ACaZoo: A Distributed Key-Value Store based on Replicated LSM-Trees

Panagiotis Garefalakis, Panagiotis Papadopoulos and Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology-Hellas (FORTH)
Heraklion GR-70013, Greece
Email: {pgaref,panpap,magoutis}@ics.forth.gr

***Abstract*—In this paper we describe the design and implementation of *ACaZoo*[1], a key-value store that combines strong consistency with high performance and high availability. *ACaZoo* supports the popular column-oriented data model of Apache Cassandra and HBase. It implements strongly-consistent data replication using primary-backup atomic broadcast of a write-ahead log, which records data mutations to a Log-structured Merge Tree (LSM-Tree). *ACaZoo* scales by horizontally partitioning the key space via consistent primary-key hashing on available replica groups (RGs). LSM-Tree compactions can hamper performance, especially when they take place at RG primaries. ACaZoo addresses this problem by changing RG leadership prior to heavy compactions, a method that can improve throughput by up to 40% in write-intensive workloads. We evaluate *ACaZoo* using the Yahoo Cloud Serving Benchmark (YCSB) and compare it to Oracle's NoSQL Database and to Cassandra providing serial consistency via an extension of the Paxos algorithm.**

I. 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 offers an eventually consistent replication mechanism with tunable consistency levels. Dynamo's open-source variants Cassandra [3] and Voldemort [4] combine Dynamo's consistency mechanisms with a BigTable-like data schema. These systems use consistent hashing to ensure a good distribution of key ranges (data partitions, or *shards*) to storage nodes.

Eventual consistency works well for applications that have relaxed semantics (such as maintaining customer carts in online stores [1]) but is not an option for a broad spectrum of applications that require strong consistency. When embarking on the *ACaZoo* project we decided to target strongly-consistent sharded data-intensive applications, a large and growing class of applications. One of our design goals for *ACaZoo* was to use a standard data model and cross-platform API. We thus opted for the Apache Cassandra/HBase column-oriented data model and the Cassandra Thrift-based API. Another design goal was to use a consistent-hashing based scheme for shard partitioning due to its simple implementation and lack of a centralized metadata service. Finally, we wanted to leverage a Log-structured Merge (LSM)-Tree [5] based storage backend due to its benefits over B+-tree organized schemes in organizing local storage, combined with a consistent replication scheme.

LSM-Trees is a particularly attractive indexing scheme used in a variety of systems (Bigtable [2], HBase [6], Cassandra [3], Hyperdex [7], PNUTS [8], etc.). In ACaZoo we decided to combine LSM-Trees with a strongly-consistent primary-backup (PB) scheme (ZAB [9], also found at the core of Apache ZooKeeper). A well known limitation of primary-backup schemes however is the requirement to go through a single master of the replica group (RG) for both read and write operations. Besides being a scalability limitation (which can be addressed by sharding over several RGs), performance can be hampered at times by periodic background activity at the master. Implementations of LSM-Trees are prone to such a challenge, especially under write-intensive workloads, as compaction operations aiming to merge data files into a smaller set drain server CPU and I/O resources.

In this paper we propose a solution to this problem that leverages the fact that compaction schedules of the nodes in a RG usually have little overlap (we experimentally validated this claim but could also enforce it if needed). Our solution ensures that the RG master is never a node that undergoes significant compaction activity. We achieve this by forcing a reconfiguration of an RG when the master is about to start a heavy compaction. Finally our solution ensures that the impact of reconfigurations on overall performance is low. This is achieved by rapidly propagating the change to clients, piggybacked as responses to standard RPCs. Our experiments show that compaction activity at the master hurts performance and that compaction-aware RG reconfiguration policies can lead to a significant performance improvement.

In *ACaZoo* we also apply an orthogonal optimization, client-coordination of I/O requests, as a means of reducing response time and relieving servers from forwarding I/O traffic.

[1] ACaZoo combines the names of the two systems it derives from: Apache Cassandra and Apache ZooKeeper.

This method has been investigated and has been shown to provide benefits in previous systems (Amazon Dynamo [1] and Petal [10] are but a few of them). The drawback of this scheme is the need to rapidly update a large and potentially dynamic client population. Configuration services such as ZooKeeper and Chubby can fill this need, operating in either *pull* (clients call into the service) or *push* (service notifies clients) mode.

Our key contributions in this paper are:

- A high-performance data replication primitive combining the ZAB [11] protocol with a single-node implementation of LSM-Trees [5]; while in principle similar to previous data replication approaches such as SMARTER [12] and BookKeeper [13], ACaZoo combines log replication with checkpointing using LSM-Trees [5] and addresses the associated challenges.
- A novel technique that addresses the impact of LSM-Tree compactions on write performance by forcing reconfigurations of RGs, changing leadership prior to heavy compactions at the master. Our experiments show that this technique can improve throughput by up to 40% in write-intensive workloads.

The rest of the paper is organized as follows: In Section II we relate ACaZoo to other work in the field. In Section III we describe the overall design and in Section IV we provide details of our implementation. In Section V we present the results of our system evaluation and in Section VI directions of ongoing and future work. Finally in section VII we conclude.

II. 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 [14] as a building block [10], [15], [16]. 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 [10], [16] for storing infrequently updated configuration metadata or to provide a distributed lock service [15]. Exposing storage metadata information to clients has been proposed in the past [1], [10], [17], although the scalability of updates to that state has been a challenge.

There have recently been several approaches to high performance replication within a local area network environment (a datacenter). SMARTER [12] implements a highly-available data store using an optimized Paxos-based replicated state machine and has demonstrated performance close to hardware limits and 12%-69% better than primary-backup versions. SMARTER replicates a periodically-checkpointed stream store whereas ACaZoo's storage backend and checkpointing mechanism is based on Cassandra's implementation of LSM Trees [5]. Cassandra recently (as of version 2.0) implemented a linearizable consistency mode using an extension of the Paxos protocol to reach consensus at each insert or update request [18]. As is validated by our experiments, this implementation (termed *Cassandra Serial*) incurs a significant performance penalty in write-intensive workloads.

A recent work that proposes explicit replication of LSM-Trees is Rose (Sears *et al.* [19]). Rose shares our goal of leveraging LSM-Trees' superior write performance in a replication context; they however focus more on the benefits of data compression and less on the performance impacts of compaction. Google's BigTable can be credited with bringing LSM-Trees to the forefront since their inception in 1996 [5] and Apache HBase for contributing an open source implementation, in both cases over a distributed replicated file system. While similar in principle, *ACaZoo* differs from these two systems in its explicit management of replication as opposed to implementing a serial LSM-Tree and relying on an underlying layer for replication.

