Improving Logical Clocks in Riak with Dotted Version Vectors: A Case Study

Ricardo Gonçalves

Universidade do Minho,
Braga, Portugal,
`tome@di.uminho.pt`

Abstract. Major web applications need the partition-tolerance and availability of the CAP theorem for scalability purposes, thus some adopt the eventual consistent model, which sacrifices consistency. These systems must handle data divergence and conflicts that have to be carefully accounted for. Some systems have tried to use classic Version Vectors to track causality, but these reveal either scalability problems or loss of accuracy if pruning is used to prevent growth.
Dotted Version Vectors is a mechanism that deals with data versioning in eventual consistent systems, which allows accurate causality tracking and scalability, both in the number of clients and servers, while limiting vector size to replication degree.
However, theories can abstract too much of the hiding properties which difficult the implementation. We discuss the challenges faced when implementing Dotted Version Vectors in Riak - a distributed key-value database -, evaluate its behavior and performance, discuss the tradeoffs made and provide further optimizations.

Keywords: Databases, Causality, Eventual Consistency, Logical Clocks, Scalability, NoSQL, Riak

1 Introduction

There is a new generation of databases on the rise, which are rapidly gaining popularity due to increasing scalability concerns. Typically these distributed systems have restrictions placed in *Consistency*, *Availability* and *Partition-Tolerance* of the CAP theorem [2]. They were grouped in a new broad class of databases called NoSQL (Not Only SQL). Examples of databases are Google's BigTable, Amazon's Dynamo, Apache's Cassandra (based on Facebook's version) and Basho's Riak [1]. Instead of providing the ACID properties, they focus on implementing what it is called a BASE (Basically Available, Soft state, Eventually consistent) system [9]. A BASE system has weaker consistency model, focuses on availability, uses optimistic replication, thus makes it faster and easier to manage large amounts of data, while scaling horizontally. As the CAP theorem says, we can only have two of the three properties that it describes. Since virtually all

[1] `http://wiki.basho.com/Riak.html`

large-scale systems must have partition tolerance, to account for hardware and networks faults, NoSQL databases focus on the *CP* for strong consistency, *AP* for high availability, or adjustable systems.

Systems like Riak or Cassandra [5] adopt an Eventual Consistency model that sacrifices data consistency to achieve high availability and partition tolerance. This means that eventually, all system nodes will be consistent, but that might not be the true at any given time. In such systems, optimistic/aggressive replication is used to allow users to both successfully retrieve and write data from replicas, even if not all replicas are available. By relaxing the consistency level, inconsistencies are bound to occur, which have to be detected with minimum overhead. This is where Logical Clocks, introduced by Lamport [6], are useful. Logical clocks are mechanisms for tracking causality in distributed systems. Causality is the relationship between two events, where one *could* be the consequence of the other (cause-effect) [7,4]. Due to constraints in global clocks and shared memory in distributed systems, these mechanisms are used for capturing causality, thus partial ordering events. It is partial because sometimes two events cannot be ordered, in which case they are considered concurrent. While some systems have tried to use classic Version Vectors (VV) [1] to track causality, they do not scale well or lose accuracy by pruning the vector to prevent its growth. Dotted Version Vectors (DVV) [8] is a novel logical clock mechanism, that allows both accurate causality tracking and scalability both in the number of clients and servers, while limiting vector size to replication degree.

From a theoretical and abstract point of view, DVV are an evolution of VV, more scalable and flexible. But in engineering, not everything that seems conceptually better, translates into better results. The main purpose of this paper is to implement DVV in a real database, proving that it can be done without major architectural changes. Moreover, we then evaluate this new implementation with the original database and discuss their tradeoffs.

In Section 2 we provide a background in Riak and DVV that is necessary to better comprehend this paper. We explain the basics of Riak, its current logical clock implementation, the classical alternatives, their shortcomings, and finally we describe briefly the DVV mechanism. Next, in Section 3 we present the major changes that had to be done, while implementing DVV in Riak. Section 4 is where the evaluation and benchmarking of this implementation can be found. Finally, we provide conclusion and future work in Section 5.

