CassandrEAS: Highly Available and Storage-Efficient Distributed Key-Value Store with Erasure Coding

Viveck Cadambe*
EE Department
Pennsylvania State University
University Park, PA, USA
viveck@engr.psu.edu

Kishori M. Konwar, Muriel Medard
Dept of EECS
MIT
Cambridge, USA
{kishori, medard}@mit.edu

Haochen Pan, Lewis Tseng
Dept of CS
Boston College
Boston, USA
{haochen.pan, lewis.tseng}@bc.edu

Yingjian Wu†
Dept of CSE
UCSD
San Diego, USA
yiw079@ucsd.edu

***Abstract*—In this work, we propose an erasure coding-based protocol that implements a key-value store with atomicity and *near-optimal storage cost*. Our protocol supports concurrent read and write operations while tolerating asynchronous communication and crash failures of any client and some fraction of servers. One novel feature is a *tunable knob* between the number of supported concurrent operations, availability, and storage cost.**

We implement our protocol into Cassandra, namely CassandrEAS (Cassandra + Erasure-coding Atomic Storage). Extensive evaluation using YCSB on Google Cloud Platform shows that CassandrEAS incurs moderate penalty on latency and throughput, yet saves significant amount of storage space.

***Index Terms*—atomicity, erasure-coding, KV store**

I. Introduction

Storage systems is the foundational platform for modern Internet services. Ever increasing reliance on massive data sets has forced developers to move towards a new class of scalable storage systems known as NoSQL key-value stores (KV-store). Most distributed NoSQL KV-stores (e.g., Cassandra [1], Riak [2], Dynamo [30]) support a flexible data schema and simple GET/PUT (or Read/Write) interface for reading and writing data items. The data items are replicated at multiple servers to provide high fault-tolerance and availability.

One drawback of the replication-based approach is the high storage cost. Erasure coding (EC) has been integrated with other types of KV-store to reduce storage cost (e.g., DepSky [8] for cloud-of-clouds, Cocytus [34] for in-memory KV-store and Giza [11] for cross-datacenter KV-store). This work is motivated by the following question: *is it possible to use erasure coding in a NoSQL KV-store to reduce storage cost while maintaining strong consistency with moderate performance penalty?* We provide an affirmative answer by introducing **CassandrEAS (Cassandra + Erasure-coding Atomic Storage)**, a customized version of Cassandra [1] with a new EC-based storage protocol that achieves *atomicity* [23], a form of strong consistency.

In this paper, we first present a novel quorum and erasure-code-based and storage cost optimized protocol called Two-Round Erasure coded Atomic Storage-Optimized (TREAS-OPT). In addition to the improved storage cost, when compared to the existing erasure-code based algorithms, TREAS-OPT algorithm is simple to implement as the high-level structure of the algorithm is very similar to the classic replication-based algorithm by Attiya, Bar-Noy and Dolev [6] and practical quorum storage [7], [30]. TREAS-OPT is a two-round protocol that uses logical timestamps, which obviates the reliance on any physical (machine) clock to implement atomic KV-store. We also modify TREAS-OPT to use physical timestamp that results in a single-round distributed storage protocol One-Round Erasure coding Atomic Storage (OREAS). Both algorithms are suitable for quorum-based KV-store.

Cassandra: We chose to develop our system on top of Cassandra. Even though Cassandra was released in 2008, it is still one of the most used NoSQL KV-stores in industry. It is also an active open-source project (recent stable release in February 2020) [4]. Plus, its quorum-based replication [7] shares similarly with other popular systems like Riak [2] and Dynamo [30]. For many big data applications, Cassandra offers a good balance of properties, e.g., high availability, scalability, and tunable consistency. Our system CassandrEAS provides similar guarantees in scalability and availability.

Cassandra offers tunable consistency guarantees, ranging from eventual consistency to strong consistency. Popular storage systems in industry, such as Google Spanner [16] and CockroachDB [3], are focusing on providing strong consistency. Inspired by the observation, this paper focuses on *atomicity* [23] (or linearizability [20]), one of the more popular forms of strong consistency, because atomicity is composable [20] and more intuitive for developing applications.

Cassandra provides high availability and scalability using a quorum-based (or peer-to-peer) design. It replicates data on multiple servers, and each server can serve any read/write

*Authors ordered alphabetically. †Yingjian worked on this project when he was affiliated with Boston College. Authors from Boston College are supported in part by National Science Foundation award CNS1816487. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

request from clients using the practical quorum approach (or Dynamo-style quorum) [7] for serving read and write operations. The approach does not use master node or consensus protocol to coordinate servers; hence, Cassandra has high performance without a single point of failure.* Cassandra's quorum-based design make it difficult to use prior EC-based solutions [8], [11], [34]. Cocytus [34] uses a master-based design, whereas Giza [11] is a consensus-based protocol. DepSky [8] does not provide atomicity.

Main Contributions

In the theoretical contribution, we propose two novel EC-based atomic storage protocols: TREAS-OPT and OREAS. From the engineering perspective, we demonstrate that our protocols can be seamlessly integrated with a real-world system, Cassandra. We made minimum modification to the original Cassandra codebase, and Cassandra users can call Cassandra's original read and write API directly *without* knowing the details of the algorithm. Our implementation can be found at https://github.com/yingjianwu199868/cassandra/

Using TREAS-OPT and OREAS, CassandrEAS also provides availability and scalability at a similar level of Cassandra, because both systems do not use master or consensus. One interesting feature of our protocols is a *tunable knob* that allows clients to choose the desired level of availability (i.e., the maximum number of tolerated server failures), storage cost, and the number of supported concurrent operations. If there are more server failures or more concurrent operations than the specified parameters, then CassandrEAS may return stale value or fail to serve client's request. A protocol for self-stabilization, reconfiguration, or recovery is left as an interesting future work.

