Fine-Grained Replicated State Machines for a Cluster Storage System

Ming Liu[*] *Arvind Krishnamurthy*[*] *Harsha V. Madhyastha*[†] *Rishi Bhardwaj*[‡] *Karan Gupta*[‡]

Chinmay Kamat[‡] *Huapeng Yuan*[‡] *Aditya Jaltade*[‡] *Roger Liao*[‡] *Pavan Konka*[‡] *Anoop Jawahar*[‡]

Abstract

We describe the design and implementation of a consistent and fault-tolerant metadata index for a scalable block storage system. The block storage system supports the virtualized execution of legacy applications inside enterprise clusters by automatically distributing the stored blocks across the cluster's storage resources. To support the availability and scalability needs of the block storage system, we develop a distributed index that provides a replicated and consistent key-value storage abstraction.

The key idea underlying our design is the use of fine-grained replicated state machines, wherein every key-value pair in the index is treated as a separate replicated state machine. This approach has many advantages over a traditional coarse-grained approach that represents an entire shard of data as a state machine: it enables effective use of multiple storage devices and cores, it is more robust to both short- and long-term skews in key access rates, and it can tolerate variations in key-value access latencies. The use of fine-grained replicated state machines, however, raises new challenges, which we address by co-designing the consensus protocol with the data store and streamlining the operation of the per-key replicated state machines. We demonstrate that fine-grained replicated state machines can provide significant performance benefits, characterize the performance of the system in the wild, and report on our experiences in building and deploying the system.

1 Introduction

Enterprise clusters often rely on the abstraction of a block storage volume to support the virtualized execution of applications. Block storage volumes appear as local disks to virtual machines running legacy applications, even as the storage service distributes volume data across the cluster. The storage system provides ubiquitous access to volumes from any node in the cluster and ensures durability and availability through replication.

Our work is in the context of a commercial enterprise cluster product built by Nutanix, a software company that specializes in building private clouds for enterprises. VMs deployed in these clusters rely on a cluster block storage system, called Stargate. As with other block storage systems [8,10,27,29,31], Stargate provides a virtual disk abstraction on which applications/VMs can instantiate any file system. However, unlike most other block storage systems, Stargate co-locates both computing and storage on the same set of cluster nodes. This approach provides cost, latency, and scalability benefits: it avoids needing to provision separate resources for computing and storage, it allows for local access to storage, and it lets both storage and compute scale with the cluster size.

A key component of such a system is the metadata index, which maps the logical blocks associated with a virtual disk to its actual physical locations. Just like the overall system, this mapping layer should provide high performance and strong consistency guarantees in the presence of failures. These requirements suggest a design with the following elements: (a) achieve high throughput and scalability by distributing the index as key-value pairs and utilizing all the cluster nodes, (b) ensure availability and consistency by replicating key-value pairs and using a consensus algorithm, such as Paxos [16] or Viewstamped Replication [25], to implement replicated state machines (RSMs), and (c) ensure durability of a node's shard of key-value state by employing a node-level durable data structure such as the log-structured merge tree (LSM).

This traditional approach to building a distributed index has drawbacks in our specific context where: (a) all operations, including metadata operations, have to be made durable before they are acknowledged, (b) there is significant variation in operation execution latency, and (c) the distributed index service has to share compute and storage with the rest of Stargate and application VMs. In particular, the use of a per-shard consensus operation log, which records the order of issued commands, introduces inefficiencies, such as short- and long-term load imbalances on storage devices, sub-optimal batching of storage operations, and head-of-line blocking caused by more expensive operations.

To address these issues, we develop a design that uses *fine-grained replicated state machine (fRSMs)*, where each key-value pair is represented as a separate RSM and can operate independently. This approach allows for flexible and dynamic scheduling of operations on the metadata service and enables effective use of the storage and compute resources. To efficiently realize this approach, we use a combination of techniques to streamline the state associated with the object radically. In particular, our approach uses no operation logs and maintains only a small amount of consensus state along with the perceived value of a key. We also address performance and consistency issues by co-designing the consensus protocol and the local node storage, providing strong guarantees on operation orderings, and optimizing failure recovery by enhancing the LSM data structure to handle the typical failure scenarios efficiently. It is worth noting that our innovation is not in the consensus protocol (as we merely borrow elements from Paxos and Viewstamped Replication), but in exploring

[*] University of Washington
[†] University of Michigan
[‡] Nutanix

an extreme operating point that is appropriate for balancing load across storage and compute resources in a managed environment with low downtimes.

We present experimental evaluations of our implementation both in a controlled testbed as well as in production deployments. Compared with traditional coarse-grained RSMs, fRSMs achieve 5.6× and 2.3× higher throughput for skewed and uniform scenarios in controlled testbeds. The resulting implementation is part of a commercial storage product that we have deployed on thousands of clusters over the past eight years. To date, we have not had a data loss event at any of these deployed production sites. We have also been able to leverage the metadata store for other applications such as write-ahead logs and distributed hypervisor management.

2 Motivation

We begin with a description of our setting and our goals. We then describe a baseline approach and discuss its shortcomings that motivate our work.

2.1 Metadata Storage Overview

Setting. Our work targets clusters that are typically used by enterprises as private clouds to perform on-premise computing. Customers instantiate virtual machines (VMs) that run legacy applications. The cluster management software then determines which node to run each VM on, migrating them as necessary to deal with faults and load imbalances.

Our Stargate storage system provides a virtual disk abstraction to these VMs. VMs perform reads and writes on the virtual disk blocks, and Stargate translates them to the appropriate accesses on physical disks that store the corresponding blocks. Stargate stores the blocks corresponding to virtual disks on any one of the cluster nodes on which user VMs are executed, thus realizing a *hyper-converged cluster infrastructure* that co-locates compute and storage. An alternate approach would be to use a separate cluster of storage nodes (as is the case with solutions such as SAN) and provide the virtual disk abstraction over the network. Nutanix employs co-location as it reduces infrastructure costs and allows the storage system to flexibly migrate data blocks accessed by a VM to the node on which the VM is currently hosted, thereby providing low latency access and lowering network traffic.

Metadata storage. In this paper, we focus on how Stargate stores the metadata index that maps virtual disk blocks to physical locations across the cluster. One can implement the virtual disk abstraction by maintaining a map for each virtual disk (vDisk) that tracks the physical disk location for every block in that vDisk. Our design, outlined below, introduces additional levels of indirection to support features such as deduplication, cloning and snapshotting. It also separates *physical maps* from *logical maps* to allow for decoupled updates to these maps.

A virtual disk is a sequence of *extents*, each of which is identified by an *ExtentID*. An extent can be shared across virtual disks either because of the deduplication of disk blocks or snapshotting/cloning of virtual disks. Extents are grouped into units called extent groups, each of which has an associated *ExtentGroupID*, and each extent group is stored as a contiguous unit on a storage device. Given this structure, the storage system uses the *vDisk Block Map* to map portions of a vDisk to ExtentIDs, the *ExtentID Map* to map extents to ExtentGroupIDs, and the *ExtentGroupID Map* to map ExtentGroupIDs to physical disk locations. These maps are shared between VMs and the cluster storage management system, which might move, compress, deduplicate, and garbage-collect storage blocks. All accesses to a given vDisk are serialized through a vDisk controller hosted on one of the cluster nodes. Stargate migrates vDisk controllers and VMs upon node failures.

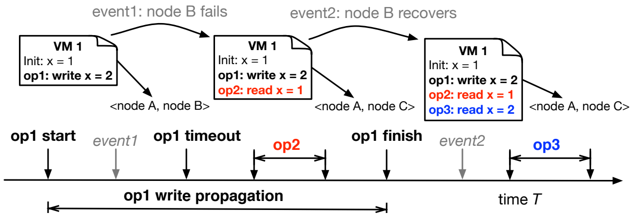


Figure 1: Example timeline that satisfies linearizability but complicates reasoning about failures. The notation <node A, node B> means that the VM is on node A and the leader of the replica group maintaining key *X* is on node B. The value of key *x* is 1 at the start of the timeline. A VM, initially running on node A, issues a write to *x*, partially performs it on node B, and suffers a timeout due to B's failure. After another node C becomes the leader, the VM reads 1 from *x* and expects to continue to see *x* set to 1, barring new writes issued subsequently. If the old leader were to recover, it could propagate its updated copy of *x* and interfere with the VM's logic.

Goals. In determining how to store Stargate's metadata index, apart from maximizing availability and efficiency, we have the following goals:

- ***Durability:*** To minimize the probability of data loss, any update to the metadata must be committed to stable storage on multiple nodes in the cluster before Stargate acknowledges the write as complete to the client. Note that our system should maintain consistent metadata even when the entire cluster comes down (e.g., due to a correlated failure).
- ***Consistency:*** Operations on the metadata index should be linearizable, i.e., all updates to a block's metadata should be totally ordered, and any read should return the last completed write. This guarantee provides strong consistency semantics to client VMs and various background services that operate on the metadata.
- ***Reasoning about failures:*** Under linearizability, even if a read issued after a failure does not reflect a write issued before the failure, this does not mean that the write failed; the update could have been arbitrarily delayed and might get applied later, causing subsequent reads to observe the updated value (see Figure 1). The system should provide stronger guarantees to client VMs so that they can reason about operation failures. In particular, any subsequent read

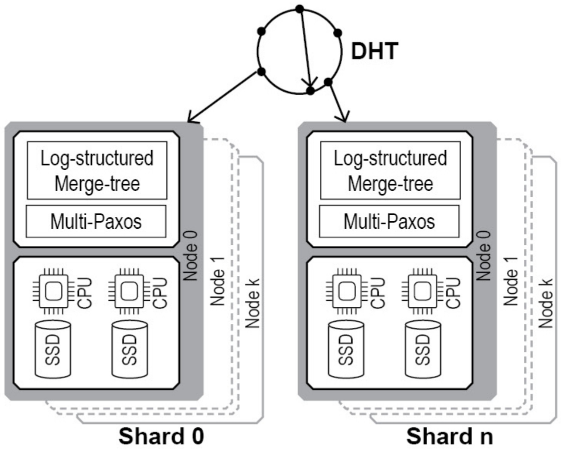


Figure 2: Baseline system architecture representing a coarse-grained replicated state machine built using LSM and Paxos.

of the metadata after an operation timeout must confirm whether the prior operation succeeded or not, and successive reads of a piece of metadata should return the same value as long as there are no concurrent updates initiated by other agents in the system.