2 Background

To understand the state of logical clocks in Riak, let us describe its basic components in this context, namely replication, system interface and data versioning. We discuss the current logical clock mechanism (VV), its shortcomings and we then briefly present an alternative: DVV - we describe in which way they differ from traditional VV.

2.1 Logical Clocks in Riak

Riak[2] is developed by Basho Technologies and is heavily influenced by Eric Brewer's CAP Theorem and Amazon's Dynamo [3]. Written mostly in Erlang, with a small amount of Javascript and C, it is a decentralized, fault-tolerant key-value store, with special orientation to document storage. Being heavily influenced by Dynamo, Riak adopts the majority of its key concepts. Being a truly fault-tolerant system, it has no single point of failure, since no machine is special or central. Next, a brief description of some relevant Riak areas.

Replication Inspired by Dynamo, Riak uses the same ring concept for replication. Consistent hashing is used to distribute and organize data. This Riak ring has a 160-bit space size, and by default has 64 partitions, each represented by virtual nodes (vnodes). Vnodes are what manages client's requests, like puts e gets. Each physical node can have several vnodes, depending both on the number of partitions the ring has and the number of physical nodes. The average number of vnodes per node can be calculated by (number of partitions)/(number of nodes). Vnodes positions in the ring are attributed at random intervals, to attempt a more evenly distribution of data across the ring. By default, the replication factor (n_val) is 3 (i.e., 3 replica vnodes per key). Also, the number of successful reads (R) and writes (W) are by default a quorum number (greater than $n\_val/2$), but can be configured depending on the consistency and availability requirement levels.

System Interface In Riak, all requests are perform over HTTP by RESTful Web Services. All requests should include the *X-Riak-ClientId* header, which can be any string that uniquely identifies the client, to track object modifications with Version Vectors (VV). Additionally, every GET request provides a context, that is meant the be given back (unmodified) in a subsequent PUT on that key. A context contains the VV.

Data Versioning Riak has two ways of resolving update conflicts on Riak objects. Riak can allow the most recent update to automatically "win" (using timestamps) or Riak can return/store (depends if it is a GET or PUT, respectively) all versions of the object. The latter gives the client the opportunity to resolve the conflict on its own. This occurs when the *allow_mult* is set to true in the bucket properties. When this property is set to false, there is a silent loss update possibility, e.g., when two clients write to the same key concurrently (almost at the same time), one of the updates is going to be discard. Hereafter, assume that *allow_mult* is always true.

Lets describe two fundamental concepts in Riak:

- Riak Object: A riak object represents the *value* in the *key-value* tuple, i.e., it contains things like metadata, the value(s) itself, the key, the logical clock, and so on. So, from now on, object is the riak object and value or values

[2] The description and tests performed on this paper are based on Riak version 0.13

are the actual data of an object that being stored, e.g., a text, a binary, an image, etc.
- Sibling: A sibling (concurrent object) is created when Riak is unable to resolve the request automatically. There are two scenarios that will create siblings inside of a single object:
  - A client writes a new object that did not come from the current local object (it is not a descendent), conflicting with the local object;
  - A client writes a new object *without* context, i.e., without a clock.

Riak uses VV to track versions of data. This is required since any node is able to receive any request, even if not replica for that key, and not every replica needs to participate (being later synchronized via read-repair or gossiping). When a new object is stored in Riak, a new VV is created and associated with it. Then, for each update, VV is incremented so that Riak can later compare two object versions and conclude:

- One object is a direct descendant of the other.
- The objects are unrelated in recent heritage (the client clock is not a descendent of the server clock), thus considered concurrent. Both values are stored in the resulting object, while both VV are merged.

Using this knowledge, Riak can possibly auto-repair out-of-sync data, or at least provide a client with an opportunity to reconcile divergent objects in an application specific manner.

