Evaluating Dotted Version Vectors in Riak*

Ricardo Gonçalves[1], Paulo Sérgio Almeida[1], Carlos Baquero[1], Victor Fonte[1], and Nuno Preguiça[2]

[1] Universidade do Minho,
{tome,psa,cbm,vff}@di.uminho.pt,
[2] Universidade Nova de Lisboa,
nmp@di.fct.unl.pt

Abstract. The NoSQL movement is rapidly increasing in importance, acceptance and usage in major (web) applications, that need the partition-tolerance and availability of the CAP theorem for scalability purposes, thus sacrificing the consistency side. With this approach, paradigms such as Eventual Consistency became more widespread. An eventual consistent system must handle data divergence and conflicts, that have to be carefully accounted for. Some systems have tried to use classic Version Vectors (VV) to track causality, but these reveal either scalability problems or loss of accuracy (when pruning is used to prevent vector growth). Dotted Version Vectors (DVV) is a novel mechanism for dealing with data versioning in eventual consistent systems, that allows both accurate causality tracking and scalability both in the number of clients and servers, while limiting vector size to replication degree.
In this paper we describe briefly the challenges faced when incorporating DVV in Riak (a distributed key-value store), evaluate its behavior and performance, and discuss the advantages and disadvantages of this specific implementation.

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

1 Introduction

There is a new generation of databases on the rise, which are rapidly gaining popularity due to increasing scalability concerns. Typically these are distributed systems where restrictions are placed in C, A and P of the CAP theorem [1]. 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. Instead of providing the ACID properties, they focus on implementing what it is called a BASE (Basically Available, Soft State, Eventually consistent) system [6]. A BASE system has weaker consistency model, focuses on availability, uses optimistic replication

* Funded by project CASTOR – Causality Tracking for Optimistic Replication in Dynamic Distributed Systems

and because of all of this, it is faster and easier to manage large amounts of data, while scaling horizontally. Systems like Riak or Cassandra 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 [3], 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) [4,2]. Due to restraints 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) to track causality, they do not scale well or lose accuracy by pruning the vector to prevent its growth. Dotted Version Vectors (DVV) [5] 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. In Section 2 we present the major changes that had to be done to implementing DVV in Riak. Section 3 is where the evaluation and benchmark this implementation is presented. Finally, conclusions and future work are in Section 4.

2 Implementing DVV in Riak

The first thing to do was to implement DVV in Erlang. After that, the file was simply integrated in Riak's Core files. Next, "Riak KV" module 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. Next are the main changes:

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 metadata, 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.

3 Evaluation

Lets resume the advantages and disadvantages - in theory - of using DVV instead of VV:

– **Simplify API**: since DVV uses the node 160-bit index as 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.
– **Possible worse performance**: 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. 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.

Also, in the DVV case, the resulting object of the coordinating replica could have siblings, which would worsen its transfer performance to the remaining replicas. With VV, only the new client object is sent to replicas.

3.1 Setup

We used Basho Bench, a benchmarking tool created to conduct accurate and repeatable performance and stress tests. Seven machines in total were used, all in the same local network. A Riak cluster running with 6 similar machines, while another machine was simulating clients requests. 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 $n\_val = 3$, also an $R = W = 2$ (R and W the size of the read and write quorum). The key-space range was $[0 - 50000]$, accessed using a Pareto distribution (20% of the keys accessed 80% of the time). Each test took 30 minutes and the same initial random seed was used for all test, to ensure the same conditions.
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 would brought.

In terms of requests proportions GET/PUT/UPD, we present two scenarios: 30%/10%/60% (S316) and 60%/30%/10% (S631). For each, we ran two different combinations of stored value's size (VS), number of clients (NC) and number of request per client per second (NR): 1) VS = 1 KB, NC = 500 clients, NR = 3 req/s and 2) VS = 5 KB, NC = 250 clients, NR = 1 req/s.
Table 1 has the latency of both GET and PUT operations, the mean size of clock metadata and the mean number of values per object (Siblings), i.e., 1 is the minimum, which represents no siblings at all, thus no concurrency. For each metric, we provide de ratio between DVV and VV. From DVV perspective, higher is worse, while lower is better (bolded in the table).