For evaluation, we deploy CassandrEAS in Google Cloud Platform (GCP) and conduct extensive experiments using the Yahoo! Cloud Service Benchamarking (YCSB) workload generator [12]. YCSB is a practical tool used to benchmark many KV-stores. Our evaluation results in Section V-B indicate that CassandrEAS incurs moderate performance penalty, yet saves significant amount of storage space.

Cassandra vs. CassandrEAS: Table I summarizes the comparison between Cassandra and CassandrEAS when storing one unit of data (i.e., data size is normalized to 1). We do not count the size of meta-data such as timestamps. Erasure coding is more useful when dealing with larger data (say in terms of 100 Bs), so meta-data size is practically negligible. In the paper, n denotes the number of servers that store the data, δ represents the maximum number of writes concurrent with any read on the *same* key-value pair, and f is the maximum number of tolerated crash servers. For compactness, let $k = n - 2f$. For Cassandra, we choose the majority quorum configuration.

In practical systems, the number of concurrent writes on the *same* KV-pair is small in most cases, since each operation completes in the order of 100 ms. If CassandrEAS with $n = 9$ is configured to tolerate $f = 2$ crashed servers, and support $\delta = 3$ concurrent writes. In this case, $k = 5$, so the data storage cost at each server becomes $\frac{1}{\lceil \frac{k}{\delta+1} \rceil} = \frac{1}{\lceil \frac{5}{3+1} \rceil} = \frac{1}{2}$.

*Cassandra uses Paxos [24] to support transactions, but it does *not* use consensus for non-transaction operations, e.g., read and write operations. Following the design philosophy, our algorithms do not support transactions.

	Cassandra	CassandrEAS	ARES
Storage cost/server	1	$\frac{1}{\lceil \frac{k}{\delta+1} \rceil}$	$(\delta+1)\frac{1}{k}$
f, max crashes	$\frac{n}{2}-1$	$\frac{n-k}{2}$	$\frac{n-k}{2}$

TABLE I: Comparison of fault-tolerance level and storage-cost for Cassandra (Quorum), CassandrEAS and ARES

.

In comparison, Cassandra's storage cost is 1, as each server stores the original data. Hence, CassandrEAS saves 50% of storage space. If we have the configuration $n = 7, f = 1, \delta = 1$, then $k = 5$ and the storage cost of CassandrEAS is $\frac{1}{3}$, a 67% reduction in storage space.

II. PRELIMINARIES

A. Erasure Coding Storage Systems

Erasure coding (EC) is a space-efficient solution for data storage. EC has been traditionally used with great success for storage cost reduction in *write-once, read-many-times* data stores (e.g., [8], [13], [21], [27], [29]). Recently there is an increasing interest in using EC in update-many-times, read-many-times data stores. As observed in [11], [34], with the advancement of hardware, it is possible to perform encoding/decoding in a real-time fashion. EC also has the potential to significantly reduce network bandwidth, as well as for system maintenance (such as repairing failed servers). Therefore, both information theory and system research communities have investigated the usage of erasure coding to reduce various kind of costs, e.g., [14], [28], [32], [33].

Three recent systems DepSky [8], Cocytus [34] and Giza [11] studied the applicability of erasure coding in other types of KV-stores. DepSky [8] uses erasure coding for efficient storage in cloud-of-clouds; however, it does *not* support atomicity. Giza [11], Microsoft's proprietary storage, is a Fast-Paxos-based multi-version cross-data center strongly consistent object store used in Microsoft's OneDrive storage system. Giza servers store erasure-coded elements instead of the original data to significantly reduce storage cost. Cocytus [34] is a master-based in-memory KV-store that guarantees strong consistency and reduces storage cost using erasure coding. For each key, value is erasure coded and the coded elements are stored among a subset of servers. In addition, the master server maintains a full copy of the value to provide high availability for read operations. As elaborated above, Cassandra's quorum-based design does not fit well with consensus protocol or master-based design. Therefore, we have to design a new EC-based protocol for using erasure coding in Cassandra.

The distributed computing community also shows interest in using erasure coding. EC-based algorithms for strongly consistent storage are an active area of research in theory community, e.g., [9], [18], [19], [26]. SODA [19] described an algorithm to achieve optimal storage cost; however, it pays a higher write communication cost. None of these algorithms

have been integrated or implemented real-world systems. It is not clear how to adapt them into practical systems.

B. Erasure Codes

In this paper, we consider the optimal erasure codes. In particular, we adopt an $[n, l]$ linear Maximum Distance Separable (MDS) code [22] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. The value refers to a specific version of the data in our context. An $[n, l]$ MDS code has the property that any l out of the n coded elements can be used to recover (or decode) the original value v. For encoding, v is divided[†] into l elements $v_1, v_2, \ldots, v_l$ with each element having size $\frac{1}{l}$ (assuming size of v is 1). The encoder takes the l elements as input and produces n coded elements $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi^{(n,l)}([v_1, \ldots, v_l])$, where $\Phi^{(n,l)}$ denotes the encoder. For ease of notation, we simply write $\Phi^{(n,l)}(v)$ to mean $[c_1, \ldots, c_n]$. The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding to the value v. Each coded element c_i also has size $\frac{1}{l}$. In our scheme, we store *one coded element* per key-value pair at each server.

C. Main Challenge

The key challenge in an EC-based storage protocol that is compatible with quorum-based system is to handle concurrent operations. To ensure high availability, the readers need to receive sufficient coded elements from the servers to be able to decode the version of the data that satisfies *atomicity*. This issue becomes complicated in practice due to the following reasons: (i) there might be concurrent write operations that write different versions of the data simultaneously; (ii) servers and clients might crash so that the servers do not have enough coded element for a particular version; and (iii) messages might arrive in an arbitrary order due to the asynchrony assumption of the underlying network.

