Replica-group leadership change as a performance enhancing mechanism in NoSQL data stores

Antonis Papaioannou*† and Kostas Magoutis*‡
*Institute of Computer Science, Foundation for Research and Technology – Hellas, Heraklion 70013, Greece
†Computer Science Department, University of Crete, Heraklion 70013, Greece
‡Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece

***Abstract*—In this paper we investigate replica-group reconfiguration as a way to mask performance bottlenecks on the primary node of a primary-backup replication group in a NoSQL data store. We investigate the benefit of changing replica-group leadership prior to resource-intensive background tasks such as LSM-tree compactions or data backups on the primary node, a method that can improve throughput by up to 23% during LSM-tree compactions and by 35% during backup tasks. Our implementation is based on MongoRocks (MongoDB 3.7 and RocksDB 5.7) using leveled compaction. We experimentally demonstrate the performance impact of compactions and data backups when they occur at replica-group primaries, and the benefits of targeted leadership-change actions. We evaluate our system using the Yahoo Cloud Serving Benchmark (YCSB) and compare to unmodified MongoRocks on dedicated infrastructure.**

I. Introduction

Data replication is a standard technique for achieving high data availability and reliability, and a range of data replication techniques are essential components of distributed storage systems today [1]. Dynamic reconfiguration of replica groups has received significant attention recently [2], [3], [4], [5] as the importance of adjusting the number and type of nodes backing replica groups with minimal downtime has become a key system requirement of Internet applications.

Primary-backup (PB) replication [6] is a strongly consistent replication technique in widespread use today [4], [7], [8], [9]. Systems implementing PB replication feature a strong leader (the primary) that coordinates write operations (in some systems, reads as well) towards secondary replicas (backups). Any PB implementation must handle failure of the primary by electing a new primary within the current configuration, as a standard reconfiguration action (a *view change*). In recent years, certain PB implementations offer APIs that externally trigger such reconfiguration actions for management purposes.

Being on the critical path of I/O activity, the primary in a PB system typically takes higher load than secondary replicas. Any additional overhead on the primary may critically affect performance of the replica group as a whole. Recent research [10] demonstrated use of targeted leadership-change actions as a driver for lightweight adaptation of replica groups, by moving the primary away from nodes that are, or will soon be, heavily loaded. In this way, a certain level of load balancing is feasible at low cost (the short availability lapse that such reconfigurations entail), occasionally leading to large benefits. Leadership change has also been proposed in the context of Byzantine fault tolerance for mitigating attacks in which a leader intentionally misbehaves [11]. Use of leadership-change in BFT settings, while partly sharing goals with our work, differs in terms of the policies and mechanisms involved.

The research in this paper aims to systematically explore the policies and mechanisms underlying replica-group leadership change and its potential for masking performance issues in NoSQL data stores. Previous work studied leadership-change under mostly write-intensive workloads in a shared cloud infrastructure using a proof of concept research prototype [10]. This paper extends previous work by describing the general design of such a system, evaluating more sources of overhead (LSM-tree [12] leveled compactions and data-backup tasks) and a wider range of workload mixes in a dedicated cluster, using an industrial-strength implementation based on MongoDB and an LSM-tree based storage manager, RocksDB [13].

We outline the general design of such a system by defining the states each replica can be in (primary or secondary, performing a background task or not) and their transitions, and highlight the key policies and mechanisms involved. We further present key implementation choices we made within MongoDB/RocksDB, in maintaining global information about node activities for ranking nodes as candidates for primary, coordinating between the independent replication (MongoDB) and storage (RocksDB) layers, and ensuring that our preferred candidate can always be elected by the PB election algorithm.

Our experimental evaluation highlights the performance benefits of reconfiguration actions in the case of LSM-tree compactions and data backup processes. There are other known performance issues, such as slow system response during checkpoint writing [14], that could be similarly addressed by our system and that we plan to evaluate in future work.

