Oak:

a Scalable Off-Heap Allocated Key-Value Map

Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, Yoav Zuriel

Yahoo Research

verizon media yahoo! research

Results Replicated acm Artifacts Available acm Artifacts Evaluated Functional acm

OAK (Off-heap Allocated Keys)

Concurrent In-memory Off-heap Key-Value Map for Big Data:

- Written in Java, but causes no JVM Garbage Collection (GC) activity
  - more performance
  - less memory

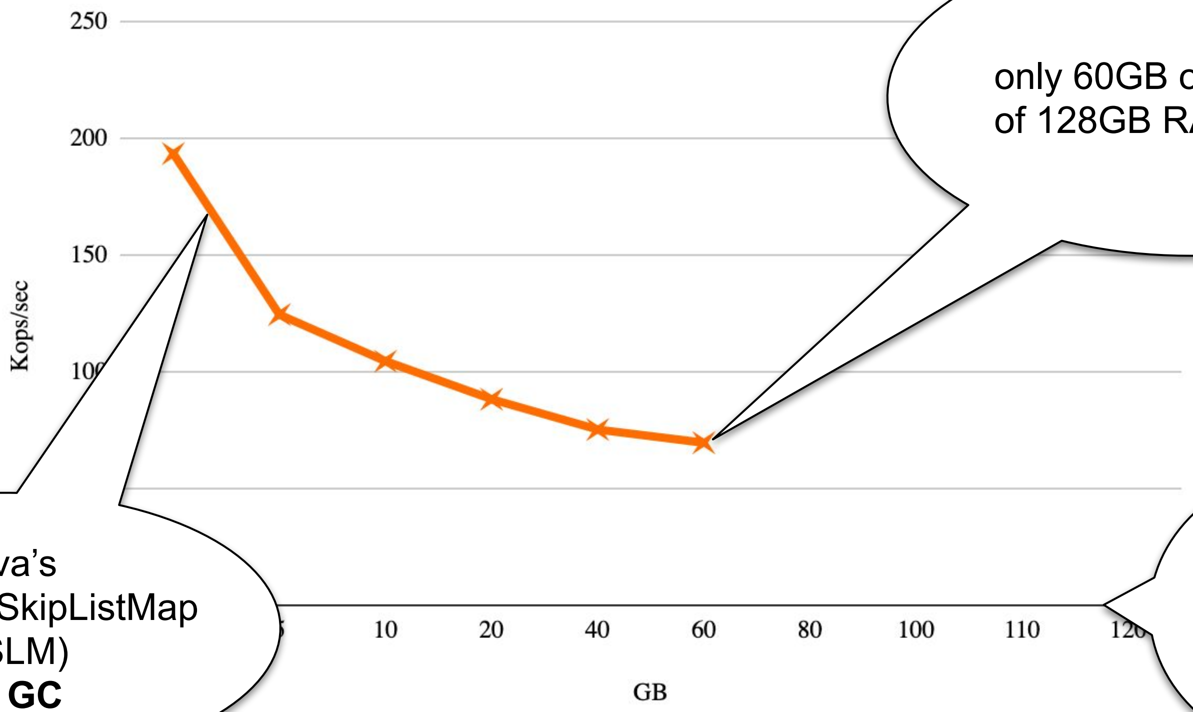
https://github.com/yahoo/Oak

Open Source Library

Big Data goes Off-Heap

How can you use 128GB RAM?

1 thread, Throughput



only 60GB of data, out of 128GB RAM

Java’s ConcurrentSkipListMap (CSLM)
G1 GC

Push data (1KB quantas) till OutOfMemory

Where had the resources been gone?

1. Internal GC structures requires memory
2. Object headers (needed for memory management) require memory
3. GC algorithms takes CPU cycles

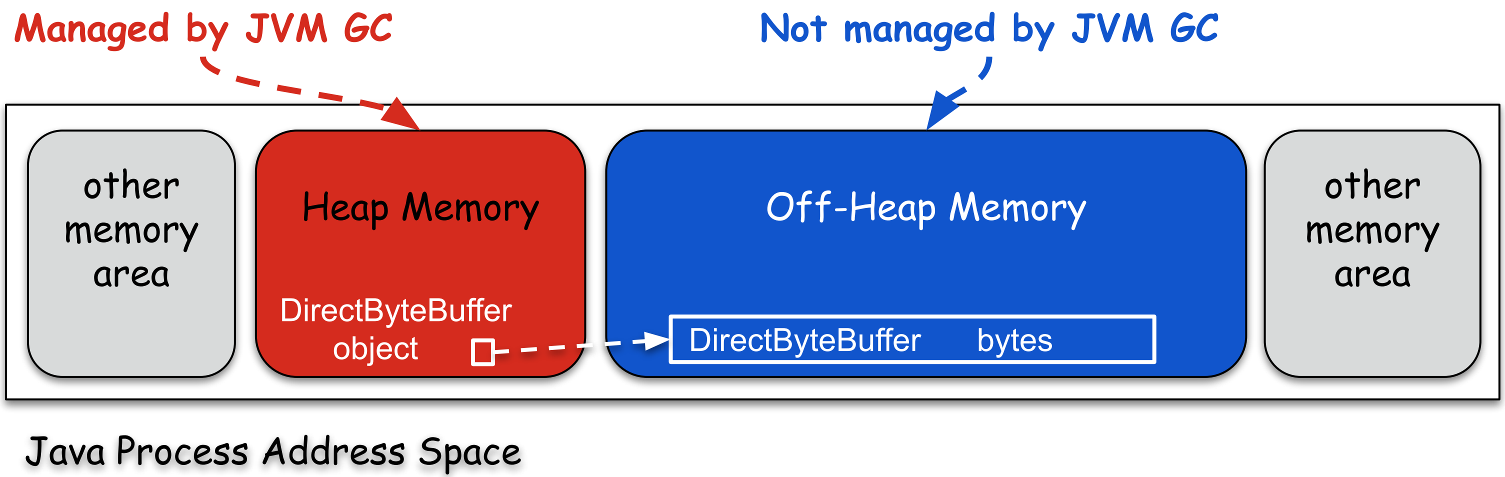
BUT…

Life is easier with managed memory language!

verizon media

Background

Quickly about Off-Heap Memory



Off-Heap Memory Pros and Challenges

+ No JVM GC costs

+ No object headers

+ Quicker access

- How to reuse memory?

- Need to (de)serialize

- How to access it concurrently?

OAK

Lean in memory

Open source library

Fast performance

Real world deploy

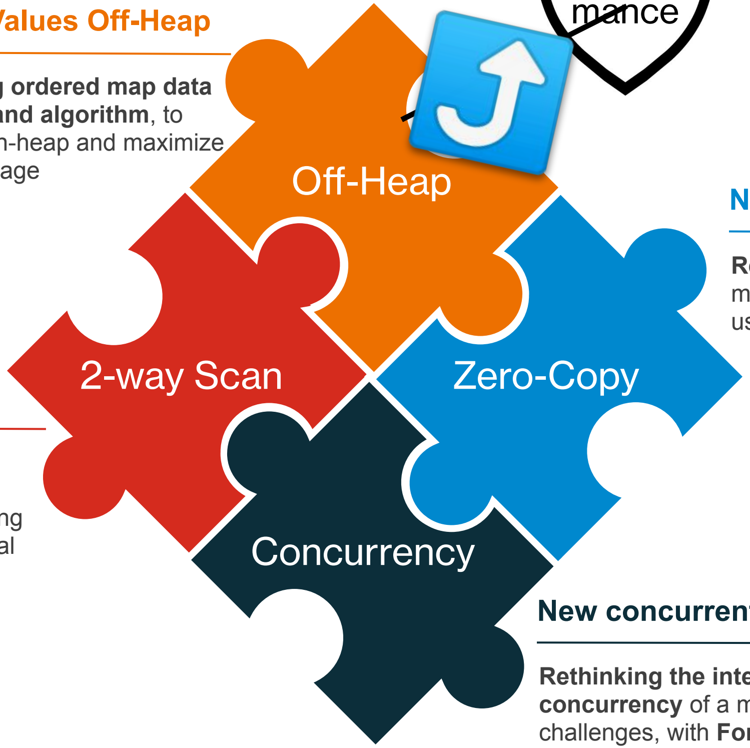
Scales better

Keys & Values Off-Heap

Rethinking ordered map data structure and algorithm, to minimize on-heap and maximize off-heap usage

Fast 2-way Scans

Rethinking how an ordered map can be traversed backward, without complicating concurrency or using additional memory



New Zero-Copy API

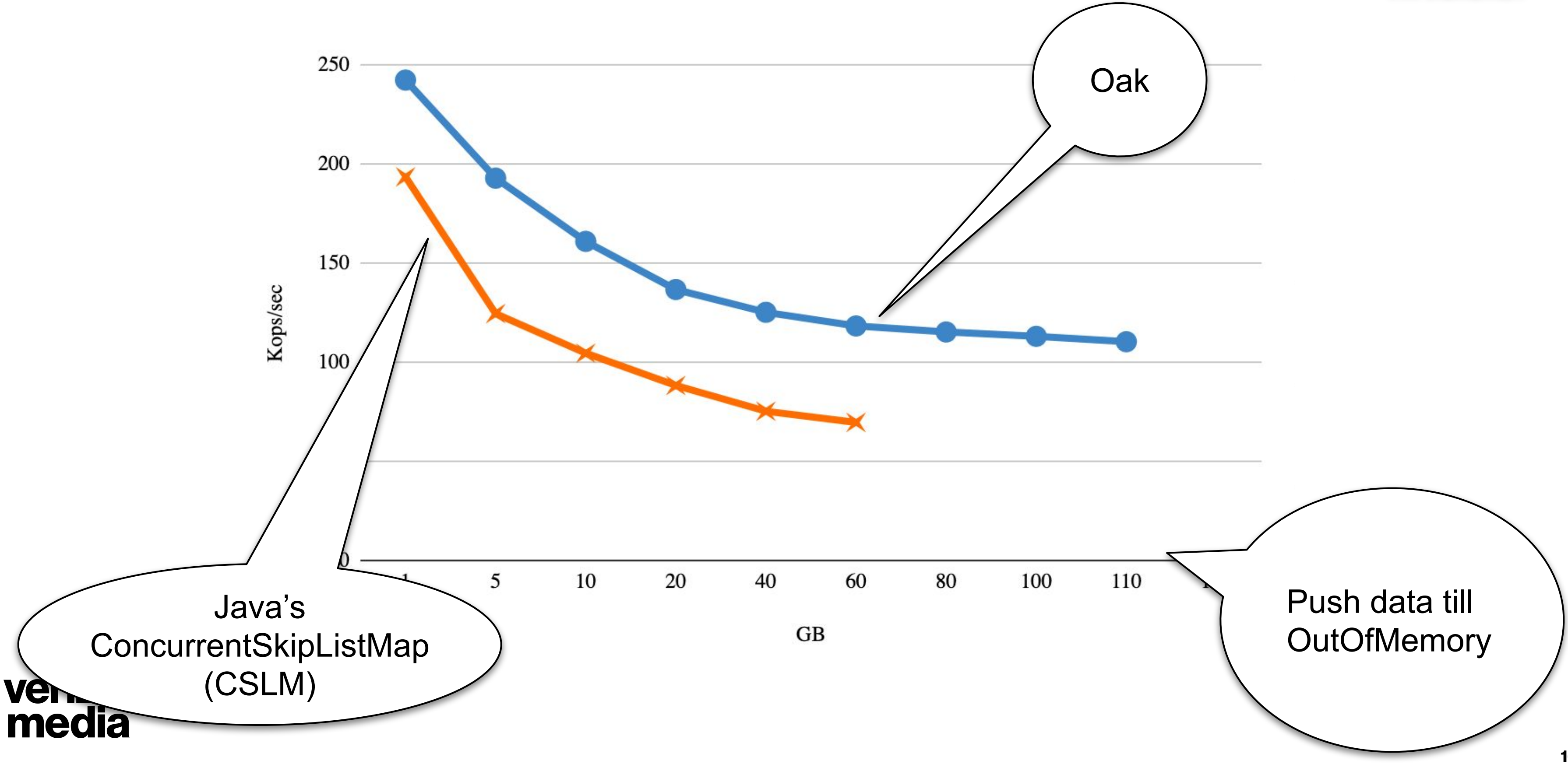
Rethinking ordered map API, t
minimize deserialization and giv
user direct memory access

New concurrent algorithm

Rethinking the internal concurrency of a map to suit new challenges, with **Formal Proof!**

verizon✓
media

How can you use 128GB RAM?