D. Model and Definitions

A shared atomic storage (or atomic KV-store) can be emulated by composing individual atomic objects. Therefore, we aim to implement a single atomic read/write memory object. A read/write object takes a value from a set $\mathcal{V}$. We assume a system consisting of three distinct sets of processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers and $\mathcal{S}$, a set of servers. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to modify the value of a shared object, and each reader is allowed to obtain the value of that object. Nodes communicate through *asynchronous*, but *reliable* channels.

Executions. An *execution* of an algorithm A is an alternating sequence of states and actions of A starting with the initial state. An execution ξ is *well-formed* if any process invokes one operation at a time and it is *fair* if enabled actions perform a step infinitely often. In the rest of the paper we consider executions that are fair and well-formed. A node *crashes* in an execution if it stops taking steps; otherwise it is *non-faulty*.

Write and Read Operations. An implementation of a read or a write operation contains an *invocation* action and a *response* action (such as a return from the procedure). An operation π is *complete* in an execution, if it contains both the invocation and the *matching* response actions for π; otherwise π is *incomplete*. We say that an operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \rightarrow \pi'$, if the response step of π appears before the invocation step of π' in ξ. Two operations are *concurrent* if neither precedes the other. An implementation A of a read/write object satisfies the atomicity property if the conditions of Lemma 13.16 in [25] holds.

Storage and Communication Costs. We define the total storage cost as the size of the data stored *across all servers*, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of negligible size, and therefore, ignore in the calculation of storage and communication cost. Further, we normalize both the costs with respect to the size of the value v; in other words, we compute the costs assuming that v has size 1 unit.

Quorum system. We define our quorum system, $\mathcal{Q}$, to be the set of all subsets of $\mathcal{S}$ that have at least $\frac{n+k}{2}$ servers. We refer to the members of $\mathcal{Q}$, as quorum sets and they satisfy the following property.

Lemma. *For any k, $1 \leq k \leq n - 2f$. (i) If Q_1 ,$Q_2 \in \mathcal{Q}$, then $|Q_1 \cap Q_2| \geq k$. (ii) If the number of faulty servers is at most f , then Q contains at least one quorum set Q of non-faulty servers.*

Liveness of operations. We require algorithms to satisfy certain liveness properties, specifically, in every fair execution that satisfies certain restrictions in terms of the number of failed nodes, we require every operation by a non-faulty client completes eventually, irrespective of the behavior of other clients. Most replication-based atomic memory algorithms guarantee liveness, as long as certain number of servers remain non-faulty; however, similar fault-tolerance levels can be achieved erasure-code based systems only in carefully designed algorithms. Moreover, there are some lower bound results on the storage-cost for erasure-coded atomic memory algorithms in the presence of faulty clients [10], [31]. As a result, to circumvent this restriction many authors assume some restricting assumptions. CASGC, ORCAS-A and ORCAS-B [15] assume a bound on the number of concurrent operations. In the same vein, our protocol assume a known bound on the number of concurrent writes with a read to achieve liveness.

III. TREAS-OPT: STORAGE-OPTIMIZED TWO-ROUND ALGORITHM

The pseudo-code for TREAS-OPT is presented in Alg. 1. TREAS-OPT is parameterized by the number of servers n, the

[†]In practice, v can be viewed as a byte array, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of represent-ability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

quorum intersection size k, and the degree of concurrency that can be tolerated δ. Note that k does not represent the dimension of the code here. The algorithm uses an MDS code, with encoding function $\Phi^{(n,\ell)} : \mathbb{F}^{\ell} \rightarrow \mathbb{F}^{n}$ where, $\mathbb{F}$ represents the finite field over which encoding is performed, n represents the length of the code, and $\ell = \lceil \frac{k}{\delta+1} \rceil$ represents the dimension of the code.

A tag τ is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$, an ID of a writer. Let $\mathcal{T}$ be the set of all tags. Notice that tags could be defined in any totally ordered domain and given that this domain is countably infinite, then there can be a direct mapping to the domain we assume. For any $\tau_1, \tau_2 \in \mathcal{T}$, where $\tau_i = (z_i, w_i)$, we define $\tau_2 > \tau_1$ if (i) $\tau_2.z_2 > \tau_1.z_1$ or (ii) $\tau_2.z_2 = \tau_1.z_1$ and $\tau_2.w_2 > \tau_1.w_1$.

Each server s_i stores one state variable, $List$, which is a set of up to $(\delta + 1)$ (tag, coded-element) pairs. Initially the set at s_i contains a single element, $List = \{(t_0, \Phi_i(v_0)\}$. A read operation at a client is implemented based on the get-data and put-data steps; and a write operation is based on the get-tag and put-data steps.

get-tag(): A client, during the execution of a get-tag() primitive, queries all the servers in $\mathcal{S}$ for the highest tags in their $Lists$, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. A server upon receiving the GET-TAG request, responds to the client with the highest tag, as $\tau_{max} \equiv \max_{(t,c) \in List} t$. Once the client receives the tags from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t and returns it .

put-data($\langle t_w, v \rangle$): During put-data($\langle t_w, v \rangle$), a client sends the pair $(t_w, \Phi_i(v))$ to each server $s_i \in \mathcal{S}$. When a server s_i receives a message $(\text{PUT-DATA}, t_w, c_i)$, it adds the pair in its local $List$, trims the pairs with the smallest tags exceeding the length $(\delta+1)$ of the $List$, and replies with an ack to the client. In particular, s_i replaces the coded-elements of the older tags with $\perp$, and maintains only the coded-elements associated with the $(\delta + 1)$ highest tags in the $List$ (see Line Alg. 2:13-17). The client completes the primitive operation after getting acks from $\lceil \frac{n+k}{2} \rceil$ servers.

get-data(): A client, during the execution of a get-data() primitive, queries all the servers in $\mathcal{S}$ for their local variable $List$, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Once the client receives $Lists$ from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t, such that: (i) its corresponding value v is decodable from the coded elements in the lists; and (ii) t is the highest tag seen from the responses of at least k $Lists$ (see lines Alg. 1:21-24) and returns the pair (t, v). Note that in the case where anyone of the above conditions is not satisfied the corresponding read operation does not complete.