Riak's implementation of VV tracks updates done by clients instead of tracking updates "written" or handle by nodes. Both are viable options, but what Riak's approach provides is what clients updated the object (and how many times), in contrast to what nodes updated the object (and how many times). Specifically, VV are a list of number of updates made per client (using *X-Riak-ClientId*), like this: $[\{client1, 3\}, \{client2, 1\}, \{client3, 2\}]$. This VV would indicate that client1 updated the object 3 times, client2 updated the object 1 time, and client3 updated the object 2 times. Timestamp data is also stored in the VV but omitted from the example for simplicity. The reason to use client IDs instead of server-side IDs in VV, is because the latter can cause silent update losses, when two or more clients concurrently update the same object on the same node [8].

There is a major difference between this and the traditional approach of using using the node's IDs, because the number of clients tends to be much greater than the actual number of nodes or replicas. Therefore, the VV would grew in size much faster, probably in an unacceptable way (both size and performance). The solution Riak adopted was to prune VV as they grow too big (or to old), by removing the oldest (timestamp wise) information. The size target for pruning is adjustable, but by default it is between 20 and 50, depending on data freshness. Removing information will not cause data loss, but it can create false conflicts. For example, when a client holds an object with an old unpruned VV and submits it to the server, where the clock was pruned, thus creating conflict, where it should not have happened.

In short, the tradeoff is this: prune to keep the size manageable, thus letting false conflicts happen. The probability of false conflict happening should not be neglected. For example, by having a large number of clients interacting with a specific object, VV can rapidly grow, thus forcing the pruning. This can lead to cases where clients have to solve false conflicts, which could be later resolved in a not so correct way, i.e., if the value that "wins", is in fact the one that would have been removed if pruning was not applied.

2.2 Dotted Version Vectors

While pruning clocks creates false conflicts, DVV prevents unbounded size growth using server-side IDs, thus eliminating the need to prune old data. But we have also said that VV with server-side IDs can cause silent updates. DVV has the same structure as VV, but addresses conflicts in a different way, by having a special case in the vector, when concurrency occurs. If a conflict is detected, the pair that should be updated is transformed to a triple (figure 1 is an example of a possible clock increment when a conflict is detected), which conflicts with the server version. So, what this accomplishes is a representation of concurrency created by two or more writes in the same key on the same node. For more details on this, see the DVV paper [8].

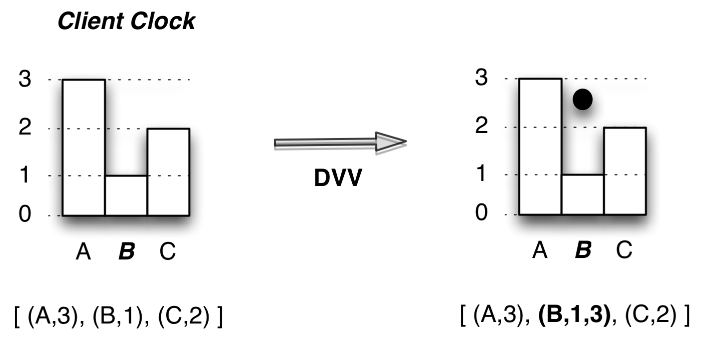

Fig. 1. A possible clock increment using DVV. Unlike VV, DVV allows non-contiguous increments in one ID per clock.

3 Implementing DVV in Riak

The first thing to be done was to implement DVV in Erlang, the programming language that Riak is written. It is a single file, that contains all the required functions. This file was placed in Riak Core, where are all the files that give core function to the system (e.g. Merkle Trees and Consistent Hashing). Then Riak KV (i.e. Riak Key-Value), which has the code for running Riak, was modified to use DDV instead of VV. This required some key changes to reflect the core differences between DVV and VV. One of them was eliminating *X-Riak-ClientId*, since we do not use the client ID anymore to update our clock. These are the main changes, in the following files:

riak_client Here we simply removed the line where the VV was previously incremented in a PUT operation.