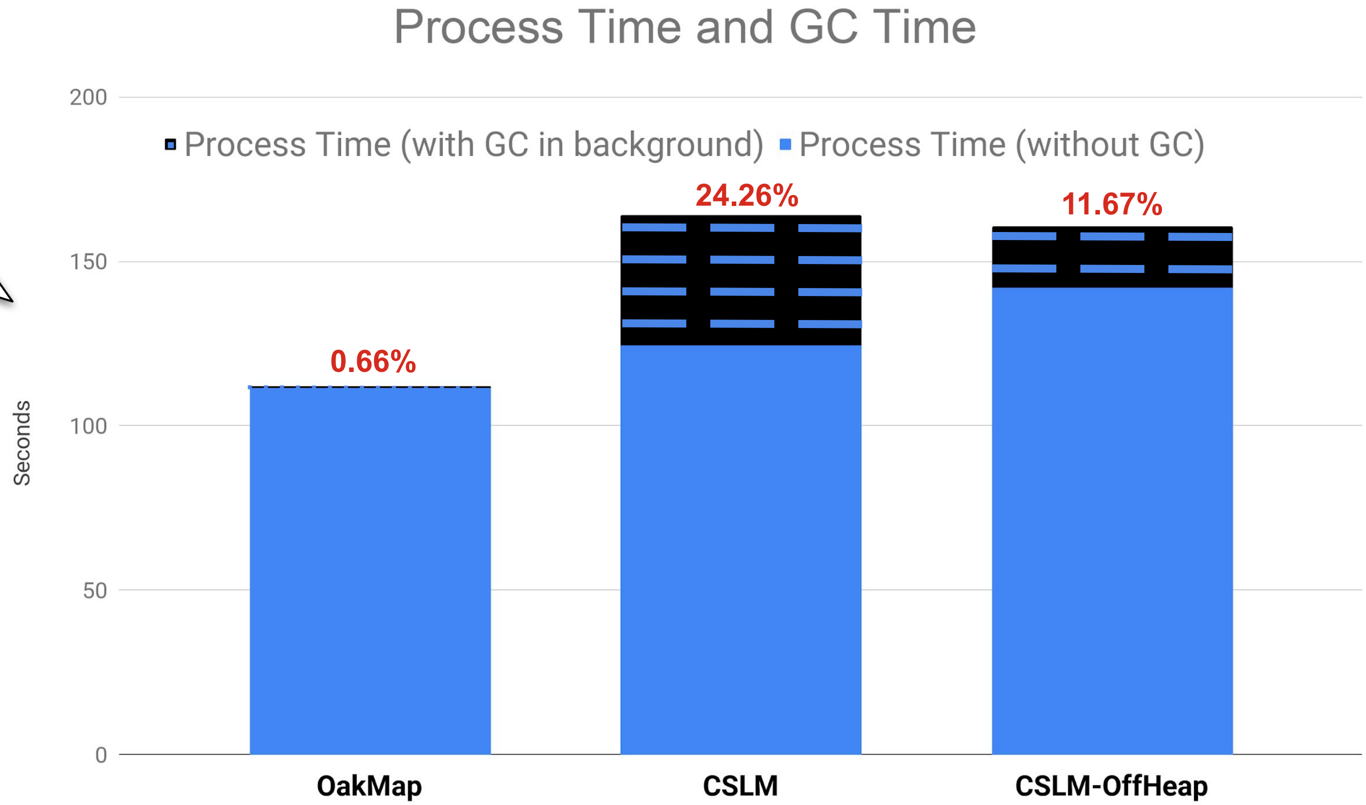
Oak

Java’s ConcurrentSkipListMap (CSLM)

Push data till OutOfMemory

How much time is spent in GC?

1 thread,
total 11GB
insertion
time



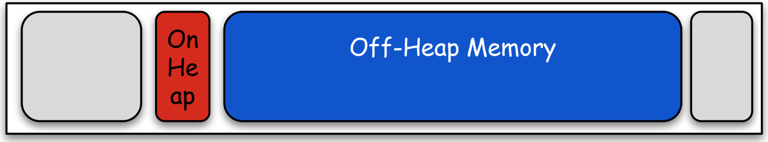
verizon media

Data Organization

verizon media

Big Data Map Design Approach

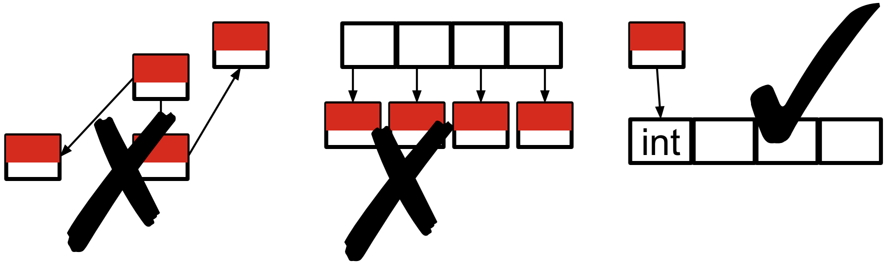
- As less metadata as possible.



verizon✓ media

Big Data Map Design Approach

- As less metadata as possible.
- Java objects and their headers are not efficient for holding data. Better primitives array



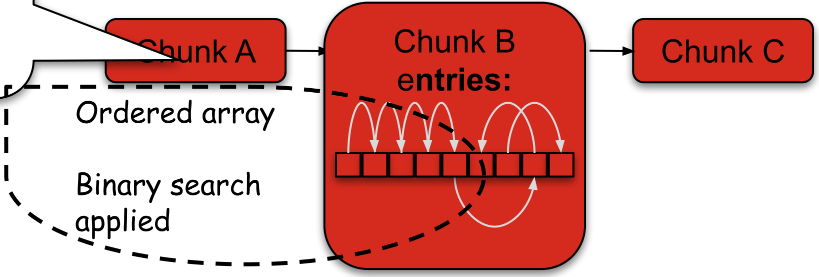
Big Data Map Design Approach

- **As less metadata as possible.**
- **Java objects and their headers are not efficient. Better primitives array**
- **Maintenance in batches:**
    - preallocate off-heap
    - manage on-heap in chunks of memory
- **Big values: write and copy on demand only**
- **Let the user access the raw data, but be on guard**
- **For Big Data traverses avoid ephemeral objects if possible, but mind NUMA architecture**

Oak's Data Organization

Chunks are contiguous, ordered key ranges

Keys are variable size, unique & immutable

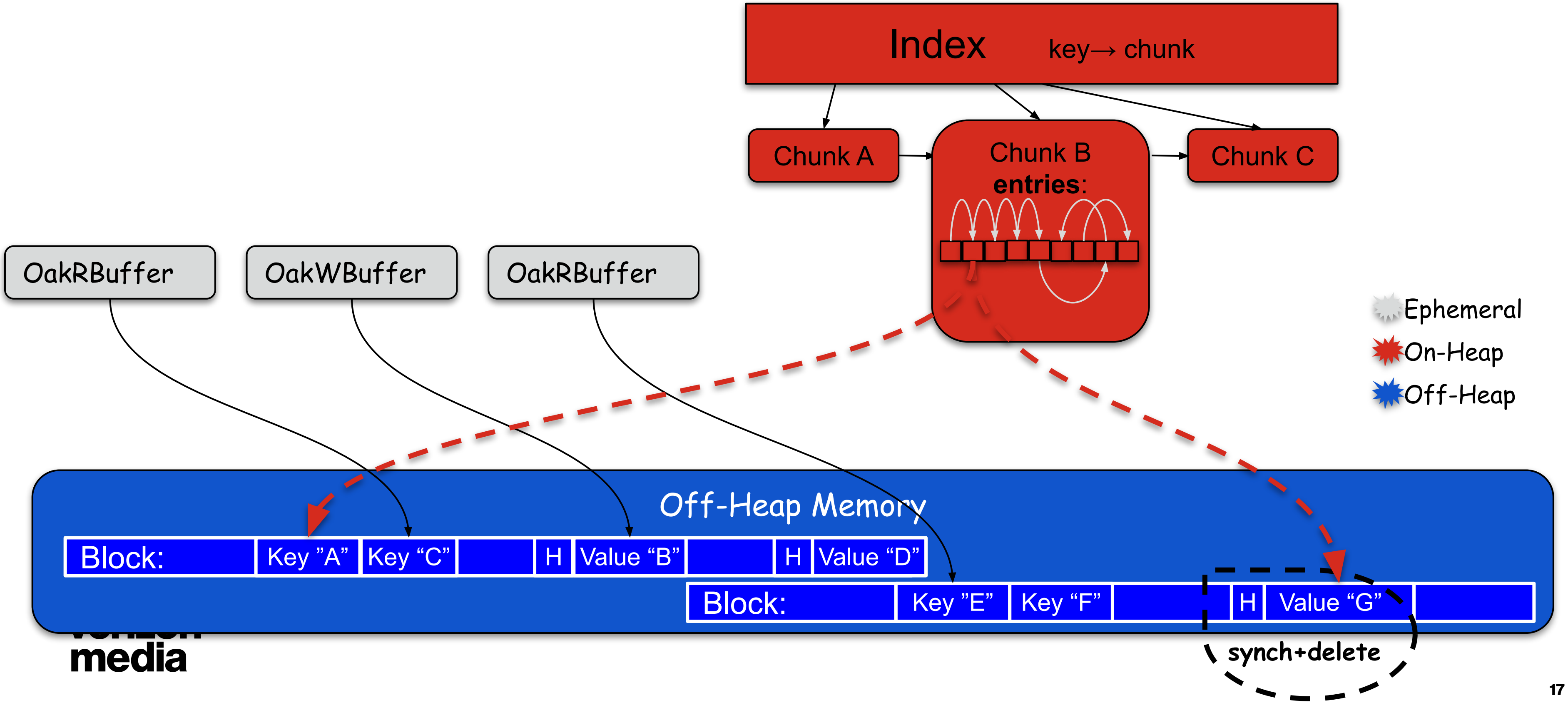


✸ Ephemeral

✸ On-Heap

✸ Off-Heap

Oak's Data Organization



Time for your questions!