The main technical aspect in which TREAS-OPT differs with ARES is the liveness proof. We argue that despite the changes, TREAS-OPT guarantees termination of every read operation whose concurrency is below δ.

Safety and Liveness properties Now we state the safety and liveness properties of TREAS-OPT.

Theorem (Atomicity)**.** *Any well-formed and fair execution of* TREAS-OPT *is atomic.*

Proof. Consider any well-formed execution β of TREAS-OPT, all of whose invoked read or write operations complete. Let Π denote the set of all completed read and write operations in β. We first define a partial order $(\prec)$ on Π. For any completed write operation π, we define $tag(\pi)$ as the variable t_w. For any completed read operation π, we define $tag(\pi)$ as the value of t_r. Now, in Π the relation $\prec$ is defined as follows: For any $\pi, \phi \in \Pi$, we say $\pi \prec \phi$ if one of the following holds: (i) $tag(\pi) < tag(\phi)$, or (ii) $tag(\pi) = tag(\phi)$, and π and ϕ are write and read operations, respectively. Atomicity is proved by using Lemma 13.16 in [25], which essentially requires any execution to hold four properties to guarantee atomicity. Let us denote them by $P1$, $P2$, $P3$ and $P4$. Property $P1$ is easily satisfied by our executions, so we show that any execution satisfies the remaining properties. Let ϕ and π denote two operations in Π such that ϕ completes before π starts in β. Let c_ϕ and c_π denote the clients that invokes ϕ and π, respectively.

Property $P2$ We want to show that $\pi \not\prec \phi$. Below we consider the four possible cases of ϕ and π.

ϕ, π are writes: It is enough to prove that $tag(\pi) > tag(\phi)$. Consider the put-data phase of ϕ, where the writer c_ϕ sends the pair (t_w, v) to all servers in $\mathcal{S}$. Let us denote the set $\mathcal{S}_\phi$ of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_ϕ during the put-data phase. Now, observe that the maximum tag in any server's $List$ is monotonically non-decreasing, because in algorithm B any server add tags to $List$ only in lines Alg. 2:13–17. Once added, a tag is never removed from $List$. Therefore, at each server in S_ϕ the maximum tag in $List$ at the time of sending the responses to c_ϕ in the put-data phase is at least $tag(\phi)$ Now, suppose S_π be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_π during the get-tag phase of π. Therefore, at the point of the execution when π is invoked, the maximum tag in $List$ of each server is at least $tag(\phi)$. Since $|S_\phi| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $S_\phi \cap S_\pi \neq \emptyset$. Therefore, there is at least one of the responses from the servers in the get-tag (see lines Alg. 2:15) has a tag at least $tag(\phi)$. So, the t_w is greater than $tag(\phi)$. So $tag(\pi) \geq t_w > tag(\phi)$ hence, $\pi \not\prec \phi$.

ϕ is a read, π is a write: By virtue of the definition of $\prec$, it is enough to prove that $tag(\pi) \geq tag(\phi)$. The rest of the argument is very similar to the previous case.

ϕ is a write, π is a read: From the definition of $\prec$, it is enough to prove that $tag(\pi) \geq tag(\phi)$. Consider the put-data phase of ϕ, where the reader c_ϕ sends the pair (t_w, v) to all servers in $\mathcal{S}$. Let us denote the set $\mathcal{S}_\phi$ of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_ϕ. Note that for each server in S_ϕ the maximum tag in $List$ at the time of sending the responses to c_ϕ in the put-data phase is at least $tag(\phi)$ (see lines Alg. 2:13–17). Now, suppose S_π be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_π, with the values in their $Lists$ during the get-data phase of π. Since the maximum tag in any server's $List$ is monotonically non-decreasing.therefore, at the point of the execution when π is invoked, the maximum tag in $List$ of each server is at least $tag(\phi)$ because for ϕ to complete we must have $t^{dec}_{max} = t^{*}_{max}$ in line Alg. 1:25. Since $|S_\phi| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $S_\phi \cap S_\pi \neq \emptyset$. Therefore, there is at least one of the reponses from the servers in the get-data (see lines Alg. 1:25) has

