# CassandrEAS: Highly Available and Storage-Efficient Distributed KV-Store with Erasure Coding

***Abstract*—Cassandra is a popular NoSQL database, often used as a distributed key-value store (KV-store) in industry. To increase availability and fault-tolerance, Cassandra replicates data on multiple servers, which unfortunately wastes storage space in the case of high replication. Recently, there has been a series of efforts on integrating erasure coding (EC) with various types of KV-store to reduce storage cost. To our knowledge, there is *no* EC-based protocol for Cassandra or similar quorum-based KV-stores owing to the difficulties in integrating erasure coding with quorum-based design.**

**In this work, we propose a EC-based protocol that achieves strong consistency with *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 of our protocol 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). Evaluation using YCSB on Google Cloud Platform with different configurations shows that CassandrEAS incurs moderate penalty on latency and throughput, yet saves significant amount of storage space.**

***Index Terms*—Strong consistency, fault-tolerence, distributed storage**

## 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 [26]) 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., Cocytus [29] for in-memory KV-store and Giza [9] 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* [20], a form of strong consistency.

*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 [6] shares similarly with other popular systems like Riak [2] and Dynamo [26]. 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 [11] and CockroachDB [3], are focusing on providing strong consistency. Inspired by the observation, this paper focuses on *atomicity* [20] (or linearizability [14]), one of the more popular forms of strong consistency, because atomicity is composable [14] 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 request from clients using the practical quorum approach (or Dynamo-style quorum) [6]. The approach does not use master node or consensus protocol to coordinate servers; hence, Cassandra has a high performance without a single point of failure. Cassandra's quorum-based design make it difficult to use prior EC-based solutions [9], [29], which either use a master-based design or consensus-based protocol.

*Main Contributions:* In this paper, we present our novel quorum-based storage protocol, One-Round Erasure coding Atomic Storage (OREAS). We implement OREAS into Cassandra [19], namely CassandrEAS. More precisely, we replace Cassandra's replication mechanism by our EC-based protocol. We try to make 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.[*] To the best of our knowledge, this is the first implementation in NoSQL KV-Store that uses erasure coding.[†]

Using 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 OREAS 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

[*]Our algorithm does not support transaction; hence, those transaction-related APIs are not supported. Note that Cassandra uses Paxos [21] to support transactions, and this is why OREAS cannot be directly applied.

[†]We will release our code on Github once the paper is accepted in hope that stimulate more discussion and evaluation on using erasure codes in similar systems.

generator [10]. YCSB is a realistic tool used to benchmark many KV-stores. Our evaluation results in Section VI-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 $B$s), so meta-data size is practically negligible. In our presentation, $n$ denotes the number of servers that store the data, $\delta$ 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. To make our formula compact, let $k = n - 2f$. For Cassandra, we choose the majority quorum configuration.

| | Cassandra | CassandrEAS |
|---|---|---|
| Storage cost at each server | 1 | $\frac{1}{\lceil \frac{k}{\delta+1} \rceil}$ |
| Fault-tolerance level $(f)$ | $\frac{n}{2} - 1$ | $\frac{n-k}{2}$ |

TABLE I: Cassandra (Quorum) vs. CassandrEAS

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

$$\frac{1}{\lceil \frac{k}{\delta+1} \rceil} = \frac{1}{\lceil \frac{5}{3+1} \rceil} = \frac{1}{2}$$

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

*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., [12], [15], [23], [25]). Recently there is an increasing interest in using EC in update-many-times, read-many-times data stores. As observed in [9], [29], 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., [13], [24], [28].

Two recent systems Cocytus [29] and Giza [9] studied the applicability of erasure coding in other types of KV-stores. Giza [9], 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 [29] 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.

*Erasure Codes:* In this paper, we consider the optimal erasure codes. In particular, we adopt an $[n, l]$ linear Maximum Distance Separable (MDS) code [16] over a finite field $\mathbb{F}_q$ to encode and store the value $v$ among the $n$ servers. Value is referred 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.

*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.

## III. Model and Definitions

A shared atomic storage 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

[‡]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.

(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. Processes communicate via *messages* through *asynchronous*, *reliable* channels.

**Executions.** An *execution* of an algorithm $A$ is an alternating sequence of states and actions of $A$ starting with the initial state and ending in a state. An execution $\xi$ 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 process $p$ *crashes* in an execution if it stops taking steps; otherwise $p$ is *correct* or *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 $\pi$ is *complete* in an execution, if it contains both the invocation and the *matching* response actions for $\pi$; otherwise $\pi$ is *incomplete*. We say that an operation $\pi$ *precedes* an operation $\pi'$ in an execution $\xi$, denoted by $\pi \rightarrow \pi'$, if the response step of $\pi$ appears before the invocation step of $\pi'$ in $\xi$. 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 [22] 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 that gets 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 under the assumption that v has size 1 unit.

