Big(ger) Sets: decomposed delta CRDT Sets in Riak

[Work in progress report]

Russell Brown
Basho Technologies, Ltd
United Kingdom
russelldb@basho.com

Zeeshan Lakhani
Basho Technologies, Inc
USA
zlakhani@basho.com

Paul Place
Basho Technologies, Inc
USA
pplace@basho.com

ABSTRACT

CRDT[24] Sets as implemented in Riak[6] perform poorly for writes, both as cardinality grows, and for sets larger than 500KB[25]. Riak users wish to create high cardinality CRDT sets, and expect better than $O(n)$ performance for individual insert and remove operations. By decomposing a CRDT set on disk, and employing delta-replication[2], we can achieve far better performance than just delta replication alone: relative to the size of causal metadata, not the cardinality of the set, and we can support sets that are 100s times the size of Riak sets, while still providing the same level of consistency. There is a trade-off in read performance but we expect it is mitigated by enabling queries on sets.

Keywords

Eventual Consistency, CRDTs, databases.

1. INTRODUCTION

Riak is an eventually consistent[26] key-value database inspired by Dynamo[16]. To ensure write availability, Riak allows concurrent writes to keys. Some users find resolving conflicts resulting from concurrency hard[19] so we added some CRDTs[23] to Riak. Our initial implementation was to create an open source library of purely functional datatypes[9] and embed them in Riak with an API[8]. In this paper we consider only the Set, an Erlang implementation of the state-based ORSWOT[13] (Observe-Remove-Set-Without- Tombstones).

In this paper we show that:

1. a naive implementation of data types in Riak performs poorly, especially for larger cardinality sets (section 2)
2. simply adding delta replication is not much better (section 3)
3. decomposing a set CRDT into its constituent parts (logical clock and set members) yields great improvements in write performance (section 4)

PaPOC'16, April 18-21 2016, London, United Kingdom

ISBN 978-1-4503-4296-4/16/04...$15.00

DOI: http://dx.doi.org/10.1145/2911151.2911156

This final point above is our work's primary contribution, and speaks to the needs of practitioners to do more than translate research into code, and consider the lifetime of the CRDT in the system, including such mundane details as durable storage. See figure 1 if you read nothing else. While decomposing the set may itself seem a reasonably obvious approach, it raises issues around anti-entropy and reads that require creative solutions.

2. MOTIVATION

When Basho released Riak DataTypes in Riak 2.0[7], the first thing users did was treat Riak Sets like Redis Sets[22] and try and store millions of elements in a set. Redis is a non-distributed, in-memory data structure server, not a distributed, fault-tolerant key/value database. Riak was unable to perform satisfactorily to user expectations when performing inserts into sets, especially as set cardinality grew.

Riak stores all data in riak-objects, a kind of multi-value-register[23]. As Riak stores each object durably on disk in a local key/value store (either bitcask or leveldb) there is a limit to the size each key/value pair can be[11], and since Riak's CRDTs are stored in a riak-object, they inherit that limit.

A riak-object has a version vector[17], and an opaque binary payload. The CRDT is stored in the binary payload.

Users report a degradation in write performance as sets grow[25]. In all the literature a CRDT is a thing in memory. There is only one of them, and every actor is a replica. In a client/server database this is not the case: a database services many clients, and stores many objects durably on disk. CRDTs in Riak must therefore be serialized for both storage and transfer over the network.

2.1 Understanding Riak Set Performance

When a client wishes to add an element to a Riak set it sends an operation: 'add element E to set S'

Riak forwards the operation to a coordinating replica that will: read a riak-object from disk, de-serialize the set, add the element to the set, serialize the set, update the riak-object's version vector, write the riak-object to disk, and send the riak-object downstream for replication.