Algorithm 1 The reader/writer client-side steps for implementing TREAS-OPT.

```
    /* at reader r */
    operation read()
2:    ⟨t_r, v⟩ ← get-data()
      put-data(⟨t_r, v⟩)
4:    return v
    end operation

    /* at writer w */
6:  operation write(v)
      t_max ← get-tag()
8:    t_w ← (t_max.z + 1, w)
      put-data(⟨t_w, v⟩)
10: end operation
     at each process p_i ∈ I

12: procedure get-tag()
      send (QUERY-TAG) to each s ∈ S
14:   until receives ⟨t_s, e_s⟩ from ⌈(n+k)/2⌉ servers
      t_max ← max({t_s : received ⟨t_s, v_s⟩ from s})
16:   return t_max
    end procedure

18: procedure get-data()
      send (QUERY-LIST) to each s ∈ S
20:   until receives List_s from each server s ∈ S_g s.t. |S_g| = ⌈(n+k)/2⌉
      Tags*_{≥k} = set of tags that appears in k lists
22:   Tags*_{dec ≥ℓ} = tags appearing in ℓ lists with values
      t*_max ← max Tags*_{≥k}
24:   t^dec_max ← max Tags*_{dec ≥ℓ}
      if t^dec_max ≥ t*_max then
26:     v ← decode value for t^dec_max
      return ⟨t^dec_max, v⟩
28: end procedure

    procedure put-data(⟨τ, v⟩))
30:   code-elems = [(τ, e_1), ..., (τ, e_n)], e_i = Φ_i(v)
      send (PUT-DATA, ⟨τ, e_i⟩) to each s_i ∈ S
32:   until receives ACK from ⌈(n+k)/2⌉ servers
    end procedure
```

Algorithm 2 The response protocols at any server $s_i \in \mathcal{S}$ in TREAS-OPT for client requests.

```
    /* at each server s_i ∈ S */
2:  State Variables:
    List ⊆ T × C_s, initially {(t_0, Φ_i(v_0))}

    Upon receive (QUERY-TAG) from q
4:    τ_max = max_{(t,c)∈List} t
      Send τ_max to q
6:  end receive

    Upon receive (QUERY-LIST) from q
8:    Send List to q
    end receive

10:
    Upon receive (PUT-DATA, ⟨τ, e_i⟩) from q
12:   τ_max = max_{(t,c)∈List} t
      if τ ≥ τ_max then
14:     List ← List\{⟨τ_max, e_i⟩ : ⟨τ_max, e_i⟩ ∈ List}
        List ← List ∪ {⟨τ, e_i⟩, ⟨τ_max, ⊥⟩}
16:   else
        List ← List ∪ {⟨τ, ⊥⟩}
18:   Send ACK to q
    end receive
```

a tag at least $tag(\phi)$. So, the t_r is as large as $tag(\phi)$. So $tag(\pi) \geq t_r \geq tag(\phi)$ hence, $\pi \not\prec \phi$.

ϕ, π are reads: From the definition of $\prec$, it is enough to prove that $tag(\pi) \geq tag(\phi)$. Consider the put-data phase of ϕ, where the reader c_ϕ sends the pair (t_r, v) to all servers in $\mathcal{S}$. Let us denote the set $\mathcal{S}_\phi$ of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_ϕ. Note that for each server in S_ϕ the maximum tag in $List$ at the time of sending the responses to c_ϕ in the put-data phase is at least $tag(\phi)$ (see lines Alg. 2:13–17). Now, suppose S_π be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_π, with the data in their $List$s during the get-data phase of π. Since the maximum tag in any server's $List$ is monotonically non-decreasing.therefore, at the point of the execution when π is invoked, the maximum tag in $List$ of each server is at least $tag(\phi)$ because for ϕ to complete we must have $t^{dec}_{max} = t^*_{max}$ in line Alg. 1:25. Since $|S_\phi| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $S_\phi \cap S_\pi \neq \emptyset$. Therefore, there is at least one of the reponses from the servers in the get-data (see lines Alg. 2:25) has a tag at least $tag(\phi)$. So, the t_r is as large as $tag(\phi)$. So $tag(\pi) \geq t_r \geq tag(\phi)$ hence, $\pi \not\prec \phi$.

Property $P3$ This follows from the construction of tags, and the definition of the partial order ($\prec$).

Property $P4$ This follows from the definition of partial order ($\prec$), and by noting that value returned by a read operation π is simply the value associated with $tag(\pi)$. □

Theorem (Liveness)**.** *Let β denote a well-formed and fair execution of* TREAS-OPT *with parameters $[n, k, \delta]$ over a system of n servers. If the number of write operations that are with any valid read operation in β bounded by δ, and the number of server failures is bounded by $\lceil \frac{n+k}{2} \rceil - 1$, then every operation in β terminates.*

Proof. Note that in the read and write operation the get-tag and put-data operations initiated by any non-faulty client always complete. Therefore, the liveness property with respect to any write operation is clear because it uses only get-tag and put-data operations. So, we focus on proving the liveness property of any read operation π, specifically, the get-data operation completes. Let α be an execution of TREAS-OPT and let c_{σ^*} and c_π be the clients that invoke the write operation σ^* and read operation c_π, respectively.

Let S_{σ^*} be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_{σ^*}, in the put-data operations, in σ^*. Let S_{σ^π} be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_π during the get-data step of π. Note that in α at the point execution T_1, just before the execution of π, none of the write operations in Λ is complete. Observe that, by algorithm design, the coded-elements corresponding to t_{σ^*} are garbage-collected from the $List$ variable of a server only if more than δ higher tags are introduced by subsequent writes into the server. Since the number of concurrent writes $|\Lambda|$, s.t. $\delta > |\Lambda|$ the corresponding value of tag t_{σ^*} is not garbage

collected in α, at least until execution point T_2 in any of the servers in S_{σ^*}.