Several systems have experimented with sharding and replication over open-source storage engines such as MySQL, including Google's F1 [20], LinkedIn's Espresso [21], and Facebook's TAO [22], with varying results. F1 faced a problem with rebalancing data partitions under expanded capacity (and eventually opted to use Spanner [23]), whereas Espresso attempted to address these problems by avoiding partition splits (by overpartitioning), mapping several partitions to a single MySQL instance, and modifying MySQL to be aware of partition identities on log updates. Espresso uses *timeline consistency* (asynchronous or semi-synchronous replication) whereas F1 is a strongly-consistent system (synchronous replication).

Perhaps the closest approaches to ours are Scatter [24], ID-Replication [25], and Oracle's NoSQL database [17]. 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. 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.

III. Design

The *ACaZoo* system architecture is depicted in Figure 1. The data model it supports, which derives from that of Apache Cassandra, HBase, and BigTable has the general structure shown in Figure 2. A unit of data (or *cell*) has the following coordinates: (*row key, column family name, column qualifier, version*). We use a consistent hashing mechanism [1] to map each row key to a replica group (RG) via the circular ring shown in Figure 1. Each RG is associated with a unique identifier that hashes to a point on the ring. Similarly each row key hashes to some point on the ring. A row key is assigned to the RG that maps nearest to it (clockwise) on the ring.

The state being replicated is a write-ahead log (WAL) recording mutations to a set of LSM-Trees as shown in Figure 3. The replication algorithm being used is a two-phase primary-backup atomic broadcast found at the core of Zookeeper [26] (ZooKeeper Atomic Broadcast or ZAB [11]), a distributed coordination service. All accesses go through the master, ensuring order. In terms of durability, *ACaZoo* supports

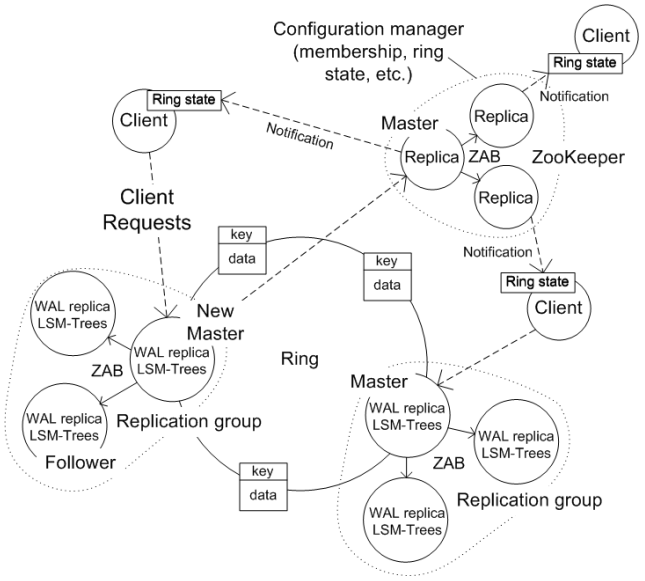

Fig. 1. The ACaZoo architecture

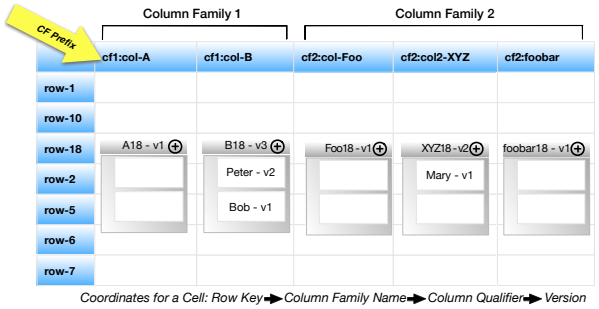

Fig. 2. ACaZoo/Cassandra data model

two modes of operation[2]: (1) writes considered durable when on disk and acknowledged by a quorum of replicas; or (2) writes considered durable when in memory and acknowledged by a quorum of replicas (while replicas periodically flush their memory buffers to disk). The second mode offers a strong level of consistency with a slightly weaker (but sufficient for practical purposes) notion of durability [27]. Once an update is committed, each node independently applies it to a memory buffer (the *memtable*) and periodically flushes it to an indexed file (the *SSTable*). SSTables should be periodically compacted (a task similar to a merge sort) to improve read performance.

A key feature of *ACaZoo* is its ability to monitor periodic node activities and to effect a change in leadership when a heavy upcoming compaction or other resource-intensive activity at a leader of an RG is expected to hamper performance of the entire group. In the current prototype, the number of SSTables awaiting compaction at a node is used as a measure of the intensity of an upcoming compaction at that node (Section IV-B provides implementation details). Figure 4 shows a 3-node RG whose leader is about to start a compaction (the proximity of a node to compaction is metaphorically

[2]Corresponding to the *batch* and *periodic* modes described in Section IV-B

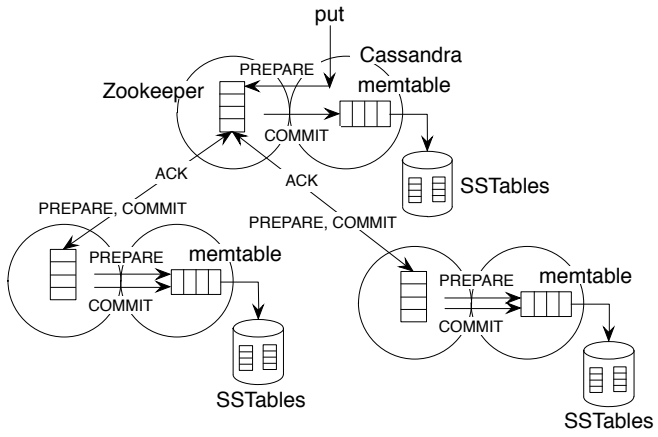

Fig. 3. ZAB-based replication of Cassandra's write-ahead log

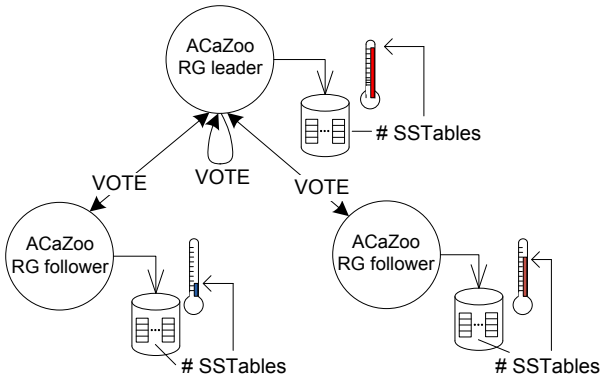

Fig. 4. An RG leader triggers an election when approaching compaction

depicted as the *temperature* of that node). At that point, the leader triggers an election. ACaZoo adapts the ZAB weighted-vote leader election algorithm as follows: Votes of nodes that undergo compactions carry zero weight. Nodes that are not undergoing a compaction, vote for themselves with a positive weight inversely proportional to the accumulated amount of compaction work. As a result, the node that is the furthest away from compaction is elected new leader of the RG.