Each replica will then read their local copy from disk. If the incoming update supersedes their local copy, determined by comparing riak-object version vectors, they simply store the incoming riak-object. If conflict is detected, the set is de-serialized from the riak-object, the CRDT join function is called, and the result serialised, the riak-object version vectors are merged, and the riak-object is finally written to

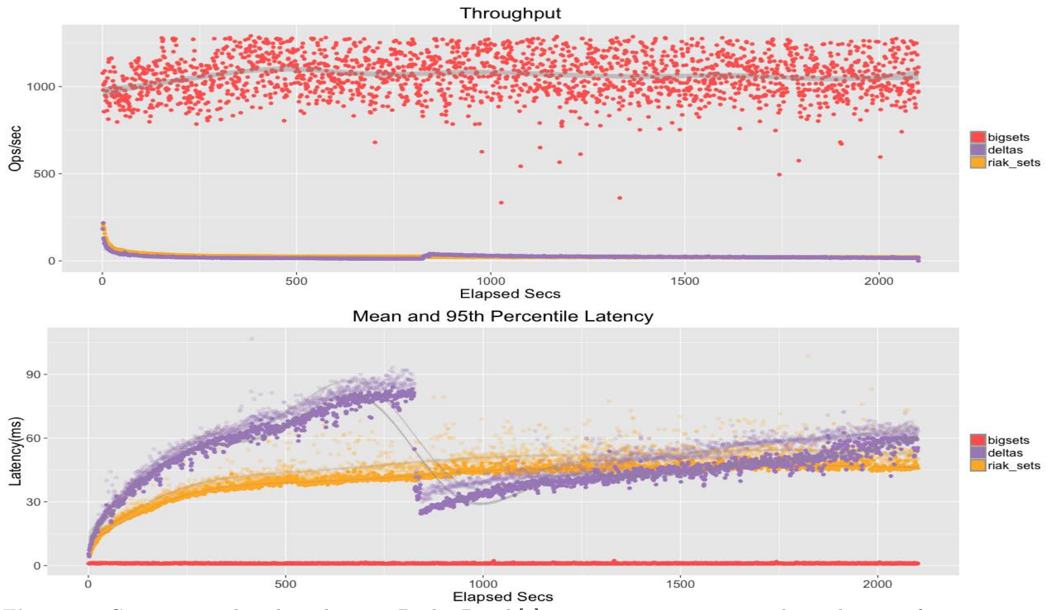

Figure 1: Comparative benchmark using Basho Bench[4], inserting up-to 10,000 4-byte elements for 35 minutes.

disk.

The key insight is that the set is stored in a single object. To add an element, Riak must read and write the whole object. This dictates both performance and the size to which a set can grow. Per-operation this yields an $O(n)$ cost, where n is the size of the set. However, over the life of the set the cost is $O(n^2)$, both in terms of bytes read and written, and transfered over the network. Where a single insert is $O(n)$ filling a set by single inserts from $0 \cdots n$ is $O(n^2)$!

3. DELTA CRDTS

State based CRDTs are join-semi-lattices. They work by "joining" or "merging" divergent replica state into a single value in a deterministic way. As described above in section 2, Riak uses "full state replication" to propagate updates to Sets. Delta-CRDTs[2] are a CRDT variant that promise a more effecient network utilisation for replication of updates. In order to improve performance we implemented delta-datatypes[10]. A delta-CRDT is one that, when updated, generates a smaller-than-full-state delta-state that can be replicated. The performance in Riak was virtually unchanged. As per the paper the upstream replica generates a delta and sends it downstream. Downstream the set ***must always*** merge the delta as an incoming delta ***never*** supersedes the local state, even without concurrency! The saving in terms of network transmission comes at the expense of always deserialising and merging downstream.

4. BIGSET: AN OVERVIEW

We wrote a prototype system, ***bigset***, to address these issues.

In bigset, as in Riak, the actors in the system are called vnodes[12]. N vnodes each store a replica of each datum. Vnodes may act concurrently. Vnodes service requests for many clients; vnodes store many data items.

The key innovation is that bigset decomposes the Set CRDT across a range of keys. An ORSWOT CRDT Set is composed of a logical clock, opaque set members, and per-member causal information called "dots". Rather than mapping a key (a set name) to an object that contains the whole CRDT Set, Bigset gives each of these component elements of a CRDT Set their own key in an ordered durable key/value datastore.

A bigset is made of a set-clock, a set-tombstone, and a collection of element-keys.

4.1 Clocks

Both the set-clock and set-tombstone are logical clocks. The former summarises events seen by the actor, the latter events removed by the actor. The set-clock grows as events occur and are replicated in the system. The set-tombstone temporalily grows as it records removal information, and then shrinks as keys are discarded.