riak_kv_put_fsm This file implements a finite-state machine, that encapsulates the PUT operation pipeline. In the initial state, we first see if the current node is a replica, and if not, we forward the request to some replica. Then, when a replica node is the coordinator, we execute the PUT operation locally first. When it is done, this replica provides the resulting object, with the updated clock and value(s), which is sent to the remaining replicas, where they synchronize their local object with this one.

riak_kv_vnode This is where the local put is done. A provided "flag" tells if this node is the coordinator, and thus the one that should do the update/sync to the clock. If this flag is false, the node will only sync the local DVV with the received one. Otherwise, this node is the coordinator, therefore it will run the *update* function with both new and local DVV, and the node ID. Then run the sync function with that resulting DVV and local DVV. Finally, the coordinator sends the results to replicas, but this time not as coordinators, thus they only run the sync function between their local object and the object provided.

riak_object This file encapsulates a Riak object, containing things like meta-data, data itself, the key, the clock, and so on. Before, an object only had one clock (one VV), even if there was more than one value (i.e. conflicting values). When conflicts were detected, both VV were merged so that there was only one new VV, which dominated both. This has an obvious disadvantage: the conflicting objects could only be resolved by a newer object. Even if by the gossip between replicas, we found that we could discard some of the conflicting values that were outdated, we could not. With DVV, we change this file so that each value has its own clock. By discarding this redundant values, we are actually saving space and simplifying the complexity of operations, since we manipulate smaller data. It worth noting that this approach to have set of clocks instead of a merged clock, could also be applied to VV. Since DVV was designed to work with set of clocks, it was mandatory to change this aspect, which introduces a little more complexity to the code, but has the advantages stated above.

4 Evaluation

In order to see if the performance was really affected, if there was savings in metadata space (smaller clocks) and if false conflicts were really gone, we had to evaluate this implementation. We executed performance benchmarks comparing the original version and the DVV version. Additionally, other metrics like clock size and conflicts, to provide some insight in what was happening and why. What follows is a description of the benchmark tool, the setup, the results and finally the tradeoffs.

4.1 Basho Bench

Basho Bench is a benchmarking tool created to conduct accurate and repeatable performance and stress tests. This tool outputs the throughput (i.e. total number of operations per second, over the duration of the test) and a range of latency metrics (i.e. 95th percentile, 99th percentile, 99.9th percentile, max, median and mean latency) for each operation. Basho Bench only requires one configuration file. The major parameters that were used are:

– Duration: 20 min;
– Number of concurrent clients: 500;
– Requests per client: 1;
– Types of requests and their relative proportions: various (detailed later);
– Key Space: [0-50000];
– Key Access: Pareto distribution, i.e. 20% of the keys accessed 80% of the time;
– Value Size: fixed 1KB or 5KB;
– Initial random seed was the same for all test, to ensure equal conditions to both mechanisms, while achieve reproducible results;
– Number of replies (R and W for the read and write operations) = 2.

4.2 Setup

For these benchmarks we used seven machines, all in the same local network. A Riak cluster running on 6 similar machines, while another independent machine was simulating the clients. The request rates and number of clients were chosen to try to prevent resource exhausting, since this would create unpredictable results. Resources were monitored to prevent saturation, namely CPU, disk I/O and network bandwidth. We also used the default replication factor $n\_val = 3$.

The following types of requests were issued from clients:

– GET: a simple read operation that returns the object of a given key;
– PUT: a *blind* write, where a value is written in a given key, with no causal context supplied, i.e. without a clock. This operation will increase concurrency (create siblings) if the given key already exists, since an empty clock does not dominate any clock, thus always conflicting with the local node clock;

– UPD: an update, that is expressed by a GET returning an object and a context (clock), followed by a 50 ms delay to simulate the latency between client and server, and finally a PUT that re-supplies the context and writes a new object, which supersedes the one first acquired in the GET. This operation reduces the possible concurrency (object with multiple values) that the GET brought.