Therefore, during the execution fragment between the execution points T_1 and T_2 of the execution α, the tag and coded-element pair is present in the $List$ variable of every in S_{σ^*} that is active. As a result, the tag and coded-element pairs, $(t_{\sigma^*}, \Phi_s(v_{\sigma^*}))$ exists in the $List$ received from any $s \in S_{\sigma^*} \cap S_\pi$ during operation π. Note that since $|S_{\sigma^*}| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $|S_{\sigma^*} \cap S_\pi| \geq k$ and hence $t_{\sigma^*} \in Tags_{dec}^{\geq k}$, the set of decodable tag, i.e., the value v_{σ^*} can be decoded by c_π in π, which demonstrates that $Tags_{dec}^{\geq k} \neq \emptyset$. Next we want to argue that $t^*_{max} = t^{dec}_{max}$ via a contradiction: we assume $\max Tags_*^{\geq k} > \max Tags_{dec}^{\geq k}$. Now, consider any tag t, which exists due to our assumption, such that, $t \in Tags_*^{\geq k}$, $t \notin Tags_{dec}^{\geq k}$ and $t > t^{dec}_{max}$. Let $S^k_\pi \subset S$ be any subset of k servers that responds with t^*_{max} in their $List$ variables to c_π. Note that since $k > n/3$ hence $|S_{\sigma^*} \cap S_\pi| \geq \lceil \frac{n+k}{2} \rceil + \lceil \frac{n+1}{3} \rceil \geq 1$, i.e., $S_{\sigma^*} \cap S_\pi \neq \emptyset$. Then t must be in some servers in S_{σ^*} at T_2 and since $t > t^{dec}_{max} \geq t_{\sigma^*}$. Now since $|\Lambda| < \delta$ hence $(t, \perp)$ cannot be in any server at T_2 because there are not enough concurrent write operations (i.e., writes in Λ) to garbage-collect the coded-elements corresponding to tag t, which also holds for tag t^*_{max}. In that case, t must be in $Tag_{dec}^{\geq k}$, a contradiction. □

TREAS-OPT *vs.* ARES *and ABD:* TREAS-OPT shares some similarities with ABD [6] as well as ARES [26]. Specifically, in TREAS-OPT, like ARES, erasure coding is used, and each server stores a list of all the tags that it receives in the execution; even if the coded element corresponding to a tag t, logical timestamp, is garbage collected, the server stores a tuple of the form $(t, \perp)$. Unlike ARES, in TREAS-OPT each server stores only coded element, similar to ABD [6]. When a new tag-coded-element pair arrives at a server, if the tag is larger than the highest locally stored tag τ_{max}, then the server simply replaces the stored coded element by the newly arriving coded element; however, the server will continue to store a tuple of the form $(\tau_{max}, \perp)$.

While in ARES, each server stores $\delta + 1$ coded elements of a code with dimension k, in TREAS-OPT, each server stores one coded element of dimension $\lceil \frac{k}{\delta+1} \rceil$. Since the size of each coded element is effectively a fraction $\frac{1}{k}$ the size of the value, the server storage cost in ARES normalized by the size of the object is $\frac{\delta+1}{k}$, whereas the storage cost of TREAS-OPT is $\frac{1}{\lceil \frac{k}{\delta+1} \rceil}$ since $\frac{\delta+1}{2k} < \frac{1}{\lceil \frac{k}{\delta+1} \rceil} \leq \frac{\delta+1}{k}$ the storage cost of TREAS-OPT is no worse than the storage cost of ARES, and can be up to twice as efficient. For instance, if we have $n = 9$ servers, and we use quorums of size 6 so that $k = 3$, for a concurrency of $\delta = 1$, the storage cost of ARES is $2/3$, whereas the cost of TREAS-OPT is $\frac{1}{2}$. For the same system, if quorums of size 7 are used, and a concurrency can be bounded by $\delta = 3$, then the storage cost of ARES is $4/5 = 0.8$ whereas TREAS-OPT has a storage cost of $1/2 = 0.5$.

Algorithm 3 The reader/writer steps for implementing OREAS.

```
    /* at reader r */
    operation read()
2:    ⟨t_r, v⟩ ← get-data()
      put-data(⟨t_r, v⟩)
4:    return v
    end operation

    /* at writer w */
6:  operation write(v)
      t_w ← (t, w) where t is machine time at w
8:    put-data(⟨t_w, v⟩)
    end operation
```

IV. ONE-ROUND ERASURE CODING ATOMIC STORAGE

Instead of logical timestamp (tags), OREAS relies on physical timestamps, i.e., machine time at the coordinator. The advantage of using logical timestamp is that strong consistency is guaranteed even in the presence of clock drift and absence of synchronized clocks among the various servers. The downside is that it requires, as in TREAS-OPT, an additional communication round in order to gather the largest tag, which increases latency. On the other hand, in OREAS, the strong consistency guarantee is reliant on correctly synchronized clocks; however, write operations complete in one round.

V. OUR SYSTEM: CASSANDREAS

A. Implementation

We share our implementation experience here in hope to lower the barrier of implementing other replication- or EC-based protocols in Cassandra in the future.[‡] We have two main goals in our implementation: (i) make minimum modification to the original Cassandra codebase; (ii) allow Cassandra users to use our implementation *without* knowing the details of OREAS. That is, users can use the same Cassandra API in our system. The implementation turns out to be a difficult task because we are constrained to using only Cassandra's internal tools and data structures for communication and coordination. We spent enormous amount of time to decompose and understand Cassandra's internal logic because the codebase is not well documented and often provide poor comments. The version of Cassandra (version 3.11) that we used to develop contains around 57,000 LoC in Java. We also implemented the algorithm ARES, which uses logical timestamp to deal with loosely synchronized clock. Our implementation of both algorithms consists of around 2,000 LoC. We did not implement our protocols as a middleware because the protocols require system information (e.g., timestamps), which is not revealed externally.

Technical Details: We mainly modify the following two files in the Cassandra codebase. In Cassandra's terminology, "mutation" is essentially a write operation that modifies the internal state of the servers. (i) *StorageProxy.java* is where the coordinator server handles the user's read or write operation. Specifically, MUTATE function handles Cassandra user's write operation whereas FETCHROWS function handles user's read operation. The size of read/write quorums is also specified in this file; and (ii) *MutationVerbHandler.java* where each server's database engine handles incoming write requests from

[‡]We also has a separate paper that documents the implementation and benchmarking of other replication-based storage protocols in Cassandra [17].

the coordinator. Specifically, DOVERB function applies the mutation onto local storage.