2.2 Baseline Design

Let us now consider a baseline approach for realizing the above-mentioned goals. This baseline takes the traditional approach of (a) sharding the metadata index across multiple nodes and multiple cores or SSDs on a given node, (b) using a consensus protocol for ordering operations on any given shard, and (c) executing operations on a durable data structure such as a log-structured merge tree.

In the baseline, all nodes in the cluster participate in implementing a distributed key-value store. We partition keys into shards, use consistent hashing to map shards to replica sets, and consider a leader-based consensus protocol wherein each node serves as the leader for one or more shards in the system. Leader-less designs (such as EPaxos [23]) can lower the communication costs as they eliminate the coordination overheads for the leader, but provide limited benefits in our setting. First, when storage and compute are co-located, there is limited value in moving communication costs from the leader to a client that is sharing network resources with a different server in the cluster. Second, as we will demonstrate later, storage and compute resources are bigger bottlenecks in our setting than the network. Due to our design choice of co-locating compute and storage, the metadata service shares resources with client VMs, which have a higher priority.

We consider a layered design wherein the lower layer corresponds to a consensus protocol, and the upper layer corresponds to a state machine implementing a durable data structure such as a log-structured merge tree. The timeline for processing a request proceeds as follows.

Consensus layer processing. For every shard, one of the replicas of the shard becomes the leader by sending “prepare" messages to a quorum of replicas. When the leader receives a mutating command such as a write, it sequences and propagates this command to all replicas (including itself) using a consensus protocol such as Paxos [16], Viewstamped Replication [25], or Raft [26]. Each shard that a node is assigned to is associated with a specific core and a specific SSD on that node; the core is responsible for sequencing updates to the shard, and the corresponding operation log is stored on the SSD. The system maximizes efficiency by committing commands to the SSD in batches, with every node batching updates destined to one of its SSDs until the prior write to that SSD is complete. Once a batched write is completed, all operations in that batch are considered “accepted". After the leader receives a quorum number of accepts for a command, it can then execute the command locally and send “learn" messages to all followers, indicating that the command has been “chosen." The chosen status does not have to be recorded in stable storage as it can be recreated upon failures. A centralized approach with primary-backup replication [3] can eliminate the use of a consensus protocol and simplify the system design. Such a design, however, limits both the operational scale and performance, and wouldn’t satisfy the system requirements that we had outlined above.

LSM layer processing. At every node, the LSM layer processes all chosen commands in the order determined by the consensus layer. LSM processing is streamlined to include just the in-memory Memtable and the stable SSTables. In particular, this is a slightly customized version of a traditional LSM implementation as the commit log, which is available from the consensus layer, can be eliminated from the LSM code. The Memtable access and compaction operations need to be synchronized with other concurrent operations to support multi-core operations. The leader acknowledges a command as complete to the client after a quorum of nodes has recorded the command, and the leader has executed the command in its chosen order because the success of some commands (e.g., compare-and-swap) can be determined only when they are executed after all previously accepted commands have been applied. Leases enable the leader to serve reads on the LSM without any communication with other nodes. However, the leader must synchronize every read on a key with ongoing updates to the same key.

Ordering guarantees. RSMs built using consensus protocols provide linearizability. Further, an RSM can guarantee in-order execution of operations issued by a client. This helps the client reason about the execution status of its operations that have timed out – if the result of a later operation implies that an earlier operation has not been performed, the client can not only deduce that the prior operation has not yet completed but also get the guarantee that the service will never perform the operation. This guarantee can be provided even after RSM reconfigurations. Upon leadership and view changes, protocols such as Viewstamped Replication ensure that operations partially performed in a previous view are not completed in sub-

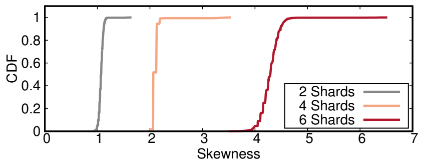


Figure 3: CDF of access skewness with 2/4/6 data shards. ***skewness*** **at any instant is defined as the ratio of the maximum to the average of outstanding IOs per shard.**

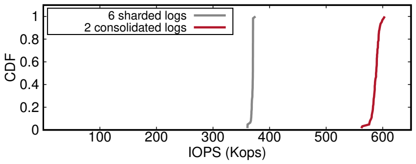


Figure 4: CDF of aggregate SSD throughput when 6 commit logs (3 per SSD) are used compared to when 2 commit logs (one per SSD) are used.

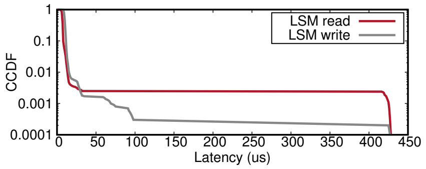


Figure 5: CCDF of LSM random 4KB read/write latencies. The 99.9th percentile latency for LSM reads/writes is 57.1×/48.4× the respective averages.

sequent views. These guarantees provide clients with some capability to infer the completion status of their operations.

2.3 Performance Implications of Baseline Design

This baseline design, however, results in several sources of inefficiency. We quantify them with micro-benchmarks using the same computing setup as our evaluations (see Section 4.1).

- ***Load imbalance due to skew:*** The skew in load across shards can lead to an imbalance across SSDs and CPU cores. For instance, differences in popularity across keys can result in long-term skew, whereas random Poisson arrival of requests can cause short-term skew. Figure 3 quantifies the skews across shards for random Poisson arrival.
- ***Sub-optimal batching:*** If there are n nodes in a replica set, each with m SSDs, the number of shards into which commands would be accumulated would be the least common multiple of m and n. (This ensures that the assignment of shard storage to SSDs and the assignment of shard leadership to nodes are statically balanced.) Batching updates independently on each of these shards can result in less than optimal latency amortization. Figure 4 shows that batching across multiple data shards can achieve 1.6× higher bandwidth than a traditional per-shard log design.
- ***High tail latency:*** Tail latency or even average latency of operations could be high due to multiple reasons. First, since the RSM abstraction requires that all replicas execute all updates in the same order, if one of the replicas for a shard is missing a command in its commit log, subsequent operations on that shard will block until this replica catches up. Second, since LSM operations vary in terms of their execution costs (shown in Figure 5), a heavyweight operation can delay the execution of lightweight operations even if processor cores are available to execute the operations.

Sub-dividing the shards into even smaller shards would mitigate the load imbalance issue. However, it suffers from three drawbacks. First, it doesn't address the request head-of-line blocking issue. Requests still have to commit and execute in sequence, as specified in the log order. Second, it further reduces batching efficiency for storage devices. Third, it doesn't provide the benefit of fast node recovery, as a recovering node cannot immediately participate in the protocol. As a result, we instead adopt a shard-less design to overcome all of these issues, as we describe next.

3 System Design

We now present the design of Stargate's metadata storage system, which provides the desired efficiency, availability, durability, and consistency properties. We use the same high-level approach as the baseline: consistent hashing to distribute metadata across replica sets, log-structured merge trees to store and access large, durable datasets, and a consensus protocol to ensure consistency of operations on replicated data.

Our approach differs in one fundamental aspect: it uses fine-grained replicated state machines (fRSMs), wherein each replicated key is modeled as a separate RSM. This approach provides the flexibility needed to effectively manage multiple storage devices and CPU cores on a server, reduces load imbalances, and enables flexible scheduling of key-value operations. However, the use of fine-grained state machines raises both performance and consistency issues, and we address them by carefully co-designing the consensus protocol, the data store, and the client stubs that interact with the storage layer.

3.1 Overview and Design Roadmap

Replicating every key-value pair as a separate RSM, though conceptually straightforward, could impose significant overheads because RSMs are rather heavyweight. For example, a typical RSM contains an operation log and consensus state for each entry in the operation log. The operation log is used to catch up replicas that are lagging behind and/or have missing entries; each operation in the log has to be propagated to laggards to get their state up-to-date.

Lightweight RSMs. Fortunately, the RSM state can be vastly streamlined for simple state machines, such as the key-value objects we use in our system.

- For normal read/write and synchronizing operations such as compare-and-swap, the next state of a key-value pair is a function of its current state and any argument that is provided along with the operator. For such operations, one can eliminate the need for an operation log; it suffices to maintain just the last mutating operation that has been performed on the object and any in-progress operations being performed on the object. We use an API that is simple and yet sufficiently powerful to support the metadata operations of a cluster storage system. (See Section 3.2.)
- The consensus state for operations on a key (e.g., promised and accepted proposal numbers) is stored along with the

key-value state in the LSM as opposed to requiring a separate data structure for maintaining this information. (See Section 3.3.1.)

- A consensus protocol typically stores accepted but not yet committed values along with its committed state and commits an accepted value when consensus has been reached. Instead, our system speculatively executes the operations, stores the resulting value in a node's LSM, and relies on the consensus protocol to update this to the consensus value for the key in the case of conflicts. This further reduces the RSM state associated with each key. It also eliminates the need for explicit learn messages.[1] (See Section 3.3.2.)
- Similar to the Vertical Paxos approach [18], leader election is performed on a per key-range granularity using a separate service (e.g., Zookeeper [13] in our case).

Enabled optimizations. We co-design the consensus protocol and the LSM layer implementing the key-value store to realize per-key RSMs.[2] This enables many optimizations.