From these three core actions we evaluated two benchmarks that considered different workload mixes. The first benchmark was to do a simple generic distribution load, with the proportion of blind puts kept at 10% and interchanged proportions of 30% versus 60% for gets and updates. The size per value was fixed at 1KB.

The second benchmark was to simulate TPC-W [10] workloads, using the "Shopping Mix" (80% reads, 20% writes) with a fixed value size of 5KB, the "Ordering Mix" (50% reads, 50% writes) and the "Browsing Mix" (95% reads, 5% writes), both with 1KB per value. Reads were done with the normal GET operation, while writes were done in UPD.

4.3 Comparison of overall latency

Workload	Clock Type	Get Mean (ms)	Get 95th (ms)	Put Mean (ms)	Put 95th (ms)	Update Mean (ms)	Update 95th (ms)	Clock Size (bytes)	Values per Key (average)
60% GET	VV	7.65	15.9	5.71	10.1	14.4	24.0	790	1.34
10% PUT	DVV	3.16	5.25	4.31	6.27	7.76	10.9	127	1.31
30% UPD	$\frac{DVV}{VV}$	**0.41**	**0.33**	**0.76**	**0.62**	**0.54**	**0.46**	**0.16**	**0.98**
30% GET	VV	10.4	21.6	7.48	13.8	18.8	31.9	859	1.20
10% PUT	DVV	3.45	5.83	4.56	6.59	8.39	11.8	123	1.16
60% UPD	$\frac{DVV}{VV}$	**0.33**	**0.27**	**0.61**	**0.48**	**0.45**	**0.37**	**0.14**	**0.97**

Table 1. DVV and VV benchmarks with a generic approach.

The first, generic, benchmark results are in table 1, while the TPC-W approach benchmark results are in table 2. Both tables show the DVV/VV ratio that helps compare the two mechanisms, values smaller than 1.0 show an improvement and are depicted in **bold**.

In all tests we find that clock size is always (much) smaller in DVV, even with the (default) pruning that occurs with Riak VV. One can also confirm that pruning is occurring, because all the tests reveal that there were more concurrent values in the VV case. The difference in the number of values per key between the two logical clocks, results from false conflicts created by pruning. We recall that since Riak VV resort to pruning they do not reliably represent concurrency, and introduce false conflicts that need to be resolved. Having no pruning, our DVV implementation accurately tracks concurrency, while still allowing an expressive reduction of metadata size. It is easy to see that even if the default pruning activation threshold was lowered in Riak VV case, although it would reduce clock sizes, this would also lead to an increase of false concurrency and higher numbers of values per key.

Regarding performance, the generic benchmark results show that using a value payload of 1KB, the write and read operations were much better then using VV. Having less conflicts, and factoring the smaller clock size, on average operations transfer smaller data (1.8KB versus 1.2KB).

Workload	Clock Type	Get Mean (ms)	Get 95th (ms)	Update Mean (ms)	Update 95th (ms)	Clock Size (bytes)	Values per Key (average)
Browsing Mix	VV	2.15	3.63	5.00	7.70	159	1.00081
	DVV	2.01	3.49	5.70	8.80	89	1.00051
	$\frac{DVV}{VV}$	**0.94**	**0.96**	1.13	1.15	**0.56**	**0.99970**
Shopping Mix	VV	2.84	5.00	6.80	11.0	117	1.00066
	DVV	2.77	4.94	7.70	12.8	82.0	1.00039
	$\frac{DVV}{VV}$	**0.98**	**0.99**	1.13	1.16	**0.70**	**0.99973**
Ordering Mix	VV	7.70	16.2	14.4	24.0	682	1.00549
	DVV	2.95	4.76	7.40	10.0	113	1.00425
	$\frac{DVV}{VV}$	**0.38**	**0.29**	**0.51**	**0.42**	**0.17**	**0.99877**

Table 2. DVV and VV benchmarks with TPC-W approach.