The ring state is stored on a Configuration Manager (or CM) depicted in the upper right of Figure 1. The CM combines a *partitioner* (a module that chooses unique identifiers for new RGs on the ring) with a distributed coordination service (we use ZooKeeper/ZAB again here). The CM contains information about all RGs, such as addresses and status of nodes (master or follower), and corresponding identifiers. 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 clients can query the CM to identify the current master of an RG or ask to be notified of any changes.

IV. Implementation

The *ACaZoo* implementation forks off the Apache Cassandra NoSQL system. It preserves the Thrift-based Cassandra client API for compatibility with existing Cassandra applications but otherwise extends Cassandra in several important

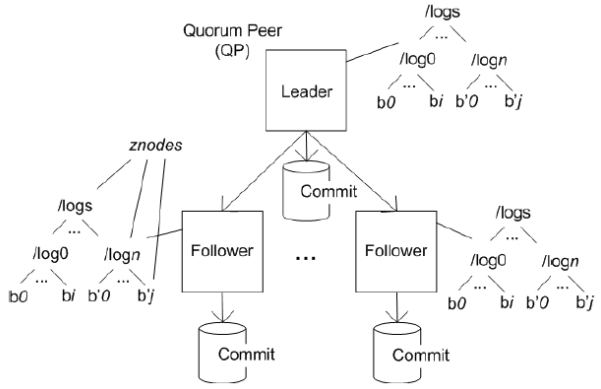

Fig. 5. ZooKeeper

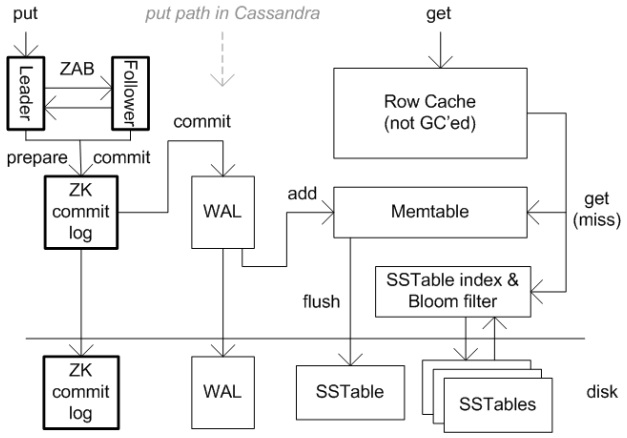

Fig. 6. ACaZoo storage server

ways: First, it replaces its eventually-consistent replication mechanism with a strongly consistent implementation using ZooKeeper ZAB [11] to replicate the LSM-Tree WAL. Second, *ACaZoo* addresses the performance impact of LSM-Tree compactions at RG leaders via appropriate reconfiguration actions. Third, it uses client-coordinated I/O in conjunction with a configuration management service. Section IV-A describes ZooKeeper at a high level, Section IV-B describes the internals of *ACaZoo*, and Section IV-C describes the ZooKeeper implementation of ZAB and optimizations we have applied to it.

A. ZooKeeper

Zookeeper implements a hierarchical namespace of fixed-size objects (referred to as *znodes*) accessed via a filesystem-like API. The Zookeeper namespace is organized as a hash-table memory structure kept consistent across a set of servers called Quorum Peers (QPs). The set of QPs is referred to as a cell. A cell is organized as a single leader and a number of followers as shown in Figure 5. Each request (otherwise known as a proposal) towards a *znode* corresponds to a transaction with a specific ID (referred to as a zxid) eventually committed into a Commit Log. Consistency is achieved via the ZAB atomic-broadcast protocol [11]. *Znodes* do not map to persistent locations on disk. Instead, the entire namespace is periodically serialized and snapshotted to disk. Zookeeper can thus recover *znode* state by loading the most recent snapshot and running its commit log, up to the most recent committed transaction. The total amount of data Zookeeper can store is bounded by the physical memory of the least-provisioned node in the system.

The leader is connected to each follower through direct FIFO channels (implemented as TCP streams) in a tree pattern. TCP connections between QPs in the chain are persistent across proposals. Each leader receives proposals from clients, moves them out of sockets, forwards them to all followers and then writes them to disk. Upon receiving a proposal, a follower writes it to disk and then acknowledges with the leader. The leader commits a proposal and responds successfully to clients only after it has received ACKs from a majority of followers. To avoid overload the leader uses an application-level flow control protocol (i.e., stop receiving data from TCP sockets) to throttle clients when the queue of outstanding (not yet committed) proposals exceeds a configurable threshold. A group commit protocol comes into action to avoid the cost of flushing dirty buffers to disk at each operation.

B. ACaZoo

The ACaZoo single-node storage backend (with the ACaZoo modifications over base Cassandra highlighted in bold) is depicted in Figure 6. We start by describing the put (write) path in base Cassandra (*"put path in Cassandra"* in Figure 6). A put is first recorded as a mutation in the WAL and then written to an ordered per-Column Family memory structure called a Memtable. When the Memtable is full, it is flushed to disk as an SSTable. Mutations represent changes to one or more tables (all belonging to the same keyspace) and refer to the same partition key. They are *deltas* rather than full representations of a cell. More information on the original Cassandra implementation can be found in Lakshman *et al.* [3].

An *ACaZoo* storage server extends base Cassandra by integrating with the core of a Zookeeper server (Figure 6, left) that is used to replicate the Memtable WAL across servers. The two modules interact at three internal interfaces: Cassandra invokes ZooKeeper's PrepRequestProcessor stage (described in Section IV-C), ZooKeeper calls Cassandra's *WAL insert* interface, and ZooKeeper notifies Cassandra of a leader election outcome. Both modules run as a single process (daemon) that starts and manages the ACaZoo storage server.

An *ACaZoo* replication group (RG) comprises a (configurable) number of such storage nodes. At startup of each RG, elections are being held to elect a leader node. The Cassandra code at that node is notified of the result and assumes the handling of read/write requests. The RG leader is responsible for invoking the ZAB agreement protocol on each put (upper left corner of Figure 6). Internally, ZAB takes the contents of a put operation (the mutation) and replicates it as the contents of a persistent *znode* treated as SEQUENTIAL (associated with a global sequence number appended to its name). As soon as the proposal is committed, the mutation is inserted into the ACaZoo WAL of all nodes that learn of the commitment and then applied to the local LSM-Trees. The *znode* and associated commit log entry can be erased as soon as the ACaZoo WAL acknowledges it. Reducing ZooKeeper state has the benefit that periodic snapshot operations are inexpensive.