Our key contributions in this paper are:

- A description of the design concepts (replica states, transition policies, and underlying mechanisms) in using replica-group leadership change as a performance enhancing mechanism in primary-backup replication
- Addressing key challenges in an implementation of the design within an industrial-strength NoSQL data store (MongoDB) and storage manager (RocksDB) using LSM-trees with leveled compaction
- An evaluation of the prototype under two sources of periodic performance impact on replica-group nodes:

LSM-tree compactions and data backup tasks.

The remainder of this paper proceeds as follows: In Section II we provide background on MongoDB replication and RocksDB. In Section III we describe the design and implementation of our system. In Section IV we discuss our experimental evaluation, and in Section V we conclude.

II. Background

As background to our design and implementation in Section III we describe the basic operation of replication on MongoDB [15] and RocksDB [13], the storage manager we use in this work. In general, MongoDB stores data in documents. It groups documents in *collections* (akin to tables), partitions data via *sharding*, and replicates shards as explained below.

MongoDB replication. MongoDB replicates shard data using primary-backup replication [6]. Each replica group has a single primary and multiple secondaries. Our brief discussion of the replication protocol is based on online documentation [16] and inspection of the source code (MongoDB version 3.7). A primary is associated with a given *term*, indicating the election number at which it was elected. The primary inserts each client update it receives to an operation log (the *OpLog*) stored as a local file and then (asynchronously) applies it to its local copy of the database. Secondaries pull OpLog entries from the primary or from another secondary, known as their *sync-source* node. Each secondary stores fetched updates to its own OpLog and applies them to their own database copy. In our experiments the sync-source is always set to be the primary.

All MongoDB nodes maintain topology and status information about other nodes in the cluster. Each node communicates regularly (in heartbeats every 2 seconds) with all other nodes to check their status, to stay up to date with their sync source (fetching OpLog entries), and to notify them of their progress.

A node runs an election when it has not seen a primary within the election timeout, or during explicit reconfiguration (termed a *priority takeover*). The election process is based on the Raft [4] protocol. When a candidate wins an election, it notifies all nodes via a round of heartbeats. It then checks if it needs to catch up with the former primary. This process tries to commit as many as possible of the entries the last primary managed to send to secondaries, but may not have managed to commit (they will otherwise be rolled back). Prior to accepting writes, the primary-elect drains its OpLog from previous-term entries by applying them to the database.

RocksDB. The implementation of MongoDB we employ in this work uses RocksDB [13] as a storage engine. RocksDB stores data in Log-Structured Merge (LSM) trees [12]. It applies each incoming update to a write-ahead log for durability and then to an ordered memory buffer (the memtable), periodically flushed to an immutable indexed ordered file (the SSTable). To retrieve a requested key, a number of SSTables may have to be consulted. Periodic compactions (merge-sort runs) of SSTables aim to maintain a small number of large SSTables, improving read performance. RocksDB implements leveled compactions [17], initially proposed in LevelDB [18], a key-value storage engine written at Google.

The MongoDB OpLog is implemented as a specific collection type called a *capped collection*, built as a circular buffer. It is a fixed-sized collection that automatically overwrites its oldest entries when it reaches its maximum size. MongoRocks monitors the size of the OpLog collection and reduces it via compaction. Due to heavy overwrite activity, the OpLog involves significant data-discarding during compactions, affecting overall performance. Thus MongoRocks triggers compaction periodically even if the size of the OpLog has not reached its limit. OpLog compactions have higher priority in the queue of pending compaction operations.

III. Design and Implementation

Design. Our design goal is to mask the impact of resource-intensive background activities on replica-group performance. Such activities, caused by internal storage-system needs (such as LSM-tree compactions) or external tasks (such as automated backup jobs), are often essential for smooth operation and cannot be avoided or postponed for too long, as doing so impacts application performance and/or data availability. The key idea is to demote a primary who is about to engage in such a resource-intensive background task into a secondary, in effect executing such tasks on secondary nodes only. The key mechanisms that facilitate this design are:

- Notifying the primary that a background task is upcoming
- Reconfiguring the group with minimal service disruption.
- Maintaining a global view of background tasks executing on replica group members. Each replica n_i periodically gossips the following status to the group: whether there is a background task (internal or external) executing on n_i; when the last background activity completed on n_i.