The clocks consist of a 2-tuple of $\{\text{BaseVV}(), \text{DotCloud}()\}$ Where BaseVV is a normal Version Vector and DotCloud is a mapping of ActorIDs to a list of integers which denote the events seen at the replica that are not contiguous to the base Version Vector[2][20]. A replica will **never** have an entry for itself in the DotCloud.

4.2 Elements

Elements are the opaque values that clients wish to store in a Set. In bigset each insert of an element gets its own key in leveldb[15]. The key is a composite made of the set name, the element, and a dot[1]. The dot is a pair of $(actorId, Event)$ denoting the logical event of inserting the element. The element-keys are also the delta that is replicated.

4.3 Writes

When writing, bigset does not read the whole set from disk; instead, it reads the set's logical clocks only and writes the updated clock(s) and any element-keys that are being inserted. For removes we need only read and write the clocks. This is how bigset acheives its write performance.

NOTE: multiple elements can be added/removed at once, but for brevity/simplicity we discuss the case of single element adds/removes only.

4.3.1 Inserts

When a client wishes to insert an element it sends an operation: 'add element E to set S with Ctx'. It may provide a causal context (hereafter just context) for the operation, which provides information as to what causal history makes up the Set the client has observed. The context will be empty if the client has not seen element E at all (as we expect to be the common case.)[1]

Bigset sends this operation to a coordinating vnode that will run algorithm 1 (see 9). Briefly: It reads the clock for the set, increments it, creates a key for the new element, and stores it and the clock, atomically.

It then sends the new key downstream as a "delta". The downstream replicas run algorithm 2 (see 9): read the local clock, if the local clock has seen the delta's dot, then it is a no-op, otherwise the clock adds the new dot to itself, and stores the updated clock and delta atomically.

4.3.2 Removes

Removes are as per-writes except there is no need to coordinate, and no element key to write. To have an effect the client ***must*** provide a context for a remove. The remove algorithm is far simpler: if the set-clock has seen the context, add the dots of the context to the set-tombstone. Otherwise, add them to the set-clock. This ensures that, if the adds were unseen they never get added, and if they were seen, they will get compacted out (see 4.3.3 below).

4.3.3 Compaction

We use leveldb[5] to store bigsets. Leveldb has a mechanism to remove deleted keys called compaction[18]. We have modified leveldb to use the set-tombstone in compaction. As leveldb considers a key K for compaction it uses the set-tombstone. If $K.dot$ is seen by the tombstone, the key is discarded. This way we remove superseded/deleted keys without issuing a delete. Once a key is removed the set-tombstone subtracts the deleted dot. This trims the set-tombstone, keeping its size minimal.

4.4 Reads

A bigset read is a leveldb iteration, or fold, over the keyspace for a set. It transforms the decomposed bigset into a traditional ORSWOT state-based CRDT. Every element key that is NOT covered by the set-tombstone is added to the ORSWOT.

As bigsets can be *big*, we don't wait to build the entire set before sending to the client, we stream the set in configurable batches (by default 10000 elements at a time.)

Riak, and bigsets, allow clients to read from a quorum of replicas. Bigset has a novel streaming ORSWOT CRDT Join operation, that is able to perform a merge on subsets of an ORSWOT. This is enabled by the fact that the set element keys are stored and therfore streamed in lexicographical element order. This ordering and incremental merging allows us to query a bigset. We can discover if X is a member of Set without reading the whole set. We can seek to a point in the set and stream, to enable pagination or range queries. We expect this to mitigate the current negative performance delta between Riak DataTypes and bigset for reads.

5. EXPERIENCE WITH BIGSETS

Not yet in production, but being developed, we do have both initial benchmark results, and property based tests for bigsets. Our property based tests using quickcheck[21] show that bigset and Riak sets are semantically equivalent. The benchmarks show us that bigsets performs far better for writes (see figure 1), while paying a read performance penalty (see figure 4) which we plan to engineer our way out of with low-level code, and by providing queries over the Set, making full set reads unnecessary in most cases.

6. SUMMARY

We've characterised the key difference between bigsets and Riak DataType sets: decomposition of the CRDTs structure, minimal read-before-write, and a set-tombstone and set-clock that make joining a delta as simple as adding its causal data to a clock and appending the key to leveldb.

The poor performance of CRDTs in Riak led to the bigsets design, which clearly demonstrates that considering the primary need to durably store a CRDT means optimising for bytes read and written. We have much work to do to bring this prototype into the Riak KV database. We plan in the future to write more about key processes we have developed including anti-entropy and hand-off, and also generality for application to other data types, including Riak Maps[14].