- *Consolidated LSM:* All the key-values replicated on a given node can be stored in a single LSM tree as opposed to the canonical sharded implementation that would require a separate LSM tree for each shard. The commit log of the unified LSM tree can be striped across the different storage devices, thus leading to more effective batching of I/O requests to the commit log.
- *Load balancing:* Per-key RSMs enable flexible and late binding of operation processing to CPU cores; a key-value operation can be processed on any core (as long as there is per-key in-memory synchronization to deal with concurrency) and durable updates can be performed on any SSD, leading to more balanced utilization of cores and SSDs.
- *Minimizing stalls:* By requiring ordering of operations only per-key, rather than per-shard, we can eliminate head-of-line blocking. Message loss and high-latency LSM operations do not impact the performance of ongoing operations on other keys, thus improving the tail latency of operations.
- *Low-overhead replication:* Each operation can be applied to just a quorum of replicas (e.g., two nodes in a replica set of three), thus increasing the overall throughput that the system can support. With coarse-grained RSMs, this optimization would result in a period of unavailability whenever a node fails, because new operations on a shard can only be served after stale nodes catch up on all previous operations on the shard. With fRSMs, lagging nodes can be updated on a per-key basis and can be immediately used as part of a quorum.

Challenges. The per-key RSM approach, however, comes with certain performance and consistency implications that we outline below.

- *Overhead of per-key consensus messages:* A coarse-grained RSM can elect a leader for a given shard and avoid the use of prepare messages for mutating operations performed on any key in the shard. In contrast, with per-key RSMs, a node would have to transmit a per-key prepare message if it had not performed the previous mutating operation on that key. Fortunately, node downtimes are low in managed environments such as ours, and a designated home node coordinates most operations on a key. We quantify the overhead associated with this using failure data collected from real deployments.
- *Reasoning about the completion status of old operations:* As discussed earlier, a coarse-grained consensus protocol such as Viewstamped Replication can discard operations initiated but not completed within a view. With fRSMs, one could perform such a view change on a per-key basis, but this would imply additional overheads even for non-mutating operations. We limit these overheads to only when a key might have outstanding incomplete operations initiated by a previous leader. (See Section 3.3.3.)

3.2 Operation API and Consistency Semantics

Operations supported: Our key-value store provides the following operations: *Create* key-value pair, *Read* value associated with a key, *Compare-and-Swap (CAS)* the value associated with a key, and *Delete* key. The *CAS* primitive is atomic: provided a key k, current value v, and a new value v', the key-value storage system would atomically overwrite the current value v with new value v'. If the current value of key k is not v, then the atomic *CAS* operation fails. Note that *Create* and *Delete* can also be expressed as *CAS* operations with a special value to indicate null objects.

We note that the *CAS* operation has a consensus number of infinity according to Herlihy's *impossibility and universality hierarchy* [12]; it means that objects supporting *CAS* can be used to solve the traditional consensus problem for an unbounded number of threads and that realizing *CAS* is as hard as solving consensus. Further, Herlihy's work shows that objects supporting *CAS* are more powerful than objects that support just reads and writes (e.g., shared registers [1]) or certain read-modify-write operations like fetch-and-increment.

We do not support blind writes, i.e., operations that merely update a key's value without providing the current value. Since all of our operations are *CAS*-like, we can provide at-most-once execution semantics without requiring any explicit per-client state as in RIFL [19]. Further, most of our updates are read-modify-write updates, so it is straightforward to express them as *CAS* operations.

Consistency model: Apart from linearizability, we aim to provide two consistency properties to simplify reasoning about operation timeouts and failures.

[1] It is worth noting that the optimization of piggybacking learn messages with subsequent commands is difficult to realize in fine-grained RSMs as a subsequent operation on the same key might not be immediate.

[2] Since we integrate the RSM consensus state into each key-value pair, we can reuse LSM APIs as well as its minor/major compaction mechanisms.

- *Session ordering:* Client operations on a given key are performed in the order in which the client issues them. This property lets a client reason about the execution status of its outstanding operations.
- *Bounded delays:* Client operations are delivered to the metadata service within a bounded delay. This property lets other clients reason about the execution status of operations issued by a failed client.

Sections 3.3.2 and 3.3.3 describe how we implement linearizable *CAS* and *read* operations using a leader-based protocol. We provide session ordering using two mechanisms: (a) leaders process operations on a given key in the order in which they were received from a client, and (b) the *read* processing logic either commits or explicitly fails outstanding operations initiated by previous leaders (see Section 3.3.3). Section 3.4 describes how coarse-grained delay guarantees from the transport layer can help clients reason about the storage state of failed clients.

Our metadata service exposes single-key operation ordering semantics as opposed to supporting transactional semantics involving multiple keys. To support multi-key operations, one can implement a client-side transaction layer that includes a two-phase commit protocol and opportunistic locking [14, 32]. This is similar to what is required of a coarse-grained RSM system to support cross-shard multi-key transactions.

3.3 Operation Processing Logic

3.3.1 Consensus State

Associated with each key is a *clock attribute* that stores information regarding logical timestamps and per-key state that is used for providing consistent updates. The clock attribute is stored along with a key-value pair in the various data structures (e.g., commit log, Memtable, and SSTables), and it comprises of the following fields.

- *epoch number* represents the generation for the key and is updated every time the key is deleted and re-created.
- *timestamp* within an epoch is initialized when the key is created and is advanced whenever the key's value is updated. The epoch number and the timestamp together represent a Paxos instance number (i.e., the sequence number of a command performed on a key-value object).
- *promised proposal number* and *accepted proposal number* associated with the key's value maintained by a given node; these represent consensus protocol state.
- *chosen bit* indicates whether the value stored along with the key represents the consensus value for the given epoch number and timestamp.

The clock attribute is a concise representation of the value associated with the key, and it is used instead of the value in quorum operations (e.g., quorum reads discussed in Section 3.3.3). Since they are frequently accessed, the clock attributes alone are maintained in an in-memory *clock cache* to minimize SSTable lookups and optimize reads/updates.

3.3.2 CAS Processing

For implementing *CAS* operations, we use a variant of the traditional Multi-Paxos algorithm, wherein we co-design different parts of the system and customize the consensus protocol for our key-value store. First, we integrate the processing associated with the consensus algorithm and the key-value store. As an example of a co-designed approach, *accept* messages will be rejected both when the promise is insufficient and when there is a *CAS error*. Second, the nodes do not maintain per-key or per-shard operation logs, but instead, skip over missed operations and directly determine and apply the accepted value with the highest associated proposal number (with a possibly much higher timestamp). Third, the processing logic speculatively updates the LSM tree and relies on subsequent operations to fix speculation errors.

Client CAS updates are built using the clock obtained via the key read previously. With each read, a client also receives the current epoch (e) and timestamp (t) for the value. The client CAS update for the key would then contain the new value along with epoch e and timestamp $t+1$. This is a logical CAS where the client specifies the new value for timestamp $t+1$ after having read the value previously at timestamp t. The request is routed to the leader of the replica group responsible for the key. It then performs the following steps.

1. **Retrieve key's consensus state:** The leader reads its local state for key k and retrieves the key's local clock. The clock provides the following values: the proposal number for a promise (p_p) and the proposal number for the currently accepted value (p_a).
2. **Prepare request:** If p_p is for a prepare issued by a different node, then the leader generates a higher proposal number, sends prepare messages to other nodes, and repeats this process until it obtains promises from a quorum of nodes. The leader skips this step if p_p and p_a are the same and refer to proposals made by the leader.

   Prepare handler: Each of the replicas, including the leader, acknowledges a *prepare* message with a promise to not accept lower numbered proposals if it is the highest prepare proposal number received thus far for the key. The replicas durably store the prepare proposal number as part of the key's clock attribute (i.e., in the commit log as well as the Memtable).
3. **Accept request:** The leader sends an *accept* message with the client-specified timestamp, i.e., $t+1$, the current epoch, and the proposal number associated with a successful prepare.

   Accept handler: At each of the replicas, including the leader, the *accept* message is processed if the current timestamp associated with the key is still t and the proposal number is greater than or equal to the local promised proposal number. If so, the key's value and the corresponding clock are recorded in the commit log and Memtable at each node. An *accept* request is rejected at one of the nodes if it has issued a promise to a higher proposal number or if the timestamp associated with the object is greater than t. In both cases, the replica returns its current value and the proposal number attached to it.

4. **Accept response processing:** The leader processes the accept responses in one of the following ways.
   - If a quorum of successful accept responses is received at the leader, the leader considers the operation to be completed and records a chosen bit on its Memtable entry for the key-value pair. It then reports success back to the client.
   - If the accept requests are rejected because the promise is not valid, then the leader performs an additional round of prepare and accept messages.
   - If the request is rejected because the (epoch, timestamp) tuple at a replica is greater than or equal to the client-supplied epoch and timestamp, then a *CAS error* is sent to the client. Further, *accept* messages are initiated to commit the newly learned value and timestamps at a quorum of nodes.

The protocol described above is faithful to the traditional consensus protocols, but it is customized for our key-value application and the use of fine-grained RSMs. In our system, a client needs to wait for a previous write to complete before issuing a subsequent write. We discuss the equivalence with coarse-grained RSM in Appendix A.4.

3.3.3 Read Processing

A read operation has to ensure the following properties upon completion: (a) the value returned should be the most recent chosen value for a given key, and (b) other previously accepted values with higher <epoch, timestamp> than the returned value are not chosen. The former requires the completion of in-progress *CAS* operations that are currently visible to the leader; this property is required for linearizability. The latter ensures that any other *CAS* operations that are in-progress but aren't visible will not be committed in the future; this is akin to a view change in the Viewstamped Replication protocol where operations that are not deemed complete at the end of a view are prevented from committing in a subsequent view.

To meet these requirements, read operations are processed in one of three different modes: *leader-only reads*, *quorum reads*, and *mutating quorum reads*. When the operation is routed to the leader, the leader checks whether it is operating in the *leader-only mode*, where all of its key-value pairs are up-to-date as a consequence of obtaining the chosen values for every key in the shard through a shard-level scan (described in Section 5.1). If the check is successful, then the leader will serve the request from its Memtable or one of the SSTables. If the leader is not operating in the *leader-only mode*, then it has to poll the replica set for a quorum and identify the most recent accepted value for a key (i.e., perform a *quorum read*). If this value is not available on a quorum of nodes, the leader has to propagate the value to a quorum of nodes (i.e., perform a *mutating quorum read*). Further, if there is an unreachable replica that might have a more recent accepted value, then the *mutating quorum read* performs an additional quorum-wide update to just the timestamp to prevent such a value from being chosen. Note that the consensus state can help determine the possibility of an update languishing in a failed/partitioned node; at least one node in a quorum set of nodes should have an outstanding promise to the failed/partitioned node, and the *read* protocol can detect this condition using a quorum operation.