A state machine describing the states a replica can be in and possible transitions is depicted in Figure 1. For concreteness and without loss of generality we assume that the background activity we aim to mask is LSM-tree compactions, however other (and different types of) activities are supported. A primary starts in state P, NC (primary, non-compacting) and each secondary in S, NC (secondary, non-compacting). A reconfiguration moves the primary from P, NC $\rightarrow$ S, NC and a secondary from S, NC $\rightarrow$ P, NC. Two key policy decisions are when to trigger a reconfiguration and when that happens, which secondary is the best choice for new primary:

Policy 1: When is reconfiguration beneficial. A primary decides that a reconfiguration of the replica group is worthwhile when a background task is upcoming on the primary and the anticipated performance impact justifies the cost (short availability lapse) of reconfiguration. The impact of the type of background tasks we consider in this paper (LSM tree compactions, data backups) nearly always justifies a reconfiguration. In future work we intend to broaden our investigation to a wider range of background activities, including shorter background activity spikes.

Policy 2: What is the best choice for new primary (if any). Another key policy is selecting the replica to promote

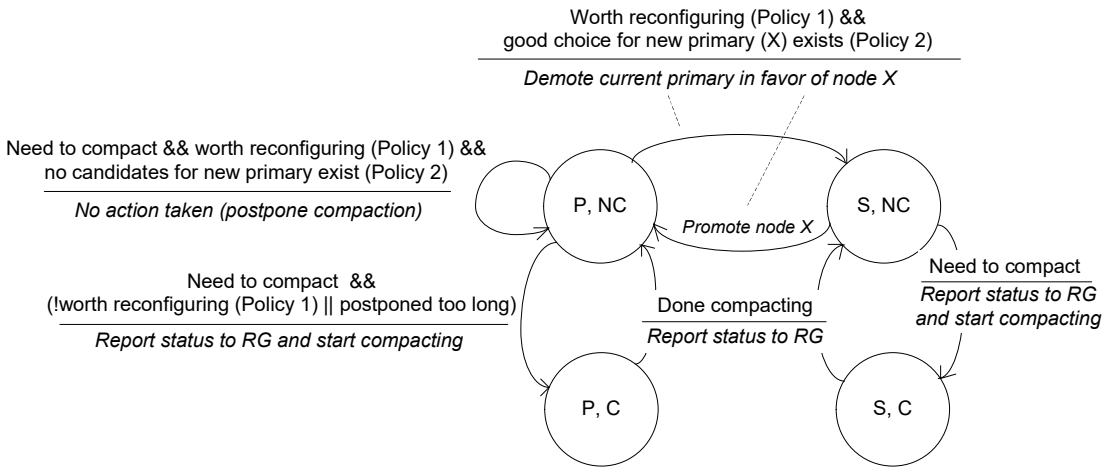


Fig. 1: States of a single replica (P: primary; S: Secondary; NC: not compacting; C: compacting) and transitions

to next primary. The selection process is based on a *replica-ranking* (RR) algorithm that takes into account the gossiped state of all replicas. The RR algorithm on the primary considers all secondaries that are not executing an internal or external background task as candidates for new primary. If there are more than one candidates, the algorithm by default favors the node that most recently completed an internal background task (*"most recently completed"*, MRC). Since internal tasks (such as compactions) are usually periodic, the expectation is that this candidate should enjoy a longer background-activity-free period until its next internal background task.

Another option is to select the candidate that served as primary for the longest time in the past (*"longest-served primary"*, LSP). Since all replicas maintain an in-memory read cache but only nodes that serve client requests (primaries) use it (secondaries only fetch updates to their OpLogs, bypassing their cache), LSP effectively favors nodes with warmer caches.

Policies 1 and 2 (*when to reconfigure* and *whom to promote*) may in some cases be inter-related: A reconfiguration may be called as soon as the background task on a recently-demoted primary is over (even though there may not be a pending compaction on the current primary), turning the leadership to the previous primary. This joint policy choice (an LSP variant we term *"preferred primary"*) in which a single node acts as primary, except for periods when it performs heavy background tasks, is geared to promote high cache hit rates.