verizon media

New API

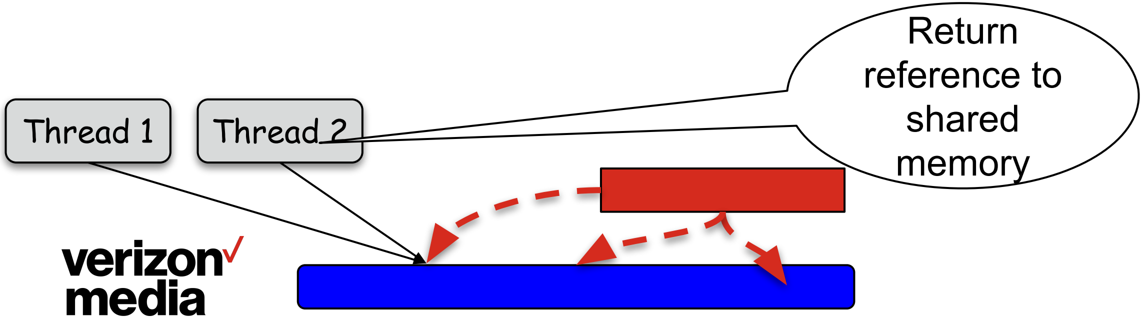
Zero-Copy API: OakMap<K,V>

ZeroCopyConcurrentNavigableMap

OakRBuffer get(K)

(Legacy) ConcurrentNavigableMap

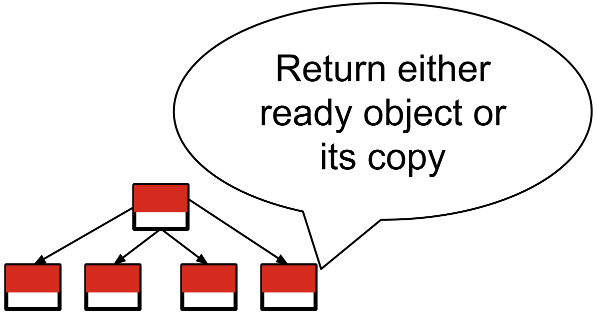
V get(K)



Thread 1

Thread 2

Return reference to shared memory



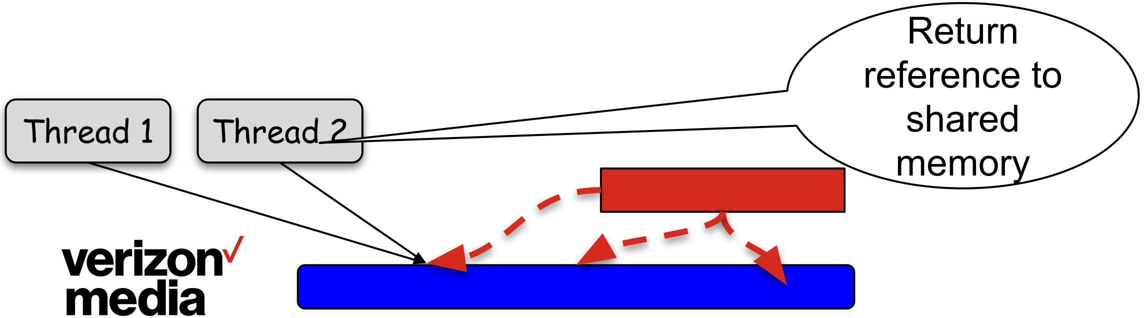
Return either ready object or its copy

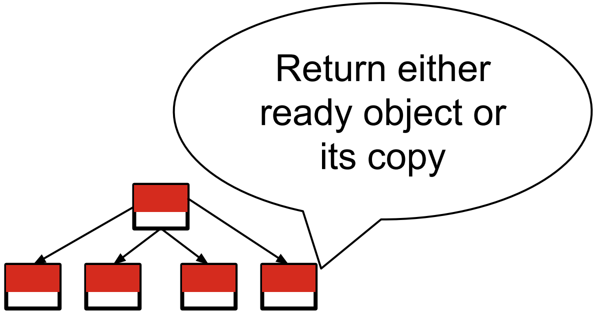
verizon✓ media

Zero-Copy API: OakMap<K,V>

ZeroCopyConcurrentNavigableMap	**(Legacy) ConcurrentNavigableMap**
OakRBuffer get(K)	V get(K)
Set⟨OakRBuffer⟩ keySet() / **keyStreamSet**()	Set⟨K⟩ keySet()

boolean **putIfAbsentComputeIfPresent**(K, V, CreateFunction(OakWBuffer), ComputeFunction(OakWBuffer))

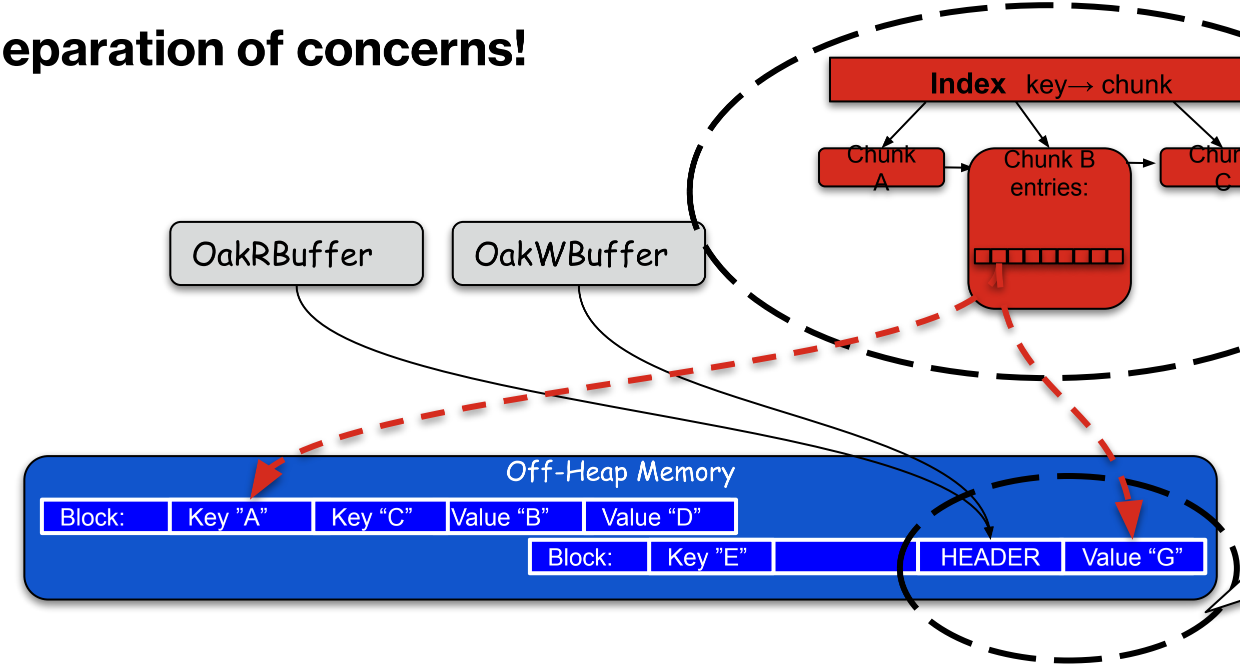




Concurrency

Oak Concurrency

Separation of concerns!

