Addressing the Read-Performance Impact of Reconfigurations in Replicated Key-Value Stores

Antonis Papaioannou and Kostas Magoutis

Abstract—Raw data are often orders of magnitude larger than main memory for many applications. As the performance of storage devices is still significantly slower than main memory, systems still rely on memory caching to improve performance. Data replication schemes are prevalent in data stores for high availability and reliability. In such schemes, while data updates are propagated to all replicas (either synchronously or in the background), reads are usually served by only a subset of replica group members (e.g., as in primary-backup and quorum systems). As a result, non-serving replicas cannot keep their memory cache state updated; thus, during a reconfiguration or a fail-over action, the system suffers from a high read-performance impact for a significant amount of time due to cold-cache misses. In our study we observed up to 70% hit after a reconfiguration due to cold cache misses, taking almost 18 minutes in some cases to fully restore to the pre-reconfiguration level of performance. In this article we propose a mechanism to maintain up-to-date read caches across replicas by sending read hints to the non-serving replicas to keep their caches warm. Thus the system is able to seamlessly achieve the same performance level even in the face of a replica group reorganization. This is especially important under the read-intensive workloads that are common today. Our evaluation shows that our mechanism has significant benefits during reconfigurations, with low performance impact under periods of resource strain. Given its advisory nature, the maintenance of read hints can be reduced or held off if needed during such periods.

Index Terms—Data replication, caching, performance

1 Introduction

The demands of Internet-scale applications for reliable, highly available data services have been growing exponentially. To meet these needs, distributed data stores consolidate large numbers of commodity servers into a single storage pool, using different forms of *data replication* [1] for reliability and high availability. Raw data is often several times larger than available memory, and performance of storage devices is still orders of magnitude lower than that of main memory. In combination with today's read-dominated production workloads and strong temporal locality among requested keys [2], application performance still depends on the efficiency of data caching. Not surprisingly, data replication systems maintain in-memory data caches to improve performance, especially for read-dominated workloads. The efficiency of a read cache impacts the overall performance of data intensive systems.

In replicated data stores, data is replicated across a set of nodes comprising a *replica group*. A range of existing replication techniques in distributed storage systems dictate how data are propagated to replicas. Based on the consistency model used, replica groups ensure that replicas within them apply updates in some specific order (strongly consistent systems require that replicas totally agree on this order). Data replication techniques focus on ensuring consistency of the persisted data set, while each node of the replica group also maintain a memory cache of that set to improve performance.

There are multiple replication models in use today. *Primary-backup* (PB) replication [3] is a traditional replication technique in widespread use [4], [5], [6], [7]. Systems implementing PB replication feature a strong leader (or primary) that coordinates read and write operations towards secondary replicas (or backups). Reads are typically served by a single replica, most commonly the primary itself. Many such systems allow clients to read from any single replica to better distribute read-intensive workloads. *Active replication* systems, such as those based on the Paxos algorithm [8], have also been in use in storage systems [9], [10]. Formally such systems involve several replicas to carry out operations, typically a majority. Efficiency considerations in today's read-dominated workloads [2] have led to active-replication systems that permit reads from a single replica, similar to PB systems, through *lease-based* mechanisms [11], [12], [13]. *Quorum-based* systems is another class of replication systems where reads involve a set of nodes [14], [15], [16]. Efficiency considerations in quorum-based systems have also led to configurations featuring a small read set (often a single replica [17]). We can thus identify a trend in practical systems to restrict reads to as few replicas as possible, to optimize for read-dominated workloads.

Involving as few replicas as possible in read operations means that the caches of the remaining replicas receive

• *Antonis Papaioannou and Kostas Magoutis are with the Institute of Computer Science (ICS), Foundation for Research and Technology – Hellas (FORTH), 70013 Heraklion, Greece, and also with Computer Science Department, University of Crete, 70013 Heraklion, Greece. E-mail: {papaioan, magoutis}@ics.forth.gr.*

Manuscript received 27 July 2020; revised 3 Oct. 2021; accepted 28 Nov. 2021. Date of publication 14 Dec. 2021; date of current version 31 Jan. 2022.
This work was supported in part by the Hellenic Foundation for Research and Innovation through the STREAMSTORE project under Grant HFRI-FM17-1998.
(Corresponding author: Antonis Papaioannou.)
Recommended for acceptance by S. Pallickara.
Digital Object Identifier no. 10.1109/TPDS.2021.3135137

delayed or no information about application reads, and thus do not fetch up-to-date content in memory, in contrast to replicas serving reads. This often does not pose a problem, when reads are satisfied consistently by a single or the same set of replicas. However, when the replica(s) satisfying reads shift to replicas that have not actively maintained their read cache, a performance impact is incurred for read operations.

A switch of the read-serving replica may occur as a result of different types of *reconfiguration* operations. One such case traditionally occurs when a serving replica fails and a backup must take over (failover) or due to load balancing actions or workload migration where parts of an application are moved across the infrastructure. In a similar case, geo-replicated systems reconfigure the set of serving replicas (e.g., leader) while the client's location shift across time zones as they serve Internet users [18], [19], [20], [21]. Another reason for frequent (e.g., every 20 minutes) replica group reconfiguration include lightweight adaptation actions as a performance enhancement mechanism to mask performance bottlenecks on the serving nodes [22] [23]. While such reconfiguration actions have a positive long term impact, a negative side effect at the time of switching the read-serving node, is that a burst of cold-cache misses at the new read-serving replica may lead to latency spikes and throughput drops that result in service-level objective (SLO) violations for significant amounts of time (several minutes).

In this paper we address this problem by maintaining up-to-date read caches across all replicas *without* increasing the number of replicas actually serving reads or modifying the replication semantics of the system. We achieve this through a low-overhead mechanism by which all replica nodes eventually see a tunable subset of read operations (loading the respective data into their in-memory data cache) in the order seen by replicas serving reads and primarily responsible for serving client requests. This method eliminates the cold-cache effect observed after a reconfiguration on the new node that serves reads, which impacts the overall system performance seen by clients. Our proposed solution is a lightweight, best-effort synchronization method that tries to minimize the overall cache divergence between the primary and replica nodes, by disseminating read requests across replicas in the background (not tied to the execution of client requests). To achieve this, read serving nodes keep a volatile buffer containing information about the reads executed by the replica group. The buffer is periodically disseminated to replicas not involved in serving reads and is used as hints to update their caches. The design of this mechanism is based on basic principles and can be easily integrated to production replicated key-value stores such as the widely used MongoDB. Our evaluation demonstrates that the overhead is minimal and that the prototype maintains stable performance eliminating SLO violations during reconfiguration actions.

The rest of this paper is organized as follows. Section 2 describes background and related work and Section 3 describes our design choices and implementation details. In Section 4 we present evaluation results, and in Section 5 we conclude.

2 Background and Related Work

Two major points relating to our work are (i) replication systems today read from a subset of replicas (typically one), leading to situations where backup (non-reader) replicas have an out-of-date memory cache; and (ii) under certain situations (such as after a failure of the primary or other reconfiguration action) reads are directed to such replicas, leading to a surge in cache misses. In what follows we will describe how different replication systems relate to points (i) and (ii). There is currently no system that directly addresses this problem (short of issuing reads to all replicas, which penalizes the common path of read operations and thus avoided in practice).

Modern Systems Often Read From One Replica Per Group (Shard). The design and implementation details of replication mechanisms (for instance, how read and write operations are performed) vary across systems, entailing consistency, availability, and performance tradeoffs [24]. A trend in practical replication systems is to restrict reads to often just one (dynamically selected) replica per shard, to optimize for read-dominated workloads.

In quorum-based [14], [15] or active-replication (Paxos) [9], [10] systems, read and write operations are typically issued towards sets of replicas (or *quorums*), which can generally differ for reads and writes, and whose sizes may range from one to all nodes.[1] Production systems using quorum replication (Apache Cassandra, Voldemort, Riak) execute reads only on a majority of nodes. Different approaches to selecting the specific majority include network proximity [25], performance history [26] or load balancing [27]. In some cases even some nodes that are part of the majority may not perform the actual read of data and rather rely on metadata (e.g., digest reads [28]). In primary-backup (PB) [3] replication implemented by several popular open-source stores (MongoDB, CouchDB, RavenDB), client read requests are served only by the primary or (relaxing consistency) by any single replica. These choices are justified by the need to optimize performance in read-dominated workloads. The system presented in this paper is applicable to data stores serving reads from any single replica at a time.

A write operation typically requires a number (k) of acknowledgements that replicas have durably stored an update, before informing the requesting client of a successful operation. In several systems, a write request is sent to a broader set of replicas (n, where $n > k$) than the number of acknowledgments needed (k), eventually reaching the entire replica group. Updates to the first k replicas to respond are synchronous with the user update, whereas the remaining updates can proceed asynchronously in the background. Although one could argue that dissemination of updates eventually reaching all replicas (as performed by most replication technologies) could be sufficient to keep memory caches of all replicas fresh, in reality the data being written may not overlap at all with the read working set of application (e.g., in the case of writing new data while performing read-intensive analytics on past values). Thus update propagation by itself cannot maintain up-to-date memory caches across all

1. A small read-quorum typically requires a large write-quorum, overlapping in at least one replica, for ensuring strong consistency.

