# Strengthening Consistency in the Cassandra Distributed Key-Value Store

Panagiotis Garefalakis, Panagiotis Papadopoulos,
Ioannis Manousakis, and Kostas Magoutis

Institute of Computer Science
Foundation for Research and Technology-Hellas
Heraklion GR-70013, Greece
{pgaref,panpap,jmanous,magoutis}@ics.forth.gr

**Abstract.** Distributed highly-available key-value stores have emerged as important building blocks for applications handling large amounts of data. The Apache Cassandra system is one such popular store combining a key distribution mechanism based on consistent hashing with eventually-consistent data replication and membership mechanisms. Cassandra fits well applications that share its semantics but is a poor choice for traditional applications that require strong data consistency. In this work we strengthen the consistency of Cassandra through the use of appropriate components: the Oracle Berkeley DB Java Edition High Availability storage engine for data replication and a replicated directory for maintaining membership information. The first component ensures that data replicas remain consistent despite failures. The second component simplifies Cassandra's membership, improving its consistency and availability. In this short note we argue that the resulting system fits a wider range of applications, and is more robust and easier to reason about.

## 1 Introduction

The ability to perform large-scale data analytics over huge data sets has in the past decade proved to be a competitive advantage in a wide range of industries (retail, telecom, defence, etc.). In response to this trend, the research community and the IT industry have proposed a number of platforms to facilitate large-scale data analytics. Such platforms include a new class of databases, often referred to as NoSQL data stores, which trade the expressive power and strong semantics of long established SQL databases for the specialization, scalability, high availability, and often relaxed consistency of their simpler designs.

Companies such as Amazon [1] and Google [2] and open-source communities such as Apache [3] have adopted and advanced this trend. Many of these systems achieve availability and fault-tolerance through data replication. Google's BigTable [2] is an early approach that helped define the space of NoSQL *key-value* data stores. Amazon's Dynamo [1] is another approach that offered an eventually consistent replication mechanism with tunable consistency levels. Dynamo's open-source variant Cassandra [3] combined Dynamo's consistency mechanisms

J. Dowling and F. Taïani (Eds.): DAIS 2013, LNCS 7891, pp. 193–198, 2013.
© IFIP International Federation for Information Processing 2013

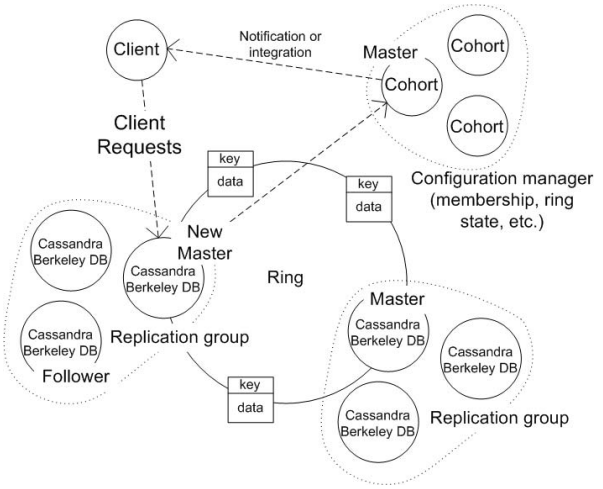


**Fig. 1.** System architecture

with a BigTable-like data schema. Cassandra uses consistent hashing to ensure a good distribution of key ranges (data partitions, or *shards*) to storage nodes.

Cassandra works well with applications that share its relaxed semantics (such as maintaining customer carts in online stores [1]) but is not a good fit for more traditional applications requiring strong consistency. We recently decided to embark on a re-design of Cassandra that preserves some of its features (such as its data partitioning based on consistent hashing) but replaces others with the aim of strengthening consistency. Our design does not utilize multiple masters on concurrent updates to a shard or techniques such as hinted handoff [1]. Instead, service availability requires that a single master per shard (part of a self-organized replication group) be available and its identity known to I/O coordinators. We reduce intervals of unavailability by aggressively publishing configuration updates. Furthermore we improve performance by using client-coordinated I/O, avoiding a forwarding step in Cassandra's original I/O path. In summary, our re-design centers on:

- Replacing Cassandra's data replication mechanism with the highly available Oracle Berkeley DB Java Edition (JE) High Availability (HA) key-value storage engine (hereafter abbreviated as BDB). Our design simplifies Cassandra while at the same time it strengthens its data consistency guarantees.
- Enhancing Cassandra's membership protocol with a highly available Paxos-based directory accessible to clients. In this way, replica group reconfigurations are rapidly propagated to clients, reducing periods of unavailability.

The resulting system is simpler to reason about and backwards-compatible with original Cassandra applications. While we expect that dropping the eventual consistency model may result in reduced availability in certain cases, we try to make up by focusing on reducing recovery time of the I/O path after a failure.