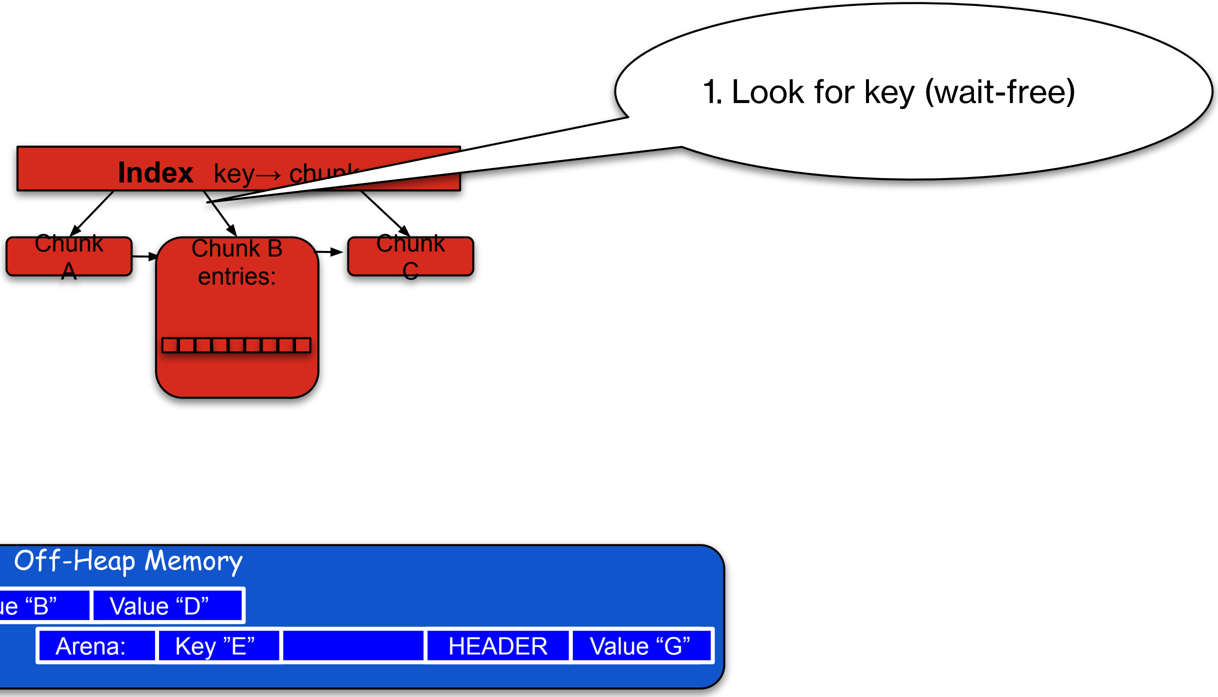


Data Structure <u>Internal</u>:
Lock-Free

User Level <u>External</u>:
Pluggable, example lock-based

Oak Concurrency

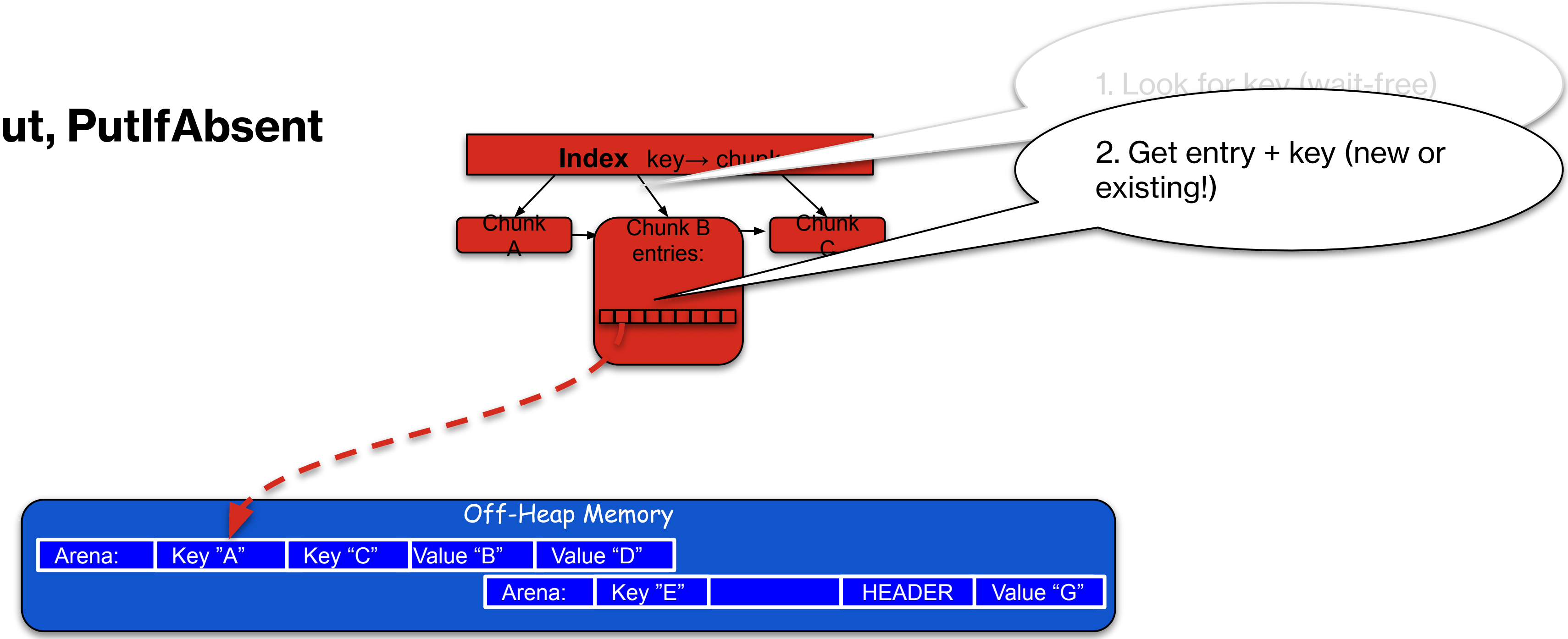
Put, PutIfAbsent



verizon✓ media

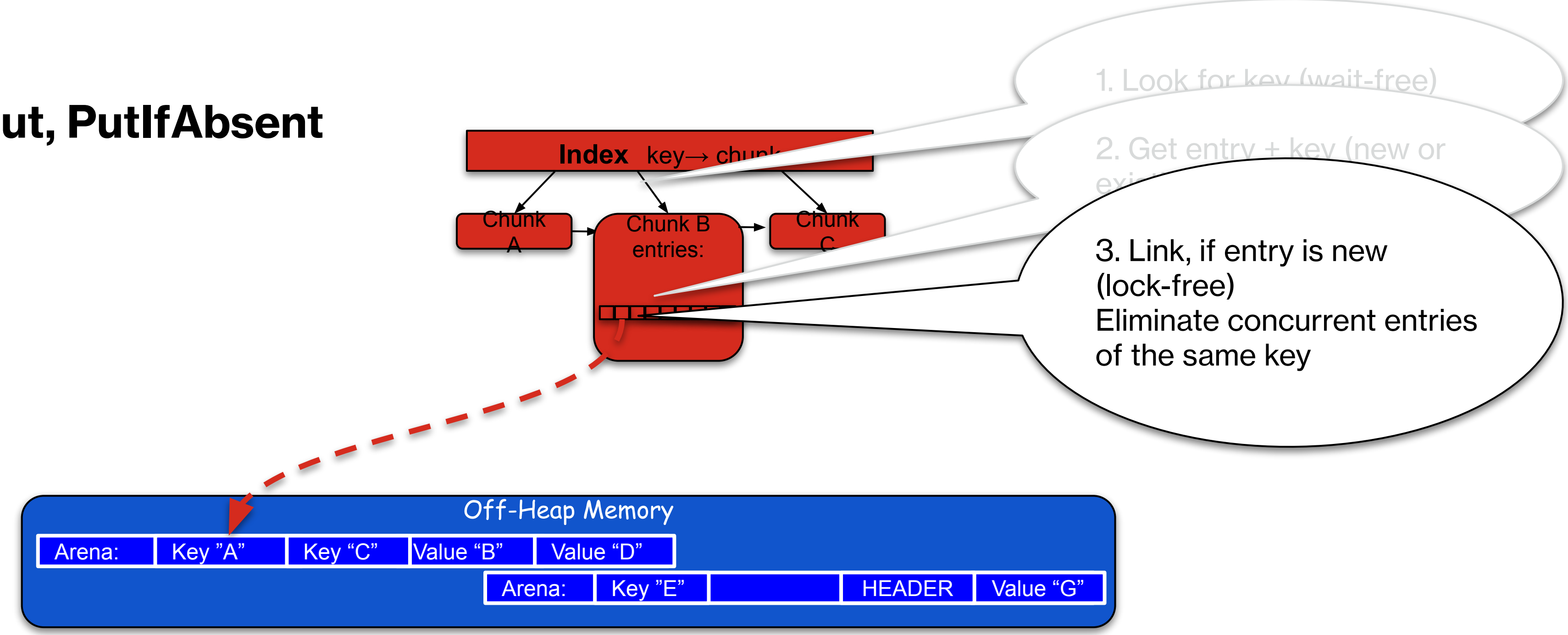
Oak Concurrency

Put, PutIfAbsent



Oak Concurrency

Put, PutIfAbsent



1. Look for key (wait-free)

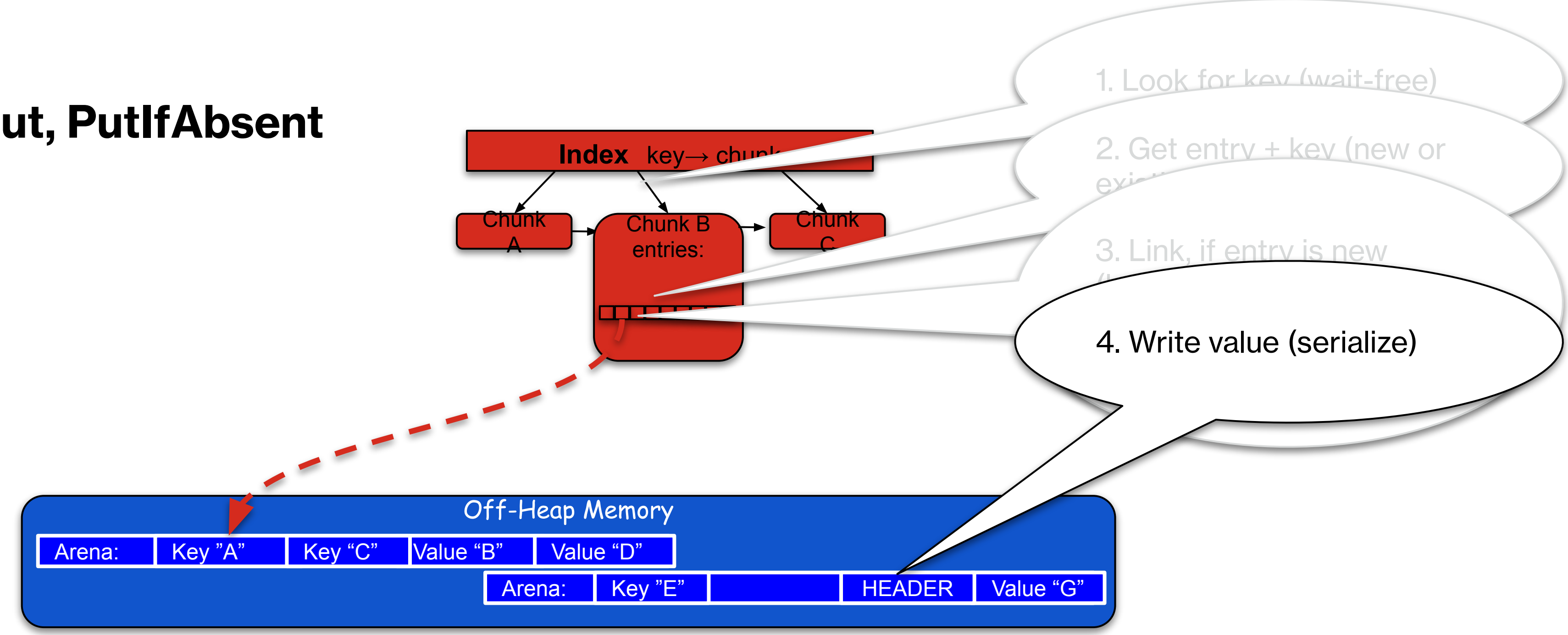
2. Get entry + key (new or exi

3. Link, if entry is new (lock-free)
Eliminate concurrent entries of the same key

verizon media

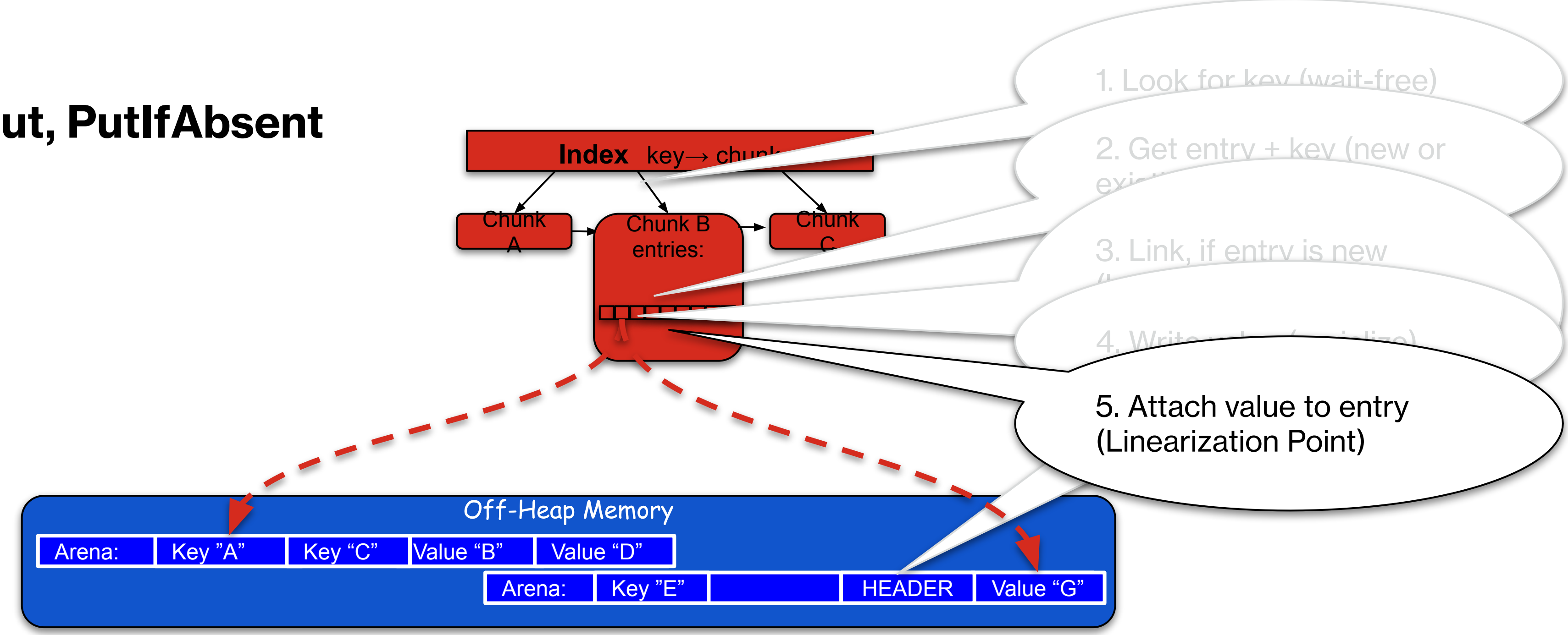
Oak Concurrency

Put, PutIfAbsent



Oak Concurrency

Put, PutIfAbsent



1. Look for key (wait-free)

2. Get entry + key (new or exi

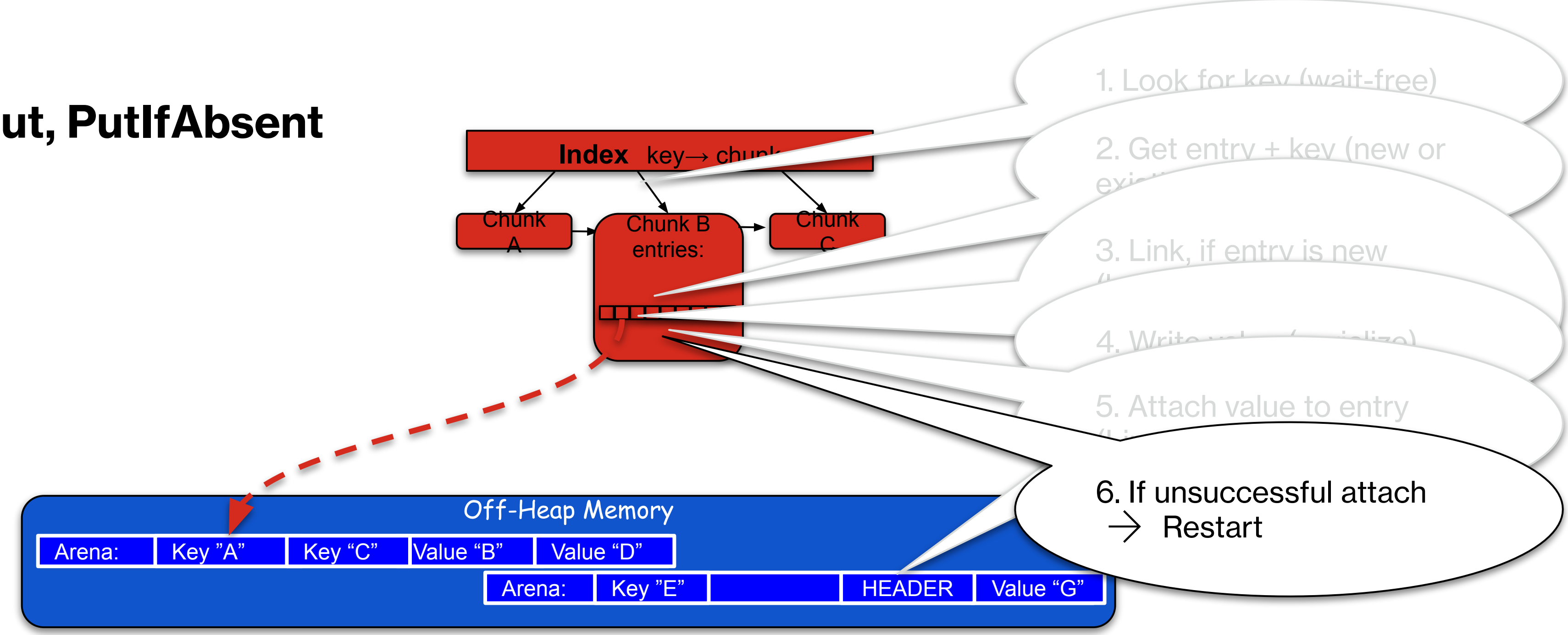
3. Link, if entry is new

4. Writ ialize)

5. Attach value to entry (Linearization Point)

Oak Concurrency

Put, PutIfAbsent



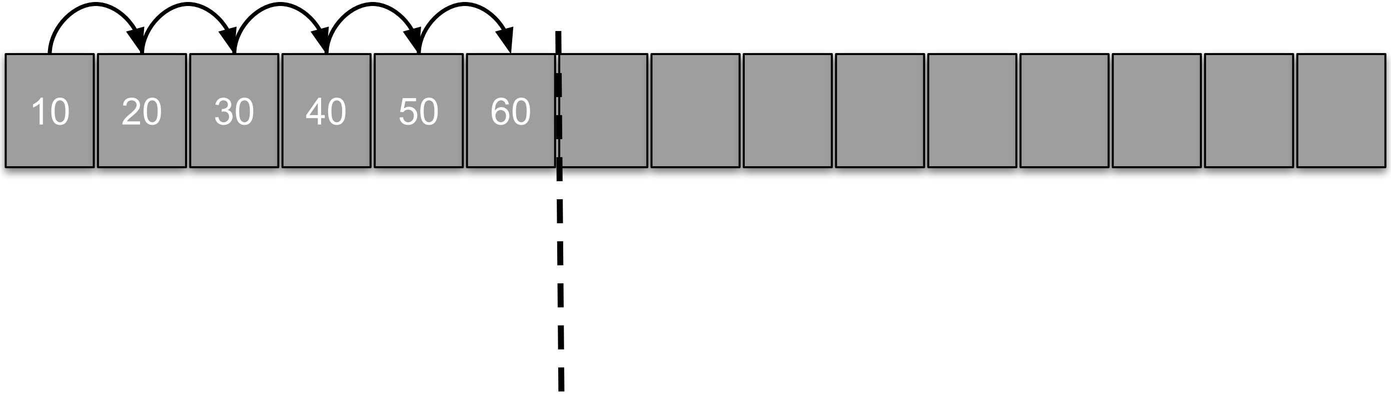
verizon media

Backward Scans

- For analytics requiring to present the results in the decreasing order

Scans (Backward)