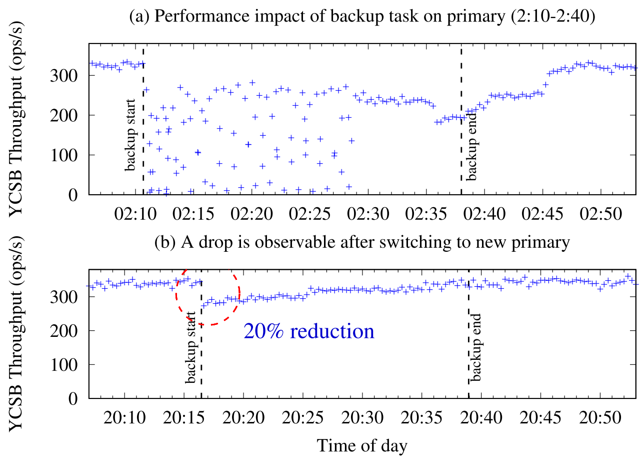

Fig. 1. Performance impact of backup activity (a) on replica group (shard) can be hidden via reconfiguration (b), however new primary suffers from cold-cache misses [22]

replicas. In addition, occasional restarting of replicas, such as in the context of rejuvenation [29] and proactive recovery in adaptive replicated services [30], completely wipes out their memory caches.

Reads May Shift to Replicas With Outdated Caches. Non-read-serving replicas are not aware of reads and thus cannot keep their memory caches warm. We previously described that systems may dynamically decide which replicas form quorums that execute requests. Another reason that triggers a shift in read-serving replicas is failures. If a serving replica fails or is brought down for maintenance, a previously non-serving replica must take over. Another reason is workload migrations, where parts of an application are moved across the infrastructure for load balancing or other maintenance operations (in such cases reads are usually directed to the nearest replica). Failover actions are frequent in large data centers [31], and adaptation actions are increasingly used for performance improvement. Geo-replicated stores automatically reconfigure the set of serving replicas (e.g., leader change) to satisfy application-defined constraints as locations and their access pattern shift across different time zones [18], [19], [20], [21]. Production systems at Google [20] trigger a reconfiguration as often as every 30 minutes when the system re-evaluates the optimal placement of a leader or its replicas based on workload characteristics, or every 2 hours in the case of Microsoft's store [18] that adapts to the shift of traffic across different time zones. Frequent reconfiguration actions may also be used as a lightweight adaptation mechanism to hide internal performance bottlenecks or in the case of colocated resource intensive background tasks on the serving replicas [22], [23]. In all these adaptation mechanisms, the newly assigned read-serving replicas (although consistent at the level of persisted data) may have missed recent reads and thus have an outdated memory cache leading to a performance impact (Section 4). Empirical evidence of the challenge addressed in this paper (and inspiration for this work) is provided by our own previous research [22], [23], [32]. Fig. 1 (from [22]) demonstrates that the action of changing the primary replica can hide the performance impact of a backup task, however the improved system (Fig. 1b) still suffers from a smaller but non-negligible performance hit (area in red circle) due to cold-cache misses at the new primary (a previously non-read-serving replica). Other work on measuring the recovery time of a replicated version of the HDFS metadata server [32] (Section 4.4.3) showed that switching the primary to a new replica with a cold memory cache can lead to significantly higher time to recover compared to a version switching to a hot spare.

Our approach aims to keep the caches of future read-serving replicas warm by disseminating read hints to them from current read-serving replicas. This approach is similar in spirit to prefetching in storage systems, which rely on high-level knowledge of future data accesses disclosed by an application [33] or history-driven predictions [34] to warm a cache ahead of time and thus increase hit rates. Our approach differs in that instead of informed guesses, replicas learn of read operations executed on a read-serving node and execute them locally to maintain an up-to-date cache view. Traditional prefetching mechanisms are complementary to our approach and can be applied in read-serving and non-read-serving replicas. Challenges investigated in the context of prefetching, especially in balancing prefetching with caching [35], are also applicable in our work.

3 Design and Implementation

We next describe the major design choices and the implementation details of our mechanism.

Design. Our goal is to ensure that non-read-serving replicas within a replica group keep track of the read working set and are always prepared to serve read requests without a performance impact due to cache misses, ensuring a smoother transition during a reconfiguration or failover. We aim to achieve this without modifying the replication protocol or system properties such as data consistency or availability. The mechanism should be transparent to users, and not have an impact on common-case performance of client read operations.

Our evaluation (Section 4) shows that in a typical application scenario over a replicated key-value store (MongoDB), a reconfiguration that changes the primary node in a replica group leads to higher latency for client requests, because of a low cache hit rate at the primary node, for the period of time (several minutes) it takes to load the working set into memory. This time depends on the amount of state that needs to be brought into memory (Section 4.3) and the speed at which the underlying storage device operates (Section 4.1). We aim to mitigate this problem by allowing the memory caches of non-read-serving replicas to keep track of the read working set using principles that can be easily implemented and integrated into existing replicated key-value stores.

A replication module is typically responsible for replicating requests across replicas in any distributed store, however the caching mechanism in each node is independent of the other replicas. A standard cache management policy over persisted data in each node is to apply every incoming read or write operation through the memory cache (implementing the necessary miss handling and write-through actions).

In order to disseminate read requests even to non-read-serving replicas and thus help them keep their caches

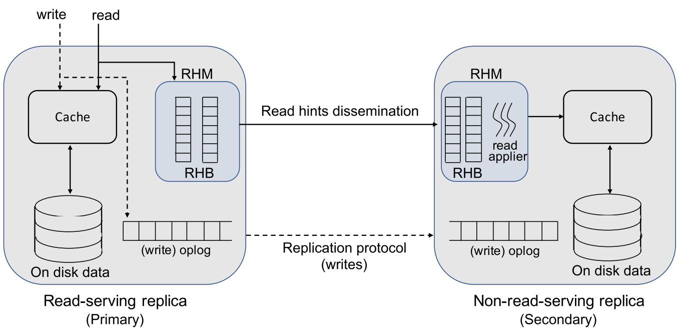

Fig. 2. The Read-Hints Module (RHM) is integrated as a plugin into the replica-maintenance path; this particular figure is based on MongoDB internals. The module passively monitors read requests and maintains a Read-Hint Buffer (RHB) that is periodically disseminated across replicas

warm, we propose recording the read operations executed on the serving replicas as *hints*, and send them over to non-read-serving replicas in the background. To achieve this, we introduce a *read-hints module* (RHM) and integrate it into all replicas. Fig. 2 presents the RHM integrated into the primary and secondary replicas. Read-serving nodes use RHM to passively monitor the executed read requests and maintain a *read-hints buffer* (RHB), which keeps a minimal description of each read operation applied on the serving replica. RHM periodically disseminates the contents of the buffer (in batches) to the non-read-serving replicas which store the received hint into their local RHB. Based on the hints a replica is able to replay read-operations (note that we are shipping read operations, not the data), leveraging the data store's existing read path and cache management mechanism.

Our mechanism does not require any modifications to the pre-existing replication and dissemination of writes, it uses separate data structures and dissemination channel for the read hints as depicted in Fig. 2 and in no way affects the data consistency or availability properties of the system (Section 3.3). As a read operation can always be satisfied by the underlying disk, our proposal is a *best-effort* mechanism that allows replicas to keep their caches warm when they become aware of the reads executed on the serving nodes. The design supports dissemination of read operations from more than one read-serving replicas and allows tuning of the degree of dissemination (how many replicas to disseminate to) and intensity of communication (degree of batching and frequency). In systems where reads are served by a different replica at a time for load balancing, RHM can be configured so that each read-serving replica disseminates read-hints to all replicas. However, we expect the biggest benefit from RHM will be realized in strongly-consistent systems that serve reads from a single primary replica.

A naive implementation that directly issues read requests to all replica nodes and waits for the first response is expected to maintain warm caches across all replicas. However, the consistency model of such approaches is weaker compared to the strong semantics achievable by reading and writing from/to the primary node. As such, naive approaches are not an option for applications requiring strong semantics from the underlying data store, whereas RHM covers such cases as well. RHM is based on an asynchronous best-effort background dissemination of read-hints and does not require any modification to the read/write protocol.

Section 3.1 describes in detail the implementation, tuning parameters, and different tradeoffs involved extending MongoDB v4.0.6. Section 3.2 supports our choice to design our mechanism as a separate module of the data store and discusses read-hints buffer properties. Section 3.3 explains that the design does not affect the replication and consistency model of the data store and that data consistency is preserved between in-cache and on-disk data within a replica node that updates its cache based on the received read hints buffer.

3.1 Capturing and Dissemination of Read Hints

Our implementation of read-hints module extends the MongoDB v4.0.6 codebase. MongoDB implements a passive replication system following the primary-backup replication model. The primary node serves client requests. In case of writes it applies the operation to its local state and updates its cache. Each update is also logged to its internal *oplog* data structure (Fig. 2). The updates are propagated to all replicas through the replication of the oplog. However the RHM is by design a separate module. The rationale for this decision, as opposed to integrating reads into the existing *oplog*, is discussed in Section 3.2.