In the TPC-W case, the first thing we can see is that concurrency (rate of conflicts, measured by values per object) is very low, as it would be expected in a more realistic setting (concurrency rates in Dynamo's paper [3] are very similar to these). Read operations were always better, or pretty even between both mechanisms. This is to be expected since the read pipeline was not modified by our implementation, but DVV is usually smaller, thus requiring less data to be transferred.

Write operations were pretty good in the ordering mix, since (like the generic approach) each value was 1KB and the difference in clock size was significant. In contrast, the browsing mix also had 1KB per value, but the difference in clock sizes was not very large (too few writes in 20 minutes for the VV clock to grew significantly, but with time, it would probably grow much larger). Then, on average, values with VV and DVV had 1.16KB and 1.09KB in size, respectively. The same can be said of the shopping mix, in this case 5.12KB and 5.08KB for the VV and DVV, respectively. Therefore, in the shopping mix and browsing mix, the difference in clock size was not sufficient to make up for the changes we had to made in the write pipeline. Simply put, using DVV in Riak, when writing some value, the coordinator has to send every conflicting value to replicas. Moreover, if the coordinator is not a replica for that key, then it has to forward the write request to a new coordinator that is also a replica. In the standard

Riak implementation, the VV case, the write pipeline is simpler, only the new client value is passed to replicas and every node can be a coordinator for any write request.

4.4 Comparison of clock sizes

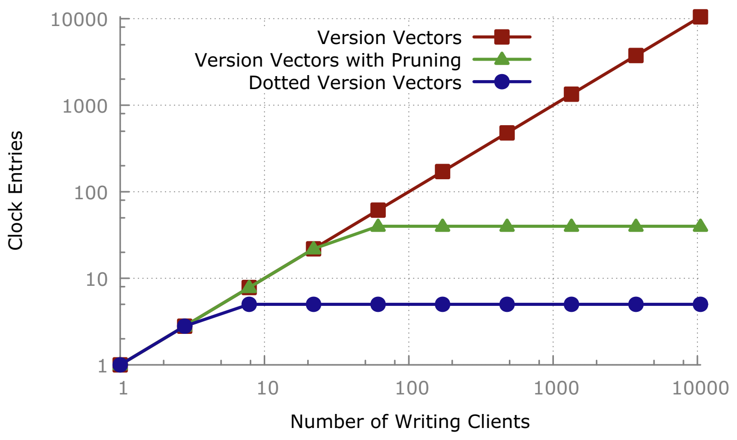

Fig. 2. Theoretical growth in clock size.

Figure 2 illustrates the theoretical effect of the number of writing clients in the number of clock entries, and thus the overall clock size. DVV stabilizes in size when the number of entries reaches the replication factor (usually 3), while VV with id-per-client grow indefinitely. Thus, in practice, systems usually resort to pruning to control its growth. In Riak's case, the imposed limit to the number of kept ids, and the trigger to pruning, is in the 20 to 50 range.

In Figure 3 we depict the evolution of the average clock size per key during the execution of the TPC-W based benchmarks in the Riak cluster. Here we can see that the DVV size becomes constant after a short time, whereas the Riak VV size is constantly increasing until stabilizing somewhere close to 1KB in the Ordering Mix case. Notice that it only stabilizes because of pruning in Riak VV, if not that it would grow linearly. The other workloads using Riak VV did not have enough time to stabilize, but would eventually be similar to the ordering mix. DVV has more or less 3 entries per clock (N=3) and size of 100 bytes, thus 1000 bytes in average for each VV means that it has roughly 30 entries, in line with the pruning range of Riak $[20 - 50]$.