We now provide additional details regarding *quorum reads* and *mutating quorum reads*. A leader not operating in *leader-only mode* satisfies a *read* request using the following steps.

1. **Quorum read request:** The leader sends the *read* request to other nodes in the replica set. Each node responds with the clock attribute associated with its local version of the key-value pair.
2. **Quorum read response:** The leader then examines the received clock attributes and checks whether any of them have a <higher epoch, timestamp> compared to the leader's clock and whether a quorum of nodes is reporting the most recent value. If the leader does not have the value associated with the highest epoch and timestamp, it obtains the value from one of the nodes reporting the most recent value. If a quorum of nodes reports not having this value, the leader propagates this value to other nodes in the quorum.
3. **Check for outstanding accepted values:** The leader then examines the received clock attributes and checks whether any of them contain a promise that satisfies the following two conditions: (1) the promise is greater than or equal to the highest proposal number associated with an accepted value, and (2) the promise is made to a node that did not respond with a clock attribute.
4. **Update timestamp to quench outstanding accepts:** If such a promise exists, then the read will perform an additional round of updates to a quorum. Let p_p be the promise associated with an unreachable node, and let v, e, and t be the value, epoch, and timestamp associated with the highest accepted proposal. The leader issues prepare commands to the replica nodes to obtain a promise greater than p_p, and then sends accept commands to the replica nodes to update their value, epoch, and timestamp fields to v, e, and $t+1$, respectively. The higher timestamp value prevents older *CAS* operations from succeeding.

The different modes for satisfying a read operation have progressively higher execution costs. In the common case, the *leader-only reads* can satisfy a read operation using local information and without communicating with the other replicas. The *quorum reads* are performed when the leader is not operating in *leader-only mode* immediately after a failover. In this case, the leader has to communicate with the other replica nodes in order to process the read request. If the most recent accepted value is not available on a quorum or if there is evidence of an unreachable node with an outstanding promise, then we resort to *mutating quorum reads* that not only incurs additional communication rounds to the replicas but also pays the overhead of writes to stable storage in order to record the updated value and timestamp. Fortunately, a *mutating quorum read* is needed only after failover and when there is an unreachable node that has obtained a promise to update the given key-value pair. Further, this is invoked only for the very first operation on the key after the failover; subsequent reads can be processed locally by the leader. This escalation of operating modes means that we incur the additional overheads associated with our use of fine-grained RSMs (e.g., per-key prepare messages and per-key timestamp updates) only in a limited number of cases.

3.4 Bounded Transport Processing

The logic outlined above allows reads to either commit or explicitly fail outstanding operations that have been received and processed by any member of the replica group. We now enhance our system to provide time bounds on the delay for propagating a command from the client to a replica node. This allows clients to also reason about the execution status of commands recently initiated by some other client in the system (e.g., the previous instance of a VM that failed unexpectedly).

CAS operations are tagged with the time at which they are initiated by the Stargate code. The leader ensures that it finishes processing the *CAS* operation within a bounded time of *T* seconds. If the time bound expires and the leader had failed to initiate any accept messages to process and propagate the new value, then it simply drops the request and returns a timeout message. As a consequence of this time bound, a *read* operation that is issued *T* seconds after an update will encounter one of the following cases: the prior update has been committed; the prior update was accepted at a subset of the nodes, in which case the read will commit it; or prior update is not at any of the responsive replicas, in which case the read will prevent the prior update from committing. The *read* can thus determine the execution status of the prior update, and repeated reads will return the same consistent value in the absence of other concurrent updates.

This bounded-time guarantee assists in handling failover of application code, migration of virtual disks across Stargate instances, and other tasks. For example, the cluster management software can delay the failover of applications until the time bound has expired to ensure that they are not affected by spurious races. For the Stargate systems code, such as that of virtual disk migration logic where stalls are not appropriate, clients directly invoke *mutating quorum read* to abort any in-flight operations from the old site until the time bound has expired.

The use of time bounds is similar in spirit to that of leases in a distributed system, and the concerns associated with the use of an implicit global clock being mitigated by the following two considerations. First, the clients of the key-value store are the block storage management services that run on the same set of nodes as the distributed key-value store and thereby share the same set of local clocks on each node. Second, in a local area enterprise cluster, time synchronization protocols such as NTP/PTP can achieve sub-millisecond time synchronization accuracy, whereas the time bounds that we provide are in the order of seconds (which is consistent with the disk timeout values in operating systems/file systems).

4 Evaluation and Deployment Measurements

Our evaluations comprise of four parts. First, we characterize the metadata service using representative traces from customer clusters. Second, we show the performance benefits of using fine-grained RSMs by comparing it with an implementation of a coarse-grained RSM (i.e., **cRSM**) approach described in Section 2. We perform these evaluations in a controlled testbed setting that runs just the metadata service and not the rest of the cluster block storage system. Note that the controlled environment has the same failure rate, request read/write ratio, and key popularity distribution that we observed in practice. Third, we present the performance of our metadata service as part of complete system evaluations. We configure a cluster with client VMs and workload generators, measure the performance of our metadata service, and characterize the performance benefits of optimizations. Finally, we report performance numbers from real-world deployments.

4.1 Experiment Setup

Our evaluations are performed on typical enterprise on-premises clusters. Specifically, our controlled testbed is a 4-node cluster, where each node is a Supermicro 1U/2U server, enclosing E5-2680 v3/E5-2620 v4 processors, 64GB/128GB DDR4 memory, two Intel DC P3600 NVMe SSDs, and a dual-port 10Gbps Intel X710 NIC. We perform the remaining evaluations on similar hardware, but at a larger scale across a large number of customer clusters. Appendix B.1 presents details of the LSM configurations that we use in practice. The replication factor for a key is three in all experiments.

4.2 Metadata Workload Characterization

We present metadata measurements from 980 customer clusters (Figure 20 in Appendix B.2). Generally, each cluster contains 3 to 30 nodes and uses 24.7TB block storage on average. The three metadata components (vDisk block, ExtentGroupID, and ExtentId) have sizes that are 0.04%, 0.12%, and 0.02% of the logical storage, respectively. Note that the size of metadata will increase when deduplication and compression are enabled due to more consolidated block storage.

Next, we characterize the metadata workload in terms of read/write ratio, value size distribution, and key access popularity by taking continuous snapshots from three custom clusters, where each cluster has at most 16 nodes. We make the following observations. First, unlike other previous key-value

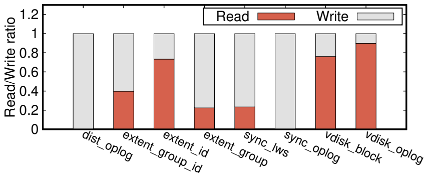


Figure 6: Read/write ratio for 8 frequently accessed metadata tables.

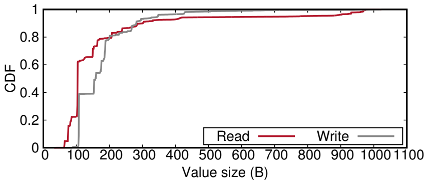


Figure 7: Value size distribution of read/write requests. Key size is less than 100 bytes.

	Cluster1	Cluster2	Cluster3
vdisk block	0.99	0.99	0.99
extent id	0.80	0.80	0.85
extent group id	0.60	0.55	0.50
extent phy_state	Uniform	Uniform	Uniform

Table 1: Key access popularity of four different metadata tables for three customer clusters. The first three types of metadata are Zipf; we show their skewness factors.

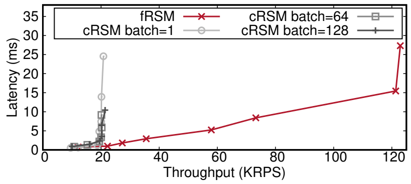


Figure 8: Latency v.s. throughput under the skewed workload for multiple shards.

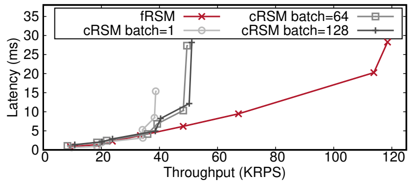


Figure 9: Latency v.s. throughput under the uniform workload for multiple shards.

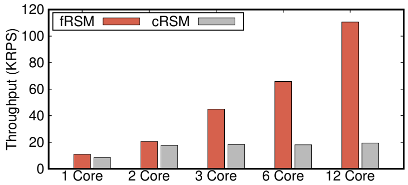


Figure 10: Maximum throu. for the skewed workload as we increase the number of cores.

store workload profiles studied under social networks/web settings [21,24], our metadata service presents various read/write ratios ranging from write-only loads for various system logs to read/write intensive ones for various filesystem metadata items (see Figure 6). Second, the read/write requests are dominated by small values, say less than 512B (see Figure 7). In fact, about 80% of reads and writes involve values that are less than 200 bytes in size. Further, requests that involve more than 1KB value sizes are about 1.0% of the reads/writes. Finally, there exist various access patterns in our metadata service. As shown in Table 1, some metadata shows highly skewed key/value accesses, while others have low skewness or even present uniform access patterns.

4.3 Benefits of Fine-grained RSMs

We now evaluate the performance benefits of fRSMs using streamlined deployments that run just the metadata service on physical nodes (as opposed to client VMs). No client workloads are executing on service nodes. We use a workload generator and configure it to issue a similar request pattern as our most frequently accessed metadata that has 43% reads and 57% writes, value size of 512B, and a Zipf distribution with skewness factor of 0.99. We also consider a uniform access case (i.e., random access pattern) as an additional workload. We inject faults into the leader using failure rate observed in the wild (Section 4.5). We evaluate fRSM and cRSM in terms of both latency and throughput.