ZooKeeper currently requires manual deletion of old snapshots and commit logs by an operator; this is something we intend to automate in ACaZoo.

Gets (reads) are handled by the leader, first looking up its cache and then (in case of a miss) its local LSM-Tree. Just as in Cassandra, *ACaZoo* uses SSTable indexes and Bloom Filters to reduce I/O overhead in case of a read miss. In a RG reconfiguration, a new leader is expected to start with a cold cache and therefore clients will experience the related warm-up phase. Note that this issue is intrinsic to any replication system where a single leader handles all read activity.

During early testing of our prototype we noticed that applying mutations in each replica's LSM-Tree was not sufficient to make the new state visible to clients. The reason was that metadata for these mutations were not being updated in the process. We solved the problem by forcing a reload of the local database schema on each put so that the schema gets rebuilt according to the current state. This was a lightweight fix that does not impact performance.

In terms of durability, ACaZoo (inheriting from Cassandra) can be configured for either *periodic* or *batch* writes to its commit WAL. Periodic (the default mode) initiates write I/Os as soon as they appear in the execution queue and acknowledges them immediately with clients. Another thread periodically (as specified in `commitlog_sync_period_in_ms`) enqueues a sync-to-disk operation on the execution queue. The batch mode offers stricter durability by grouping multiple mutations over a time window (defined by `commitlog_sync_batch_window_in_ms`) and executes them in a batch. After each batch, it performs a sync of the commit WAL and then acknowledges the writes with clients.

We have two optimizations in mind to further improve the I/O performance of our prototype. One optimization has to do with avoiding a redundant commit step in ZK and the Cassandra WAL. Another has to do with further integrating the prepare disk write with the WAL (so that the data is not written twice to disk). Based on our performance results we believe that these optimizations are not critical and defer their implementation to a future edition of our prototype.

Masking the impact of SSTable compactions

In *ACaZoo*, LSM-Tree compaction activities that are expected to impact a RG leader (and thus the entire RG) are detected and acted upon. Our implementation inherits Cassandra's compaction thresholds (by default a compaction is scheduled as a low priority task as soon as there are 4 same-sized SSTables but can be delayed –due to resource constraints– up to the point there are 32 such SSTables, when a compaction is forced). Our implementation intercepts compaction trigger events at RG leaders (disabling Cassandra's AutoCompaction feature) and performs a leader election (demoting it to a follower) before allowing compaction to proceed. Variability in node activities –especially under resource constraints– means that compactions are in effect unsynchronized across nodes.

In Section III we described the ACaZoo leader election process (also shown in Figure 4). We currently support two election policies: choosing a random non-compacting node excluding the current leader (RANDOM policy), or choosing the next non-compacting node in some sequence (round robin, RR policy). When the leader is nearing the point of having to perform a compaction, it sends an election event (a NEW_LEADER proposal in ZooKeeper terminology, Section IV-C) to followers. A new leader is elected as soon as a quorum commits to following him. Normally during leader election, ZooKeeper nodes broadcast their votes containing their current epoch, last transaction seen, and preferred leader. By default the preferred leader in that vote is themselves; nodes that are up to date have more chances to win. In *ACaZoo*, the node triggering the leader election is voting for someone else according to the policy (RANDOM or RR). *ACaZoo* followers that have a low anticipated compaction load respond positively to the leader election according to the policy.

During the election, which typically lasts between 200-500 ms the outgoing leader keeps accepting requests. When a new leader is elected, the previous leader returns a CUSTOM EXCEPTION with the identity of the new leader in RPCs. Thus the identity of the new leader is rapidly propagated to the requesting clients. An alternative path to learn of leadership changes is through ZooKeeper watch event notifications.

A caveat here is that frequent leadership changes can hurt performance. To avoid this situation, we estimate the compaction load of a server at all times (taking into account the amount of data that need to be compacted) and decide to trigger a leader election only when that load exceeds a configurable threshold (and thus expected to be a hard hit on performance).

Client-coordinated I/O and configuration management

ACaZoo clients are allowed to maintain ring state and thus be able to route operations appropriately rather than through a (possibly random) server. They obtain that state by connecting to the Configuration Manager (CM), the integration of a ZooKeeper cell and a Cassandra partitioner that jointly maintain RG identities and tokens, leader and follower IPs, and the key ranges on the ring. We decided to use actual IP addresses rather than elastic ones 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 Zookeeper *znode* directory so that a newly elected RG leader can retrieve it and identify itself to the CM.

In more detail, CM creates a ZooKeeper directory named `system_state` and a *znode* in it for each token range in the system. A new leader, aware of the token range it represents modifies the state of the corresponding *znode* to store information about its RG. We additionally store in `system_state` information about the RG followers and other system state to empower clients with the ability to take action as soon as possible when a failure occurs. The CM exports two RPC APIs to storage nodes: *register/deregister RG*, *new leader for RG*; and a *get ring info* API to both storage nodes and clients.

ACaZoo clients have the option to either use the exposed CM API or request notifications of any changes by setting *watches*[3] on the ZooKeeper `system_state` *znode*. The CM is responsible for maintaining this *znode* up to date by monitoring the ring state using a JMX API. The CM thus learns of a new RG joining the ring or a new leader election in an existing RG and triggers watch notifications.

[3]A *watch* is a request by a client to receive a notification by a ZooKeeper server should there be any modification to the referenced *znode*.

C. ZooKeeper data path and optimizations

This section provides insight to the internals of the ZooKeeper server system architecture. While important for understanding the operation of the ZAB protocol, its efficiency, and the integration of ZooKeeper into ACaZoo (Section IV-B), this is not prerequisite reading and the reader can skip to Section V. ZooKeeper has a staged event-driven architecture (Figure 7). Each QP is structured as a sequence of Request Processors that operate in a pipelined fashion. Each RP is associated with a thread that performs operations on its input passing over the result to the next RP via a shared FIFO queue. The requests are passed by reference through the queues.