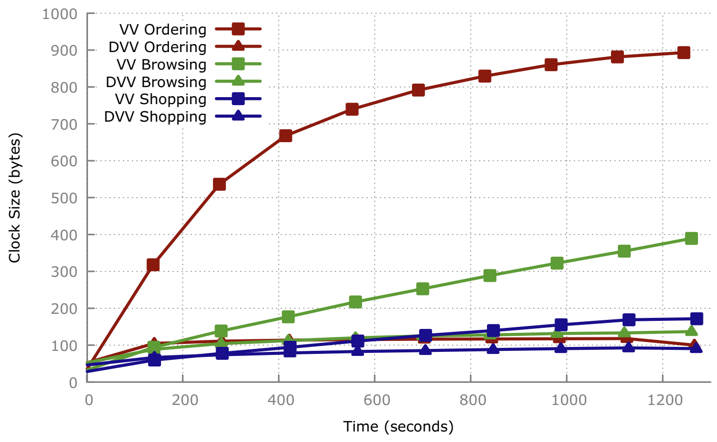

Fig. 3. Real growth in clock size using TPC-W workload mixes.

4.5 Tradeoffs

Lets resume the advantages and disadvantages of using DVV instead of VV. First, the advantages:

- **Simplify API**: since DVV uses the node's internal 160-bit ID, there is no need for clients to provide IDs, thus simplifying the API and avoiding potential ID collisions;
- **Save space**: DVV are bounded to the number of replicas, instead of the number of clients that have ever done a PUT. Since there is a small and stable number of replicas, the size of DVV would be much smaller than traditional VV;
- **Eliminates false conflicts**: clock pruning does not cause data loss, but it does cause false conflicts, where data that could be discard, is viewed as conflicting. Using DVV, the clock is bound to the number of replicas, therefore pruning is not necessary, thus eliminating false conflicts;

And now the disadvantages:

- **More complex write pipeline**: when a *non-replica* node receives a PUT request, it must forward it to a replica node. This overhead can be considerable if the transferred data is big. Which is even worse if the replica is not in the same network as the non-replica. Another thing that may affect negatively the performance is the fact that clock update and synchronization has to first be done in the coordinating replica, and then sent to the remaining replicas, whereas in VV the object goes directly to all replicas simultaneously. This is made worse when in the DVV case, the resulting object of the coordinating replica has siblings, which means that all siblings will be transferred to the remaining replicas. With VV, only the client object is sent to replicas.

5 Conclusions

Logical clocks in Riak are pruned when the number of entries exceeds some threshold, and consequently does not reliably represent concurrency, thus it introduces false conflicts. Having no pruning, DVV accurately tracks concurrency while still allowing an expressive reduction of metadata size. In terms of performance, the results showed that if we have many clients with high reads and low writes, DVV performs much better. On the other hand, fewer clients and more writes tend to mitigate DVV advantages, and in some cases it is worse than VV.

As future work, DVV performance should be addressed. The extra hop for non-replica nodes could be avoid, if we use a partition-aware client library or load balancer that knows which replica to communicate, thus reducing the response time. Another other problem: if the resulting replica coordinator's object has siblings, then it has to transfer all the siblings to the others replicas. This can be somewhat minimized if use a simple LRU cache to store keys and the corresponding clock. Since we often do not have conflicts, the coordinator can check first if the client object is more recent than the local one using the cache. If it is, we can send immediately the client object to all replicas before writing locally.

References

1. Almeida, P., Baquero, C., Fonte, V.: Version stamps-decentralized version vectors. In: Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on. pp. 544 – 551 (2002)
2. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. p. 7. ACM, New York, NY, USA (2000)
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. ACM, New York, NY, USA (2007)
4. Fidge, C.J.: Partial orders for parallel debugging. In: Workshop on Parallel and Distributed Debugging. pp. 183–194 (1988)
5. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (April 2010)
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (July 1978)
7. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms. pp. 215–226. North-Holland (1989)
8. Preguiça, N.M., Baquero, C., Almeida, P.S., Fonte, V., Gonçalves, R.: Dotted version vectors: Logical clocks for optimistic replication. CoRR abs/1011.5808 (2010)
9. Pritchett, D.: BASE: An acid alternative. ACM Queue 6(3), 48–55 (2008)
10. (TPC)., T.P.P.C.: Tpc benchmark w(web commerce) specification version 1.8 (2002)