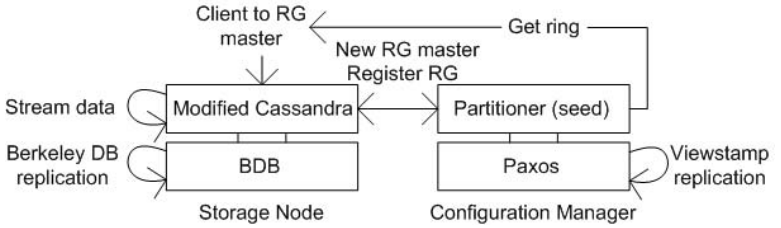


**Fig. 2.** System components and their interactions

The rest of the paper is organized as follows: In Section 2 we describe the overall design and in Section 3 we provide details of our implementation and preliminary results. In Section 4 we describe related work and in Section 5 directions of ongoing and future work. Finally in section 6 we conclude.

## 2 Design

Our system architecture is depicted in Figure 1. We preserve the Thrift-based client API for compatibility with existing Cassandra applications. We also maintain Cassandra's ring-based consistent hashing mechanism (where keys and storage nodes both map onto a circular ring [1]) but modify it to map each key to a BDB replication group (RG) instead of a single node. BDB implements a B+-tree indexed key-value store via master-based replication of a transaction log, using Paxos for reconfiguration. In our setup, all accesses go through the master (ensuring order) while writes are considered durable when in memory and acknowledged by all replicas. Periodically, replicas flush their memory buffers to disk. These settings offer a strong level of consistency with a slightly weaker (but sufficient for practical purposes) notion of durability [4].

Each node in an RG runs a software stack comprising a modified Cassandra with an embedded BDB (left of Figure 2). On a master, Cassandra is active and serves read/write requests; on a follower, Cassandra is inactive until elected master (election is performed by BDB and its result communicated to Cassandra via an upcall). The ring state is stored on a Configuration Manager (or CM, right of Figure 2). The CM complements Cassandra's original metadata service which uses a gossip-based protocol [3]. It combines a *partitioner* (a module that chooses tokens for new RGs on the ring) with a primary-backup viewstamp replication [5] scheme where a group of nodes (termed *cohorts*) exchange state updates over the network. The CM can be thought of as a highly-available alternative to Cassandra's *seed* nodes. It contains information about all RGs, such as addresses and status (master or follower), and corresponding tokens. Any change in the status of RGs (new RG inserted in the ring or existing RG changes master) is reported to the CM via RPC. The CM is queried by clients to identify the current master of an RG (by token).

We improve data consistency over original Cassandra by prohibiting multi-master updates. For a client to successfully issue an I/O operation, it must have

access to the master node of the corresponding RG. Causes of unavailability include RG reconfiguration actions after failures and delays in the new ring state propagating to clients. Our implementation supports faster client updates by either eager notifications by the CM [6] or by integrating with the CM. Additionally, clients can explicitly request RG reconfiguration actions if they suspect partial failure (i.e., a master visible to the RG but not to the client).

Our partitioner subdivides the ring to a fixed number of key ranges and assigns node tokens to key-range boundaries. This method has previously been shown to exhibit advantages over alternative approaches [1]. Each key range in our system corresponds to a different BDB database, the total number of key ranges on the ring being a configuration parameter. Finally, data movement (streaming) between storage nodes takes place when bootstrapping a new RG.

## 3 Implementation and Preliminary Results

Our implementation replaces the original Cassandra storage backend with Oracle Berkeley DB JE HA. One of the challenges we faced was bridging Cassandra's rich data model (involving column families, column qualifiers, and versions [3]) with BDB's simple key-value get/put interface where both key and value are opaque one-dimensional entities. Our first approach mapped each Cassandra cell (row key, column family, column qualifier) to a separate BDB entry by concatenating all row attributes into a larger unique key. The problem we faced with this model was the explosion in the number of BDB entries and the associated (indexing, lookup, etc.) overhead. Our second approach maps the Cassandra row-key to a BDB key (one-to-one) and stores in the BDB value a serialized HashMap of the column structure. Accessing a row requires a lookup for the row and subsequent lookup in the HashMap structure to locate the appropriate data cell. Our current implementation following this approach performs well in the general case, with the exception of frequent updates/appends to large rows (the entire row has to be retrieved, modified, then written back to BDB). This is a case where Cassandra's native no-overwrite storage backend is more efficient by writing the update directly to storage, avoiding the read-modify-write cycle.

Our Configuration Manager (CM) uses a specially developed Cassandra partitioner to maintain RG identities, master and follower IPs, RG tokens, and the key ranges on the ring. We decided to use actual rather than elastic IP addresses due to the long reassignment delays we observed with the latter on certain Cloud environments. Each RG stores its identifier and token in a special BDB table so that a newly elected RG master can retrieve it and identify itself to the CM. The CM exports two RPC APIs to storage nodes: *register/deregister RG*, *new master for RG*; and one to both storage nodes and clients: *get ring info*. The CM achieves high availability of the ring state via viewstamp replication [5, 7].