If there is no suitable candidate (e.g., if all secondaries are compacting), the primary postpones its background activity (P, NC $\rightarrow$ P, NC) and re-evaluates its decision as soon as it receives a status update from any of the secondaries that may point to a suitable candidate. The primary decides to start the activity anyway (P, NC $\rightarrow$ P, C) if the reconfiguration is not worthwhile or if it has been postponing it for too long. Secondaries that need to perform a resource-intensive background task are allowed to proceed (S, NC $\rightarrow$ S, C). During that period they are ineligible to become primary.

A key challenge for our work is that a suitable candidate for primary may not be always easy to elect in existing variants of primary-backup [4], [7], [8], [16]: Most such election algorithms require that the elected primary is a node that has seen all committed proposals, whereas our optimal choice for next primary may be a node that has not participated in a committing majority for a long time (e.g., due to a heavy activity in the past on their side). To ensure that our choice for new primary does not lead into a state where the node cannot be elected within a reasonable amount of time, if the candidate fails to get a majority of votes (voters' OpLog being ahead of the candidate's OpLog) the current primary steps down (prohibiting new updates) so that the primary-elect catches up with the latest OpLog entries and request that replicas vote again, eventually ensuring success.

Implementation. Our implementation is based on MongoDB [15] with the RocksDB engine [13] (MongoRocks [19]).

MongoDB has built-in support for reconfiguring a replica group. Replicas regularly exchange a ReplicaSetConfig structure, listing all nodes in the replica group at a particular configuration *version*. Changes in the configuration are indicated by a change in the version number and propagated from the primary (downstream) to all nodes. Each replica-group member has a *priority* property that affects the timing and thus the outcome of elections for primary. When a node learns of the new ReplicaSetConfig, it ranks all of the priorities listed there and assigns itself a timeout proportional to its rank

$$(\text{priority rank} + 1) \times \text{election timeout}$$

After the timeout expires, it checks if it is eligible to run for election, and if so it starts an election.

In our implementation, we programmatically assign priorities to each node in a new ReplicaSetConfig as indicated by our replica-ranking (RR) algorithm. The node we aim to elect as new primary has higher priority than the current primary. We modified the priority takeover timeout to activate the action as soon as possible. In practice, reconfiguration completes within a few hundreds of milliseconds.

Each replica-group member maintains a view of the current background tasks running on all replicas as input to the RR algorithm. All members periodically report their status over the regularly-exchanged MongoDB heartbeats. We use the RocksDB internal *compaction statistics* API to expose the counter of currently-running compactions (*rocksdb.num-running-compactions* property) to the MongoDB layer and embed this information to every heartbeat message. The receiver extracts information from heartbeats from each replica-group member and derives when the last compaction ran on each node. Our RR algorithm selects only between non-compacting nodes and by default favors the one with the most-recently completed compaction (*MRC* policy). In addition to LSM-tree compactions, the current implementation carries information about and handles externally-triggered data backup tasks.

Reconfigurations of the replica group can only be triggered by the primary. To re-instate the previous primary (*"preferred primary"* policy), the current primary monitors the status of its predecessor (now a secondary) based on its periodically reported status. When the current primary notices that the "preferred primary" has completed its internal background activities, it triggers elections proposing it as a candidate. We have observed in our experimental evaluation that the preferred-primary policy promotes good cache behavior as the primary maintains a fresh cache for a long period of time.

RocksDB communicates with MongoDB over a sockets channel to notify it of a compaction task and ask its permission to start it. If the MongoDB replica is a secondary, it responds positively. If it is primary, MongoDB asks RocksDB to defer the compaction and starts a reconfiguration. RocksDB uses a *pending-compaction-queue* to remember the database that requested the compaction. When the node transitions to secondary, RocksDB goes over entries in the *pending-compaction-queue* to reconsider delayed compactions. We use MongoDB's *ReplicationCoordinator* public API to expose information about replica state transitions to the storage engine.