**Erasure Codes.** In TREAS-OPT, we use an $[n, k]$ linear MDS code [16] over a finite field $\mathbb{F}_q$ to encode and store the value $v$ among the $n$ servers. An $[n, k]$ MDS code has the property that any $k$ out of the $n$ coded elements can be used to recover (decode) the value $v$. For encoding, $v$ is divided[§] into $k$ elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of $v$ is 1). The encoder takes the $k$ elements as input and produces $n$ coded elements $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi^{(n,k)}([v_1, \ldots, v_k])$, where $\Phi^{(n,k)}$ denotes the encoder. For ease of notation, we simply write $\Phi^{(n,k)}(v)$ 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 $\frac{1}{k}$. In our scheme we store one coded element per server. We use $\Phi_i^{(n,k)}$ to denote the projection of $\Phi^{(n,k)}$ on to the ith output component, i.e., $c_i = \Phi_i^{(n,k)}(v)$. Without loss of generality, we associate the coded element $c_i$ with server $i$, $1 \leq i \leq n$. Below, when the dimension of the MDS code is clear we omit the code parameter $\Phi^{(n,k)}(\cdot)$ to write simply as $\Phi(\cdot)$.

[§]In practice $v$ is a file, 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 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 failed servers is at most $f$ , then $Q$ contains at least one quorum set $Q$ of non-failed 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, irrespective of the behavior of other clients. Although most replication-based based atomic memory algorithm guarantee liveness as long as certain no of servers remain non-faulty. Unfortunately, certain lower bounds on the storage-cost of erasure-coded atomic memory algorithm in the presence of faulty clients [8], [27]. As a result, to circumvent this restriction many authors assume certain restricting assumption. CASGC, ORCAS-A and ORCAS-B assume a bound on the number of concurrent operation. In the same vein, TREAS-OPT assumes a known bound on the number of concurrent writes with a read to achieve liveness.

## IV. TREAS-OPT: STORAGE-OPTIMIZED TWO-ROUND ALGORITHM

In this section, we present TREAS-OPT. The pseudo-code for TREAS-OPT is presented in Alg. 1. TREAS-OPT is parameterized by the number of servers $n$, the quorum intersection size $k$, and the degree of concurrency that can be tolerated $\delta$. 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 $\tau$ 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 $\tau_1, \tau_2 \in \mathcal{T}$ we define $\tau_2 > \tau_1$ if $(i)$ $\tau_2.z > \tau_1.z$ or $(ii)$ $\tau_2.z = \tau_1.z$ and $\tau_2.w > \tau_1.w$.

TREAS-OPT shares some similarities with ABD as well as TREAS. Specifically, in TREAS-OPT, like TREAS, 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 $t$ is garbage collected, the server stores a tuple of the form $(t, \perp)$. However, unlike TREAS, in TREAS-OPT each server stores only coded element, similar to ABD. When a new tag-coded-element pair arrives at a server, if the tag is larger than the highest locally stored tag $\tau_{max}$, 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 TREAS, 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 TREAS 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 TREAS, 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 TREAS 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 TREAS is $4/5 = 0.8$ whereas TREAS-OPT has a storage cost of $1/2 = 0.5$.

The main technical aspect in which TREAS-OPT differs with, in terms of correctness, in comparison with TREAS is the liveness proof. We argue that despite the changes, TREAS-OPT guarantees termination of every read operation whose concurrency is below $\delta$.

**Safety and Liveness properties** Now we state the safety and liveness properties of TREAS-OPT. Due to lack of space the proofs are deferred for a later extended version of the paper.

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

*Proof.* Consider any well-formed execution $\beta$ of TREAS-OPT, all of whose invoked read or write operations complete. Let $\Pi$ denote the set of all completed read and write operations in $\beta$. We first define a partial order $(\prec)$ on $\Pi$. For any completed write operation $\pi$, we define $tag(\pi)$ as the variable $t_w$. For any complete read operation $\pi$, we define $tag(\pi)$ as the value of $t_r$. Now, in $\Pi$ 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 $\pi$ and $\phi$ are write and read operations, respectively. The proof of atomicity is based on proving the properties $P1$, $P2$ and $P3$. Let $\phi$ and $\pi$ denote two operations in $\Pi$ such that $\phi$ completes before $\pi$ starts in $\beta$. Let $c_\phi$ and $c_\pi$ denote the clients that invokes $\phi$ and $\pi$, respectively.

Property $P1$ We want to show that $\pi \not\prec \phi$. Below we consider the four possible cases of $\phi$ and $\pi$.

$\phi, \pi$ are writes: It is enough to prove that $tag(\pi) > tag(\phi)$. Consider the put-data phase of $\phi$, where the writer $c_\phi$ 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 $c_\phi$ 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 $S_\phi$ the maximum tag in $List$ at the time of sending the responses to $c_\phi$ in the put-data phase is at least $tag(\phi)$ Now, suppose $S_\pi$ be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to $c_\pi$ during the get-tag phase of $\pi$. Therefore, at the point of the execution when $\pi$ 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. **??**:**??**) 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, $\pi \not\prec \phi$.

$\phi$ is a read, $\pi$ 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.

