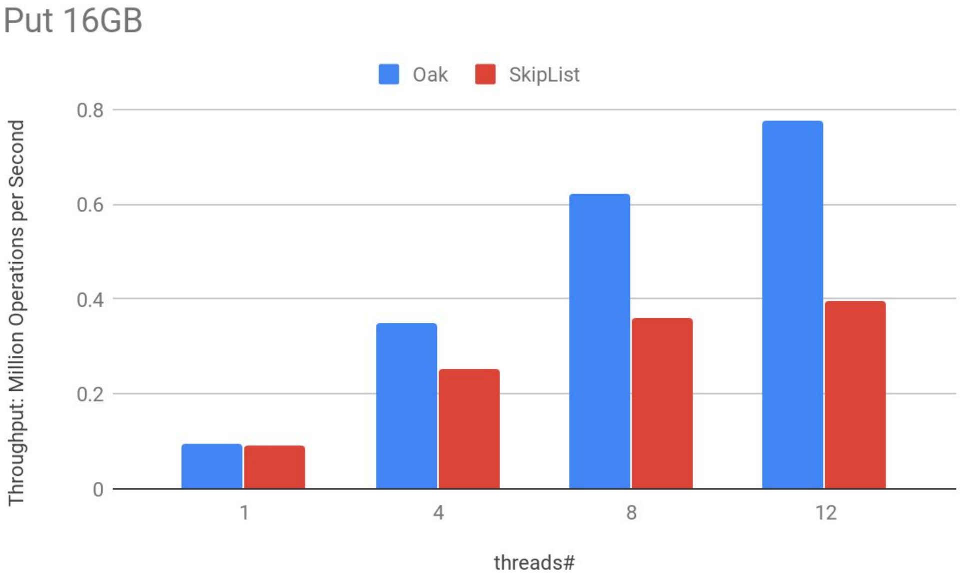
OAK PERFORMANCE

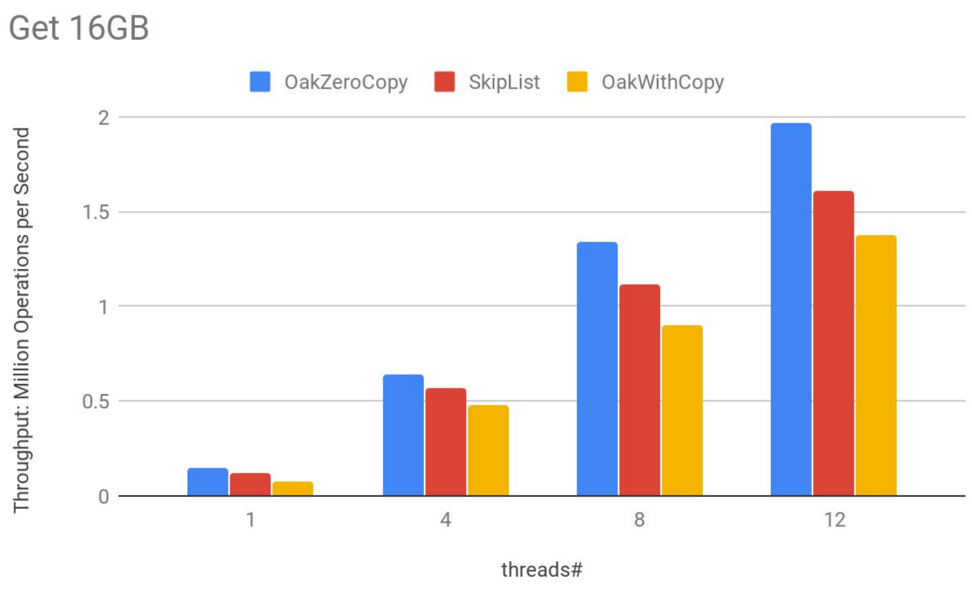
First we would like to present isolated benchmarks comparing OakMap (ZeroCopy) to Java's ConcurrentSkipListMap. We present a very simple benchmarks to start with. All benchmarks start with the warm-up phase inserting 5M random key-value pairs. Key takes 100 bytes and value 1KB. 5.5GB is written in warm-up phase.

After warm-up we proceed with measurements of the get-only, put-only or ascending/descending scans workloads. The highest memory usage after puts (with maximal throughput) is about 9.6GB (including warm-up). All experiments exercise 1,4,8, or 12 threads within 16GB on-heap for SkipList and 4GB on- with 12GB off-heap for Oak (also 16GB in total).