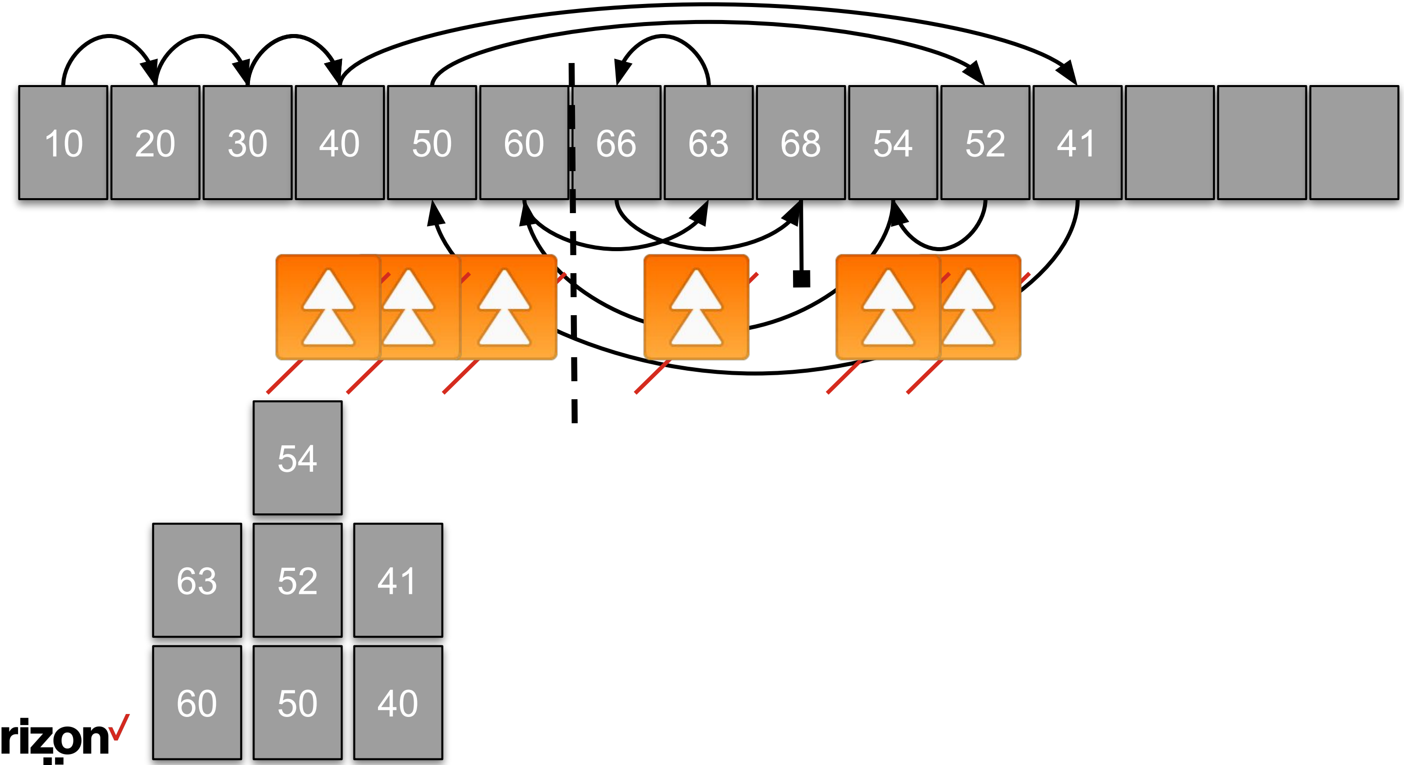
ZeroCopy API
Concurrency
☞ Backward Scans
Working with Off-Heap
Evaluation



verizon✓
media

Scans (Backward from 63)

ZeroCopy API
Concurrency
☞ Backward Scans
Working with Off-Heap
Evaluation



verizon✓
media

Working with off-heap

Off-heap Usage Commons

Creation

```
ByteBuffer block =
        ByteBuffer.allocateDirect
                           (this.capacity);
```

JVM GC Management

when **block** object is released by JVM GC, the OS memory is also released

Cost

frequent allocation and deallocation of DirectByteBuffers requires ***3 times more*** memory compared to ad-hoc management

Off-heap memory is usually used for

- **immutable data**
- **allocated once and released by the end of the program**

Off-heap Usage Ad-hoc

Block

ByteBuffer **block** =

ByteBuffer.*allocateDirect*(**~256MB**);

Block Pool

blocks are allocated for OakMap instance lifetime, then reused via pool for other OakMaps

Slice

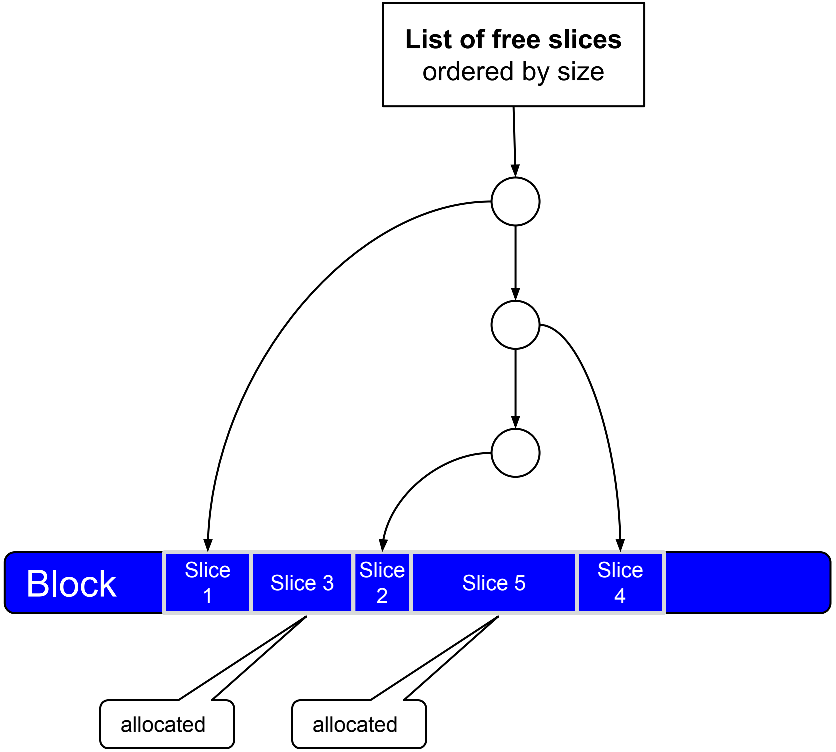
small part of big **block** defined by *reference*:

<BlockID, offset(in block), length>

- *Extra Tip:*
- *don't use ByteBuffer#duplicate() and ByteBuffer#slice(),*
- *do use only **absolute access** on the main big ByteBuffer - block (recall we do not want many ephemeral objects floating around)*

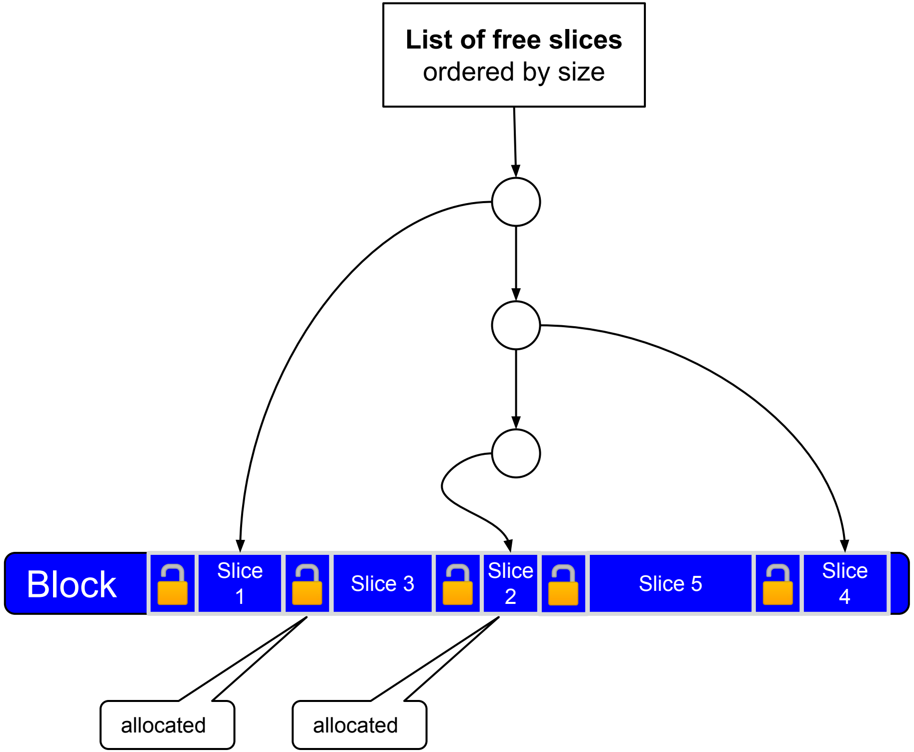
Off-heap Reuse Possibilities

- **Sometimes off-heap memory is never reused**
- **Otherwise...**
- **If there is <u>no concurrency</u>, add deleted slices to the free-list and use it for new allocations**
  - either look for suitable slice size, or merge nearby slices to get bigger allocation possibilities
  - no concurrency -- easy life! :)



Off-heap Concurrent Reuse

- **Finally, for concurrency you may use locks**
  - each slice protected by a lock for access/delete
  - memory used for locks isn't reused (!)
  - slice is deleted under lock, thus all belated threads see deleted slice and release the lock
  - off-heap based locks (are explained next)
- **OR wait for our next paper and Oak release :)**



Off-heap Modifications

01 **WRITES**

```
DirectBuffer buff = ByteBuffer.allocateDirect(capacity);
// use ByteBuffer absolute put instructions
buff.putInt/Long(int index, int/long value);
```

02 **CAS**

```
unsafe.compareAndSwapLong
(null,   buff.address() + buff.position(),      expectedValue, newValue);
```

03 **JDK11**

```
String[] sa = ...
VarHandle avh =
MethodHandles.arrayElementVarHandle(String[].class);
boolean r = avh.compareAndSet(sa, 10, "expected", "new");
```

Time for your questions!

verizon
media

Evaluation

Machine

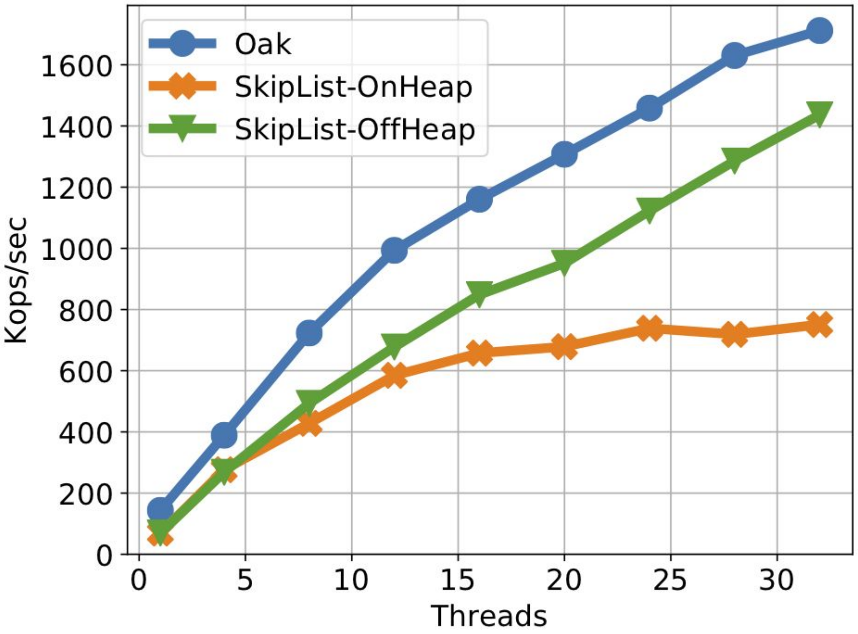
- AWS instance m5d.16xlarge
- utilizing 32 cores (with hyper-threading disabled)
- on two NUMA nodes

Experiment Parameters

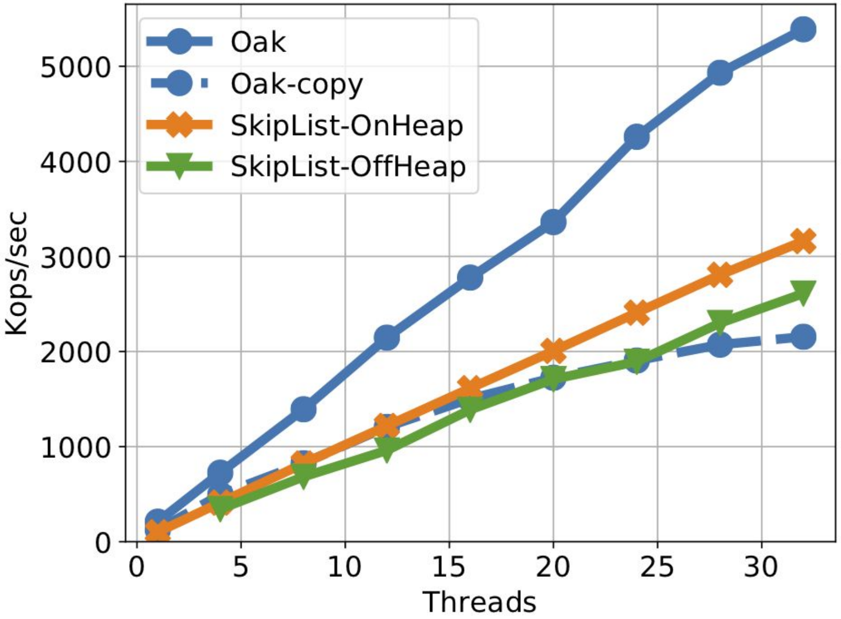
- Keys size 100B
- Value size 1KB
- Limit to 32GB (Inserting 12GB raw data)

Scaling with Parallelism (11M KV-pairs)

Put

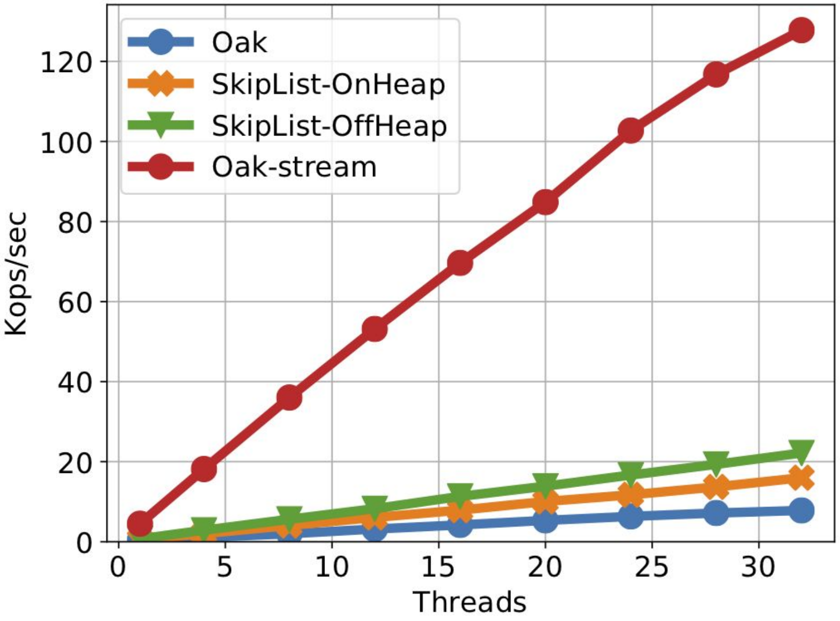


Get

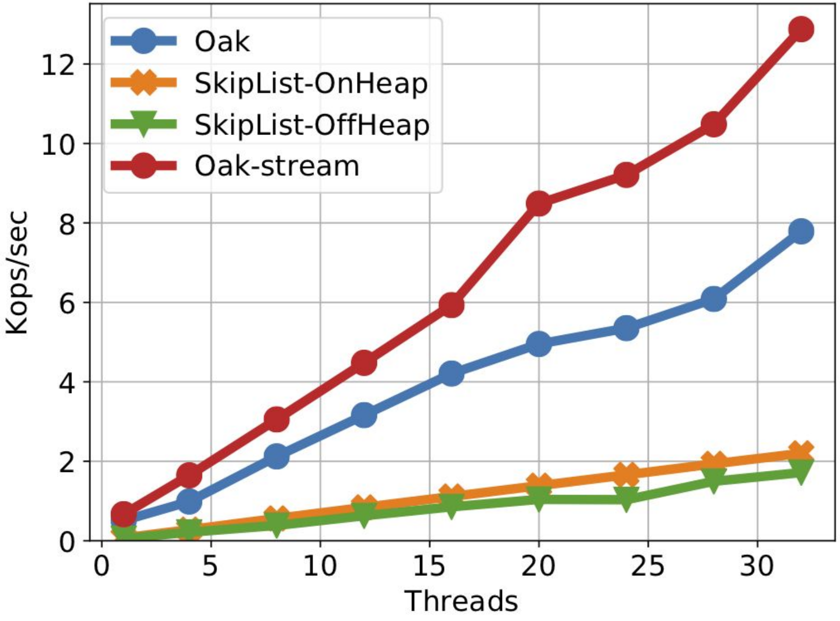


Scaling with Parallelism (11M KV-pairs)