The RHM monitors reads served on a primary extending the code path that handles the read request, cloning the read operation and channeling it to the RHM code. It keeps into RHB a minimal description of each read operation to allow a replica to locate the data, avoiding unnecessary information such as request type, session ID and hash signatures used for sanity checking on the primary node only. The RHB is periodically synced across replicas, with inter-node communication within a replica group specifically built for this purpose and implemented over TCP/IP sockets. The replication is performed by default in batches over 90ms intervals or when the RHB reaches its maximum capacity (by default 10,000 entries). We have empirically determined that these settings allow replicas to be reasonably up-to-date with minimal overhead (Section 4.7), but both parameters are configurable. We experiment with different settings in Section 4.

On a read-serving node, a background thread is responsible for periodically sync'ing the RHB to the replicas. To avoid contention between threads that update and disseminate RHB across replicas we apply double-buffering techniques, i.e., when the buffer is marked as ready to be shipped over the network, it is marked as read-only and swapped with a new empty buffer to log the read-hints. We also pre-allocate the memory buffers that support RHB to avoid allocation overheads. After the RHB synchronization is complete, the read-only buffer is marked as clear and ready to be reused.

Batching multiple reads allows for summarization, an optimization where multiple occurrences of the same request in the RHB may be consolidated to the last read of each key. Thus several requests for the same data are logged just once in the buffer, enabling a compact view of the requested data and reducing the total RHB size to be shipped across replicas. Other techniques [36] that further reduce the amount of transferred data over the network could also be applied.

On the receiving replica, a receiver thread, part of the RHM, is responsible for handling incoming connections from the serving node RHM and storing hints into its local buffer. Threads off of a read-applier thread pool are in charge of taking the read operations off the local RHB and applying them to the local copy of the database. During periods of resource strain, a secondary replica can apply a subset of read hints received or even turn RHM off while overload conditions last (Section 4.8). Threads from the read-applier pool shepherd requests through the same code path followed by a normal user read request (except this code path was previously only executed by the primary replica). In this way, the replica cache tracks the read working set along with the primary node's cache.

RHM allows the configuration on the number of replicas that will sync the RHB. It supports three options: *all, one, region-one*. With option *all* (*one*), all (one) replica(s) receive RHB updates from the primary node. Option *one* is useful when CPU and network bandwidth are scarce; on the downside, only one replica will be well prepared to serve reads. *Region-one* is used in geo-replicated systems and allows syncing the read log with one replica in each region. Furthermore in resource-constrained nodes where secondary replicas are co-located with primaries, the RHM can selectively apply or even turn off read hints, similar to adaptation strategies followed in prefetching systems [35].

When reconfiguration actions are known in advance, a potential alternative is to delay the dissemination of read hints to that time. However, such a delay is not expected to be beneficial. Storing the history of read hints at the read-serving node for a long time would require significant amounts of memory. In addition, disseminating read-hints in a burst just before a reconfiguration action could pose a significant performance hit and also delay the reconfiguration until the replica is prepared. Thus we believe that even in the case of frequent reconfiguration actions (such as in systems described in Section 2), there are practical benefits to the continuous dissemination of read-hints.

3.2 Read-Hints Buffer Properties

Our mechanism is by design a self-contained module (Fig. 2), ensuring that our implementation is generally applicable (not too tied to the design principles of a specific stores replication module) and can be easily integrated into most distributed data stores. Using existing data structures or mechanisms of the existing replication subsystem would require tight coupling with a specific store. In addition, our read hints buffers have different durability properties. Since the cache is essentially *soft state* that need not be persisted as a separate entity, we can treat the dissemination of the read as a hint mechanism to the replicas; thus in case of node failure, a lost RHB has no impact on data availability or recovery mechanism. This also allows to delay the replication of the hint buffers and transfer read ops in batches to amortize the transmission cost. Batching allows us to benefit from summarization and to further reduce the amount of transferred data (Section 4.10).

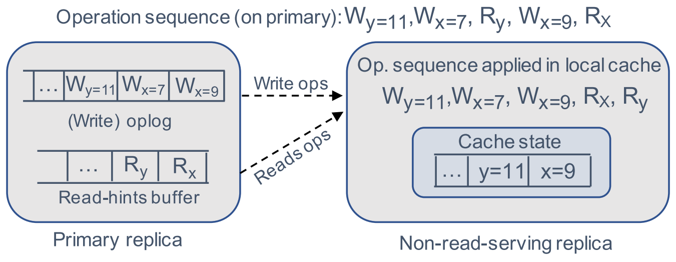
Operation sequence (on primary): $W_{y=11}, W_{x=7}, R_y, W_{x=9}, R_X$
... $W_{y=11}$ $W_{x=7}$ $W_{x=9}$
(Write) oplog
... R_y R_x
Read-hints buffer
Primary replica
Write ops
Reads ops
Op. sequence applied in local cache
$W_{y=11}, W_{x=7}, W_{x=9}, R_X, R_y$
Cache state
... y=11 x=9
Non-read-serving replica

Fig. 3. Reads may be re-ordered relative to writes, however caches always contain the latest state written to disk

The format of an RHB entry is simple: it carries only minimal description of the requested data (e.g., the key and some filters) derived from a read request –there is no need to include information about client, session or timestamps as described in Section 3.3– further contributing to a reduction of the amount of information sent over the wire.

3.3 Consistency

Consistency Across Nodes. Our mechanism does not affect the store's consistency model leaving the replication mechanism intact. Standard replication mechanisms dictate how committed state updates are eventually applied to all replicas. Updating replica caches by our read-operation dissemination mechanism ensures that a cache is refreshed with the most recent state that has already been applied on the same replica (stored on its disk). The shipping of read-operations (rather than shipping the data) and their execution, reflects the latest state known to replicas in their local cache. As such, the cache remains consistent with the on-disk state, which in turn follows whatever consistency guarantees are implemented in the specific distributed store.

Consistency Between in-Cache and On-Disk State. As reads are shipped independently of writes across replicas, there is a probability that reads may be reordered relative to writes (Fig. 3). This does not cause a problem as a read operation applied in the local cache of another replica will always reflect in memory the latest state written to disk at the time. In this way, cache contents always remain valid. The order of writes will remain as decided by the primary, and each write will consistently update cache and disk.

We note that correctness relies on correct implementation of atomic execution and serialization of read and write (updating cache and disk) operations in each replica's storage backend.

Cached Objects View Across Nodes. Even with our mechanism, caches in different nodes may have different contents due to the delay in disseminating operations as well as due to actions of the cache replacement policy. Typically, most systems feature a cache eviction policy that favors recently or frequently accessed objects (e.g., LRU, LFU). We expect that with our mechanism, a cache will contain the objects of the last applied RHB and the most recently accessed objects for any node (with the primary or read-serving nodes being somewhat ahead of others). It is not a goal of our work to keep caches fully in sync, and we think that there is little benefit in trying to achieve such a stringent objective. Our approach is a best effort mechanism aiming to help non-read-serving replicas track the read working set and thus be well prepared to serve future reads. In Section 4 we show

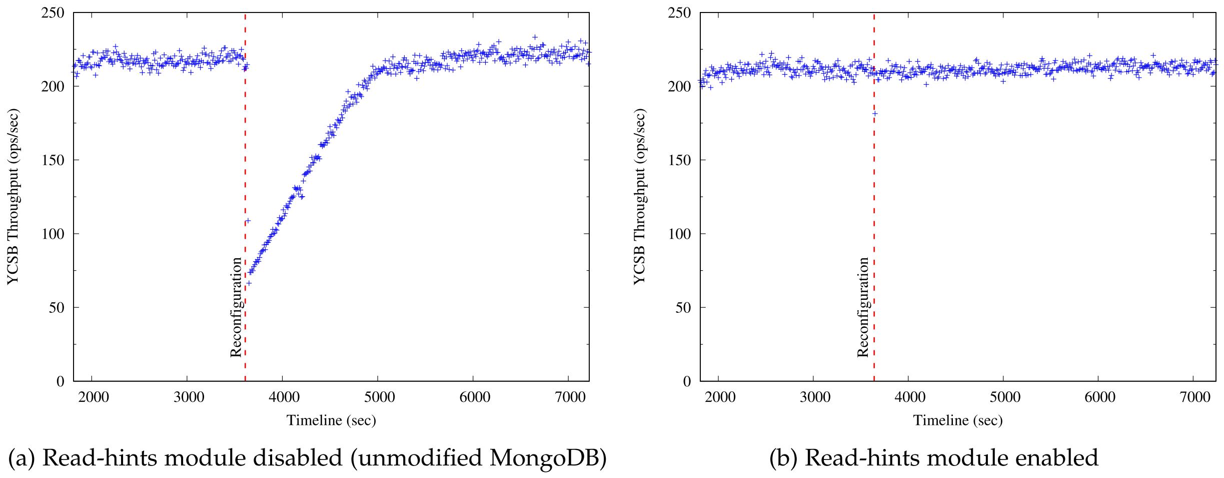

Fig. 4. Throughput under read-only workload, HDD used as back-end store.

that our mechanism is able to achieve our goal with no noticeable overhead.