7. ACKNOWLEDGEMENTS

Many thanks to Scott Lystig Fritchie and John Daily for their time reviewing drafts.

Funding from the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement 609551, SyncFree project.

8. REFERENCES

[1] P. S. Almeida, C. B. Moreno, R. Gonçalves, N. Preguiça, and V. Fonte. Scalable and accurate causality tracking for eventually consistent stores. In *Distributed Applications and Interoperable Systems*, volume 8460, Berlin, Germany, June 2014. Springer, Springer.

[2] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In A. Bouajjani and H. Fauconnier, editors, *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, volume 9466 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2015.

[3] C. Baquero. Delta crdts. https://github.com/CBaquero/delta-enabled-crdts.

[4] Basho. Basho bench. http://docs.basho.com/riak/latest/ops/building/benchmarking/.

[5] Basho. Leveldb. http://docs.basho.com/riak/latest/ops/advanced/backends/leveldb/.

[6] Basho. Riak. http://basho.com/products/riak-kv/.

[7] Basho. Riak 2.0. http://basho.com/posts/technical/introducing-riak-2-0/.

[8] Basho. Riak datatypes. https://docs.basho.com/riak/2.1.3/dev/using/data-types/.

[9] Basho. Riak dt. https://github.com/basho/riak_dt.

[10] Basho. Riak dt deltas. https://github.com/basho/riak_dt/tree/delta_data_types.

[11] Basho. Riak object max size. http://docs.basho.com/riak/latest/community/faqs/developing/#is-there-a-limit-on-the-size-of-files-that-can-be.

[12] Basho. Vnodes. http://docs.basho.com/riak/latest/theory/concepts/vnodes/.

[13] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[14] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[15] J. Dean and S. Ghemawat. Leveldb. https://rawgit.com/google/leveldb/master/doc/index.html.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[17] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug. 1991.

[18] Google. Leveldb file layout and compactions. https://leveldb.googlecode.com/svn/trunk/doc/impl.html.

[19] D. Macklin. Key lessons learned from transition to nosql at an online gambling website. http://www.infoq.com/articles/key-lessons-learned-from-transition-to-nosql.

[20] D. Malkhi and D. B. Terry. Concise version vectors in winfs. *Distributed Computing*, 20(3):209–219, 2007.

[21] T. A. QuviQ, John Hughes. Erlang-quickcheck. http://www.quviq.com/products/erlang-quickcheck/.

[22] S. Sanfilippo. Redis. https://en.wikipedia.org/wiki/Redis.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[24] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

[25] K. M. Spartz. Benchmarking large riak data types. http://kyle.marek-spartz.org/posts/2014-12-02-benchmarking-large-riak-data-types-continued.html.

[26] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[27] Basho. Basho OTP. https://github.com/basho/otp/releases/tag/OTP_R16B02_basho8.

[28] J. L. Petersen. Estimating the Parameters of a Pareto Distribution. http://www.math.umt.edu/gideon/pareto.pdf.

[29] Amazon. Previous Generation Instances. http://aws.amazon.com/ec2/previous-generation.

Notes

[1]Why a context? Adds are removes. If the client has already seen some element E in the set a new insert of E replaces the old. Adding element E at replica X will cause all dots for E at X to be removed and the single new event will be the sole surviving dot for E. This optimisation comes from the reference implementation[3] and assumes that the actor ***is*** a replica. With action-at-a-distance it is more complex: the client is not a replica, so must tell the coordinating replica what it has seen, it must read the set (or at least the element) to get a context. To clarify why, imagine a client reads from vnode A, but writes to vnode B.