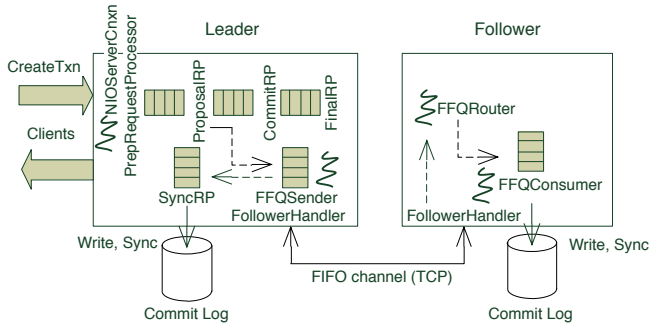

Fig. 7. ZooKeeper data path

The lifecycle of a proposal comprises the following steps: The client sends a request to the leader by invoking the CreateTxn API. A thread (NIOServerCnxn) allocates a new buffer in which it copies the proposal's data payload. The leader may decide to throttle clients if it is running low on the number of available pre-allocated buffers. The leader queues the request into the FIFO queue of the PrepRequestProcessor stage. The PrepareRequestProcessor encapsulates the client request into a Quorum Packet and queues it into the queue of ProposalRP which run in the same context. ProposalRP will broadcast the quorum packet using another thread (FFQSender), which handles message transmission to all peers. The proposal is then passed to SyncRP, which appends it to the Commit Log. SyncRP periodically flushes written proposals to the disk using group commits. SyncRP passes the proposal on to CommitRP, which is responsible for counting ACK messages sent by QPs for this proposal. QPs may ACK a proposal only after they ensure that it has been successfully flushed to disk. When the leader receives a majority of ACKs it sends a COMMIT message to all QPs over the chain and the proposal is applied on receiving the COMMIT message. When a proposal is committed, the FinalRP stage sends a reply back to the client.

Followers have a simple structure in which a FFQRouter thread performs equivalent tasks to the leader's FFQSender. FFQRouter is spawned when the follower receives a NEW_LEADER message and is responsible for listening for new incoming TCP connections. It passes its output to the FFQConsumer thread, which decides if this message should be consumed or not based on its FollowerEpoch field. If the message is from a previous epoch (i.e., stale), it rejects it, otherwise it consumes it. If the message is a PROPOSAL it appends it to the Commit Log and then sends an ACK message to the leader. If the message belongs to any other category it appends it to the commit log and performs the appropriate proposal-specific action to it.

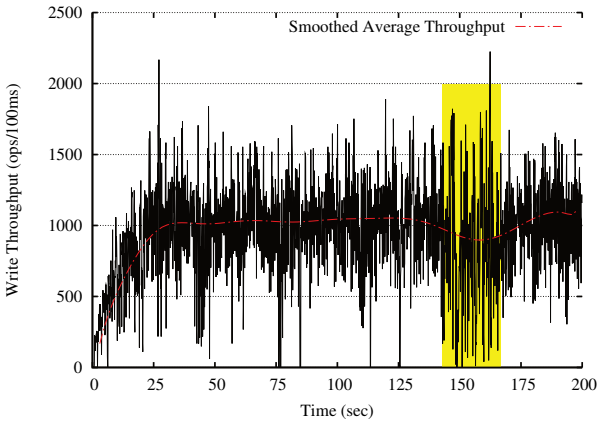

Fig. 8. ACaZoo without RG leader changes

Optimizations

Writes to the ZK commit log use a group-commit mechanism. To achieve high I/O throughput the system needs to ensure efficient operation all the way to the disks. In a heavily write-intensive setup the filesystem buffer-cache should be continuously writing to disks to avoid stalling for too long at the periodic sync operation. The standard operating system setup (in Linux and other general-purpose operating systems) is to delay (defer) writes. We had to change the standard behavior by tuning the thread responsible for destaging data from the buffer cache to disk to be invoked whenever there is anything in the buffer cache to be written. Note that this optimization requires platform-specific knowledge and is thus testimony to the fact that despite improved support in Java [28] it is not always possible to achieve fully platform-independent systems software in Java alone.

V. Evaluation

Our evaluation platform (unless stated otherwise) is the Flexiant FCO cloud with virtual machines (VMs) having 2 CPUs, 2GB memory, and a 20GB remotely-mounted disk. Certain performance-intensive experiments were performed on a similarly configured private OpenStack-based cloud to minimize interference with concurrently executing workloads. The version of Apache Cassandra used is 2.0.l, Apache Zookeeper is 3.4.5, and the Oracle NoSQL database is 2.1.54. We chose the Oracle NoSQL commercial database as one point of comparison to ACaZoo since it closely relates to it in several aspects (key-value store with similar API and data model, sharding over replica groups, primary-backup replication, client routing of I/O requests, Java implementation) but differs from it in the use of a B+-tree storage organization vs. ACaZoo's LSM-Trees.

Our load generator is the Yahoo Client Serving Benchmark (YCSB) [29] version 0.1.4, which can be configured to produce a specific access pattern, I/O size, and read/write ratio. YCSB performs 1KB accesses (it creates and accesses a single table with one column family comprising ten columns of 100-byte cells) with configurable read/write ratio using Zipf (featuring locality) or uniformly-random probability distributions.

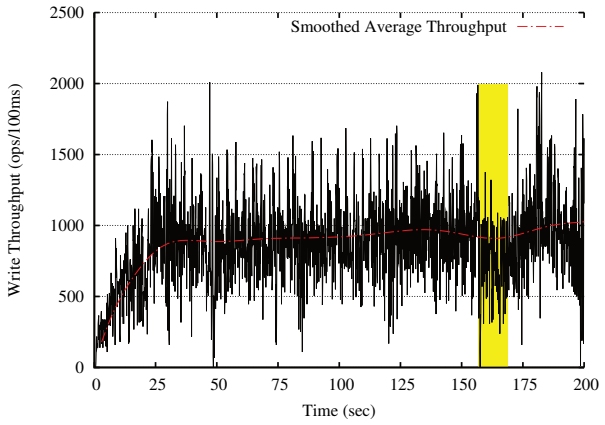

Fig. 9. ACaZoo with RG leader changes (RANDOM policy)

A. Performance impact of LSM-tree compactions

We first quantify the performance impact of LSM-tree compactions on an ACaZoo RG of three storage nodes with a fixed leader (no RG leader changes). Figure 8 depicts the throughput of a write-intensive workload (YCSB with 64 threads performing 100% writes) over time. We observe that at 45″, 75″, and 113″ performance drops briefly but drastically due to memtable flushes taking place. From 143″-165″ a compaction task is seen to have a significant and enduring impact on system performance. This compaction task involves 4 SSTables and takes 22.34 sec to complete for a total of 310MB of data compacted (an effective throughput of 8.5 MB/s). Besides Memtable flushes and compaction activities, Java garbage collection is seen to have infrequent but measurable impact on performance.

Insert operations are indeed resource intensive due to message deserialisation, Memtable flushes, compactions, and intensive memory use. Through `iostat` monitoring we observed that our VMs are nearly always CPU-bound, turning I/O-bound when the amount data processed grows significantly. While we believe that this picture can be somewhat improved by increasing the allocation of resources to servers (e.g., assigning an additional disk spindle dedicated to the commit WAL as well as additional CPU cycles) this increases system cost and may not always be an option in large-scale deployments.