4 Evaluation

In this section we evaluate the benefits of allowing non-read-serving replicas track the read working set using our extended prototype of MongoDB v4.0.6. The MongoDB binaries used have been compiled with debug symbols. Our results show that the system exhibits stable performance after shifting the node that serves reads from primary to the nearest secondary replica after the workload migration. The overhead of our mechanisms does not have a measurable impact on overall system performance (compared to the baseline implementation). Unless otherwise stated, all experiments use a non-sharded MongoDB installation with one replica group consisting of one primary and two replica nodes. Our main experimental testbed consists of four servers, each equipped with a Intel Xeon Bronze 3106 8-core 1.70GHz CPU, 16GB DDR4 2666MHz DIMMs, 256GB Intel D3-S4610 SSD and 2TB Ultrastar 7K2 HDD, running Ubuntu Linux 16.04.6 LTS, interconnected via a 10Gb/s Dell N4032 switch. Whenever a different testbed is used (such as AWS EC2 in Section 4.2), this is clearly stated in the text.

Our evaluation includes three workloads: the Yahoo Cloud Serving Benchmark (YCSB) [37] v0.11, TPC-C [38], a popular OLTP benchmark adapted for NoSQL systems [39]; and an analytical processing workload modeled after the popular TPC-H [40] benchmark. In all cases the workload driver executes on a dedicated server. In experiments where the workload switches the read-serving node to the nearest replica (to achieve the shortest possible latency), we side-step the fact that our testbed has a flat network topology (nearly identical latency across all replicas) and emulate the selection of the nearest replica via MongoDB's server tagging feature [41]. The dataset is loaded fresh onto MongoDB nodes before each experiment and caches are dropped before each run.

4.1 Performance Impact During Reconfiguration

In this section we evaluate our method under the YCSB workload generator. The benchmark is configured to produce two different mixes of reads versus updates/writes: a 100% read workload to stress our mechanism as the primary node has to keep up with a busy read-hints module, and 80% (reads)-20% (updates) to demonstrate performance under a mixed workload including both reads and writes.

Unless otherwise stated, the requested keys are randomly selected from a uniform distribution. The dataset consists of 50 million unique records result in 60GB of data on disk (including indexes) per node. The number of YCSB client threads (number of parallel connections between database client and servers) is set to 8, empirically determined to stress the cluster while keeping average response time under 20ms (considered a reasonable threshold). MongoDB is configured with 10GB of cache size. Read and write concern use the *majority* option by default.

Fig. 4 depicts YCSB throughput (ops/sec) under the read-only workload, with data stored on HDDs on each node. As in all subsequent figures, time (x-axis) starts 30 minutes into the experiment, when the system is deemed to have reached a steady state. The y-axis depicts throughput in YCSB ops/sec. Fig. 5 presents the MongoDB cache miss rate on node 1 (the primary node that serves client requests at the beginning of the run), and node 2 (the nearest replica serving reads after the workload migration). Figs. 4a and 5a correspond to unmodified MongoDB, while Figs. 4b and 5b correspond to our prototype. The vertical dashed line represents the transition of the read-serving node from node 1 to node 2.

Fig. 4a exhibits a clear performance hit for unmodified MongoDB (read-hints module is disabled), right after the reconfiguration at 3600 seconds. We observe system throughput (217 ops/sec before the reconfiguration) dropping to 65 ops/sec right after reconfiguration (a 70% reduction), taking 18 minutes to reach its pre-reconfiguration serving rate again. The performance hit is explained by the new serving node (node 2) high cache miss rate (up to 86%) at the early stages of its serving operation phase (Fig. 5a), significantly higher than the 31% miss rate that either node experiences while at steady state. Note that the reported cache miss rate takes into account data and index accesses, as MongoDB loads indexes into its internal cache to speed up read requests. In Fig. 5a, node 2 reports 0% cache activity before it starts serving client requests (as it does not perform any read operations); similarly node 1 does not serve reads after the workload migration

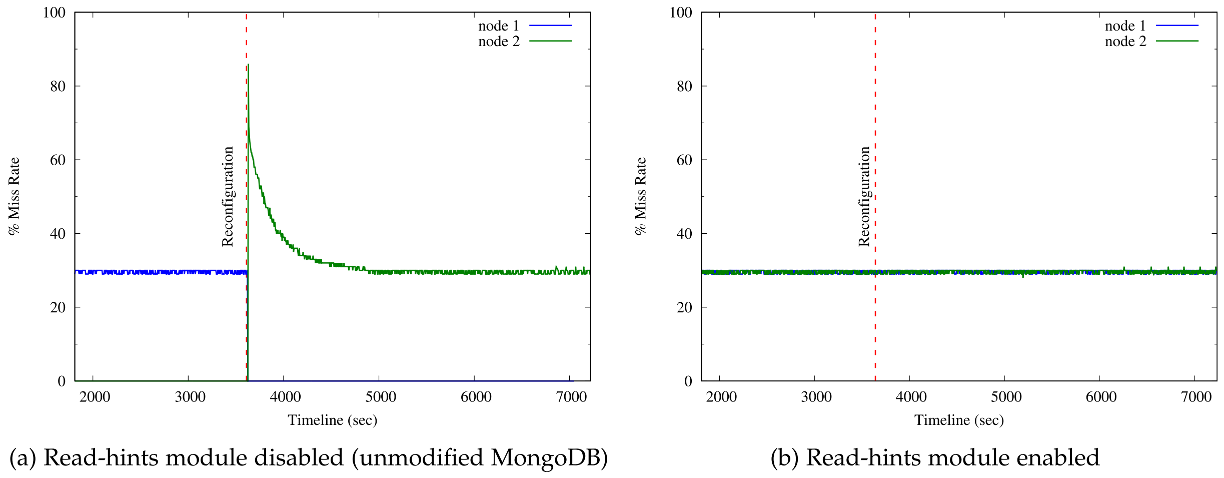

Fig. 5. Cache miss rate under read-only workload, HDD used as back-end store.

starts directing its requests to the nearest replica. We note that had node 2 been operating as a read-serving node in the recent past, the performance impact may be lower than observed in this experiment.

Fig. 4b depicts throughput with our RHM enabled. The reported throughput is about 211 ops/sec during the whole run of the experiment, while the MongoDB cache miss rate remains stable at 31% at all nodes during the entire run (Fig. 5b). We observe that our mechanisms are effective in keeping replica caches up to date, saving a 55% spike on the cache miss rate experienced by unmodified MongoDB when the nearest replica starts serving client requests. Unmodified MongoDB cannot meet its pre-reconfiguration throughput for a long period of time, even with a cache size of 10 GB.

For further insight into internal resource use, we report the total amount of physical memory used in each node (Fig. 6). In line with observations in Fig. 5b, it takes almost 20 minutes to bring data in memory right after reconfiguration at 3600 seconds with unmodified MongoDB (Fig. 6a, node 2). Our prototype (Fig. 6b) is able to maintain data in memory (application cache) in node 2, resulting in a smoother transition to the new configuration. Note that the memory reported as used exceeds 10GB (the MongoDB cache size) as it includes the memory used by the MongoDB process and the OS as well.

Next we study the effect of a faster storage device (SSD) on the performance impact during reconfiguration. In general, using SSDs we expect to be able to serve client requests and to recover (restore a replica's working set) at a higher rate. Fig. 7a depicts throughput with unmodified MongoDB using SSD as a back-end store. The system initially serves requests at a rate of 2260 ops/sec. At 1250 seconds we trigger the workload migration and the serving replica moves from node 1 to node 2, which causes a clear performance hit. The throughput drops at 1510 ops/sec (a 33.1% reduction), taking over 2 minutes to reach its stable state again of 2260 ops/sec, caused by cold-cache misses at the new serving node. Fig. 8a depicts MongoDB cache experiencing a 87% miss rate right after reconfiguration, before reaching again its stable rate at 31% until the end of the run. Our prototype is able to maintain throughput stable during the whole run (Fig. 7b) as the MongoDB cache-miss rate is not affected when moving the serving replica from node 1 to node 2 (Fig. 8b).

We note that with the read-hints module disabled, MongoDB cache-miss rate at the early stages of reconfiguration does not depend on the back-end store (HDD or SSD). This is expected as the dataset and cache capacity are the same in both cases. In the case of SSD, the system is able to recover faster as it can serve client requests at a higher rate. However,

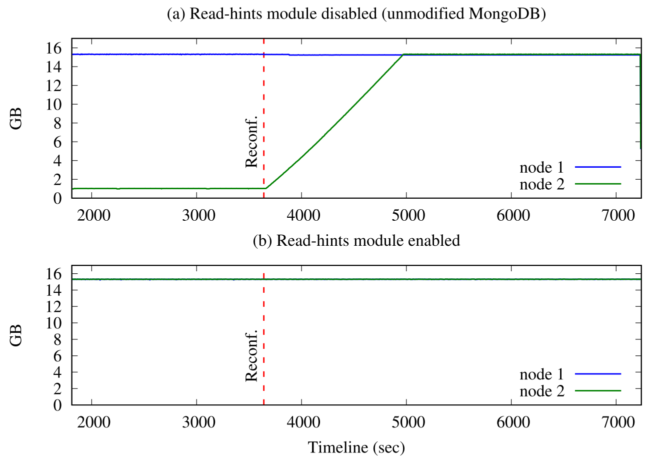

Fig. 6. Monitoring memory use.

