Oak for Druid Summary:

1. Oak (**O**ff-heap **A**llocated **K**eys) is a scalable concurrent KV-map for real-time analytics. Oak is a next generation of our previous research in KV-map field. The idea raised more than a year ago and Oak was designed during discussions with [cheddar] and [himanshug], so Oak is modeled based on the requirements of Druid.
2. Oak implements the industry standard Java NavigableMap API. It provides strong (atomic) semantics for read, write, read-modify-write, and range query (scan) operations (forward and backward). Oak is optimized for big keys and values, in particular for incremental maintenance of objects (e.g., aggregation). It is faster and scales better with the number of CPU cores than popular NavigableMap implementations, e.g., Doug Lee's ConcurrentSkipListMap (Java's default).
3. We suggest to integrate Oak-based Incremental Index as an alternative to currently existing Druid's Incremental Index. Because Oak is naturally built for off-heap memory allocation, has greater concurrency support, and should show better performance results.

Oak Design Points:

- Oak's internal index is built on contiguous **chunks** of memory, which speeds up the search through the index due to locality of access.
- Oak provides an efficient implementation of the NavigableMap.**compute(*key, updateFunction*)** API – an atomic, zero-copy update in-place. Specifically, Oak allows user to find an old value associated with the *key* and to update it (in-place) to *updateFunction(old value)*. This allows the Oak user to focus on business logic without taking care of the hard issues of data layout and concurrency control.
- Further on, Oak supports atomic ***putIfAbsentComputeIfPresent(key, buildFunction, updateFunction)*** interface. That allows to look for the *key*, if key is not yet exists the new *key-->buildFunction(place to update)* mapping is added, otherwise the key's value is updated to *update(old value)*. The above interface works concurrently with other updates and requires only one search traversal.
- Oak works off-heap and on-heap. In the off-heap case the keys and the values are copied and stored in a self-managed off-heap ByteBuffer. For Oak, the use of off-heap memory is simple and efficient thanks to its use of uniform-sized chunks. Its epoch-based internal garbage collection has negligible overhead.
- Oak's forward and reverse scans are equally fast (interestingly, prior algorithms as Java's ConcurrentSkipListMap did not focus on reverse scans, and provided grossly inferior performance).

The attached PowerPoint presentation provides additional Oak details.

Oak Performance:

We implement Oak in Java, and compare its performance, with on- and off-heap allocators, to the standard ConcurrentSkipListMap. The experiments were run on a hardware platform that features four Intel Xeon E5-4650 processors, each with 8 cores. The values are 1KB. The maps are initially filled up with 100K randomly selected keys and later the keys distribution is zipfian.

Figure 1 depicts the scalability of the throughput (millions operations in a second) of the mix of put, remove and get operations. The operation are mixed equally about 33% each. Figure 2 presents the mix of compute and get operations. The compute of the ConcurrentSkipListMap is not atomic. In Figures 3 and 4 we can see the performance of the ascending (forward scan) and descending (backward scan) iterator respectfully. The range scanned is 100 keys. The great performance of the Oak's descending

iterator is due to each next step is done in O(1) complexity (same as ascending iterator). While in ConcurrentSkipListMap's descending iterator each next step is done in O(logN) complexity.
The results are not the same as we expect to see with Oak integrated in Druid, as Druid brings some additional overhead. However we can get some insights from the following results.

