Brief Announcement: A Key-Value Map for Massive Real-Time Analytics

Dmitry Basin
Yahoo Research
Haifa, Israel
dbasin@yahoo-inc.com

Edward Bortnikov
Yahoo Research
Haifa, Israel
ebortnik@yahoo-inc.com

Anastasia Braginsky
Yahoo Research
Haifa, Israel
anastas@yahoo-inc.com

Guy Golan Gueta
Yahoo Research
Haifa, Israel
ggolan@yahoo-inc.com

Eshcar Hillel
Yahoo Research
Haifa, Israel
eshcar@yahoo-inc.com

Idit Keidar
Technion and Yahoo Research
Haifa, Israel
idish@ee.technion.ac.il

Moshe Sulamy*
Tel-Aviv University
Tel-Aviv, Israel
msulamy@gmail.com

ABSTRACT

Modern big data processing platforms employ huge in-memory key-value (KV-) maps. Their applications simultaneously drive high-rate data ingestion *and* large-scale analytics. These two scenarios expect KV-map implementations that scale well with both real-time updates and massive atomic scans triggered by range queries. However, today's state-of-the art concurrent KV-maps fall short of satisfying these requirements – they either provide only limited or non-atomic scans, or severely hamper updates when scans are ongoing.

We present KiWi, the first atomic KV-map to efficiently support simultaneous massive data retrieval and real-time access. The key to achieving this is treating scans as first class citizens, whereas most existing concurrent KV-maps do not provide atomic scans, and some others add them to existing maps without rethinking the design anew.

1. INTRODUCTION

The ordered *key-value* (KV) map abstraction has been recognized as a popular programming interface since the dawn of computer science, and remains an essential component of virtually any computing system today. It is not surprising, therefore, that with the advent of multi-core computing, many scalable concurrent implementations have emerged, e.g., [2, 3, 4, 9, 10, 12].

Today, KV-maps have become centerpiece to web-scale data processing systems, e.g., Google's F1 [15], which powers its AdWords[1] business, and Yahoo's Flurry[2], the technology behind its Mobile Developer Analytics business. These applications raise novel use cases and push scalability requirements to new levels. Namely, they require real-time performance for both (1) ingestion of the incoming data streams, and (2) analytics of the resulting dataset. For example, as of early 2016, the Flurry platform systematically collected data of 830,000 mobile apps[3] running on 1.6 billion user devices[4]. Flurry streams this data into a massive index, and provides application developers with tools that produce a wealth of reports over the collected data.

Analytics platforms exploit KV-stores like Google's Bigtable [7] and Apache HBase [1]. These technologies combine on-disk indices for persistence with an in-memory KV-map for real-time data acquisition [7]. The latter's scalability has a major impact on overall system performance, (as shown in [8]). The stream scenario requires the KV-map to support fast *put* operations, whereas the analytics aspect relies on (typically large) scans (i.e., range queries). The consistency (atomicity) of scans is essential for correct analytics. The new challenge that arises in this environment is allowing consistent scans to be obtained *while the data is being updated in real-time*.

In this brief announcement we present KiWi, the first KV map to efficiently support large atomic scans as required for data analytics, alongside real-time updates. Most existing concurrent KV-maps do not support atomic scans at all [2, 11, 3, 4, 9, 12, 10]. Others support a single scan at a time [13], do not ensure progress for scans [5, 14, 6, 11], or severely hamper updates when scans are ongoing [5, 14]. Table 1 summarizes the properties of state-of-the-art concurrent data structures that support scans, and compares them to KiWi.

*This work was done in part while interning with Yahoo Research, Haifa.