MongoRocks uses compaction tasks to reduce the size of the OpLog. By default it triggers OpLog compactions at least every 30 minutes, even if the OpLog has not reached its limit. Assuming that all members of a replica set start at the same time, this would normally lead all nodes to perform their OpLog compactions in sync. In this way there would be no available non-compacting replica to replace the primary. In our implementation we randomize this time interval on every node to be between 30 to 45 minutes. In our experiments the system was always able to find a non-compacting replica.

IV. Evaluation

Our experimental testbed is a cluster of 4 servers, each equipped with a dual-core AMD Opteron 275 processor clocked at 2.2GHz with 12GB of main memory. All servers run Ubuntu 14.04 64-bit with a 3.14.1 Linux kernel and are interconnected via a 1Gb/s Ethernet switch. Servers used as MongoDB servers have a base 72GB 10,000 RPM SCSI drive with an additional 300GB 15,500 RPM SAS drive dedicated to storing data. All hard drives are formatted with ext4.

We use MongoDB 3.7 with RocksDB 5.7 as storage engine. Our evaluation workload is the Yahoo Cloud Serving Benchmark (YCSB) [20] version 0.11 executing on a dedicated server. The benchmark is configured to produce two different mixes of reads vs. updates/writes: 90%-5% (read intensive) and 50%-50% (write intensive), with requested keys selected randomly with the Zipf distribution. The YCSB evaluation dataset consists of 12 million unique records or 15GB of data per node. Our initial MongoDB cluster consists of one shard and 3 servers in the replica group. The number of YCSB client threads (number of parallel connections between database client and servers) is set to 2. We set MongoDB's *write concern* to one, meaning that writes are considered complete when acknowledged by the primary. We use journaling, which requires OpLog entries to be durable (written to disk) before acknowledging. We use the default settings for RocksDB: *max_background_jobs*, the maximum number of concurrent background jobs (including flushes and compactions) is set to 2, and *compaction_style* set to *level*. The dataset is loaded fresh onto MongoDB nodes before each experiment. In the following two sections we evaluate the benefit of reconfiguration actions in the presence of LSM-tree compactions and data backups. The default RR policy, MRC, is used in all experiments. A comparative evaluation of RR policies is beyond the scope of this paper and subject of future work.

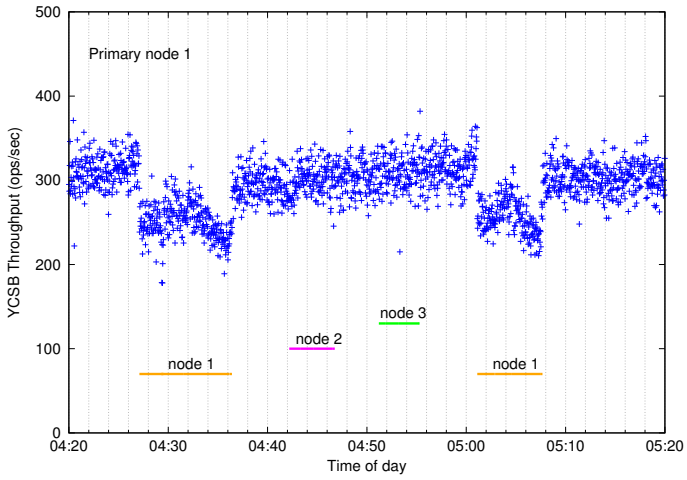
A. LSM-tree compactions

Figure 2 depicts YCSB throughput (ops/sec) under the read-intensive workload mix. As in all subsequent figures, time (x-axis) starts one hour into the experiment, when the system is deemed to have reached a steady state. Figures 2a and 2b depict performance without and with reconfigurations. Colored squares at the top of a graph (and corresponding vertical dashed lines) represent elections of a node as primary. Horizontal solid lines represent compaction tasks performed on a node. Each line may represent more than one sequentially-executing compaction tasks at different levels [17].