9. APPENDIX: ALGORITHMS

Algorithm 1 bigset coordinate insert

```
Require: set S, element E, op-context Ctx
  SC = read set-clock
  TS = read set-tombstone
  for Dot in Ctx do
    if SC not seen Dot then
      SC = join(SC, Dot)
    else
      TS = join(TS, Dot)
    end if
  end for
  SC.increment()
  Dot = SC.latest-dot()
  Val = (S, E, Dot)
  atomic-write([SC, TS, Val])
  send-downstream(Val, Ctx)
```

Algorithm 2 bigset replica insert

```
Require: set S, val V, op-context Ctx
  SC = read set-clock
  TS = read set-tombstone
  for Dot in Ctx do
    if SC not seen Dot then
      SC.join( Dot)
    else
      TS .join(Dot)
    end if
  end for
  if SC not seen V.dot then
    SC.add(V.dot)
    atomic-write([SC, TS, V])
  else {SC seen V.dot}
    atomic-write-if-changed([SC, TS])
  end if
```

10. APPENDIX: EVALUATION

In-progress evaluations are being run on many setups and environments. For this paper, we used an Amazon EC2 *cc2.8xlarge*[29], compute-optimized, 64-bit instance type, which includes 2 Intel Xeon processors, each with 8 cores, 3.37 TB of internal storage, and 60.5 GB of RAM. For our *Riak Sets* and *Deltas*[10] runs, we used a 4-node Riak/Riak-core cluster built atop on Basho's *OTP_R16B02_basho8* fork[27] of Erlang OTP.

To exercise our clusters, we used Basho Bench[4], a benchmarking tool created to conduct accurate and repeatable stress tests and produce performance graphs. The information in table 1 centers on writes on a single key with a single worker process. For table 2, we used 25 concurrent workers.

The focus of our runs are on write-performance across increasing set cardinality. We've also included some preliminary results with 1000-key pareto-distributed[28], 5-minute, read loads, as well as a mixed write/read load, comparing current Riak Sets against the ongoing Bigset work.

10.1 Write Runs

	Riak Sets Avgs.			*Deltas Avgs.*			*Bigsets Avgs.*		
Write Runs	TP	M	95th	TP	M	95th	TP	M	95th
5k/4bytes	81.57	14.54	16.4	48.81	24.48	27.26	7094.66	7.07	18.71
10k/4bytes	45.58	27.62	30.98	26.79	47.28	52.43	1040.00	0.94	1.12
45k/4bytes	6.89	173.60	183.49	12.61	91.22	98.91	1060.67	0.94	1.11

Table 1: Writes *Cardinality/Size-Per-Element*
TP - *Throughput* in operations per second (Ops/sec)
M - *Mean latency* in microseconds
95th - *95th percentile latency* in microseconds

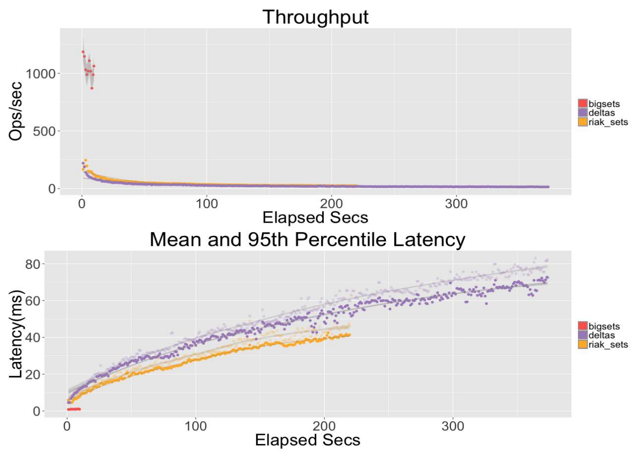

Figure 2: Writes on a 10,000-cardinality set of 4-byte elements. Bigsets completes getting to the chosen cardinality much more quickly than the others.

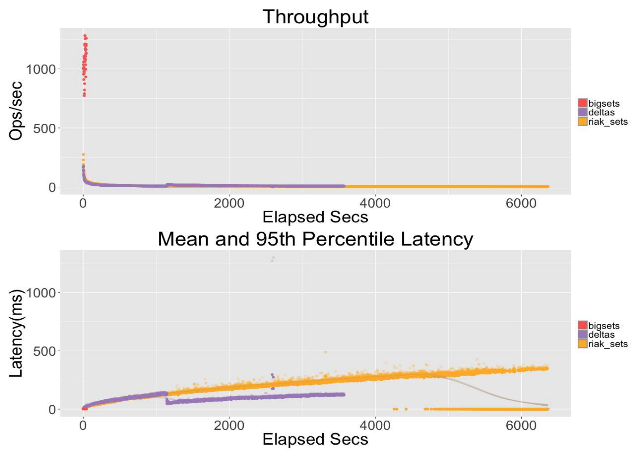

Figure 3: Writes on a 45,000-cardinality set of 4-byte elements. Like figure 2, it reaches its chosen cardinality much faster.