Figure 1:

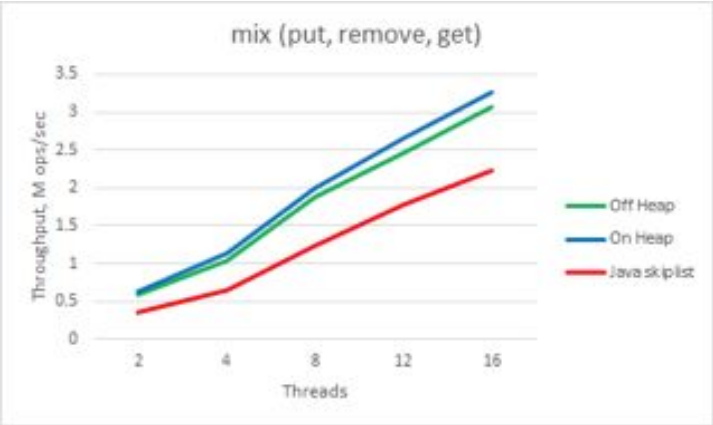


Figure 2:

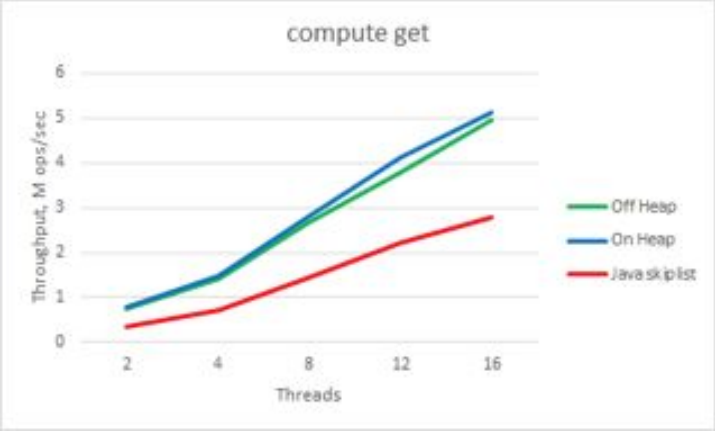


Figure 3:

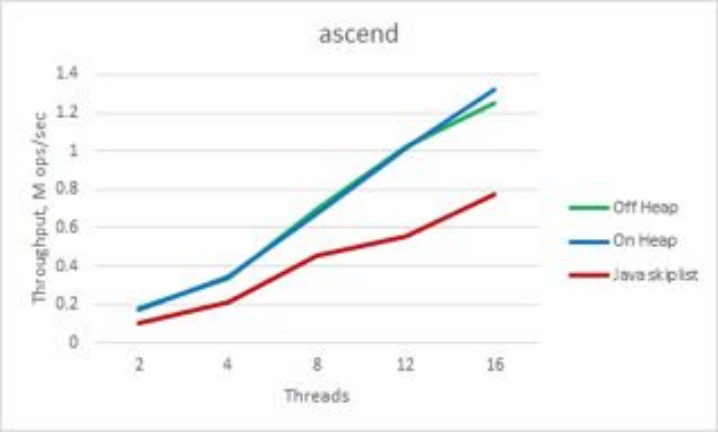
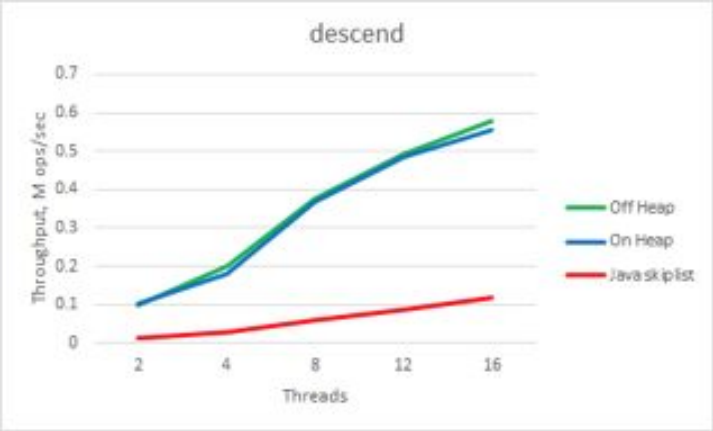


Figure 4:



Integration of the Oak as an Oak-based Incremental Index (as an alternative to currently existing Druid's Incremental Index) requires a restructure to the Druid's Incremental Index module. The details of the suggested refactoring will soon follow.