימושים קיימים של מפות בזיכרון, במיוחד מימושים בשפות שבהן הזיכרון מנוהל כדוגמת ג'אווה, לא מתאימים לעבודה עם כמויות הולכות וגדלות של זיכרון גישה אקראית. למרות ההישגים האחרונים בתחום, אלגוריתמים של איסוף אשפה מתקשים להתמודד עם כמויות הזיכרון בפלטפורמות של ביג דאטה. מנגנוני איסוף האשפה הקיימים מתמודדים עם זיכרון של כעשרות בודדות של ג'יגה. המגבלות הללו מובילות את מפתחי פלטפורמות ביג דאטה לחשוב על פתרונות לעניין זה ע"י שימוש בהקצאת זיכרון לא מנוהל.

אנו עוסקים בביקוש למפות מקבילות גדולות בזיכרון עבור פלטפורמות של אנליטיקות, ושמים את הקצאת הזיכרון בראש סדר העדיפויות. בעבודה זו אנו מציגים מפה של מפתחות וערכים שהיא סקילבילית ומקבילית המתאימה למערכות המבצעות אנליטיקות של ביג דאטה בזמן אמת. המפה מאחסנת את המפתחות והערכים בזיכרון שאינו מנוהל, ולא בזיכרון ששפות כדוגמת ג'אווה מציעות, ובזאת מצמצת את עלויות השימוש במנגנון

המחקר בוצע בהנחייתה של פרופסור עדית קידר, בפקולטה להנדסת חשמל.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

אחסון מפתחות בזיכרון לא מנוהל עבור אנליטיקות של ביג דאטה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת חשמל

הגר מאיר

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
טבת התשע"ט חיפה דצמבר 2018

אחסון מפתחות בזיכרון לא מנוהל עבור אנליטיקות של ביג דאטה

הגר מאיר