Higher throughput. We set up a three-node replica group with twelve data shards, running across two SSDs and twelve CPU cores. In the case of cRSM, each node is a leader for four shards, each shard allocated a separate core, and six shards share each SSD. In the case of fRSM, there is a consolidated commit log striped across the two SSDs, and each operation is dynamically scheduled to a CPU core. We consider cRSM configured to perform batched commit using different batch sizes. fRSM achieves 5.6× and 2.3× higher throughputs over cRSM (with batch size of 128) for skewed and random cases, respectively (see Figures 8 and 9). This is because fRSM (1) allows requests accessing different keys to be reordered and committed as soon as they complete; (2) eliminates the computation cost associated with scanning the RSM log to identify and retry uncommitted entries; (3) avoids unnecessary head-of-line blocking caused by other requests; (4) achieves better load balance across SSDs and cores even in skewed workloads. The first three benefits can be observed even in the single shard case (Figures 11 and 12), while the next experiment further examines the load balance benefits.

Better load balancing. To examine the load-balancing benefits of fRSM , we again consider a three-node replication group with twelve data shards but vary the number of CPU cores used to process the workload. We consider the skewed workload, and we configure cRSM to use a batch size of 64. We then measured the maximum throughputs achieved and the average/p99 latency of operations when we achieve the maximum throughput (see Figures 10 and 13). fRSM provides a 1.9×, 4.1×, 6.1×, 11.0× throughput improvement and 1.9×, 2.4×, 3.3×, 5.3× (1.3×, 2.5×, 3.3×, 4.9×) avg(p99) latency reduction as we increase the number of cores from 1 core to 2, 4, 6, and 12 cores, respectively. The performance of cRSM, on the other hand, does not improve with more than two provisioned cores. Under load skews, fRSM allows balanced and timely execution of operations on different key-based RSMs, while cRSM has to commit requests in the RSM log sequentially and is subject to skews and head-of-line blocking.

4.4 Performance of Commercial Offering

We now evaluate the fRSM approach when implemented inside a commercial product providing a cluster-wide storage abstraction. This introduces many additional overheads as the metadata service is executed inside a controller virtual

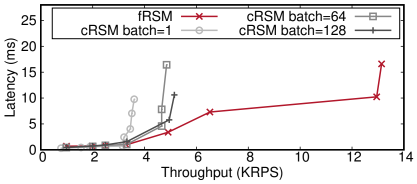


Figure 11: Latency v.s. throughput under the skewed workload for a single shard.

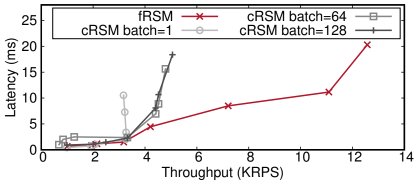


Figure 12: Latency v.s. throughput under the uniform workload for a single shard.

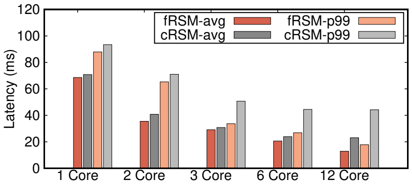


Figure 13: Average/p99 latency for the skewed workload as we increase the number of cores.

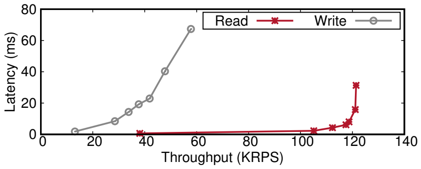


Figure 14: Latency versus throughput for reads and writes inside a Stargate cluster.

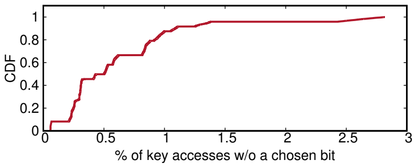


Figure 15: Operations requiring a multi-phase protocol when the leader has no chosen value.

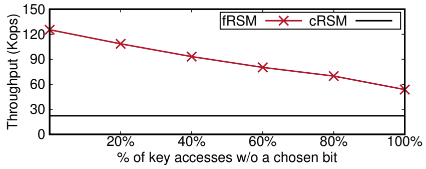


Figure 16: Throughput for the skewed workload varying the fraction of multi-phase operations.

machine, there is virtualized access to the network, and storage/CPU resources are shared with the rest of the cluster management system as well as client VMs.

We use an internal three-node cluster and an in-house workload generator that mimics various types of client VM behavior. Figure 14 reports the performance. The node is able to support peak throughputs of 121.4KRPS and 57.8KRPS for reads and writes, respectively. Under a low to medium request load, the average latency of reads and writes is 0.63ms and 1.83ms, respectively. In the appendix, we provide additional measurements of the internal cluster that quantify the benefits of using a gradation of read execution modes and utilizing the appropriate read variant for a given key. Overall, the throughput performance of fRSM inside the commercial offering is in the same ballpark as the stand-alone evaluation, but the access latency is significantly higher due to various queueing delays and interference with other storage operations that are concurrently performed by the VMs and the cluster management software.

4.5 Measurements from Real-world Deployments

High availability. We collect failure data over a two-week period (from 2018/09/12 to 2018/09/25) from about 2K customer clusters. On average, there are 70 software detached events (due to unanswered heartbeat messages) and 26 hardware failures (e.g., disk failures) per day, respectively. Crucially, our measurements show that a recovering node is able to integrate itself into the service within 30 seconds irrespective of the number of key-value operations that might have been performed when it was down. Appendix B.4 reports detailed failure handling performance. The reason for this fast recovery is that a recovering node only replays the operations in its commit log before it can participate in the consensus protocols. Each key accessed subsequently would allow the recovering node to update just that particular key-value state given the fine-grained nature of the RSMs in our system. The node can also lazily update the remaining key-value state in the background, and we observe that our system does so in about 630secs on average. In other words, the fRSM approach speeds up node integration significantly by more than 20x.

Multi-phase operations. The primary overhead associated with fRSM is the need for one or more additional rounds of protocol messages when a leader invokes an operation on a key that was previously mutated through a different leader. cRSM also incurs leadership change overheads, but they are at a shard-level, whereas fRSM incurs the overheads on a per-key basis. We quantify how often this happens in practice by measuring the fraction of instances where a leader does not have the chosen bit set and has to perform additional protocol phases. Figure 15 shows that fRSM incurs additional overheads for less than 1% of the key accesses in more than 90% of the cluster snapshots. We then performed an analysis of how the fRSM throughput degrades as we vary the number of accesses requiring multi-phase operations given the skewed workload discussed earlier. Figure 16 shows that, even though the throughput of fRSM degrades in our controlled testbed, fRSM's throughput is still higher than that of cRSM's even when 100% of the operations require multiple phases.

Cluster throughputs. We report the node/cluster throughput of the metadata layer from real deployments. Figure 17 shows the cluster throughput, where (1) every point represents a cluster data point; (2) the left y-axis represents both the throughput as well as the number of Paxos state machines that are executing per second (since every operation corresponds to a Paxos instance of a key-value pair); (3) the right y-axis is the number of nodes in the cluster. It varies from 3 nodes to a maximum of 33 nodes in the cluster; (4) the red line represents the throughput measurements per node. We can observe that our metadata layer can scale from a few thousand state machine invocations to about 393K state machine invocations per second across the cluster. The cluster with the maximum number of cluster-level operations had eight nodes,

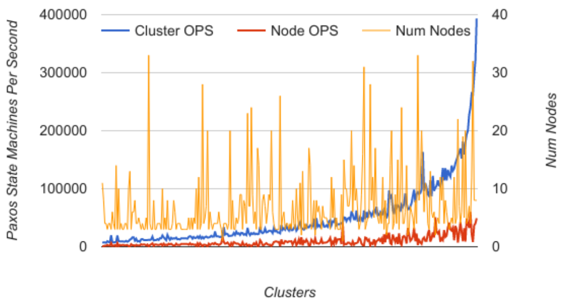


Figure 17: Cluster-level and node-level throughput for the metadata layer in the custom cluster.

and the per-node throughput is ∼59K operations per second, which is consistent with the stress tests performed on the internal cluster. Note that the peak system throughput for the other clusters could be higher, as the observed throughput is a function of the offered load.

5 Deployment-based Experience

From our experience developing the system and troubleshooting issues, we have not only improved the robustness of the system but also have learned a number of non-obvious lessons.

5.1 Fault Tolerance

Stargate provides highly-available block storage, and we describe how the metadata layer handles various cluster failures.

Transient failure. This is a typical failure scenario where the failed node recovers within a short period, e.g., a node taken offline for upgrades. When the node is the leader of a replica group, one of the other replicas will elect itself as the leader. The new leader initially processes reads and writes using quorum operations instead of transiting into *leader-only mode* (since *scan* is an expensive operation). The system keeps track of newly created SSTables on the leader and ensures these newly created SSTables are not compacted with older ones. This guarantees that new updates are segregated from older ones. When the failed node recovers, it elects itself as the leader of the replica group, provided it is the natural leader of the shard. We then transfer the newly created SSTables to the recovered node to enable it to catch up on lost updates and enter *leader-only mode* after it does so. If a significant period of time has elapsed without the failed node recovering (e.g., 30 minutes in our current system), the current leader attempts to transition to *leader-only* mode. For this, it has to *scan* the entire keyspace, by performing batched quorum reads or mutating quorum reads as necessary, to discover the up-to-date state for all keys in its shard.

Correlated or group failure. Generally, this is an uncommon event but will happen when (1) the rack UPS (uninterruptible power supply) or rack networking switch goes down; (2) the cluster undergoes planned maintenance. We apply a rack-aware cluster manager, where Stargate creates different location independent failure domains during the cluster creation and upgrade phases. Upon metadata replication, based on the replication factor (or fault tolerance level), we place replicas across different failure domains to minimize the probability that the entire metadata service is unavailable.

Optimization. It is worth noting that the choice of the LSM tree as a node's local data storage is beneficial in optimizing the handling of failures. With appropriate modifications to the LSM tree, we are able to keep the newly created data segregated. It also helps optimize the transfer of state to new nodes that are added to the replica set (to restore the replication factor in the case of persistent failures) by enabling the bulk transfer of SSTable state to the new nodes. Further, our system has a background process that periodically checks the integrity of stored data and re-replicates if necessary. This accelerates the recovery process. If a node goes down for a while, the system starts a dynamic healing approach that proactively copies metadata to avoid a two-node failure and unavailability.