We next evaluate the performance improvement from RG leader changes. Figure 9 depicts YCSB throughput with 64 threads and a 100%-write workload under the RANDOM leader change policy. We observe the impact of Memtable flush events at 48″, 85″, 121″, and 199″. There is also a new leader election at 157″ just before the leader starts a compaction task. At that point, the client experiences a short (100ms) interval of unavailability (throughput drops to 0 ops/sec) and then continuing on with the new leader. This can be contrasted to the long (about 23 sec) interval of performace degradation due to compaction observed in Figure 8. In Section V-C we show that when the master is compacting, the probability that a majority of RG nodes simultaneously compacting is low (and decreasing with RG size). Therefore a RG leader change is expected with high likelihood to produce a configuration that can make progress.

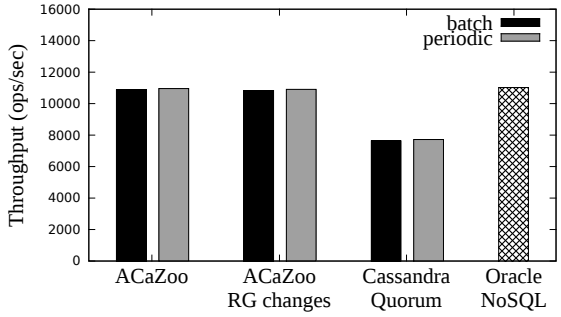

Fig. 10. YCSB throughput: 100% reads

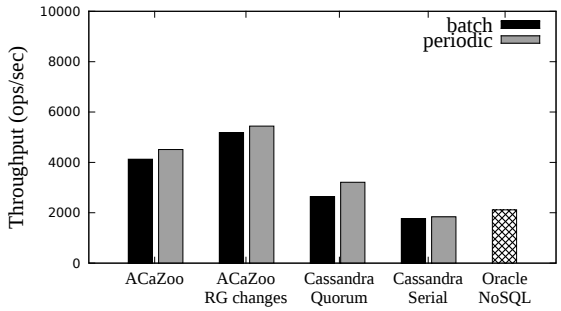

Fig. 11. YCSB throughput: 50% reads, 50% writes

B. Performance of single replication group

We next compare the performance of a single ACaZoo RG of three storage nodes to an equivalent setup of Oracle NoSQL, and two Cassandra setups of three storage nodes in a ring with replication factor 3: a setup with *quorum consistency* reading/writing 2 out of 3 replicas (termed Cassandra Quorum) and another similar setup performing linearizable writes [18] (termed Cassandra Serial). Cassandra Quorum - a relaxed consistency system- is configured conservatively so as to approximate the semantics of the other three systems. We evaluate all systems under a YCSB workload of 256 concurrent threads with three different operation mixes (100/0, 50/50, 0/100 reads/writes) on a private OpenStack based cloud with VMs having 2 CPUs, 2GB memory, and a 20GB remotely-mounted disk. We test ACaZoo both with and without RG leader changes using the RANDOM policy, to evaluate the performance improvement from enabling this feature. The database created by YCSB contains 10^6 1KB records for a total of 1GB of data.

ACaZoo performs reads from the master replica only, just as Oracle NoSQL does (Oracle calls this *absolute consistency*). As discussed in Section IV-B, ACaZoo and Cassandra can be configured for either *periodic* (relaxed durability) or *batch* (strict durability) writes to their commit WAL. Oracle NoSQL is configured to perform writes in WRITE_NOSYNC mode, i.e., it initiates them as soon as they arrive at replicas but syncs them to disk only when a buffer of configurable size fills up (similar to the *batch* mode of ACaZoo and Cassandra).

Figure 10 depicts YCSB throughput during a read-only workload. The small database size relative to VM memory

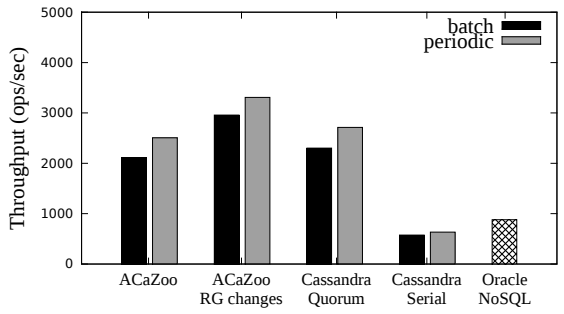

Fig. 12. YCSB throughput: 100% writes

	Count (#)	Longest (sec)	Average (sec)	Total (sec)
Compaction (RA)	11	78.44	17.96	197.64
Memtable flush (RA)	53	-	-	-
Garbage Collection (RA)	197	0.91	0.148	29.33
Compaction (RR)	12	72.65	15.94	191.39
Memtable flush (RR)	52	-	-	-
Garbage Collection (RR)	192	0.85	0.147	27.84

TABLE I. TASKS DURING A 100%-WRITE YCSB WORKLOAD

means that reads are mostly cached. ACaZoo performs on par with Oracle NoSQL, while Cassandra trails due to the requirement to read from two rather than one replica. We omit Cassandra Serial from Figure 10 since its performance is identical to Cassandra Quorum (the two systems share their read path). ACaZoo with RG leader changes does not lead to a performance improvement in this case due to the absence of writes (and therefore significant compaction) activity.

As the share of writes increases into the mix, performance drops for all systems. We observe in Figures 11 and 12 that ACaZoo in batch mode outperforms Oracle NoSQL (for both the 50%/50% and 100%-write mixes). We attribute the difference to the better pipelining of I/O operations in ACaZoo compared to Oracle NoSQL (we have empirically determined that BerkeleyDB JE –the underlying Oracle NoSQL storage engine– allows a single outstanding batch write transfer between replicas at a time). For 50% reads, 50% writes (Figure 11) ACaZoo with RG leader changes outperforms standard ACaZoo by 25% and 20% for batch and periodic operations. For 100% writes (Figure 12) ACaZoo with RG leader changes outperforms standard ACaZoo by 40% and 33% for batch and periodic operations respectively. Cassandra Serial trails all systems due to the need for four roundtrips between replicas in each put transaction [18]. Results with the RR policy are nearly identical to those with the RANDOM policy leading us to conclude that both are equally effective in our workloads.

To get a deeper understanding of the intensity of compactions and other periodic tasks in ACaZoo we logged these activities on the (initial) leader node during a 20-minute 100%-write workload (results are shown in Table I). Although the leader will change during the course of the run, the events are representative of the activity taking place at each of the nodes in a RG. The recorded events in a typical run with the RANDOM policy (depicted as RA) included 11 compactions, 53 Memtable flushes, and 197 Java garbage collections. It is interesting to note that garbage collection events are brief; compactions however are long and very intensive. The longest compaction lasts for more than a minute and merges more than 700MB of data. Results with the ROUND ROBIN policy (depicted as RR in Table I) are similar; a slight difference is that RA spends more time on garbage collection and compaction in this run (the two are related: a larger compaction (700MB in RA lasting 78″ vs. 600MB lasting 72″ in RR) leads to a longer garbage collection in RA).