3.2 Results

In all combinations we find that clock metadata size is always smaller in DVV, even with the (default size) pruning that occurs in VV. We also know that

Scenario		GET			PUT				
GET/PUT/UPD	Clock	Mean	Median	95th	Mean	Median	95th	Meta	Siblings
VS/NC/NR		(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(bytes)	(average)
60/30/10	VV	15.3	13.7	30.5	11.3	9.8	23.4	875	4.35
1/500/3	DVV	12.4	10.1	28.9	13.8	11.6	28.8	228	3.61
	DVV/VC	**0.81**	**0.74**	**0.95**	1.22	1.18	1.23	**0.26**	**0.83**
60/30/10	VV	9.57	7.60	24.3	6.08	4.24	16.1	449	3.06
5/250/1	DVV	9.97	7.76	26.1	11.4	9.22	27.1	183	2.98
	DVV/VC	1.04	1.02	1.07	1.88	2.17	1.68	**0.41**	**0.98**
30/10/60	VV	10.4	9.07	21.6	7.48	6.74	13.8	859	1.20
1/500/3	DVV	3.45	3.14	5.83	4.56	4.29	6.59	123	1.16
	DVV/VC	**0.33**	**0.35**	**0.27**	**0.61**	**0.64**	**0.48**	**0.14**	**0.97**
30/10/60	VV	4.72	4.35	9.15	3.98	3.50	7.64	458	1.20
5/250/1	DVV	3.92	3.56	7.42	5.50	5.08	9.97	110	1.15
	DVV/VC	**0.83**	**0.82**	**0.81**	1.38	1.45	1.30	**0.24**	**0.95**

Table 1. Scenario 3: value size 5KB, 250 clients, each with 1 req/s.

pruning is occurring, because the majority of tests reveal that there were more siblings in the VV case, when compared with same DVV run. This means that, the major difference between the siblings average results from false conflicts created by pruning. Therefore, we can conclude that, indeed DVV is smaller than VV, while preventing false conflicts from happening. Even if the default size pruning was lowered, metadata would be smaller, but false conflicts rate would rise.
In terms of performance, things are a bit more complex. First, in the S631 scenarios, DVV performance was almost always worse than VV. These scenarios have 30% writes of new objects, which could generate siblings if that key already had an object. If we look at the siblings average in this case, we see an absurd number of concurrency happening. Since only a UPD can resolve and simplify concurrency, it is obvious why this is the use case where most concurrency occurs. This in a rather extreme and unrealistic use case, but shows that DVV suffers in fact from the problem of the extra hop to a replica, and also suffers from the fact that it has to send all the possible siblings of the coordinator replica, to the other replicas. Since the number of siblings is extremely high, this problem is only augmented in DVV case.
Scenarios S316 are a bit more positive in terms of performance, and only strengthens the previous conclusions. Being update-heavy and having little new writes, we can see that performance in some cases for DVV, is actually better than VV, even in some PUT cases. Having less siblings and smaller clock metadata on average, operations transfer smaller data. Thus GET operations tend to perform faster, since they have the same protocol in both VV and DVV, but in DVV case, data is smaller. The write speed can also benefit from DVV when the data is small, like the scenarios where values have 1KB in size. Since the major thing that harms DVV performance is transferring objects between replicas in writes,

having a small value relatively to the clock size, can be enough to actually gain performance even in write operations. As we can see, when the data size is bigger like the 5KB scenarios, savings on metadata and number of siblings are not enough for writes performance to be better than VV case.

4 Conclusions

The Riak implementation of VV resorts to pruning, when the number of entries exceeds some threshold, and consequently does not reliably represent concurrency, introducing false conflicts. Having no pruning, our DVVs implementation accurately tracks concurrency while still allowing an expressive reduction of metadata size. In general, the bigger the number of clients interacting with the system, and the more read-heavy the system, the better DVV would compare to VV. On the contrary, the bigger the object size, the worse DVV would compare to VV.
As future work, the two main things that worsens DVV performance should be address. 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. The other problem is when the resulting object of the coordinator has siblings, thus having to transfer all the siblings to the others replicas. This can be somewhat minimized if we adopt an optimistic approach, by transferring only the client value and the new coordinator's DVV (updated and synchronized). If the replicas do not have the values that should have been transferred (this would be checked using DVV), it would later synchronize through gossiping or read repair. This approach could also create a bit of an overhead on the server side, but that remains to be tested.

References

1. E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA, 2000. ACM.
2. C. J. Fidge. Partial orders for parallel debugging. In *Workshop on Parallel and Distributed Debugging*, pages 183–194, 1988.
3. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
4. F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
5. N. M. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *CoRR*, abs/1011.5808, 2010.
6. D. Pritchett. BASE: An acid alternative. *ACM Queue*, 6(3):48–55, 2008.