10.2 Read and Mixed Runs

	Riak Sets Avgs.			*Bigets Avgs.*		
Reads Runs	TP	M	99th	TP	M	99th
1k/1k/4bytes	6353.83	3.93	6.85	785.23	32.70	202.18
1k/10k/4bytes	1266.48	19.78	32.72	221.14	116.77	632.67
1k/100k/4bytes	64.11	390.96	693.71	38.55	652.56	965.75
∼1k/Mix-**R**	1198.45	4.36	40.60	2062.72	16.48	121.97
∼1k/Mix-**W**	”	40.00	1301.82	”	14.40	202.50

Table 2: Reads/Mixed *Keys/Cardinality/Size-Per-Element*
Mix - 60/40 write-to-read ratio for 5 minute *Write/Read*
TP - *Throughput* in operations per second (Ops/sec)
M - *Mean latency* in microseconds
99th - *99th percentile latency* in microseconds

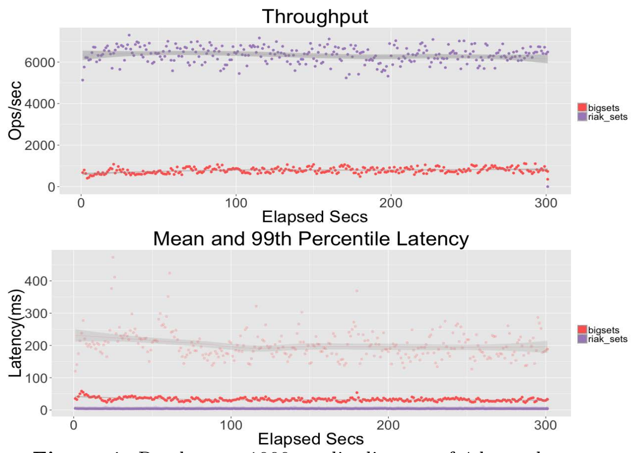

Figure 4: Reads on a 1000-cardinality set of 4-byte elements.

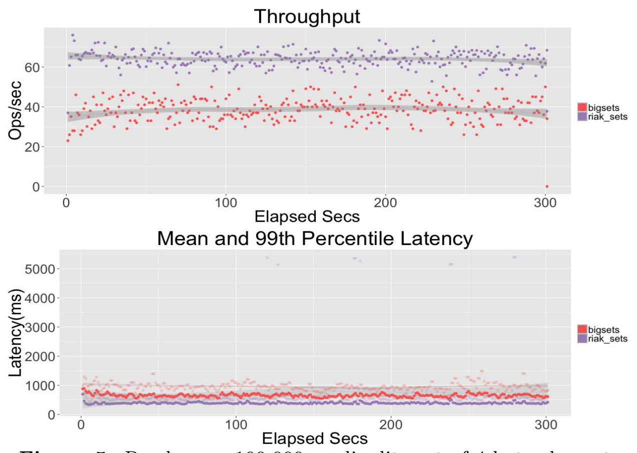

Figure 5: Reads on a 100,000-cardinality set of 4-byte elements.

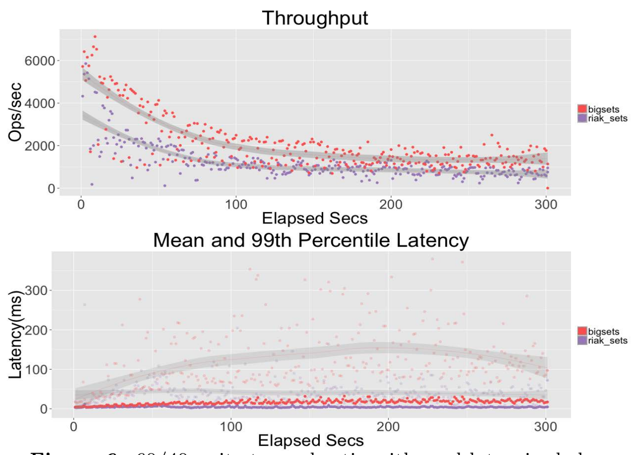

Figure 6: 60/40 write-to-read ratio with read-latencies below.