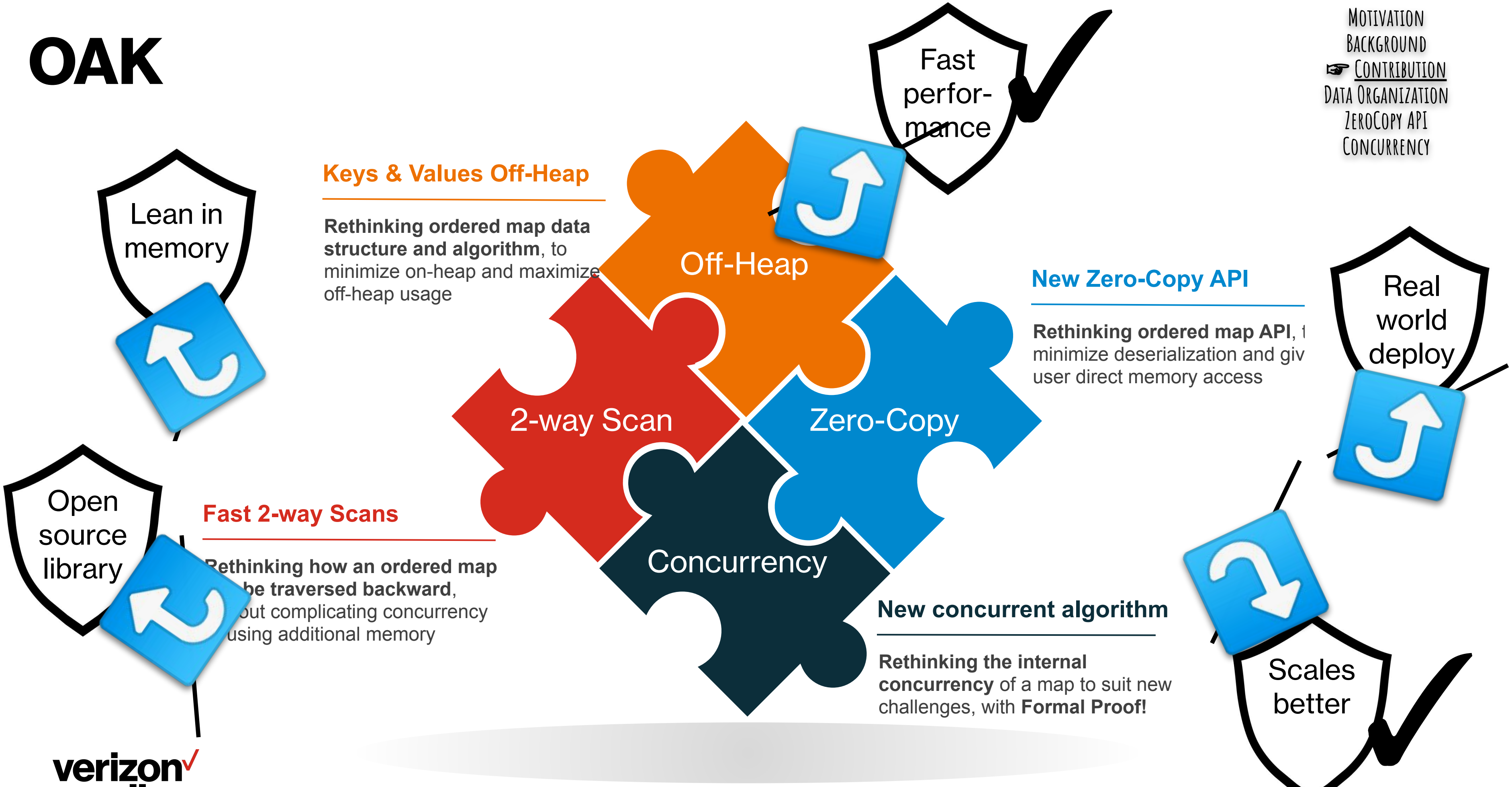
Ascending scan , 10K pairs/scan



Descending scan, 10K pairs/scan



OAK



Keys & Values Off-Heap

Rethinking ordered map data structure and algorithm, to minimize on-heap and maximize off-heap usage

New Zero-Copy API

Rethinking ordered map API, t minimize deserialization and giv user direct memory access

Fast 2-way Scans

Rethinking how an ordered map be traversed backward, out complicating concurrency using additional memory

New concurrent algorithm

Rethinking the internal concurrency of a map to suit new challenges, with **Formal Proof!**

Oak in Apache Druid

a popular open-source real-time analytics database

Re-implement Druid's centerpiece Incremental Index (I^2) component around Oak

OakIncrementalIndex

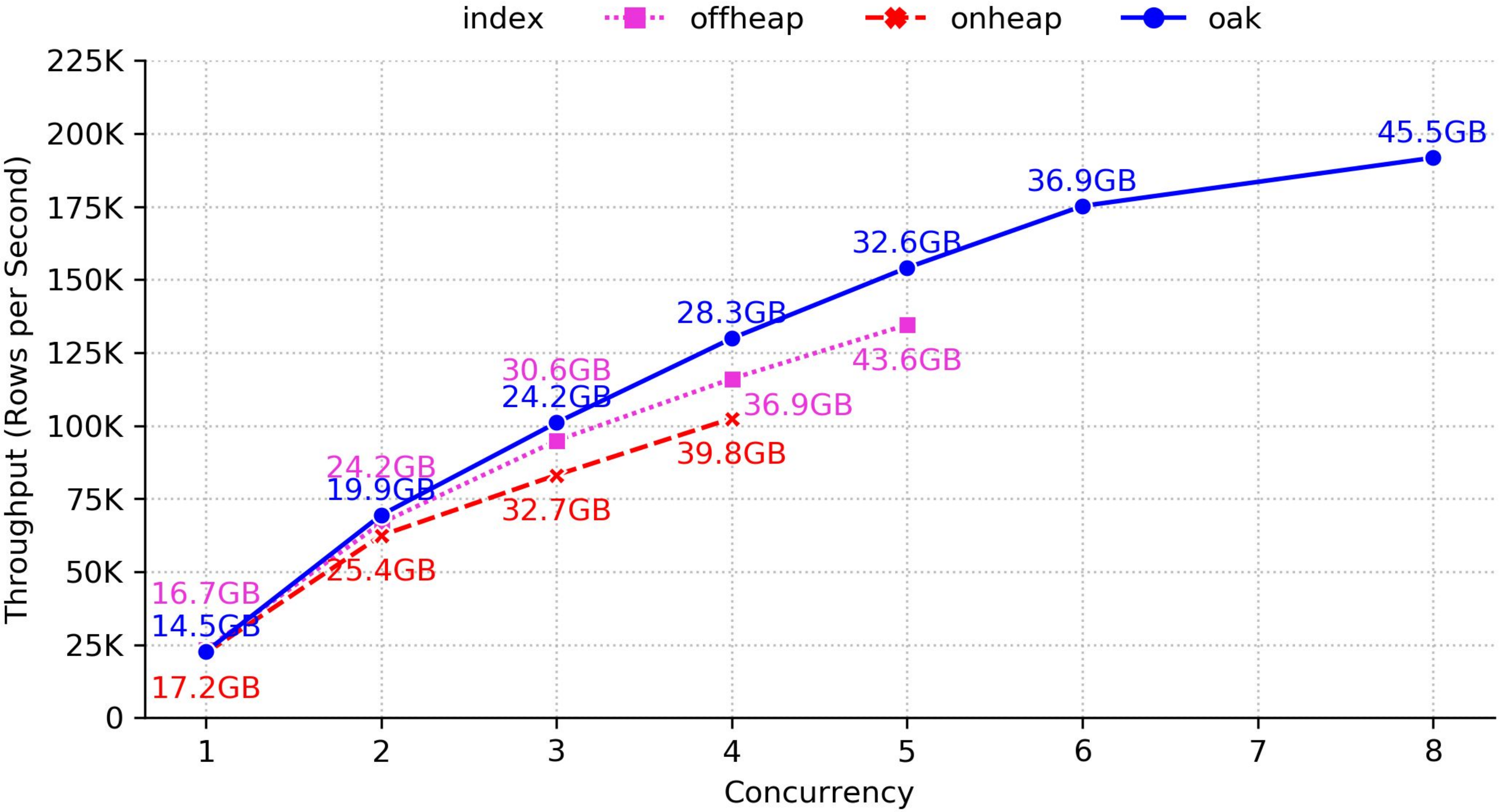
Decreasing memory consumption

Faster Ingestions

Experimental Setup

- **We compare (1) OakMap-based IncrementalIndex (OakI^2) with the legacy Druid implemented CSLM-based index**
  - **(2) both the keys and the values are (on-heap) Java objects (the default)**
  - **(3) the keys are Java objects whereas the values are stored in (individual) off-heap ByteBuffers.**

- **The hardware testbed is**
  - 12-core (24-hyperthread) Intel server (E5-2620 v2 @ 2.10GHz)
  - with 46GB of RAM and SSD storage
  - Runtime OS is RedHat 6 with Java 8 (build 1.8.0_241-b07).


index
offheap
onheap
oak
Throughput (Rows per Second)
0
25K
50K
75K
100K
125K
150K
175K
200K
225K
Concurrency
1
2
3
4
5
6
7
8
16.7GB
14.5GB
17.2GB
24.2GB
19.9GB
25.4GB
30.6GB
24.2GB
32.7GB
28.3GB
36.9GB
39.8GB
32.6GB
43.6GB
36.9GB
45.5GB

media

I^2-Oak

I^2 implementation on top of OakMap

Configurable at system level (the legacy I^2 is still a default).

Minor refactoring of the Druid code (I^2 API abstraction).

Implemented as core part of Druid but could be an extension to reduce friction.

Details

Druid I^2 schema mapped to OakMap keys and values.

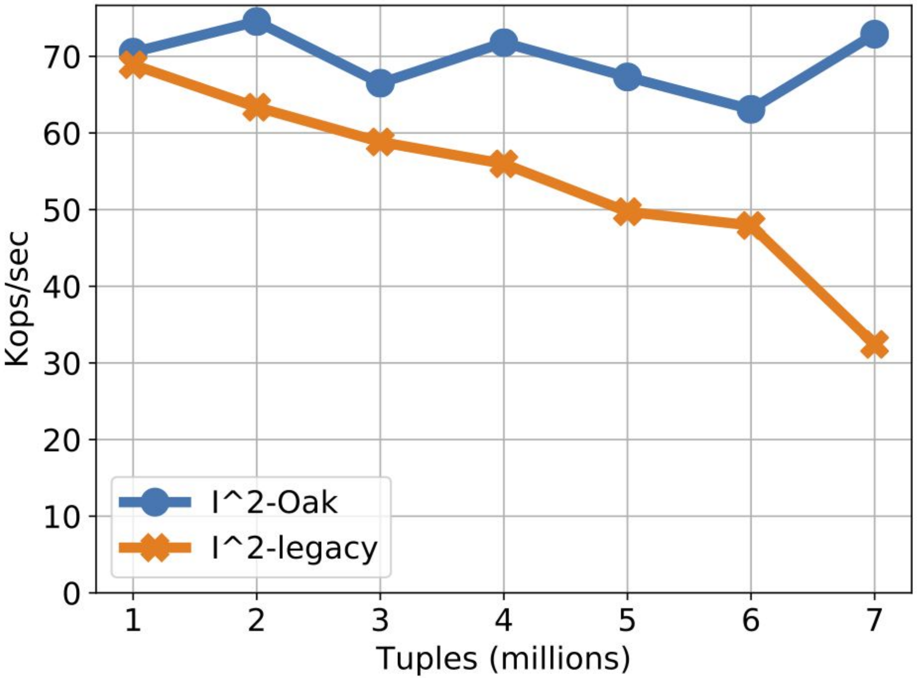
Leverages the ZC API for queries and in-place aggregation.