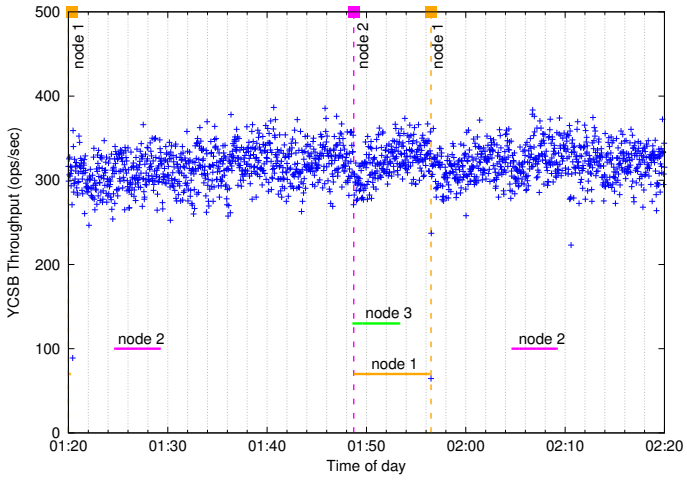
Figure 2a exhibits a clear performance hit during the compaction tasks at the primary node (indicated by horizontal orange lines labeled "node 1"). The average throughput during the one-hour window shown is 295 ops/sec. During primary compactions the average throughput is 253 ops/sec, whereas non-compacting time periods have an average throughput of 310 ops/sec. Thus there is a 18.4% performance hit during compaction tasks at the primary.

Performance exhibits a more stable behavior under the automated replica-group leadership change mechanism (Figure 2b), achieving an average of 318 ops/sec over the one-hour window. Overall our system exhibits 7.2% higher throughput during the one-hour window. Compaction tasks in this experiment were mostly due to growth of the OpLog database.

Figure 3 presents YCSB throughput under the write-intensive workload. Figure 3a exhibits a clear performance hit during the compaction tasks on node 1 (primary). The average throughput for the one-hour window is 239 ops/sec. Average throughput during compactions is 194 ops/sec, whereas non-

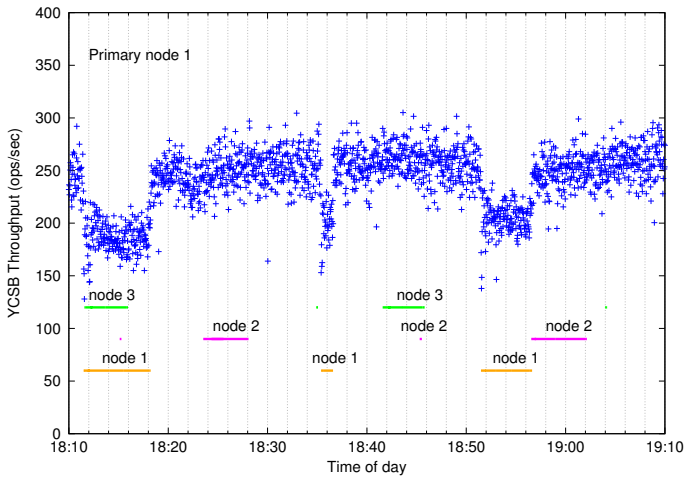


(a) Standard MongoRocks (primary node: 1)

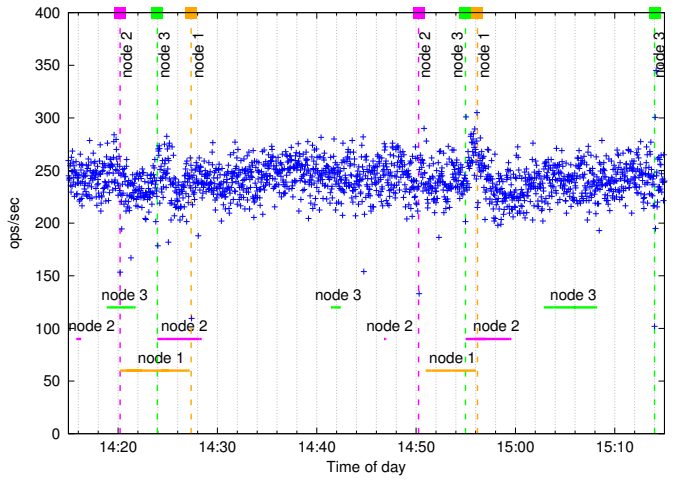


(b) MongoRocks with reconfigurations

Fig. 2: 90% reads-10% writes



(a) Standard MongoRocks (primary node: 1)