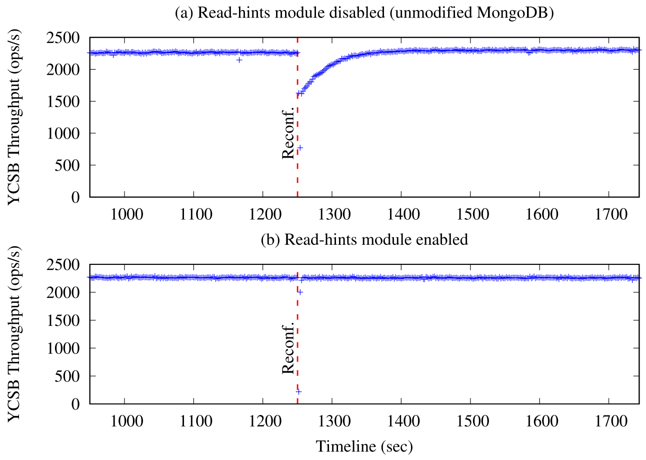

Fig. 7. Throughput under read-only workload, SSD used as back-end store.

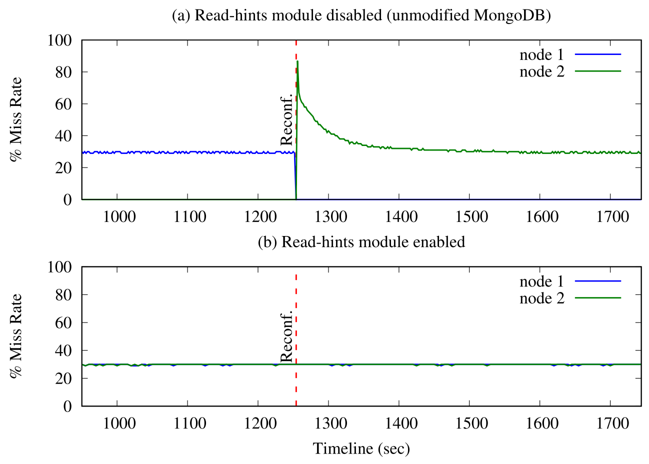

Fig. 8. Cache miss rate under read-only workload, SSD used a back-end store.

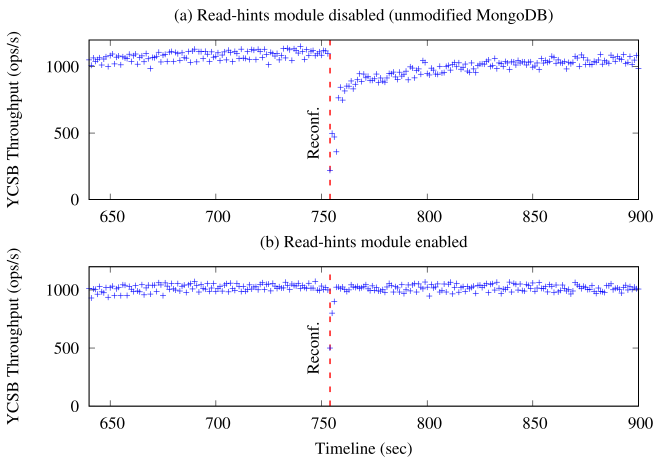

Fig. 9. Aggregate throughput of a multi-sharded replicated database on AWS EC2 under read-only workload.

recovery time would increase as cache sizes grow, as seen in Section 4.3. Overall we observe that even with faster storage devices, a primary-backup replication management system that executes reads only at the primary, exhibits a steep performance degradation for long periods of time (several minutes) right after reconfiguration, due to cold-cache misses at the new serving node. Our mechanisms are able to mask this impact with no measurable performance overhead in steady-state performance.

4.2 Multi-Shard Deployments on AWS EC2

Next we evaluate the RHM mechanism over a sharded MongoDB deployment using multiple replica groups. The evaluation was carried out on Amazon EC2 cloud platform, to validate portability and reproducibility of our results. Our database is split in three shards. Each shard comprises a replica group. Each server node hosts two MongoDB instances, a primary and a secondary replica, belonging to different shards. A multi-shard deployment requires a MongoDB metadata config server and a query router process (mongos), interfacing between client applications and the sharded cluster[2]. The 3 EC2 VMs allocated for the database nodes are of type r5a.xlarge featuring 4 vCPUs and 32GB of memory. Each MongoDB process is configured with 14GB of cache size. The metadata server is hosted on a c5.xlarge (4 high performance vCPUs) instance. We use a dedicated c5.xlarge instance for the YCSB workload generator.

Fig. 9 depicts the aggregate throughput of unmodified MongoDB versus with RHM enabled. In the case of unmodified MongoDB (top graph) we observe a clear performance hit during reconfiguration, with system throughput dropping from 1090 ops/sec before reconfiguration to below 500 ops/sec right after reconfiguration, requiring almost 2 minutes to fully recover to the pre-reconfiguration throughput. MongoDB with RHM enabled exhibits a smooth transition to the new configuration (Fig. 9b). The cache-miss ratio for unmodified MongoDB is up to 75% for the new primary node right after the reconfiguration, exhibiting a similar trend to the cache-miss ratio of experiments in Section 4.1.

2. https://docs.mongodb.com/manual/sharding

The aggregate steady-state throughput in this experiment is lower than reported in the experiments of Section 4.1. This is due to different specs of the two platforms and the additional entities (metadata configuration server and query router) required for a multi-shard MongoDB deployment.[2] However, both cases exhibit a similar trend: unmodified MongoDB suffers a significant performance hit due to cold-cache misses after a reconfiguration, whereas MongoDB with RHM enabled results in a much smoother transition.

4.3 Effect of Cache Size on Time to Restore Performance

In previous experiments we noted that the time to restore performance of unmodified MongoDB after the read-serving node changes, improves with a faster back-end store. In this section we investigate experimentally the impact of the size of the cache, an important issue in light of larger memory capacities typical of high-end enterprise servers. To evaluate the relationship between the time to restore performance after a read-serving replica change versus cache size, we perform experiments using unmodified MongoDB with an SSD back-end and different cache sizes, measuring the time to reach its pre-reconfiguration throughput after a primary change.

To ensure that we can control the memory available for caching, we had to regulate both MongoDB's own internal cache implemented within its storage engine (WiredTiger) as well as the filesystem cache. MongoDB' internal WiredTiger cache loads collection data and indexes and is of configurable size; however, the filesystem also indirectly caches MongoDB data and automatically uses free memory left unused by the WiredTiger cache or other processes. MongoDB thus benefits from both caches to reduce disk I/O. To effect a system-wide limit on cache size, we resorted to the Linux control groups (cgroups) feature that can limit the total memory available for all caches (MongoDB internal and filesystem caches).

Fig. 10 shows that there is a direct relationship between recovery time (time to refill the cache) and cache size. Increasing cache capacity from 1GB to 10GB leads to longer time to refill the cache, taking over 2 minutes to reach steady

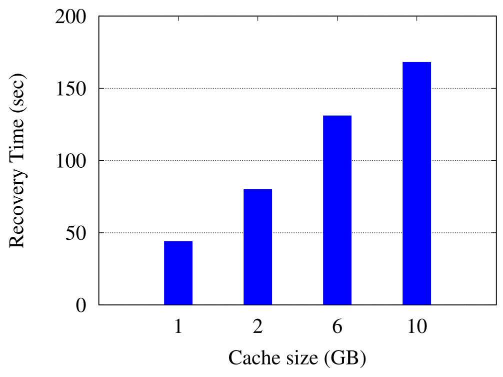

Fig. 10. Time to restore performance level versus cache size.

state performance level with 10 GB of cache as it has to bring more data into memory to fill the cache and reach steady state. Production deployments with even larger memory capacities (orders of magnitude larger memories are very common in enterprise environments) are expected to result in much longer periods of low system performance. Large caches are expected to be characterized by low response times (during steady-state performance) and long time to refill after a reconfiguration, yielding prolonged periods of large (as a ratio of recovery versus steady-state performance) SLO violations, making a strong case for the mechanisms proposed in this paper.

4.4 Effect of Cache Access Pattern

To determine what (if any) is the impact of the cache access pattern, we run experiments with the read-only workload and a Zipf-distributed (rather than uniform) access pattern. We thus configure YCSB to select keys using the Zipf distribution, which exhibits a stronger temporal locality and is expected to increase the efficiency of cache and result in higher overall throughput.

In Fig. 11 we observe that throughput is indeed increased to 2689 ops/sec. MongoDB (both the unmodified version and our prototype) exhibits a stable miss rate at 18% over the entire run. Following a reconfiguration, the throughput of unmodified MongoDB drops by 39% due to an increase of the cache miss rate. The cache behavior follows a similar trend to the experiments under uniform distribution (Fig. 8). After the reconfiguration, the cache miss rate increases to 80% (Fig. 12). As the new serving node is warming up its cache, it takes almost up to 2 minutes for the system to reach the steady-state throughput following the reconfiguration. Our prototype exhibits a smooth transition to the new serving replica, just as in previous cases. Thus, even though we have stronger locality and more efficient use of cache in this case, we observe similar behavior to the experiment with uniform distribution.