PODC'16, July 25-28, 2016, Chicago, IL, USA
ACM 978-1-4503-3964-3/16/07.
http://dx.doi.org/10.1145/2933057.2933061

[1]https://www.google.com/adwords/
[2]https://developer.yahoo.com/flurry/docs/analytics/
[3]http://flurrymobile.tumblr.com/post/144245637325/appmatrix
[4]http://flurrymobile.tumblr.com/post/117769261810/the-phablet-revolution

	atomic scans	balanced	multi-scan	wait-free scans	fast puts
snapshot iterator [13]	✓	✓	✗	✓	✓
SnapTree [5]	✓	✓	✓	✗	✗
Ctrie [14]	✓	✓	✓	✗	✗
k-ary tree [6]	✓	✗	✓	✗	✓
BW-Tree [11]	✗	✓	✓	✗	✓
Java-Skiplist [2]	✗	✓	✓	✓	✓
KiWi	✔	✔	✔	✔	✔

Table 1: Comparison of concurrent data structures that support scans. Fast puts means not hampering updates (e.g., by cloning nodes) when scans are ongoing.

An important emphasis in KiWi's design is on facilitating synchronization between scans and updates. Since scans are typically long, our solution avoids livelock and wasted work by always allowing them to complete (without ever needing to restart). On the other hand, updates are short (since only single-key puts are supported), therefore restarting them in cases of conflicts is practically "good enough" – restarts are rare and when they do occur, little work is wasted. Formally, KiWi opts to provide *wait-free* scans and *lock-free* put operations.

2. DESIGN PRINCIPLES

To support atomic wait-free scans, KiWi employs multi-versioning. But in contrast to the standard approach where each put creates a new version for the updated key, KiWi only keeps old versions that are needed for ongoing scans, and otherwise over-writes the existing version. Moreover, version numbers are managed by scans rather than updates, and thus put operations may over-write data without changing its version number. This unorthodox approach offers significant performance gains given that scans typically retrieve large amounts of data and hence take much longer than updates. It also necessitates a fresh approach to synchronizing updates an scans, which is a staple of KiWi's design.

A second important feature of KiWi's design is its efficient memory management. Data in KiWi is organized as a collection of *chunks*, which are large blocks of contiguous key ranges. Such data layout is cache-friendly and suitable for non-uniform memory architectures (NUMA), as it allows long scans to proceed with few fetches of new data to cache or local memory. Chunks regularly undergo maintenance to improve their space utilization (via *compaction*) and internal organization, as well as the distribution of key ranges into chunks (via splits and merges). All these issues are handled by KiWi's *rebalance* abstraction, which performs batch processing of multiple maintenance operations that are vital for the data structure's health. The synchronization of rebalance operations with ongoing puts and scans is subtle, and much of the KiWi algorithm is dedicated to handling possible races in this context.

KiWi is a balanced data structure, providing logarithmic access latency in the absence of contention. This is achieved via a combination of (1) indexing chunks for fast lookup and (2) partially sorting keys in each chunk to allow for fast in-chunk binary search. To facilitate concurrency control, we separate chunk management from indexing: KiWi employs a *search index* separately from the (chunk-based) data storage layer. The index is updated lazily after rebalancing of the data storage layer completes.

3. RESULTS

KiWi is a practical algorithm, with multiple optimizations implemented on top of the theoretical underpinnings. We benchmark its Java implementation extensively under multiple representative workloads. In the vast majority of experiments, KiWi significantly surpasses the best-in-class concurrent algorithms. KiWi's advantages are particularly pronounced in our target scenario with long scans in the presence of concurrent puts, where KiWi's atomic scans are twice as fast as the non-atomic scans offered by the Java Skiplist [2], and compared to existing atomic solutions [6, 5], KiWi not only performs all operations faster, but actually executes either updates or scans an order of magnitude faster than every other solution.

Figure 1 presents evaluation results for a *mixed* workload – half the threads perform scans, whereas the second half perform puts. Each scan picks the range's lower bound uniformly at random, and iterates through S keys for a parameter S. We compare KiWi with other state-of-the art KV-maps supporting scans.. Figure 1(a) shows the throughput scalability with scan size, with maximal parallelism (32 threads). KiWi dominates universally except for very short scans, for which k-ary tree [6] is 10% faster. For long scans, the latter deteriorates fast (Figure 1 (b)). This happens because k-ary tree restarts iterating every time a put falls within the scanned range – i.e., puts make progress but scans get starved. For large values of S, SnapTree [5] is the second-fastest because it shared-locks the scanned ranges in advance and scans unobstructed. From the puts perspective (Figure 1 (c)-(d)), KiWi is the fastest again, whereas SnapTree is the slowest since its locking starves concurrent updates. In contrast to its competitors, KiWi serves both scans and puts well.