C. Time correlation of compactions across replicas

In this experiment we observe compaction events on all replicas of an ACaZoo RG in configurations of 3, 5, and 7 nodes. Our goal is to determine the degree of overlap of compaction events in different replicas over a long (90 minute) workload. We use YCSB with 256 threads producing a 50% read, 50% write mix for 60 minutes, then switch to a 100% write mix for 30 minutes. The ACaZoo configuration used has the RG leader-change feature disabled. Our goal is to determine the probability P that at any point in time only a minority of nodes in the RG are simultaneously compacting and therefore there is a majority that can make progress. Because of significant non-determinism in the system we expect that in practice P (expressed in Eq. (1)) would be non-zero.

$$P(\text{ Any minority in RG compacting }) = 1 - P(\text{ No compactions in RG }) - P(\text{ Any majority in RG compacting }) \quad (1)$$

Figures 13–15 depict the time breakdown per 10-minute interval between the following three states: No compaction anywhere in the RG; some (any) minority of RG nodes compacting simultaneously; a quorum of RG nodes compacting simultaneously. The fraction of time spent in the second of those states is a measure of P. Going from 3 to 7 nodes in an RG, P ranges (on average) from 21% (for 3 nodes) to 32% (5 nodes) to 44.5% (7 nodes), indicating that the RG leader-change technique is expected to be increasingly effective with larger RG sizes. The average probability that a quorum of RG nodes compacts simultaneously (the regime where the RG leader-change technique does not help) diminishes from 23% (3 nodes) to 13% (5 nodes) to below 12% (7 nodes). We note that if a higher degree of non-determinism is desirable, it could be achieved by using different configurations (e.g., min/max compaction thresholds) for each replica.

D. Availability of RG under leader failure

In this section we compare the availability of a RG when the leader fails, comparing ACaZoo to Oracle NoSQL Database under a YCSB read-only Zipf-distributed workload produced by 64 threads. Figure 16 shows an outage of about 3.24 sec from the time the leader of the RG crashes until service resumes at the YCSB client. This interval breaks down to the following segments: (a) 1.19 sec between the time the leader crashes until the client notices; (b) 2 sec until the client establishes a connection with the new leader and restores service. Interval (a) further breaks down into: (1) 220 ms for the RG to reconfigure (elect a new leader); (2) 970 ms to propagate the new-leader information (e.g., its IP address) to the client through the CM.

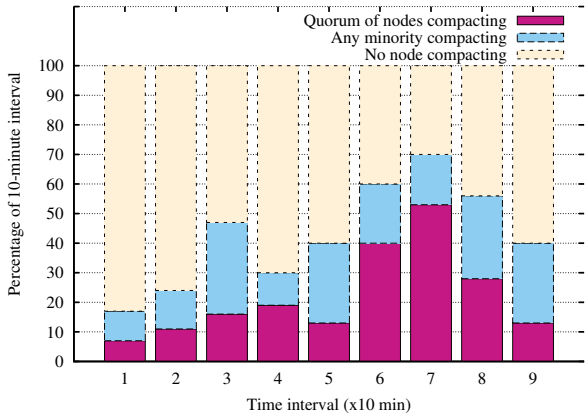

Fig. 13. ACaZoo RG of 3 nodes

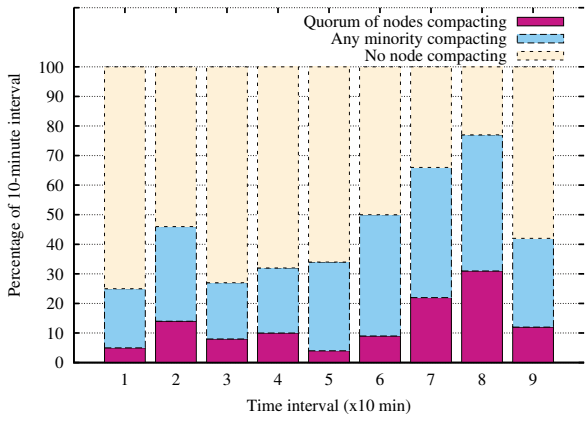

Fig. 14. ACaZoo RG of 5 nodes

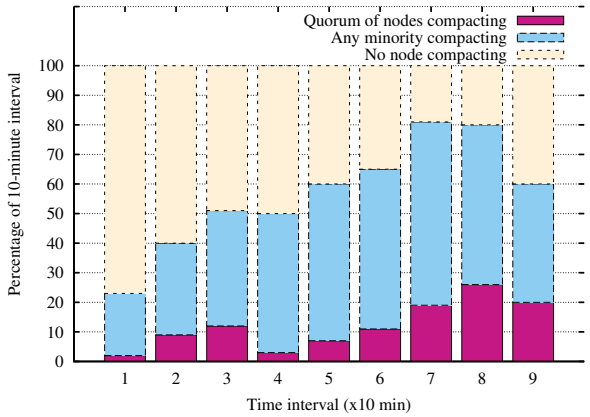

Fig. 15. ACaZoo RG of 7 nodes

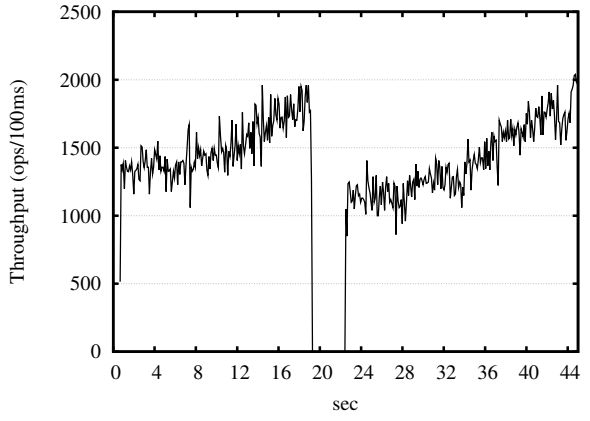

Fig. 16. YCSB throughput of ACaZoo under leader failure

In Figure 16 we observe ACaZoo performance ramping up from about 1300 ops/100ms to about 1900 ops/100ms as the RG leader cache warms up. The initial cache hit rate is expected to be low since the leader's data cache can only fit about 15-20% of the database during the load phase preceding each YCSB run. We observe the same ramp-up phase after a failover since the new leader also starts with a cold cache.