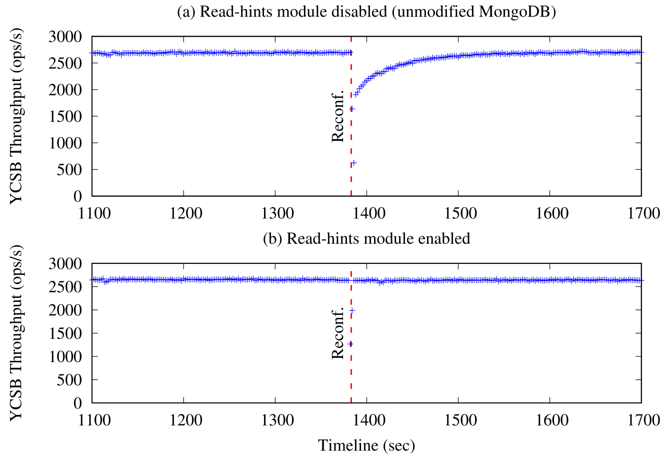

Fig. 11. Throughput under read-only workload, Zipf distribution, SSD back-end store.

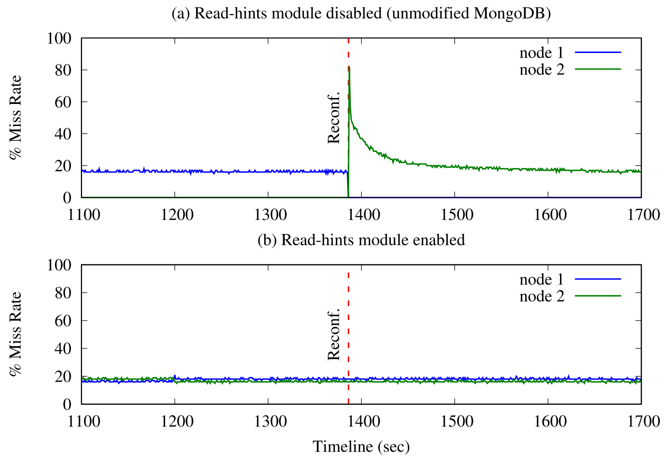

Fig. 12. Cache miss rate under read-only workload, Zipf distribution, SSD back-end store.

4.5 Read-Write Workload

In the 100%-read workload evaluated so far, backup replicas in unmodified MongoDB did not see any of the read operations, explaining the high miss rates experienced after reconfiguration. As soon as writes are introduced into the mix, however, backup replicas have a way to learn of updates (as writes are propagated through the stores replication mechanism). Thus, *if the working set of future reads has strong spatial and temporal overlap with the working set of past writes*, it is expected that the performance impact of a reconfiguration action will be reduced. Indeed, our experiments showed that when the percentage of writes issued by YCSB increase, the performance impact experienced by unmodified MongoDB diminishes, due to the fact that reads and updates access the same set of keys. However, in several applications (e.g., in typical big-data analytics scenarios where updates are inserting new records, whereas reads are accessing past -historical- records) this is not expected to be the case. To characterize such a scenario, we set up an experiment with an 80%-20% read-write workload mix. However, in this case the read and writes are performed on a different set of keys, using a uniform distribution for both operation types.

Fig. 13a depicts results with unmodified MongoDB, with read throughput exhibiting a clear performance hit, up to 58% during reconfiguration. This is due to the high cache miss rate (89%) on the new serving replica right after the workload migration (Fig. 14a). Write throughput is not affected by the reconfiguration as caching does not directly affect write performance.

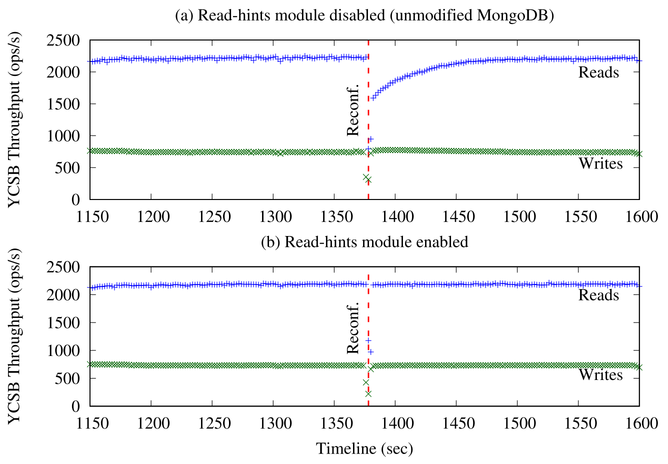

Fig. 13. Throughput under read-write workload, uniform distribution, SSD back-end store.

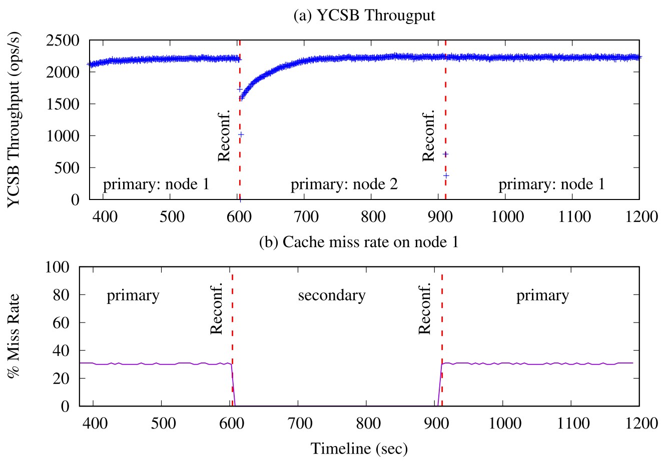

Fig. 15. Node 1 serves as primary again at 600 sec after a second reconfiguration (RHM disabled, same working set).

4.6 Re-Electing a Past Primary

In previous sections, the replica node that takes over as a primary has an empty cache as would be the case if that node recently joined the group. In a system that has been online for some time and, especially for a node that has served as a replica-group primary in the past, it may be the case that its cache has some or all of the key-values in the current working set of the client application, thus resulting in few or no cache misses after the reconfiguration. In Fig. 15a (YCSB throughput) we exhibit the outcome of a second reconfiguration re-electing as leader node 1 at 610 sec, after having served as leader in 0-300 sec, with RHM disabled. The throughput remains unaffected during the second reconfiguration as YCSB clients access the same working set during the run. In Fig. 15b we observe that node 1's cache miss rate does not increase when node 1 becomes primary again. Several practical factors however can still render a past leader's cache cold: One such case is when an application's working set changes over time, a fact that has been supported by analyses of real environments [2]. Other factors that may occasionally wipe out the memory contents of replica nodes are (1) replica migrations that often take place in the background for load balancing or management needs, or (2) power losses and reboots.

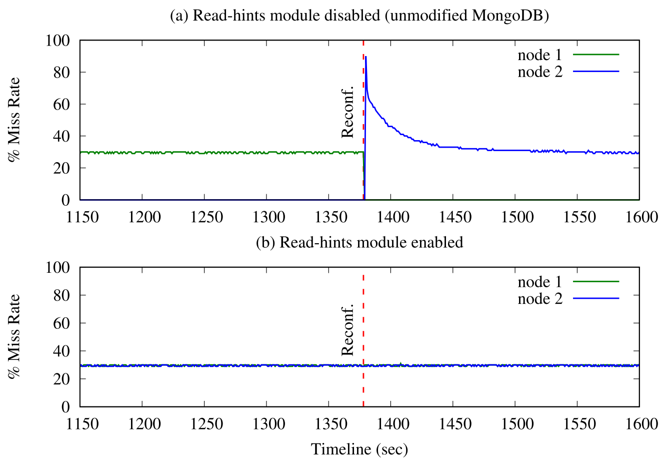

Fig. 14. Cache miss rate under read-write workload, uniform distribution, SSD back-end store.

To highlight the impact of a time-varying working set when RHM is disabled, we set up an experiment[3] featuring two synthetic workloads (two YCSB instances) with different working sets. Again we perform two reconfigurations, with node 1 re-elected primary for a second time at 580 sec (Fig. 16). Initially, node 1 is primary and serves requests from clients accessing key set 1. At 435 sec it steps down and node 2 takes over as primary. At about the same same time, an additional set of clients starts accessing a different set of keys (key set 2). As node 2 has an empty cache (first time serving as primary), we observe a performance hit of up to 65% (Fig. 16). We note the reduced throughput per client group in 435-560 sec as they share the replica group. At 560 sec the set of clients accessing key set 1 stop executing, briefly improving throughput for clients accessing key set 2. At 580 sec, node 1 is promoted to primary again. While node 1 has keys in its cache, they are no longer useful as the working set has changed, resulting again in a performance drop. Enabling RHM (not shown in Fig. 16) results in a smooth transition from node 1 to node 2 (for clients accessing key set 1) and back to node 1 (for clients accessing key set 2).

4.7 Performance Overhead

In this section we study the performance overhead of our mechanisms, namely the monitoring and logging of read requests into in-memory buffers (the read-hits buffers) and their communication to replica nodes. We focus on the performance impact on throughput under a 100% read workload, in order to fully stress our mechanisms which only apply to read operations.

We repeat the experiment of Section 4.1 using SSD as back-end store and we measure the performance overhead in terms of throughput while the system is at steady state without applying a change of the serving node. In this case the CPU utilization is at 80% (combined user and system time). We break overhead in two parts: (a) the impact of logging every read request and (b) the impact of replicating the read log under different configurations. In our first

3. We use a slightly different testbed (Intel Xeon Bronze 3206 8-core CPU 1.90GHz, 32GB DDR4, Micron 5200 MAX SSD) in this experiment.