One major challenge we encountered is that Cassandra does *not* provide an easy way to fetch users' requests and modify their mutations. We have to figure out a way to construct a new mutation when we need to add new fields such as timestamp and coded elements. Recall that we do not want to introduce new data structure, so we choose to use Cassandra's column family data model (i.e., an ordered collection of rows) to store the List variable at each server. CassandrEAS uses BackBlaze Reed-Solomon Code.

B. Evaluation

We evaluate the performance of CassandrEAS by comparing it to the vanilla Cassandra (version 3.11). Cassandra-All requires the coordinator to hear from every server, whereas Cassandra-Quorum only requires to hear from a majority quorum of servers.

Cluster Configuration and Workload: We use the Google Cloud Platform (GCP) as our testing platform. Each virtual machine (VM) is equipped with 4 virtual CPUs, 16 GB memory, and hosting Ubuntu 14.04 LTS. All VMs are located in datacenter us-east1-c (South Carolina). VMs use internal IP's for communication. The average RTT between any two VMs is around 0.3 ms, and TCP bandwidth measured by Iperf is around 7.5 Gbits/sec. For most of evaluation, our cluster consists of 5 VMs, and 3 YCSB single-threaded clients. Thus, $\delta = 3$. We use YCSB to generate realistic workload. We first insert a total of 30,000 KV-pairs, and each user performs 100,000 read or write operations. Recall that we have 3 YCSB clients, so we have 300,000 operations in total. We report the aggregated throughput, i.e., the sum of total operations per second across three YCSB clients.

Performance: Figure 1 presents our evaluation results in different configurations. Figures 1a to 1c show latency and throughput under different write ratio with data size equal to $128B$. Read latency is comparable across all four algorithms. ARES has poor write latency due to the usage of logical timestamps, which requires an extra round-trip. OREAS has moderate penalty in write latency. Throughput is comparable in the case of write ratio 0.1, a common case in NoSQL KV-stores. Figures 1d to 1f show latency and throughput under different data size with write ratio 0.1. We only show average latency, as 95 percentile has the same pattern. Figure 1f demonstrates that CassandrEAS suffers moderate penalty on throughput.

Availability, Fault-tolerance, and Scalability: CassandrEAS is highly available, fault-tolerant, and scalable, because its core is based on Cassandra. It can continue serving client's operations as long as the conditions specified by f and δ are satisfied. We have tested our system with 7 servers and 1 crashed server, and observed minimal disruption on throughput (ranging from 0.1% to 2.5% decrease). We also observed that each YCSB client's throughput decreased a bit right after a server crashed, and later came back to normal throughput (compared with a cluster without any fault). Finally, we also tested clusters with 7, 9, and 11 servers. The throughput of OREAS is in the range of $77 - 80\%$ of Cassandra's. This demonstrates that CassandrEAS' performance is also scalable, i.e., the throughput increases when n increases.

Correctness: We develop a consistency checker based on the approach specified in [25] to ensure that our implementation also provides this guarantee. Validating strong consistency requires precise clock synchronization across all servers. This is impossible to achieve in a distributed system where clock drift is inevitable. To circumvent it, we deploy our CassandrEAS servers on a single machine and use Mininet [5] to simulate the underlying network communication. Our checker then uses the machine time as the global clock. We collect multiple traces under different configurations with failures. The checker verifies that all traces we tested satisfy atomicity.

VI. SUMMARY

Strong consistency, storage efficiency, and availability are three key features for NoSQL KV-stores. We demonstrated that through CassandrEAS – erasure coding can be used to reduce storage cost while incurring moderate performance penalty. One interesting future work is to investigate how to apply TREAS-OPT and OREAS in other NoSQL KV-stores, or to Cassandra to support transaction-related primitives beyond the simple read or write.

REFERENCES

[1] The apache cassandra project. http://cassandra.apache.org/.

[2] basho. http://basho.com/products/riak-s2/. [Online; accessed 30-October-2018].

[3] cockroachdb. https://www.cockroachlabs.com/.

[4] Db-engines: Cassandra system properties. https://db-engines.com/en/system/Cassandra. [Online; accessed 14-March-2020].

[5] Mininet. http://mininet.org/. [Online; accessed 14-March-2020].

[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. Journal of the ACM, 42(1):124–142, 1996.

[7] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. PVLDB, 5(8):776–787, 2012.

[8] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. ACM Transactions on Storage, 9(4):1–33, 2013.

[9] Viveck R. Cadambe, Nancy A. Lynch, Muriel Médard, and Peter M. Musial. A coded shared atomic memory algorithm for message passing architectures. Distributed Computing, 30(1):49–73, 2017.

[10] Viveck R Cadambe, Zhiying Wang, and Nancy Lynch. Information-theoretic lower bounds on the storage cost of shared memory emulation. In Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, pages 305–313. ACM, 2016.