Figure 17 shows YCSB throughput under Oracle NoSQL. The client-observed outage in this case is about 3.5 sec. Oracle NoSQL starts from about half the throughput of ACaZoo (700 ops/100msec) and again from that level on after a failover. We also observe a more noisy behavior in Oracle NoSQL's throughput curve compared to ACaZoo. We believe that both observations are due to ACaZoo's use of a highly effective *key cache* (inherited from Cassandra) in addition to its standard data cache, which aids it in directly locating rows in SSTables in case of data cache misses, reducing its indexing overhead.

E. Impact of client-coordinated I/O

In this final experiment we quantify the performance benefit due to client-coordination of requests. We run a YCSB read-only workload over a cluster of six RGs and observe improvement of 26% and 30% in average response time and throughput respectively, compared to server-coordination of requests (Table II summarizes our results).

VI. Future Work

Efficient elasticity and data re-distribution is an important research area that we plan to focus on next. A brute force approach of streaming a number of key ranges to a newly joining RG is a starting point but our focus will be on alternatives that exploit the underlying replication mechanism (as in Lorch *et al* [30]). The immutability (write-once) characteristics of LSM-Trees lend themselves to efficient data movement primitives. 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).

VII. Conclusions

In this paper we described ACaZoo, a NoSQL system offering strong consistency, high performance, and high availability for sharded data intensive NoSQL applications. These properties are achieved via the combination of a high performance replicated data store based on LSM-Trees, client-coordinated I/O, and fast client notifications of cluster configuration changes. The impact of heavy periodic background activity at the master (a challenge with frequent compactions in LSM-Trees) is handled via replica-group leader switches.

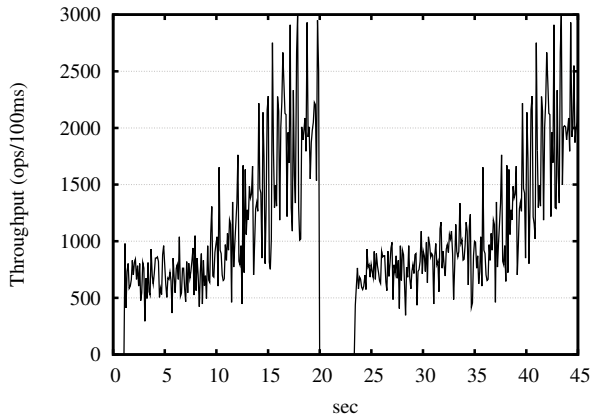

Fig. 17. YCSB throughput of Oracle NoSQL under leader failure

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

TABLE II. YCSB READ-ONLY WORKLOAD

We examined two policies for RG leader changes (random and round-robin) and found them both effective in delivering a performance improvement of up to 40% in write-intensive workloads. We note that the RG leader-change technique is generally applicable to any primary-backup replication system. ACaZoo overall is shown to exhibit excellent performance and availability compared to a commercial database with comparable architecture and consistency semantics.

VIII. ACKNOWLEDGMENTS

We thankfully acknowledge the support of the PaaSage (FP7-317715) EU project.

REFERENCES

[1] 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," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[2] 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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, p. 35, 2010.

[4] A. Auradkar, C. Botev, S. Das *et al.*, "Data Infrastructure at LinkedIn," in *Proc. of the 28th IEEE International Conference on Data Engineering (ICDE)*, Washington, DC, April 1-5, 2012.

[5] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: http://dx.doi.org/10.1007/s002360050048

[6] Apache Inc., "HBase," Apache Software Foundation, 2013. [Online]. Available: http://hbase.apache.org/

[7] R. Escriva, B. Wong, and E. Sirer, "HyperDex: A Distributed Searchable Key-vale Store," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 25–36, Aug. 2012.

[8] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!s Hosted Data Serving Platform," in *Proc. of the VLDB 08*, Auckland, New Zealand, August 2008.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems (2Nd Ed.)*, S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=302430.302438

[10] E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 84–92, 1996.

[11] F. Zunqueira, B. Reed, and M. Serafini, "ZAB: High-performance broadcast for primary-backup systems," in *Proc. of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN'11)*, Hong Kong, HK, June 2011.

[12] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li, "Paxos Replicated State Machines as the Basis of a High Performance Data Store," in *Proc. of the 2011 USENIX Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, March 2011.

[13] Apache Inc., "BookKeeper," Apache Software Foundation, 2013. [Online]. Available: http://zookeeper.apache.org/bookkeeper/

[14] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[15] M. Burrows, "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.

[16] J. MacCormick *et al.*, "Niobe: A Practical Replication Protocol," *ACM Transactions on Storage (TOS)*, vol. 3, no. 4, 2008.

[17] Oracle, Inc., "Oracle NoSQL Database," http://www.oracle.com/technetwork/products/nosqldb/learnmore/nosql-wp-1436762.pdf, 2012.

[18] J. Ellis, "Lightweight Transactions in Cassandra 2.0," http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0, DataStax, Inc., 2013.

[19] R. Sears, M. Callaghan, and E. Brewer, "Rose: Compressed, log-structured replication," in *Proc. of PVLDB'08*, Auckland, New Zealand, August 2008.

[20] J. Shute, S. Oancea, B. Ellner *et al.*, "F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business," in *Talk given at ACM SIGMOD/PODS 2012 (industrial presentations)*, New York, NY, June 22-27, 2013.

[21] L. Qiao, K. Surlaker, T. Quiggle *et al.*, "On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform," in *Proc. of ACM SIGMOD/PODS 2013 (industrial presentations)*, New York, NY, June 22-27, 2013.

[22] N. Bronson, A. Z., G. Cabrera *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," in *Proc. of 2013 USENIX Annual Technical Conference (ATC'13)*, San Jose, CA, June 26-28, 2013.

[23] J. C. Corbett, J. Dean, M. Epstein *et al.*, "Spanner: Google's globally-distributed database," in *Proc. of 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, Holywood, CA, October 8-10, 2012.

[24] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable Consistency in Scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal, 2011.

[25] Shafaat, T. and others, "ID-Replication for Structured Peer-to-Peer Systems," in *Proc. of Euro-Par 2012*, Rhodes, Greece, 2012.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proc. of the 2010 USENIX Annual Technical Conference (ATC 2010)*, Boston, MA, June 2010.

[27] Birman, K., and others, "Overcoming CAP with Consistent Soft-State Replication," *IEEE Computer*, vol. 45, no. 2, pp. 50–58, 2012.

[28] "Java New I/O API and Direct Buffers," Oracle, 2013. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/jni-14.html

[29] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proc. of 1st ACM Symposium on Cloud computing (SoCC'10)*, Indianapolis, IN, Jun. 2010.

[30] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART Way to Migrate Replicated Stateful Services," in *Proc. of EuroSys'06*, Leuven, Belgium, 2006.