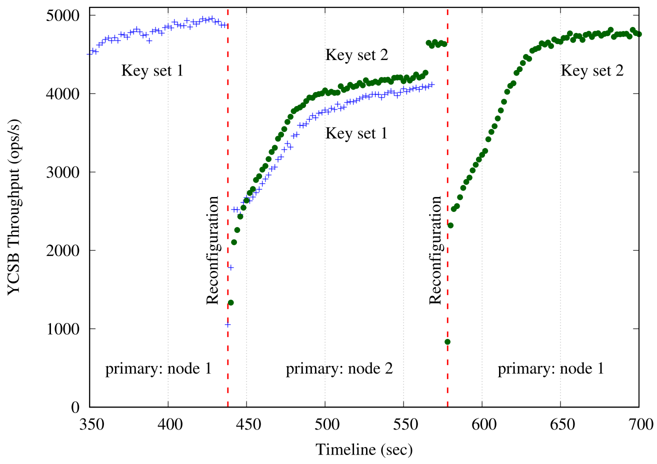

Fig. 16. Node 1 serves as primary again at 580 sec after a second reconfiguration (RHM disabled, working set changes).

experiment we enable the logging mechanism but do not replicate the read log. Although we do not replicate the read log across replicas, we still swap the read-log buffer (part of the double buffering being performed, Section 3.1) every 90 ms. The average performance impact of logging is found to be 0.6% (average of 10 runs with a relative standard deviation of 0.16%).

We next study the impact of our mechanism with the replication of read log enabled. The reported impact is cumulative, meaning that it includes the cost of logging of read requests and the cost of replication of the read log buffers. We experiment with different replication intervals and number of replicas to sync the read log with. When only one replica gets the read-log updates the performance overhead is 0.9%, increasing to 1.6% when all replicas receive the updates (average of 10 runs for each configuration, with relative standard deviation 0.25% and 0.21% respectively). Experimenting with different replication intervals (90ms and 1000ms) does not seem to have any impact on the overall system performance. A short replication interval creates small frequent batches of read log that are replicated across replicas while a long interval creates less frequent but bursty batches. Overall, we find that the performance impact of our mechanism to maintain up-to-date read caches across replicas is minimal even under stress.

Finally, we focus on the cost of RHM maintenance under resource strain, caused by the sharing of resources between a primary and a backup replica co-located on the same node on AWS EC2. We use a single-shard deployment and focus on the performance (throughput) of a single node hosting the primary replica and a backup. To fully stress the RHM system, we use a read-only workload. We compare unmodified MongoDB to the same system with RHM enabled with a workload that drives the former at an average CPU utilization of 70%, considered fully utilized. At that point, we observe 4.65% lower throughput for MongoDB with RHM versus the unmodified, as seen by clients. For a lighter loaded server, we observe no noticeable impact on overall system performance. While the impact is considered low even under stress, RHM could be temporarily stopped or selectively apply read hints to reduce it even further (Section 4.8).

TABLE 1
Applying Fewer Read-Hints Lowers RHM Overhead, at the Cost of Reduced Cache Efficiency After Reconfiguration

%Hints	Cache	CPU		Disk reads	
Applied	miss rate	usr+sys	I/O wait	MB/s	ops/s
10%	39%	5.2%	0.37%	8.8	484
20%	33%	7.3%	2.36%	16.2	794
33%	31%	10.2%	4.26%	25.1	1137
50%	30%	13.4%	5.97%	34.9	1501
100%	30%	24.7%	8.62%	60.9	2426

4.8 Selectively Applying Read Hints

As noted in Section 3, a secondary replica can selectively apply the read hints it receives via RHM from the primary to reduce the overhead of the mechanism. Here we quantify the overhead (CPU and read I/O) attributed to RHM when applying a fraction of the read hints received (10%, 20%, 33%, 50%) under a YCSB workload with a uniform access pattern. Our results highlight the point that RHM can be applied in an adjustable manner, in line with the amount of resources available to a specific deployment at any point in time. In addition, RHM can be switched off during periods of excessive load, and turned back on after such periods or after more resources are provided to a deployment.

Table 1 shows that as the amount of hints applied is reduced, RHM overhead (CPU and read I/O[4]) is also reduced while the cache miss rate on the node that takes over as primary after reconfiguration increases. The cache miss rate on the primary node (which applies all read requests) is 30% at steady state. We also observe that applying 50% of read hints on replicas matches the miss rate on the primary allowing a smooth primary transition between nodes. This is evidence that significant benefits are possible by applying even a small fraction (10-33%) of read hints.

Another key point is that the RHM mechanism can be turned off during periods of overload (90-100%). These are not expected to last too long, either because they are typically due to bursts and expected to recede or will be absorbed by adding new resources (elasticity actions), at which point RHM dissemination can resume at a level (% of hints) adjusted to the current level of load.

4.9 Space Overhead

Read-oplog entries contain just the minimum description of requested data (IDs and some filters) necessary to identify the requested data. To reduce the read log size and the amount of data sent over the wire, we omit (unnecessary to the replicas) information regarding the session id, signature hash and other fields of the read request used on primary node. A replica that receives read log batches can reconstruct and apply the corresponding read requests. In this case the space overhead of the read log entry is 39 bytes (the collection name and the contents of the filter).

As the read log is replicated in batches across replicas, the communication interval affects the size of each batch but not the total data transferred over the network. A

4. We use `mpstat` and `iostat` to collect CPU and I/O statistics respectively; the monitored I/O device is used exclusively by MongoDB

replication interval of 1000 ms corresponds to a batch size of 2,260 log entries (matching the serving rate of read requests reported in Section 4.1 using SSD as back-end store). A replication interval of 90 ms (default in our prototype) corresponds to a batch size of 205 entries. In both cases, 2,260 requests/sec are communicated to replicas, resulting in 88 Kbytes/sec per replica. The required network bandwidth to replicate the read log is analogous to the serving rate of reads on primary. However we communicate only a minimal description of read requests as hints and not the data itself. In the experiments described in this section, we log all read requests. In Section 4.10, we evaluate our optimizations to further reduce the size of the read log.

4.10 Optimizations to Reduce Read-Hints Buffer Size

Batching reads operations provides an opportunity to reduce the total size of the read log to be shipped across replicas as multiple requests for the same data can be logged just once (summarized) in the buffer. This depends on the access pattern of data, the replication interval of read log (as summarization can be performed for requests within the same interval) as well as the serving rate of the system. A system with high serving rate using a long replication interval and featuring strong locality in the access pattern, can greatly reduce the amount of transferred data across replicas.

In this section we study two different access patterns (uniform, Zipf) under two different read log replication interval settings using the fixed serving rate of our system using SSD back-end. Under uniform accesses, almost all the requested keys in every interval are unique, even when the duration of the interval is set to 1,000 ms. Due to the relatively large dataset size (50 million unique keys) it is highly unlikely that during an interval the driver selects the same keys more than once. Under the Zipf distribution, we find that 12% fewer data are transferred to replicas when we use the default read log replication interval of 90 ms. Setting the interval to 1000 ms, 21% fewer read log data are transferred over the network to replicas.

When network bandwidth is scarce, one could apply compression or use more sophisticated deduplication techniques [36] on the read log to further reduce network traffic.

4.11 TPC Workloads

This section extends our evaluation using workloads modeled after two popular TPC benchmarks, TPC-C (online transaction processing) and TPC-H (online analytical processing).

4.11.1 TPC-C

We experimented with TPC-C [38], a popular online transaction processing benchmark emulating a commerce system with five types of transactions. While initially designed to test traditional RDBMS systems, we used a recent implementation that adapts it to match the NoSQL document data model and to test transactional features [39] [42].

The database is populated to simulate 500 warehouses resulting in 66 GB of data size (41.5 GB on disk space using compression). To have a continuous view of performance, we adapted the benchmark to report throughput and response time for each query every 2 seconds.

Queries are served by the primary node at the start of the run. Initially we aimed to shift the read-serving node to the nearest secondary during the run. However, in this way we could re-target only the STOCK LEVEL query (a read-only query) [39] towards the secondary node. We thus moved towards scenarios where we switch the primary node within the replica group. This may occur during a failover action or to improve performance [18], [19], [22], [23]. In this case, all transactions are executed in the node that serves as primary. However, switching the primary yields minimal impact on performance (we observe no cold-cache misses). An analysis showed that this is because most of the queries have a similar pattern (*read, then update (modify) the same keys*), thus eventually propagating the read working set to all replica caches, achieving as a side effect the benefits of our mechanism. We wanted to investigate whether a different type of workload, online analytical processing, has different characteristics. Our results are summarized next.

4.11.2 TPC-H

TPC-H [40] emulates a decision support system or business intelligence database environment tasked with providing answers for business analyses on a dataset, initially designed for traditional RDBMSs. The workload is generally read- and scan-intensive. We implemented a subset of the benchmark,[5] namely the ”Pricing Summary Report Query (Q1)” compatible with the MongoDB query model. This query reports the amount of business that was billed, shipped, and returned. The query exhibits a high degree of complexity. As it has to scan and fetch data from disk and then perform aggregation operations on them (e.g., sum, average), it is both I/O and CPU intensive. We load the database using the DBGen[6] tool using a scaling factor of 10. To improve transaction performance, we built indexes on fields used for the query. The total database size is 29 GB (resulting in 13 GB on-disk allocated space using compression). The query parameters (e.g., the shipdate interval) are randomly selected.