5.2 Addition/Removal of Nodes

Recall that, in Stargate's metadata store, keys are spread across nodes using consistent hashing. Since we apply every update for a key to only a quorum of the key's replicas to maximize system throughput, the addition of nodes to the cluster must be handled carefully. For example, consider the addition of node A (in between Z and B) to a four-node cluster with nodes Z, B, C, and D. Say a key in the range (Z, A] has previously been written to only B and D, i.e., two out of the key's three replicas B, C, and D. Now, a read for that key could potentially return no value since two of the key's three new replicas (A, B, and C) have no record of it.

To prevent such issues, we introduce a new node by temporarily increasing the replication factor for the keys assigned to it, until the node is caught up. Having a new node catch up by issuing Paxos reads for all of its keys is, however, terribly slow; this process has taken as long as 18+ hours at one of our customers! So, we also had to develop a protocol that enables a new node to directly receive a copy of relevant portions of other nodes' SSTables. Since a new node starts serving new operations while receiving LSM state in the background, we disable caching until the new node is caught up, so as to prevent inconsistency between in-memory and on-disk state. This bulk copy method is also used during the node removal process. Besides that, we place the removed node into a *forwarding state* such that replication requests won't be accepted, but local requests will be forwarded to another node. After affected token ranges are scanned, and a quorum of the remaining nodes can respond to the request, the removed node is excised from the DHT ring.

5.3 Deletion of Keys

Consensus protocols such as Paxos are silent on the issue of deletion; it is assumed that Paxos state must be kept around forever. Therefore, when a key is deleted, correctly removing that key's Paxos state from all replicas proved to be tricky to get right for several reasons. (We describe our delete protocol in Appendix A.2.) Even after all replicas commit to their LSMs, a tombstone record indicating a key's deletion, we found that the key's old value could resurface for multiple reasons. Example causes include faulty SSDs failing to write an update to stable storage despite acknowledging having done so, or misbehaving clients issuing mutating reads with an epoch number lower than the key's epoch value when it was deleted, causing the old value to be re-propagated to all replicas. To avoid such scenarios, apart from using high-quality SSDs, we set a key's tombstone record in the LSM to be deleted only 24 hours after the third record was created. Since we use the current time to pick epoch numbers, 24 hours is sufficiently large that clock skew cannot prevent epoch numbers from monotonically increasing.

6 Related Work

Our work is related to recent research efforts in consensus protocols, consistent storage systems, metadata management, relaxed consistency, and cluster storage.

Consensus protocols: To provide consistency in the presence of node faults, we use a consensus protocol that is an extension of protocols such as Multi-Paxos [16], Viewstamped Replication [25], and Raft [26]. The crucial difference is that we integrate request processing (which in our case is read/CAS/delete operations of a key-value store) with the consensus protocol logic. This approach allows us to realize fine-grained replicated state machines that enable effective use of storage and compute units in a setting where they are scarce (since client VMs are co-located with the storage service).

Many replication protocols reduce coordination by identifying operations that can be performed independently (e.g., Generalized Paxos [17], EPaxos [23]). We employ a similar technique but use it to optimize the use of storage and computing on a server node. Our work is related to foundational algorithmic work on atomic distributed registers [1,9], but we support synchronization operations that have an unbounded consensus number (such as *CAS*).

Consistent storage systems: Our work is also related to recent work on various types of consistent key-value storage systems. Unlike Spanner [5], RIFL [19], FaRM [7], and TAPIR [35], our key-value store does not directly support transactions but rather limits itself to single key operations. Instead, it provides the atomic *CAS* primitive, which is used by the block storage management layer to make mutating updates and limited types of transactional operations. Our key-value store, however, provides bounded time operations and stronger ordering constraints that are required by legacy applications in virtualized settings. Its node-local data structures are based on those of BigTable [4] and HBase [11], and we make some modifications to aid in fast failure recovery. Our consistent storage system is also related to MegaStore [2], which provides per-row transactional updates using Paxos. Our approach integrates the Paxos algorithm with the key-value store logic in order to both enhance performance as well as provide stronger operation ordering guarantees.

Metadata management in P2P systems: Traditional DHT-based P2P storage systems (like DHash [6], Pastry [28], OceanStore [15], Antiquity [33], Ceph [34]) provide a management layer that maps physical blocks to node locations. Such metadata is a read-only caching layer that only changes when nodes join/leave. However, our metadata service maintains mappings between physical and virtual blocks, which could frequently change under VM migration. Hence, our system has a stronger consistency requirement.

Relaxed consistency: Researchers have proposed a couple of relaxed consistency models to reduce request execution latency, especially for geo-replicated key-value storage. For example, Walter [30] supports parallel snapshot isolation and conducts asynchronous replication. Within each site, it uses multi-version concurrency control and can quickly commit transactions that write objects at their preferred sites. COPS [22] is a geo-replicated key-value store that applies causal consistency across the wide area. RedBlue [20] defines two types of requests: blue operations execute locally and lazily replicate in eventually consistency manner; red operations serialize with respect to each other and require cross-site coordination. The metadata layer of our enterprise cloud storage has a linearizable requirement.

7 Conclusion

Enterprise clusters today rely on virtualized storage to support their applications. In this paper, we presented the design and implementation of a consistent metadata index that is required to provide a virtual disk abstraction. Our approach is based on using a distributed key-value store that is spread across the cluster nodes and is kept consistent using consensus algorithms. However, unlike other systems, our design uses fine-grained RSMs with every key-value pair represented by a separate RSM. Our design is motivated by the effective use of storage and computing on clusters that is achieved by flexible scheduling of unrelated operations. Our work tackles a range of challenges in realizing fine-grained RSMs and provides useful ordering guarantees for clients to reason about failures. We build and evaluate our system, compare it with coarse-grained RSMs in controlled testbed settings, and provide measurements from live customer clusters.

Acknowledgments

This work is supported in part by NSF grants CNS-1714508 and CNS-1563849. We would like to thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their comments and feedback. In addition, we thank the following people for their contributions to the initial design and implementation of the system: Radhanikanth Guturi, Mohit Aron, and Dheeraj Pandey.

References

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1):124–142, 1995.

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data System Research*, 2011.

[3] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems*, 2:199–216, 1993.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[8] EMC. EMC Isilon OneFS: A Technical Overview, 2016.

[9] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, 2009.

[10] Gluster. Cloud Storage for the Modern Data Center: An Introduction to Gluster Architecture, 2011.

[11] HBase Reference Guide. `https://hbase.apache.org/book.html`.

[12] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010.

[14] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[15] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[16] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] Leslie Lamport. Generalized Consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[18] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *The ACM Symposium on Principles of Distributed Computing*, 2009.

[19] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[20] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[21] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[22] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[23] Iulian Moraru, David G Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[25] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988.

[26] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, 2014.

[27] Ohad Rodeh and Avi Teperman. zFS - A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[28] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2001.

[29] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[30] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[31] Sun. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System, 2007.

[32] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[33] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: Exploiting a Secure Log for Wide-area Distributed Storage. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[35] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

A More Details of fRSM

In this appendix, we present additional details regarding the design of our system.

A.1 Read and CAS Algorithmic Description

Algorithm 1 READ procedure

```
procedure READ_LEADER($key$)
    if $leader\_only = 1$ then                          ▷ In leader-only read mode
        $< val_{local}, CLOCK_{local} > \leftarrow lsm\_read(key)$
        if $CLOCK_{local}.chosen$ then
            return $send\_client\_reply(key, val_{local})$
        end if
    end if
    ▷ Do a quorum read.
    for $N$ in $Replica_{group}$ do
        $send\_read(key, N)$
    end for
end procedure

procedure PROCESS_READ_REPLY_AT_LEADER($req$, $msg$)
    if $msg_{type} = SUCCESS$ then
        $++num\_responses$
    else
        $++num\_errors$
    end if
    if $CLOCK_{msg} > CLOCK_{local}$ then
        ▷ Replica has a newer value. Perform mutating quorum read.
        $write\_result \leftarrow$
            $CAS\_LEADER(key_{req}, val_{local}, CLOCK_{local}.epoch, CLOCK_{local}.ts)$
        if $write\_result = CAS_{success}$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else
            return $send\_client\_reply(key_{req}, READ_{error})$
        end if
    else if $CLOCK_{msg} = CLOCK_{local}$ then
        $++num\_responses\_with\_same\_clock$
    end if
    if $CLOCK_{msg}.P_p = CLOCK_{local}.P_p$ then
        ▷ Local leader holds promise on the replica.
        $++num\_promise\_to\_local\_leader$
    end if
    if $num\_responses\_with\_same\_clock \geq QUORUM$ then
        if $num\_promise\_to\_local\_leader \geq QUORUM$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else if $num\_responses = Replica_{group}.count$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        end if
    end if
    if $num\_responses + num\_errors = Replica_{group}.count$ then
        ▷ Local leader doesn't hold promise on quorum nodes and some replicas
         didn't reply. Some of these replica nodes may have a value with higher
         timestamp, do a mutating quorum read.
        $write\_result = CAS\_LEADER(key_{req}, val_{local}, CLOCK_{local}.epoch, CLOCK_{local}.ts)$
        if $write\_result = CAS_{success}$ and
         $write\_result_{num\_responses} < Replica_{group}.count$ then
            $ts_{new} \leftarrow CLOCK_{local}.ts + 1$
            $CLOCK_{new} \leftarrow get\_clock(CLOCK_{local}.p_p, CLOCK_{local}.epoch, ts_{new})$
            $write\_result \leftarrow$
                $CAS\_LEADER(key_{req}, val_{local}, CLOCK_{new}.epoch, CLOCK_{new}.ts)$
        end if
        if $write\_result = CAS_{success}$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else
            return $send\_client\_reply(key_{req}, READ_{error})$
        end if
    end if
end procedure

procedure PROCESS_READ_AT_REPLICA($req$)
    $< CLOCK_{local} > \leftarrow lsm\_read(key_{req})$
    $send\_read\_reply(key_{req}, CLOCK_{local})$
end procedure
```