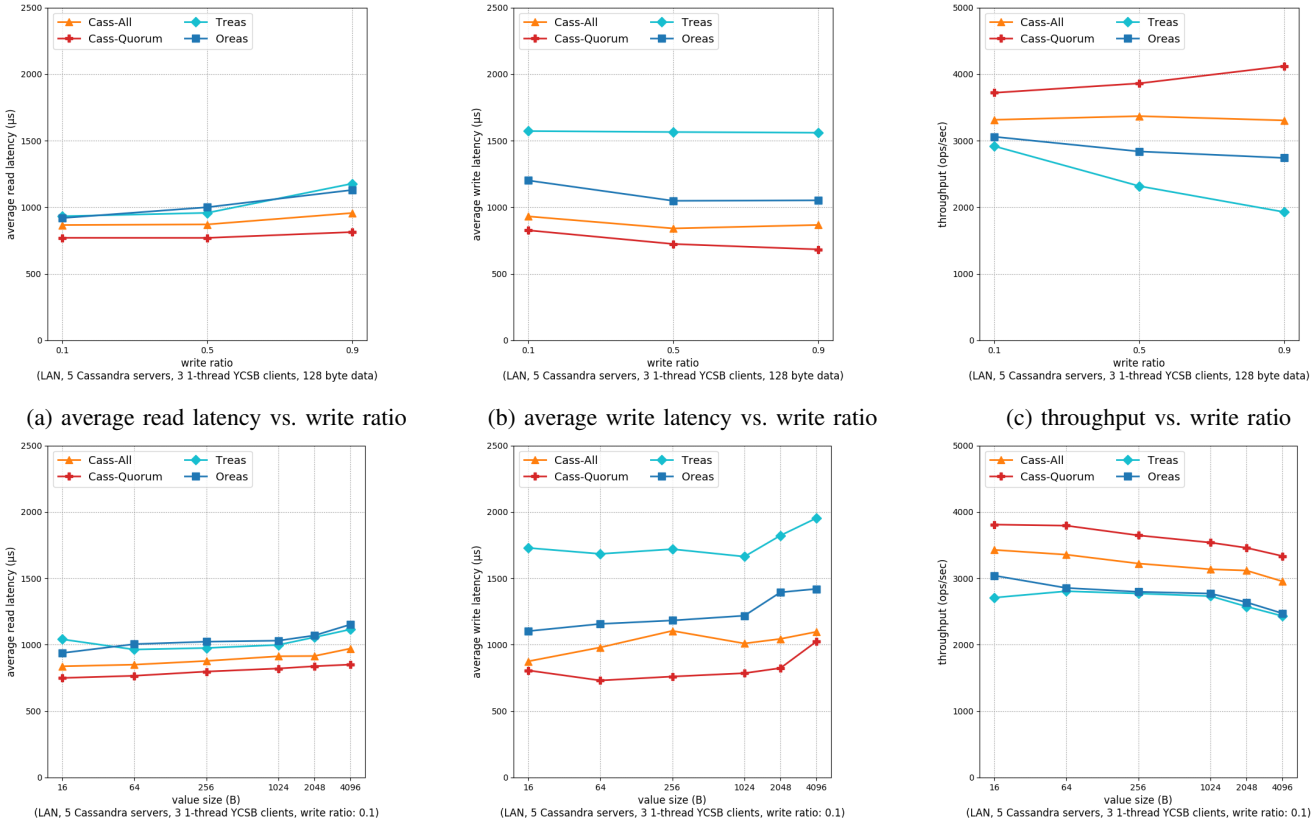
[11] Yu Lin Chen Chen, Shuai Mu, and Jinyang Li. Giza: Erasure coding objects across global data centers. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17), pages 539–551, 2017.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, pages 143–154, 2010.

[13] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. Proceedings of the IEEE, 99(3):476–489, 2011.

[14] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. IEEE transactions on information theory, 56(9):4539–4551, 2010.

[15] Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In DISC '08: Proceedings of the 22nd international symposium on Distributed Computing, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag.



(a) average read latency vs. write ratio

(b) average write latency vs. write ratio

(c) throughput vs. write ratio

(d) avg read latency vs. data size

(e) avg write latency vs. data size

(f) throughput vs. data size

Fig. 1: Comparison of latency and throughput of different configurations

[16] Corbett et al. Spanner: Google's globally-distributed database. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[17] Guo-Shu Gau et al. Practical experience report: Cassandra+: Trading-off consistency, latency, and fault-tolerance in cassandra. In 22nd International Conference on Distributed Computing and Networking, ICDCN 2021, 2021.

[18] Kishori M. Konwar et al. A layered architecture for erasure-coded consistent distributed storage. In Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, pages 63–72, 2017.

[19] Konwar et al. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2016.

[20] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 12(3):463–492, 1990.

[21] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In Proc. USENIX Annual Technical Conference, pages 15–26, 2012.

[22] W. C. Huffman and V. Pless. Fundamentals of error-correcting codes. Cambridge university press, 2003.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565, July 1978.

[24] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[25] N.A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.

[26] Nicolas C. Nicolaou, Viveck R. Cadambe, N. Prakash, Kishori M. Konwar, Muriel Médard, and Nancy A. Lynch. ARES: adaptive, reconfigurable, erasure coded, atomic storage. In 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019, pages 2195–2205. IEEE, 2019.

[27] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In 13th USENIX Conference on File and Storage Technologies (FAST), pages 81–94, 2015.

[28] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In OSDI, pages 401–417, 2016.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In Proceedings of the 39th international conference on Very Large Data Bases, pages 325–336, 2013.

[30] Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In Proceeding of the 4th Int'l Workshop on Large Scale Distributed Systems and Middleware (LADIS 2010), 2004.

[31] Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. Space bounds for reliable storage: Fundamental limits of coding. Technical report, arXiv:1507.05169v1, 2015.

[32] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. IEEE Transactions on Information Theory, 60(8):4661–4676, 2014.

[33] Zhiying Wang and Viveck R Cadambe. Multi-version coding—an information-theoretic perspective of consistent distributed storage. IEEE Transactions on Information Theory, 64(6):4540–4561, 2018.

[34] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In 14th USENIX Conference on File and Storage Technologies (FAST 16), pages 167–180, 2016.