Project Status

Code complete. Component- and system-level benchmarks.

Community: Git issue, PR.

Druid Ingestion - Scaling with Data Size



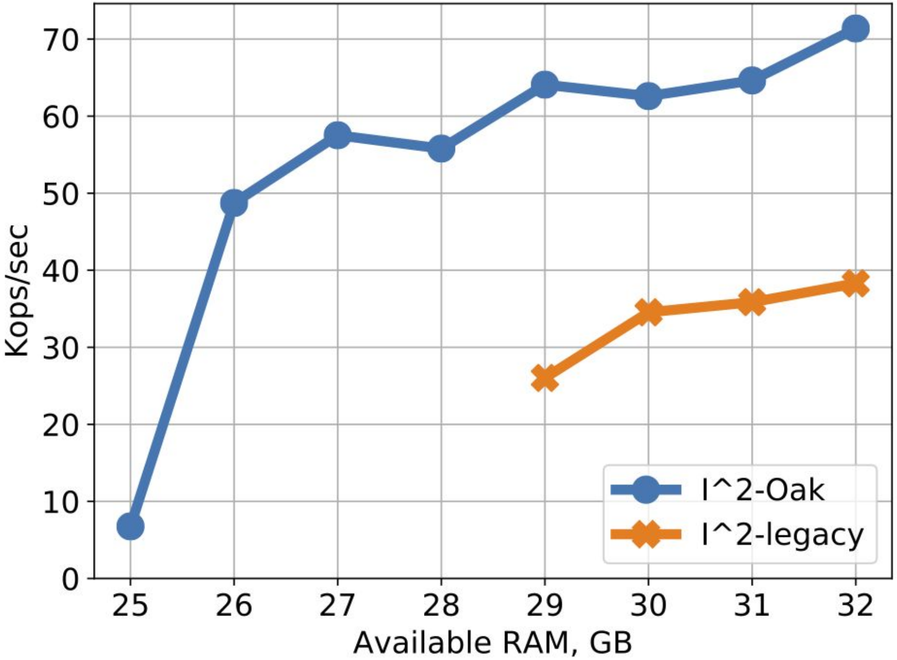
Ingesting 1M to 7M tuples

Tuple size 1.25KB

30GB available RAM

verizon✓
media

Druid Ingestion - Scaling with RAM



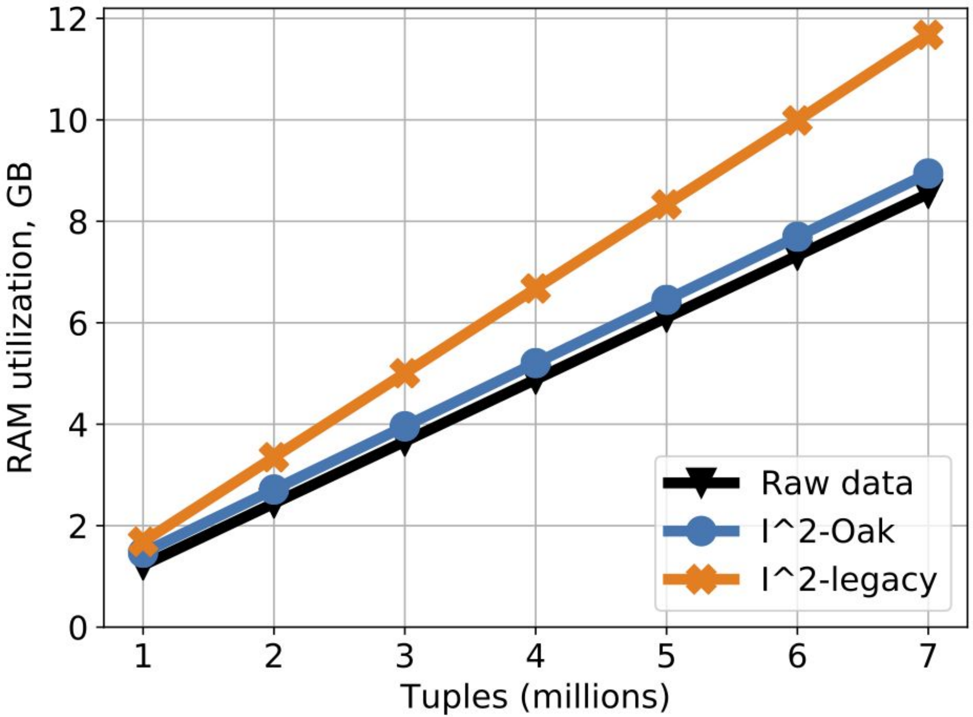
Ingesting 7M tuples

Tuple size 1.25KB

RAM scaling 25GB to 32GB

verizon√ media

Druid Ingestion - RAM overhead



OAK

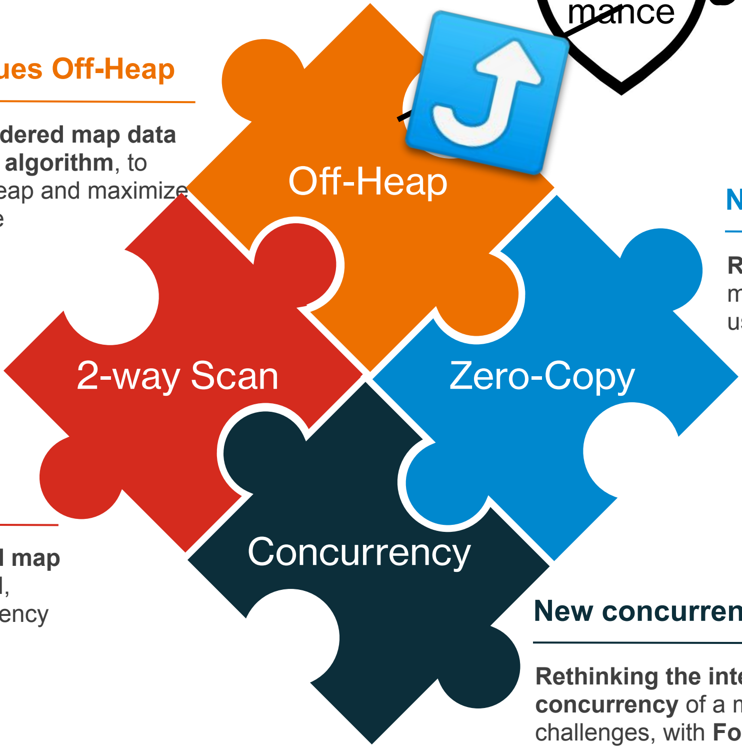
Lean in memory ✓

Fast performance ✓

Real world deploy ✓

Open source library ✓

Scales better ✓



Keys & Values Off-Heap

Rethinking ordered map data structure and algorithm, to minimize on-heap and maximize off-heap usage

New Zero-Copy API

Rethinking ordered map API, t
minimize deserialization and giv
user direct memory access

Fast 2-way Scans

ethinking how an ordered map be traversed backward, out complicating concurrency using additional memory

New concurrent algorithm

Rethinking the internal concurrency of a map to suit new challenges, with **Formal Proof!**

verizon media

How to use OakMap?

What for Oak?

Go to https://github.com/yahoo/Oak

1. **Clone or fork it for yourself**
2. **User needs to create Serializer for Keys and for Values**
    - serialize()
    - deserialize()
    - calculateSize()
3. **User needs to create Keys Comparator**
    - For primitives like Integer/String there are Serializer & Comparator available
4. **Create an OakMapBuilder**
    - OakMapBuilder<K,V> builder = ... \\ create a builder
    - OakMap<K,V> oak = builder.build();
5. **Decide about ZeroCopy API**
6. **Use it! :)**
7. **A problem? Contact anastas@verizonmedia.com**

Oak usages

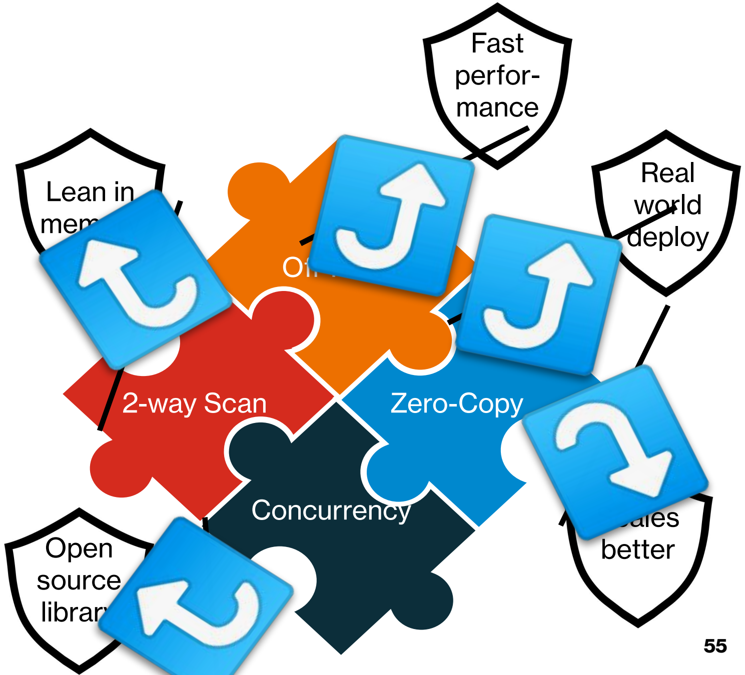
1. If you are using Java's ConcurrentSkipListMap for more than 2-4GB
2. If you are using Java and experience GC related issues or it takes too much memory
3. More than that OakHash its on its way!
4. If you are unsure, but want to check, contact anastas@verizonmedia.com
5. If you think that Oak might be useful, but see some problems, contact anastas@verizonmedia.com
6. Bottom line: contact anastas@verizonmedia.com

Oak: a concurrent ordered KV-map with...

Questions?

anastas@verizonmedia.com

1. **First off-heap managed memory data structure**
   - off-heap data vs on-heap metadata
   - managed programming experience
2. **Novel Zero-Copy API**
   - minimize deserialization
3. **Novel Concurrent Algorithm**
   - conditional and unconditional update-in-place
   - fast 2-ways scans
4. **Fast && Lean compared to CSLM**
   - 2.5% metadata
   - up to x2 faster than CSLM
5. **Real world application**
   - Druid
6. **Open Source Library:** https://github.com/yahoo/Oak



verizon✓
media

Thank you!