(b) MongoRocks with reconfigurations

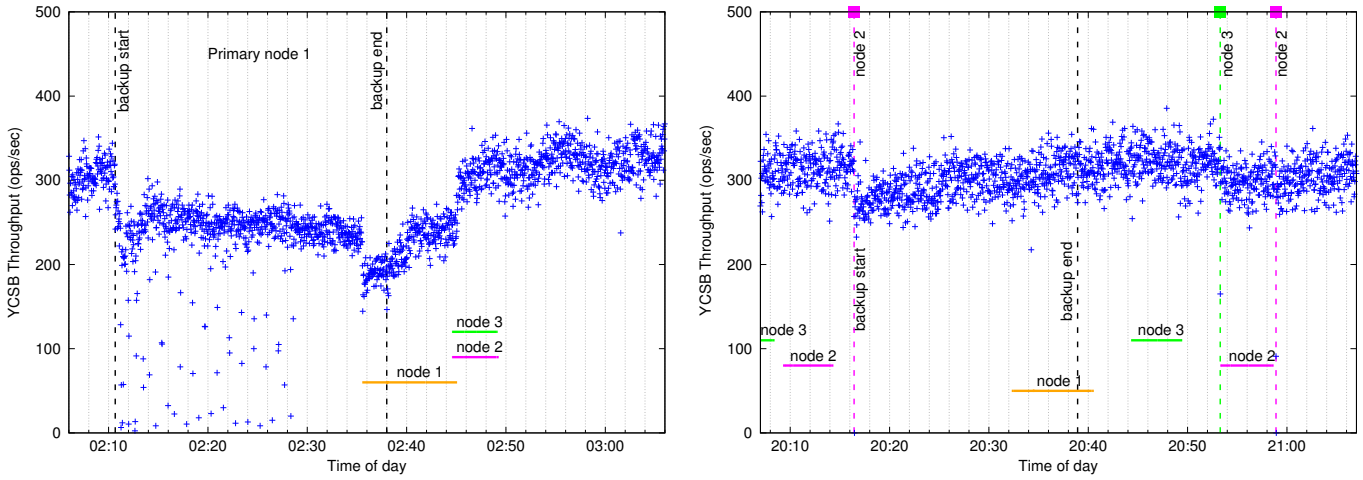
Fig. 3: 50% reads-50% writes

compacting periods (on primary) have an average throughput of 252 ops/sec. There is thus a 23% performance hit during compactions on the primary. Throughput using reconfigurations exhibits a more stable rate (Figure 3b) achieving on average 250 ops/sec. Under the write-intensive workload the system achieves on average 4.2% higher throughput. We observe that our results are not sensitive to the read/write ratio as compactions are mostly OpLog-driven and in both cases have relatively low frequency. However, the performance improvement of reconfigurations in these representative runs is significant (18%-23%) during compactions in both cases.

B. Data backup

We extend our evaluation to external background tasks, and study performance while taking an online data backup with the MongoDB mongodump utility [21]. We use the *–oplog* option to capture incoming writes that occur during the mongodump operation into a file, ensuring that backups are point-in-time snapshots of the database state. During the backup process, the tools force a running database instance to read all data through memory. Reading infrequently-used data has the side effect of evicting frequently-accessed data from the cache, causing the performance of the regular database workload to deteriorate.