OAK vs Java ConcurrentSkipListMap Results

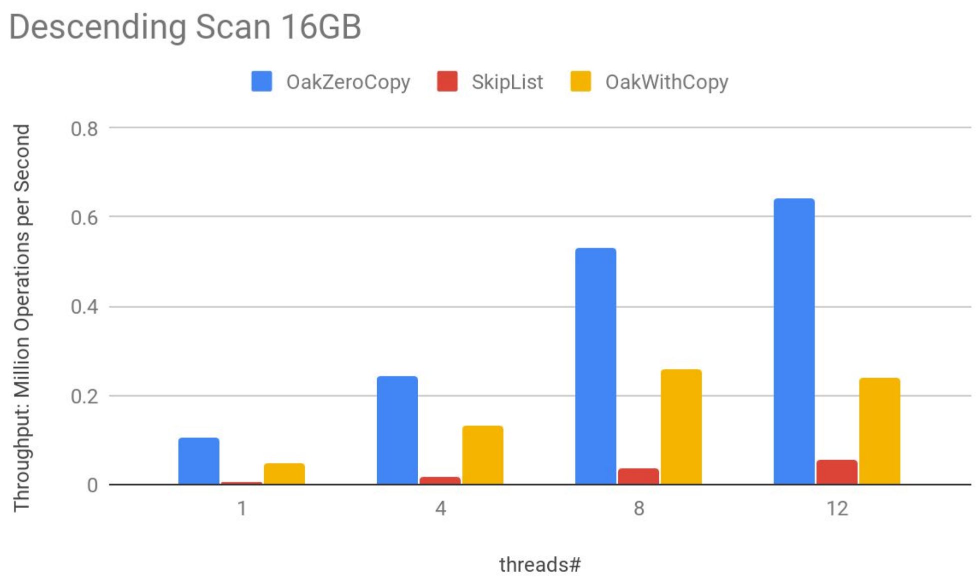
Please find below the put-only and get-only (zero-copy and with copy) throughput comparison. For data ingestion, Oak throughput is up to twice higher that of ConcurrentJavaSkipList. For zero-copy gets-only, Oak improves ConcurrentJavaSkipList throughput in up to 30%. When Oak is requested to copy the data upon retrieval, Oak copies the data out of the off-heap buffers and to builds the objects. Due to this overhead, Oak performs about 10-15% worse than ConcurrentJavaSkipList. Note that for data ingestion the copy is done anyway.

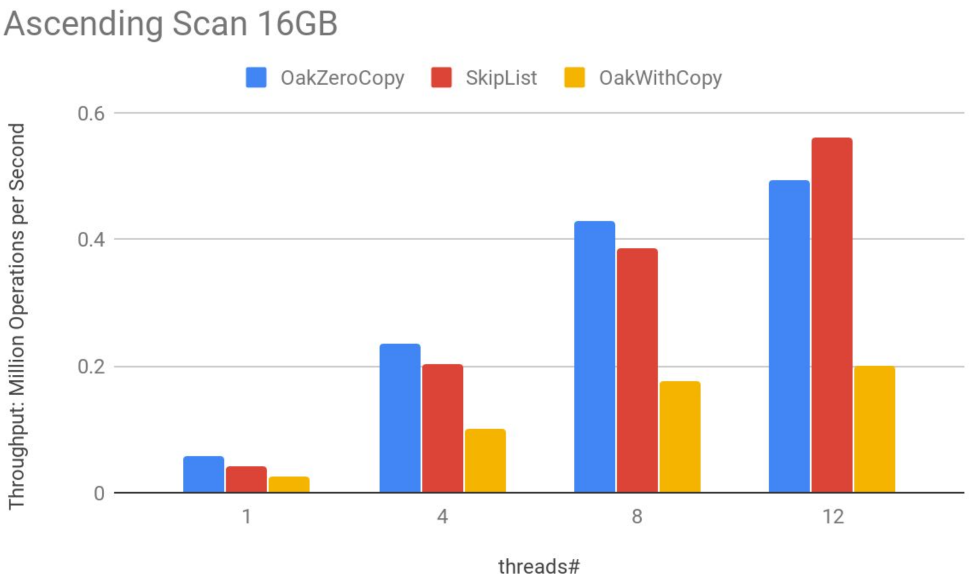




Hereby are the throughputs for ascending and descending scans. Those are short scans retrieving 100 keys from a random start keys.

For descending scan Oak outperforms SkipList dramatically. This is because Oak's scan steps are done in constant time vs logarithmic steps of ConcurrentJavaSkipList's scan. Again if building of the objects is requested upon data retrieval, it deteriorates the performance results of OakWithCopy. For ascending scans Oak's performance is comparable to SkipList.




Ascending Scan 16GB
🟦 OakZeroCopy
🟥 SkipList
🟨 OakWithCopy
Throughput: Million Operations per Second
0.6
0.4
0.2
0
1
4
8
12
threads#