Algorithm 2 CAS procedure

```
procedure CAS_LEADER($key, val_{new}, epoch_{new}, ts_{new}$)          ▷ Client sent write.
    $< val_{local}, CLOCK_{local} > \leftarrow lsm\_read(key)$
    if ($CLOCK_{local}.epoch > epoch_{new}$ or
        ($CLOCK_{local}.epoch = epoch_{new}$ and $CLOCK_{local}.ts > ts_{new}$)) then
        return $send\_client\_reply(key, CAS_{error})$
    end if
    if $CLOCK_{local}.p_p$ is not valid then                      ▷ Issued by another node.
        $CLOCK_{new} \leftarrow get\_clock\_with\_higher\_proposal(CLOCK_{local}.p_p)$
        for $N$ in $Replica_{group}$ do            ▷ Replica group includes the leader itself.
            $send\_prepare(key, CLOCK_{new}, N)$
        end for
    else                        ▷ This leader holds the Multi-Paxos promise for this key.
        $CLOCK_{new} \leftarrow get\_clock(CLOCK_{local}.p_p, epoch_{new}, ts_{new})$
        for $N$ in $Replica_{group}$ do            ▷ Replica group includes the leader itself.
            $send\_accept(key, val_{new}, CLOCK_{new}, N)$
        end for
    end if
end procedure

procedure PROCESS_MSG_LEADER($req$, $msg$)
    if $msg_{type}$ is prepare_reply or accept_reply then
        $++num\_responses$
        if $CLOCK_{msg} > CLOCK_{req}$ or $val_{msg}$ is present then
            ▷ Replica has a higher clock or an accepted value for this clock.
            $update\_highest\_clock\_seen(key_{req}, val_{msg}, CLOCK_{msg})$
            $err \leftarrow 1$
        end if
        if $num\_responses \geq QUORM$ then
            if $err = 1$ then
                ▷ We have to run Paxos for a replica replied value
                 or with a higher clock.
                $< val_{resp}, CLOCK_{resp} > \leftarrow get\_highest\_clock\_seen(key_{req})$
                $CAS\_LEADER(key_{req}, val_{resp}, CLOCK_{resp}.epoch, CLOCK_{resp}.ts)$
                return $send\_client\_reply(key_{req}, CLOCK_{req}, CAS_{error})$
            else
                if $msg_{type}$ is prepare_reply then
                    for $N$ in $Replica_{group}$ do
                        $send\_accept(key_{req}, val_{req}, CLOCK_{req}, N)$
                    end for
                else if $msg_{type}$ is accept_reply then
                    ▷ Chosen bit is added to the local leader's memtable.
                    $send\_client\_reply(key_{req}, CLOCK_{msg}, CAS_{success})$
                end if
            end if
        end if
    else if $msg_{type}$ is prepare or accept then
        ▷ Same as the way follower works.
        return $PROCESS\_MSG\_FOLLOWER(req, msg)$
    end if
end procedure

procedure PROCESS_MSG_FOLLOWER($req$, $msg$)
    ▷ Regular Paxos protocol at the replica.
    $< CLOCK_{local}, val_{local} > \leftarrow lsm\_read(key_{req})$
    if $msg_{type}$ is prepare then
        if $CLOCK_{local} >= CLOCK_{req}$ then
            ▷ Epoch and timestamp match, value for this clock exists.
            $send\_prepare\_reply(key_{req}, CLOCK_{local}, val_{local})$
        else if $CLOCK_{local}.p_p > CLOCK_{req}.p_p$ then
            ▷ Previously accepted Promise greater than request Proposal.
            $send\_prepare\_reply(key_{req}, CLOCK_{local})$
        else
            ▷ This proposal can be accepted.
            $lsm\_write\_clock(key_{req}, CLOCK_{req})$
            $send\_prepare\_reply(key_{req}, CLOCK_{req})$
        end if
    else if $msg_{type}$ is accept then
        if $CLOCK_{req} \geq CLOCK_{local}$ then
            ▷ Received clock's epoch, timestamp, and promise are greater or
             equal to the local clock's corresponding values.
            $lsm\_write\_whole(key_{req}, val_{req}, CLOCK_{req})$
            $send\_accept\_reply(key_{req}, CLOCK_{req})$
        else
            $send\_accept\_reply(key_{req}, CLOCK_{local}, val_{local})$
        end if
    end if
end procedure
```

Algorithm 1 and 2 presents how read/CAS requests are handled at the leader and followers. They follow our protocol description in Sections 3.3.3 and 3.3.2.

A.2 Delete Processing

The key-value store also supports a delete operation, which is used by the block storage system to remove index map entries that are no longer necessary (e.g., when a virtual disk snapshot is deleted). A delete request from a client is similar to a regular *CAS* update where the client provides the epoch e and timestamp $t+1$. The leader processes a delete operation by first getting a quorum of nodes to update the value associated with the key to a special *DeleteForCell* value for epoch e and timestamp $t+1$. If the *DeleteForCell* value was not accepted by all replicas but only by a quorum, then a *DeletedCellTombstoned* message is sent to ensure replicas keep the key-value pair until the next deletion attempt. As far as the client is concerned, quorum nodes accepting a *DeleteForCell* is considered as a successful *CAS* update.

Periodically, the leader attempts to complete a two-phase deletion process to delete the value completely. When it has gotten all replicas to accept the delete request, the first phase is considered complete. It then sends a second message to instruct replica nodes to schedule the key for deletion and to remove all state associated with it. This request is recorded in *Memtable/SSTable* individually on every replica. The next major compaction on a replica will remove the state. Until then the deletion record persists at each replica with its associated clock containing epoch e and timestamp $t+1$.

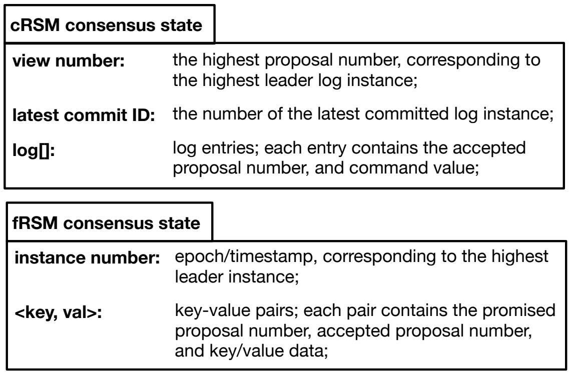
Once the key deletion is successful (quorum nodes have accepted the deletion request), any new CAS updates with epoch $\leq e$ are rejected as *CAS errors*. New client updates for the key (i.e., key creation) must use a new (higher) epoch with timestamp 0.

A.3 fRSM Operation Summary

Table 2 summarizes read/write operations under various cases in terms of request latency, message count, and metadata storage operation count.

A.4 The Relationship between cRSM and fRSM

Note that fRSM works exactly the same as cRSM but in a fine-grained way. In terms of the consensus state (Figure 18 in the appendix), cRSM maintains a per-shard view number, the latest commit ID, and a log of RSM instances (where each instance has an accepted proposal number and the command value). fRSM essentially maintains information only for the most recent instance and directly encodes the promised/accepted proposal number along with the key/value pair. As a result, it doesn't require the latest commit ID. In terms of the way they handle the leader change event, cRSM uses a full two-round consensus protocol to synchronize the latest commit ID, preparing for all future commands. fRSM also takes a full two-phase consensus protocol to synchronize the consensus state for each key. In the example shown in Figure 19 (in the appendix), where there are eight operations accessing three different keys, cRSM issues the leader prepare message at op1, while fRSM performs this prepare at op1, op3, and op7.



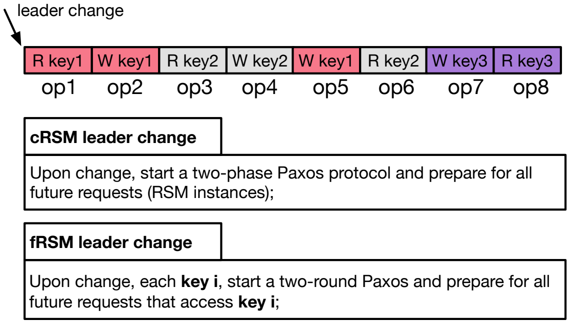
cRSM consensus state

view number:	the highest proposal number, corresponding to the highest leader log instance;
latest commit ID:	the number of the latest committed log instance;
log[]:	log entries; each entry contains the accepted proposal number, and command value;

fRSM consensus state

instance number:	epoch/timestamp, corresponding to the highest leader instance;
<key, val>:	key-value pairs; each pair contains the promised proposal number, accepted proposal number, and key/value data;

Figure 18: Consensus state comparison between cRSM and fRSM .



cRSM leader change

Upon change, start a two-phase Paxos protocol and prepare for all future requests (RSM instances);

fRSM leader change

Upon change, each **key i**, start a two-round Paxos and prepare for all future requests that access **key i**;

Figure 19: Leader change comparison between cRSM and fRSM .

B More Real-world Evaluation

B.1 LSM Configuration

Table 3 shows key LSM parameters. They are configured based on the physical storage media, cluster setup, and metadata characteristics. The table presents the default values.

B.2 Deployment Scale

Figure 20 presents the deployment scale in terms of node number, storage size, and metadata size.