Figure 4 exhibits the impact of the backup process on performance while executing a read-intensive workload. Figure 4a shows a clear performance hit during the backup task (whose duration is indicated by the vertical dashed lines) when there is no reconfiguration action. The average throughput achieved during the one-hour window is 266 ops/sec. At 02:30 mongodump completes the extraction of data and begins to copy OpLog entries. The OpLog dump has no effect on the cache in contrast to the previous phase, where throughput was more noisy. At the end of the backup process there is an overlapping compaction task that results in a further performance degra-



(a) Standard MongoRocks

(b) MongoRocks with reconfigurations

Fig. 4: Data backup (mongodump –oplog), 90% reads-10% writes

dation. The average throughput during the background tasks (backup and compaction) is 229 ops/sec, whereas average throughput when no background task executes is 314 ops/sec. The system achieves 35% higher throughput when there are no background activities on the primary.

Figure 4b exhibits performance when applying a reconfiguration action prior to the backup. At 20:16 the system is notified about the upcoming external background task and the primary initiates a replica-group reconfiguration to mask its impact. The RR algorithm selects node 2, the replica with the most recently completed compaction (default policy), as the new primary. Node 2 serves as primary for the first time and therefore it initially suffers from low cache-hit ratio (causing the performance dip at 20:16), gradually increasing performance. This has not been a problem in previous experiments as the role of primary had been rotated around nodes a few times already before the measurement phase began, warming up caches. In this case, the average throughput during the one-hour window is 306 ops/sec, 15.8% higher average throughput than standard (non-reconfiguring) MongoDB. In addition, our system benefits the data backup task itself, as there is lower contention for resources in secondary nodes. The elapsed time of the backup task in our system is 21.5% shorter than the non-reconfiguring system (1,349 vs. 1,640 sec).

V. Conclusions

In this paper we evaluate the performance benefits of automated replica-group reconfigurations under the impact of periodic sources of overheads such as LSM-tree compactions and data backup tasks, using the YCSB benchmark. We find that targeted leadership change actions lead to 18-23% improved performance during execution of compaction tasks and 4-7% improved performance overall for LSM-tree compactions. Replica-group leadership change prior to a data backup task on the primary leads to 35% better performance and 21.5% faster backup compared to standard MongoRocks.

References

[1] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*. Berlin, Heidelberg: Springer-Verlag, 2010.

[2] S. Benz and F. Pedone, "Elastic Paxos: A Dynamic Atomic Multicast Protocol," in *Proc. of 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, GA, USA, June 2017.

[3] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira, "Dynamic reconfiguration of primary/backup clusters," in *Proc. 2012 USENIX Annual Technical Conference (ATC 12)*, Boston, MA, 2012.

[4] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, 2014.

[5] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed systems (2nd ed.)," S. Mullender, Ed., 1993, pp. 199–216.

[7] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," in *Proc. of 7th ACM Symp. on Princ. of Distr. Computing (PODC)*, 1988.

[8] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *DSN'11*, Hong Kong, 2011.

[9] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI'04*, San Francisco, CA, 2004.

[10] P. Garefalakis, P. Papadopoulos, and K. Magoutis, "ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees," in *Proc. of 33rd IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, 2014.

[11] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault-tolerant systems tolerate byzantine faults," in *Proc. of the 6th USENIX NSDI'09*, Boston, MA, April 2009.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-structured Merge-tree," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[13] (2018) Facebook RocksDB. [Online]. Available: http://rocksdb.org

[14] (2018) MongoDB Checkpointing Issues. [Online]. Available: https://goo.gl/idyog2

[15] (2018). [Online]. Available: https://www.mongodb.com

[16] (2018) MongoDB Wiki: replication internals. [Online]. Available: https://github.com/mongodb/mongo/wiki/Replication-Internals

[17] (2018) Leveled compaction. [Online]. Available: https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

[18] J. Dean and S. Ghemawat. (2011) Leveldb: A fast persistent key-value store. [Online]. Available: https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html

[19] (2018). [Online]. Available: http://mongorocks.org

[20] B. F. Cooper *et al.*, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of the 1st ACM Symp. on Cloud Computing (SoCC '10)*, 2010.

[21] (2018) Mongodump. [Online]. Available: https://docs.mongodb.com/manual/reference/program/mongodump/