Initially the query is executed on the primary node with a warm cache (having previously executed another instance of the query), achieving an execution time of 1.2 sec with a nearly 0% cache miss rate. After workload migration (changing read replica) with our mechanism enabled, the query execution time is the same as the new read node has a warm cache. After disabling our mechanism however, replica caches cannot stay in sync with the primary node's cache. Thus, after a migration, query execution time increases to 6.3 sec (a 5.25 times slower query execution time) due to initial cold-cache misses. Fig. 17 shows that the cache miss rate is between 60% and 40% during the execution of the query that performs read and scan operations. Our mechanism is able to maintain warm caches across replicas, achieving an 80% reduction of execution time.

5. The database schema could be adapted to better fit the NoSQL document data model but this is out of scope of this work

6. DBGen is a TPC provided software package that must be used to produce the data used to populate the database.

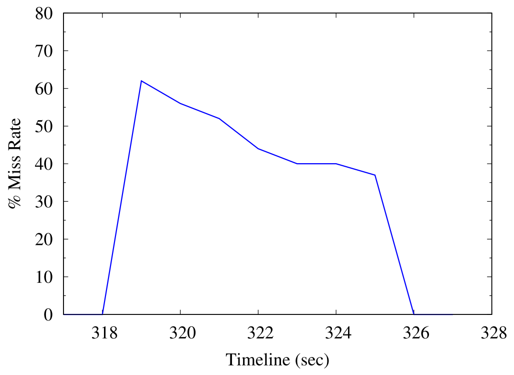

Fig. 17. Cache miss rate after migration of the analytics query.

5 Conclusion

In this paper we propose a low-overhead mechanism for maintaining warm read caches across all replicas in systems that serve reads from a small number of (typically one) replicas. We find that the mechanism can have significant performance benefits during reconfiguration actions, avoiding large performance drops for long periods of time (order of minutes). The performance drops avoided by our mechanism are observable even with faster storage devices and are proportional to the size of the cache. Mixed (read/write) workloads featuring non-overlapping read and write working sets are also exposed to the problem described in this paper and benefit from our solution. All in all, the proposed mechanism is low cost, easy to implement and retrofit in existing systems, and results in significant benefits during reconfiguration actions. Its low performance impact observed under periods of resource strain can be avoided by reducing or stopping the maintenance of read hints during such periods. While the benefits of our mechanism are not on the common path of system performance, the challenge addressed in this work has the potential to be even more impactful in the future, as reconfigurations become more frequent for management actions such as replica rejuvenation, proactive recovery, adaptive placement, and to mask the performance impact of resource-heavy activities.

References

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

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *Proc. ACM SIGMETRICS Perform. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, “Distributed systems,” S. Mullender, Ed., 2nd ed. USA: Addison Wesley, pp. 199–216, 1993.

[4] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 305–320.

[5] B. Oki and B. Liskov, “Viewstamped replication: A new primary copy method to support highly-available distributed systems,” in *Proc. Annu. ACM Symp. Princ. Distrib. Comput.*, 1988, pp. 8–17.

[6] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2011, pp. 245–256.

[7] R. van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *Proc. Symp. Oper. Syst. Des. Implementation*, 2004, pp. 91–104.

[8] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[9] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, “Paxos replicated state machines as the basis of a high-performance data store,” in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 141–154.

[10] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proc. Annu. ACM Symp. Princ. Distrib. Comput*, 2007, pp. 398–407.

[11] C. Gray and D. Cheriton, “Leases: An efficient fault-tolerant mechanism for distributed file cache consistency,” in *Proc. ACM Symp. Operating Syst. Princ.*, 1989, pp. 202–210.

[12] B. W. Lampson, “How to build a highly available system using consensus,” in *Proc. Int. Workshop Distrib. Algorithms*, 1996, pp. 1–17.

[13] I. Moraru, D. G. Andersen, and M. Kaminsky, “Paxos quorum leases: Fast reads without sacrificing writes,” in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–13.

[14] D. K. Gifford, “Weighted voting for replicated data,” in *Proc. ACM Symp. Oper. Syst. Princ.*, 1979, pp. 150–162.

[15] G. DeCandia *et al.*, “Dynamo: Amazon’s Highly Available Key-value Store,” in *Proc. ACM SIGOPS Symp. Oper. Syst. Princ.*, 2007, pp. 205–220.

[16] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, “Probabilistically bounded staleness for practical partial quorums,” *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[17] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme, “Are quorums an alternative for data replication?,” *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 257–294, 2003.

[18] M. S. Ardekani and D. B. Terry, “A self-configurable geo-replicated cloud storage system,” in *Proc. USENIX Conf. Oper. Syst. Des. Implementation*, 2014, pp. 367–381.

[19] S. Liu and M. Vukolić, “Leader set selection for low-latency geo-replicated state machine,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1933–1946, Jul. 2017.

[20] A. Sharov, A. Shraer, A. Merchant, and M. Stokely, “Take me to your leader! online optimization of distributed storage configurations,” in *Proc. Int. Conf. Very Large Data Bases*, 2015, pp. 1490–1501.

[21] O. Wolfson, S. Jajodia, and Y. Huang, “An adaptive data replication algorithm,” *ACM Trans. Database Syst.*, vol. 22, no. 2, pp. 255–314, 1997.

[22] A. Papaioannou and K. Magoutis, “Replica-group leadership change as a performance enhancing mechanism in NoSQL data stores,” in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1448–1453.

[23] P. Garefalakis, P. Papadopoulos, and K. Magoutis, “ACaZoo: A distributed key-value store based on replicated LSM-trees,” in *Proc. IEEE Int. Symp. Reliable Distrib. Syst.*, Oct. 2014, pp. 211–220.

[24] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[25] Dynamic snitching in Cassandra: Past, present, and future. Accessed: Dec. 2021. [Online]. Available: https://www.datastax.com/blog/2012/08/dynamic-snitching-cassandra-past-present-and-future

[26] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, “C3: Cutting tail latency in cloud data stores via adaptive replica selection,” in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 513–527.

[27] Riak, “Load balancing and proxy configuration,” Accessed: Dec. 2021. [Online]. Available: https://docs.riak.com/riak/kv/2.2.3/configuring/load-balancing-proxy

[28] “How are read requests accomplished?,” Accessed: Dec. 2021. [Online]. Available: https://docs.datastax.com/en/ddac/doc/datastax_enterprise/dbInternals/dbIntClientRequestsRead.html

[29] N. Kolettis and N. D. Fulton, “Software rejuvenation: Analysis, module and applications,” in *Proc. Int. Symp. Fault-Tolerant Comput.*, 1995, pp. 381–390.

[30] V. V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. Bessani, “FITCH: Supporting Adaptive Replicated Services in the Cloud,” in *Proc. Distrib. Appl. Interoperable Syst.*, 2013, pp. 15–28.

[31] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Synthesis Lectures on Computer Architecture*, M. Martonosi, Ed., 3rd ed. San Rafael, CA, USA: Morgan & Claypool, 2019.

[32] D. Stamatakis, N. Tsikoudis, E. Micheli, and K. Magoutis, “A general-purpose architecture for replicated metadata services in distributed file systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2747–2759, Oct. 2017.

[33] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan, "A status report on research in transparent informed prefetching," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 2, pp. 21–34, 1993.

[34] Y. Chen, S. Byna, and X.-H. Sun, "Data access history cache and associated data prefetching mechanisms," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, pp. 1–12.

[35] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *SIGMETRICS Perform. Eval. Rev.*, vol. 23, no. 1, pp. 188–197, 1995.

[36] L. Xu, A. Pavlo, S. Sengupta, J. Li, and G. R. Ganger, "Reducing replication bandwidth for distributed document databases," in *Proc. ACM Symp. Cloud Comput.*, 2015, pp. 222–235.

[37] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[38] TPC-C. Accessed: Dec. 2021. [Online]. Available: http://www.tpc.org/tpcc/

[39] A. Kamsky, "Adapting TPC-C Benchmark to Measure Performance of Multi-Document Transactions in MongoDB," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2254–2262, Aug. 2019.

[40] TPC-H. Accessed: Dec. 2021. [Online]. Available: http://www.tpc.org/tpch/

[41] MongoDB Server Manual: Read preference. Accessed: Dec. 2021. [Online]. Available: https://docs.mongodb.com/manual/core/read-preference/

[42] TPC-C in Python for MongoDB. Accessed: Dec. 2021. [Online]. Available: https://github.com/mongodb-labs/py-tpcc

Antonis Papaioannou received the PhD degree from the Computer Science Department, University of Crete, in November 2021. He is currently a postdoctoral researcher with the Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH). His research interests include distributed systems, adaptation mechanisms in replicated data stores and stream processing systems.

Kostas Magoutis received the BSc degree from AUTH, Greece, in 1993, the MA degree from BU, in 1996, and the PhD degree in computer science from Harvard University, in 2003. He is currently an associate professor with the Computer Science Department, University of Crete, Greece, and collaborating researcher with ICS-FORTH. His research interests include scalable and highly-available distributed systems and data-intensive services.