B.3 Internal Cluster Measurements

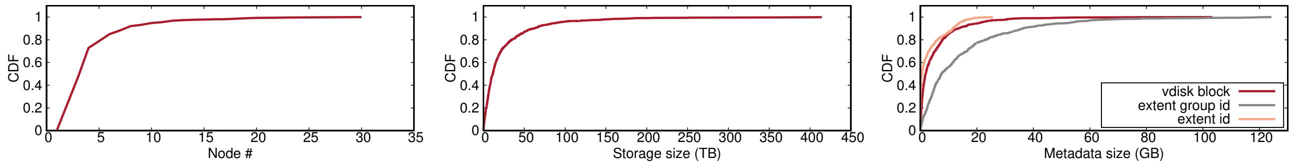
We consider again the internal cluster running the complete storage and virtualization system along with client VMs invoking stress tests on the metadata and storage layer (as discussed earlier in). We report the average/p99 latency distribution of read/write requests (Figure 21), showing comparable end-to-end performance for read and write operations. We also evaluate the performance of leader-only reads. Leader-only mode significantly reduces the number of protocol messages and storage accesses, enabling fast metadata access. Figures 22 and 23 show that leader-only mode results in benefits across different value sizes. On average, across various sizes,

Operations	Latency (RTT)	Message #	Leader LSM RD. #	Leader LSM WR. #	Follower LSM RD. #	Follower LSM WR. #
Cold CAS	2	$4\lceil n/2\rceil$	1	2	2	2
Warm CAS	1	$2\lceil n/2\rceil$	1	1	1	1
Leader-only Read	0	0	1	0	0	0
Quorum Read	1	$2n$	1	0	1	0
Mutating Quorum Read	3	$2n+4\lceil n/2\rceil$	2	2	3	2

Table 2: Message RTTs and LSM read/write counts with no cache for the leader and the follower under different settings. *n* is the number of replicas. *cold CAS* refers to the case that the proposal of a key is issued by another node so that the leader has to invoke the two round Paxos. *warm CAS* means that the leader is able to skip the 1st round of prepares.

Parameter	Description	Default value
max_heap_size	Maximum heap size of the metadata store	2∼4GB
flush_largest_memtables	Heap usage threshold when flushing the largest memtable	0.9
default_memtable_lifetime	Life time in minutes for any memtable	30
min_flush_largest_memtable	Minimum memtable size forced flush when heap usage is high	20MB
max_commit_log_size_on_disk	Maximum disk usage by commit logs before triggering a cleanup task	1GB
commitlog_rotation_threshold	Maximum size of an individual commit log file	64MB
number_of_compaction_threads	Number of threads to perform minor/major compaction	2
compaction_throughput_limit	Maximum disk throughput consumed by compaction on a disk	64MB

Table 3: LSM performance-sensitive parameters.



(a) Node #.

(b) Storage size.

(c) Metadata size.

Figure 20: Node #, storage size, and metadata size CDF across 980 custom clusters.

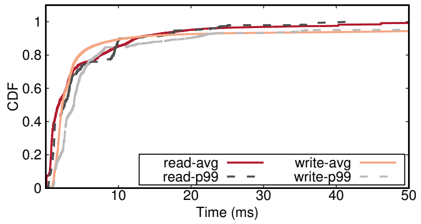


Figure 21: Average/p99 read/write latency CDF inside a Stargate cluster.

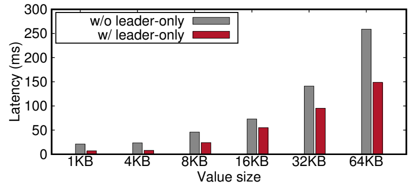


Figure 22: Latency versus value size, compared between with and without leader-only mode.

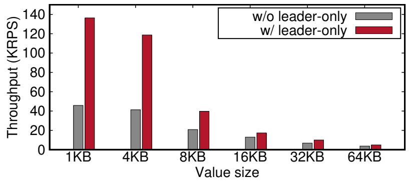


Figure 23: Throughput versus value size, compared between with and without leader-only modes.

leader-only mode halves the latency and more than doubles the throughput. This underscores the benefits of using a gradation of read execution modes and utilizing the appropriate read variant for a given key.

B.4 Failure and Recovery Measurements

We provide additional details on the failure and recovery measurements from our customer clusters. Figure 24 shows the number of software detached events and fatal hardware errors across the measurement period across all of the 2K clusters. Both of them are detected by the DHT health manager. Under software failures, our system will quickly restart the metadata service and rejoin the DHT ring, consuming 2.7s on average. Upon fatal hardware errors, we reboot the server box and then walk through some device checks (e.g., storage media and network). Figure 27 presents our observed server downtime distribution. After node failure, the system follows a 3-phase node handling failure to recover to the leader-only mode, i.e., regaining leadership (*T1*), performing local recovery (*T2*), and performing a leader scan (*T3*). Based on our collected traces, we observe that *T1* consumes 1.0ms. During *T2* phase, the node reads the commit log and executes missing requests. Figures 25 and 26 present the CDF of local node recovery (*T2*) and the number of recovered operation records (from the committed log) for 4 clusters, respective. Note that our cluster node is able to serve client requests starting from *T2* in a non-leader-only mode and enters the leader only mode after the scan finishes (*T3*).

The duration of the *T3* phase depends on scan performance. To enable leader-only reads, the new leader must scan through its owned range to learn the latest values. In some cases, Paxos writes must be done, and this imposes additional la-

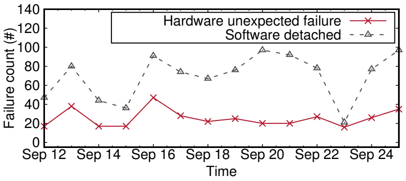


Figure 24: Failure rate among 2K custom VMs (Year 2018).

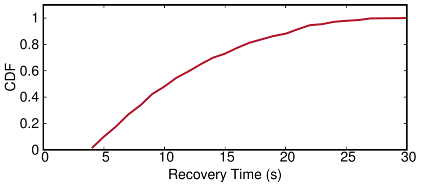


Figure 25: CDF of node recovery time inside a Stargate cluster.

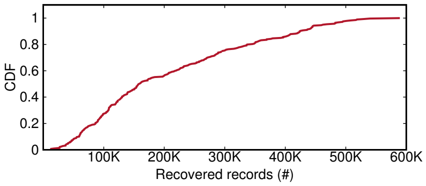


Figure 26: CDF of recovered records inside a Stargate cluster.

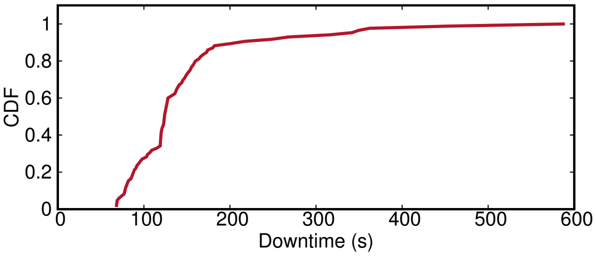


Figure 27: Node downtime CDF (after the hardware failure).

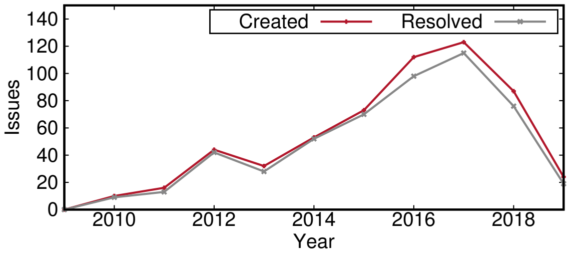


Figure 28: Metadata service corruption report over years.

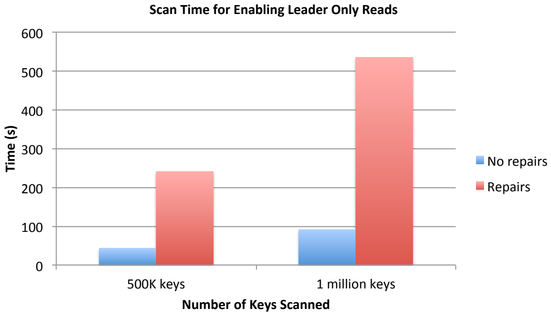


Figure 29: Time to perform a scan in order to enable leader-only reads.

tency costs for the scanning process. The worst-case repair time occurs when Paxos operations must be performed for every key. Conversely, the best-case scan time occurs when no consensus operations need to performed (i.e., the node has all of the latest data). Figure 29 provides the time associated with scans when the nodes are loaded with data comprising of 32-byte keys and 8KB values. When repairs need to run for every value, the total scan time is about 6× long. These measurements show the quick integration of recovering nodes into the metadata service.

B.5 Metadata Corruption Reports

Figure 28 shows the number of cases that have been reported by our QA based that have caused data unavailability or corrupt data being returned to the client based on the tests. We have not culled for duplicate issues, where the single cause manifested in multiple ways. The broad category of failures has changed over the years. Initially, it was the interaction with the local filesystems (fsyncs, o_directs), persistent media corruption, cluster misconfiguration. In recent years, it has been due to the addition of new features like leader-only reads, fast range migrations, balancing with no node downtimes. There have been a handful of protocol implementation issues that were weeded out fairly quickly.

C Testing Framework

Our testing strategy and framework has evolved over the years. Based on experience, we have found white box testing to be one of the key ways to identify implementation issues in new features and avoid regressions. We have instrumented code to simulate various scenarios and probabilistic error conditions like replica packet drops, timeouts, and erroneous key states. Whenever a bug is discovered in the field or in black box testing, we add a white box test to simulate the same condition along with making the fix.

We also have multiple test clusters that do end-to-end black box testing with error injections. Errors can be in the form of service restarts, service down, corruption over the network, timeouts, replays, and corruption of persistent store. As an example, we use a test devised specifically for fRSM that performs atomic increments on value(s) stored in key(s) *n* times (where each of the atomic increment is a CAS update) and, at the end, when all clients are done, check whether the final value of each key is *n* times the number of clients. While these clients are incrementing values using CAS, we randomly kill replica/leader nodes, insert failures, randomly drop messages between leader/replica nodes, add a delay in replying to messages, etc. Apart from incrementing values in the keys, we also delete keys in the middle of the test to go through the delete workflow and re-insert the key(s) with the value(s) seen just before the deletes, so the clients can continue incrementing the values. We also add/remove replicas to the metadata service while this test is underway to test add/remove node scenarios and different read variants. These type of tests can be performed within a developer environment and have aided in building a robust system.

It is non-trivial to pinpoint performance bottlenecks due to the complexity of our system. We instrument our logic across the metadata read/write execution path and report runtime statistics at multiple places, such as the number of outstanding requests at the Paxos leader, the hit rate of the key clock attribute, read/write/scan latency at leader and follower of one key-range, etc. This instrumentation has been helpful in identifying various performance issues.