Finally, KiWi implements a balanced data structure, which allows operations to run orders of magnitude faster than unbalanced ones in case insertion order is not random. For example, if we build the map from an ordered stream of keys, KiWi's throughput is close to that obtained in experiments with a random insertion order. In contrast, implementations that do not address balancing directly fail to provide adequate performance – e.g., k-ary tree's get/put throughput is 700 times slower than KiWi's in this scenario.

4. REFERENCES

[1] Apache HBase – a Distributed Hadoop Database. https://hbase.apache.org/.

[2] Java Concurrent Skip List. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

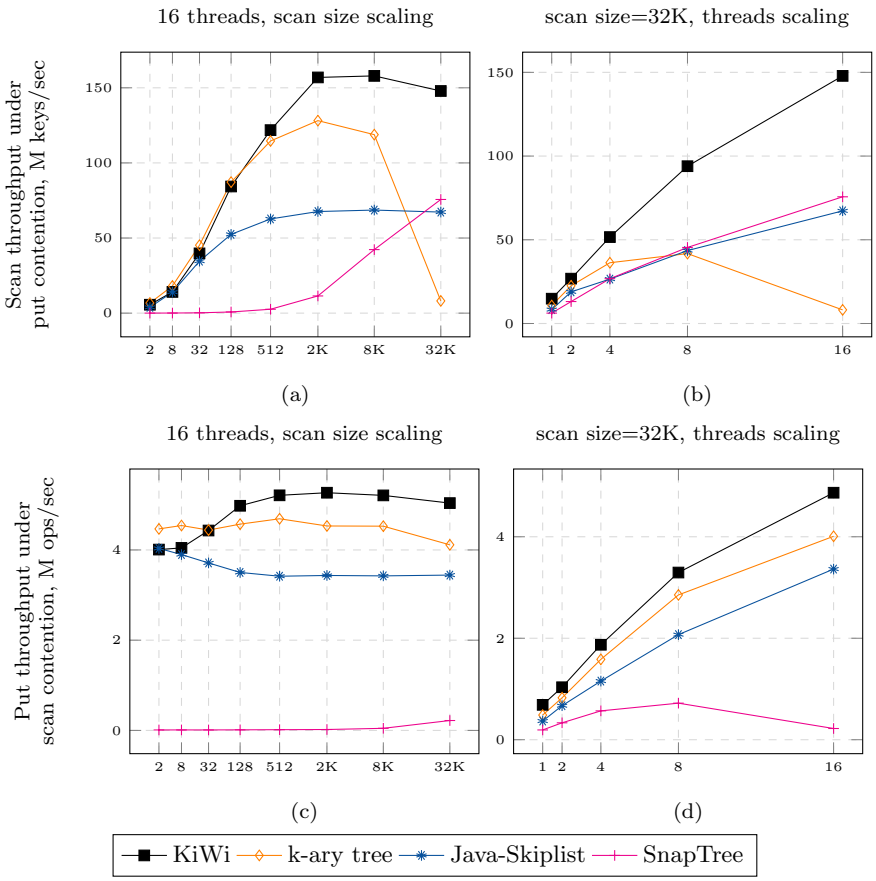

Figure 1: KiWi's throughput: (a-b) Scans with background put contention, (c-d) Puts with background scan contention.

[3] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *ICDCN*, pages 107–118, 2011.

[4] A. Braginsky and E. Petrank. A lock-free B+tree. In *SPAA*, pages 58–67, 2012.

[5] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, pages 257–268, 2010.

[6] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *OPODIS*, pages 31–45, 2012.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[8] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *EuroSys*, pages 32:1–32:14, 2015.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215, 2004.

[10] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *PPoPP*, pages 141–150, 2012.

[11] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.

[12] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.

[13] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *DISC*, pages 224–238, 2013.

[14] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *PPoPP*, pages 151–160, 2012.

[15] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.