Preliminary results with the Yahoo Cloud Serving Benchmark (YCSB) over a cluster of six Cassandra nodes (single-replica RGs) on Flexiant VMs with 2 CPUs, 2GB memory, and a 20GB remotely-mounted disk indicate improvement by 26% and 30% in average response time and throughput respectively, compared

**Table 1.** YCSB read-only workload

| | Throughput (ops/sec) | Read latency (average, ms) | Read latency (99 percentile, ms) |
|---|---|---|---|
| Original Cassandra | 317 | 3.1 | 4 |
| Client-coordinated I/O | 412 | 2.3 | 3 |

to original Cassandra (Table 1 summarizes our results). This benefit is primarily due to client-coordination of requests. Our ongoing evaluation will further focus on system availability under failures and scalability with larger configurations.

## 4 Related Work

Our system is related to several existing distributed NoSQL key-vale stores [1–3] implementing a wide range of semantics, some of them using the Paxos algorithm [8] as a building block [6, 9, 10]. Most NoSQL systems rely on some form of relaxed consistency to maintain data replicas and reserve Paxos to the implementation of a global state module [9, 10] for storing infrequently updated configuration metadata or to provide a distributed lock service [6]. Exposing storage metadata information to clients has been proposed in the past [1, 9, 11], although the scalability of updates to that state has been a challenge.

Perhaps the closest approaches to ours are Scatter [12], ID-Replication [13], and Oracle's NoSQL database [11]. All these systems use consistent hashing and self-managing replication groups. Scatter and ID-Replication target planetary-scale rather than enterprise data services and thus focus more on system behavior under high churn than speed at which clients are notified of configuration changes. Just as we do, Oracle NoSQL leverages the Oracle Berkeley DB (BDB) JE HA storage engine and maintains information about data partitions and replica groups across all clients. A key difference with our system is that whereas Oracle NoSQL piggybacks state updates in response to data operations, our clients have direct access to ring state in the CM, receive immediate notification after failures, and can request reconfiguration actions if they suspect a partial failure. We are aware of an HA monitor component that helps Oracle NoSQL clients locate RG masters after a failure, but were unable to find detailed information on how it operates.

## 5 Future Work

Integrating the CM service into Cassandra clients (making each client a participant in the viewstamp replication protocol) raises scalability issues. We plan to investigate the scalability of our approach as well as the availability of the resulting system under a variety of scenarios. Another research challenge is in provisioning storage nodes for replication groups to be added to a growing cluster. Assuming that storage nodes come in the form of virtual machines (VMs) with local or remote storage on Cloud infrastructure, we need to ensure that nodes in an RG fail independently (easier to reason about in a private rather

than a public Cloud setting). Elasticity is another area we plan to focus on. A brute force approach of streaming a number of key ranges (databases) to a newly joining RG is a starting point but our focus will be on alternatives that exploit replication mechanisms [14].

## 6 Conclusions

In this short note we described a re-design of the Apache Cassandra NoSQL system aiming to strengthen its consistency while preserving its key distribution mechanism. Replacing its eventually-consistent replication protocol by the Oracle Berkeley DB JE HA component simplifies the system while making it applicable to a wider range of applications. A new membership protocol further increases system robustness. A first prototype of the system is ready for evaluation while the development of more advanced functionality is currently underway. This work was supported by the CumuloNimbo (FP7-257993) and PaaSage (FP7-317715) EU projects.

## References

1. DeCandia, G., et al.: Dynamo: Amazon's Highly Available Key-value Store. ACM SIGOPS Operating Systems Review 41, 205–220 (2007)
2. Chang, F., et al.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) 26, 4 (2008)
3. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review 44, 35 (2010)
4. Birman, K., et al.: Overcoming CAP with Consistent Soft-State Replication. IEEE Computer (45) 50–58
5. Mazieres, D.: Paxos Made Practical. Technical report (2007)
6. Burrows, M.: The Chubby Lock Service for Loosely-coupled Distributed Systems. In: Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA (2006)
7. Oki, B.: Liskov. B: Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In: Proc. of the 7th ACM Symposium on Principles of Distributed Domputing (PODC), Toronto, Canada (1988)
8. Lamport, L.: Paxos made simple. ACM SIGACT News 32, 18–25 (2001)
9. Lee, E., Thekkath, C.: Petal: Distributed Virtual Disks. ACM SIGOPS Operating Systems Review 30, 84–92 (1996)
10. MacCormick, J., et al.: Niobe: A Practical Replication Protocol. ACM Transactions on Storage (TOS) 3 (2008)
11. Oracle, Inc.: Oracle NoSQL Database: An Oracle White Paper (2011)
12. Glendenning, L.: et al.: Scalable Consistency in Scatter. In: Proc. of 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal (2011)
13. Shafaat, T.M., Ahmad, B., Haridi, S.: ID-Replication for Structured Peer-to-Peer Systems. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 364–376. Springer, Heidelberg (2012)
14. Lorch, J., et al.: The SMART Way to Migrate Replicated Stateful Services. In: Proc. of EuroSys 2006, Leuven, Belgium (2006)