$\phi$ is a write, $\pi$ is a read: This is very similar to the same case in the proof of Theorem **??**.

$\phi, \pi$ are reads: This is very similar to the same case in the proof of Theorem **??**.

Property $P2$ This follows from the construction of tags, and the definition of the partial order $(\prec)$.

Property $P3$ This follows from the definition of partial order $(\prec)$, and by noting that value returned by a read operation $\pi$ is simply the value associated with $tag(\pi)$. □

**Theorem** (Liveness). *Let $\beta$ 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 $\beta$ bounded by $\delta$, and the number of server failures is bounded by $\lceil \frac{n+k}{2} \rceil - 1$, then every operation in $\beta$ 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 of the DAP. So, we focus on proving the liveness property of any read operation $\pi$, specifically, the get-data operation completes. Let $\alpha$ be and execution of TREAS-OPT and let $c_{\sigma^*}$ and $c_\pi$ be the clients that invokes the write operation $\sigma^*$ and read operation $c_\pi$, respectively.

Let $S_{\sigma^*}$ be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to $c_{\sigma^*}$, in the put-data operations, in $\sigma^*$. Let $S_{\sigma^\pi}$ be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to $c_\pi$ during the get-data step of $\pi$. Note that in $\alpha$ at the point execution $T_1$, just before the execution of $\pi$, none of the the write operations in $\Lambda$ is complete. Observe that, by algorithm design, the coded-elements corresponding to $t_{\sigma^*}$ are garbage-collected from the $List$ variable of a server only if more than $\delta$ 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 $t_{\sigma^*}$ is not garbage collected in $\alpha$, at least until execution point $T_2$ in any of the servers in $S_{\sigma^*}$.

Therefore, during the execution fragment between the execution points $T_1$ and $T_2$ of the execution $\alpha$, the tag and coded-element pair is present in the $List$ variable of every in $S_{\sigma^*}$ 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 $\pi$. Note that since $|S_{\sigma^*}| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $|S_{\sigma^*} \cap S_\pi| \geq k$ and hence

**Algorithm 1** The the steps for implementing TREAS-OPT.

```
    /* at reader $r$ */
    operation read()
2:      $\langle t_r, v\rangle \leftarrow$ get-data()
        put-data($\langle t_r, v\rangle$)
4:      return $v$
    end operation

    /* at writer $w$ */
6:  operation write($v$)
        $t_{max} \leftarrow$ get-tag()
8:      $t_w \leftarrow (t_{max}.z + 1, w)$
        put-data($\langle t_w, v\rangle$)
10: end operation
     at each process $p_i \in \mathcal{I}$

12: procedure get-tag()
        send (QUERY-TAG) to each $s \in \mathcal{S}$
14:     until receives $\langle t_s, e_s\rangle$ from $\left\lceil \frac{n+k}{2} \right\rceil$ servers
        $t_{max} \leftarrow \max(\{t_s :$ received $\langle t_s, v_s\rangle$ from $s\})$
16:     return $t_{max}$
    end procedure

18: procedure get-data()
        send (QUERY-LIST) to each $s \in \mathcal{S}$
20:     until receives $List_s$ from each server $s \in \mathcal{S}_g$ s.t. $|\mathcal{S}_g| = \left\lceil \frac{n+k}{2} \right\rceil$
        $Tags_{*}^{\geq k}$ = set of tags that appears in $k$ lists
22:     $Tags_{dec}^{\geq \ell}$ = tags appearing in $\ell$ lists with values
        $t^*_{max} \leftarrow \max Tags_{*}^{\geq k}$
24:     $t^{dec}_{max} \leftarrow \max Tags_{dec}^{\geq \ell}$
        if $t^{dec}_{max} \geq t^*_{max}$ then
26:         $v \leftarrow$ decode value for $t^{dec}_{max}$
        return $\langle t^{dec}_{max}, v\rangle$
28: end procedure

    procedure put-data($\langle \tau, v\rangle$))
30:     code-elems $= [(\tau, e_1), \ldots, (\tau, e_n)]$, $e_i = \Phi_i(v)$
        send (WRITE, $\langle \tau, e_i\rangle$) to each $s_i \in \mathcal{S}$
32:     until receives ACK from $\left\lceil \frac{n+k}{2} \right\rceil$ 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 $s_i \in \mathcal{S}$

2:  State Variables:
    $List \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$

    Upon receive (QUERY-TAG) $_{s_i, c_k}$ from $q$
4:      $\tau_{max} = \max_{(t,c) \in List} t$
        Send $\tau_{max}$ to $q$
6:  end receive

    Upon receive (QUERY-LIST) $_{s_i, c_k}$ from $q$
8:      Send $List$ to $q$
    end receive
10:
    Upon receive (PUT-DATA, $\langle \tau, e_i\rangle$) $_{s_i, c_k}$ from $q$
12:     $\tau_{max} = \max_{(t,c) \in List} t$
        if $\tau \geq \tau_{max}$ then
14:         $List \leftarrow List \backslash \{\langle \tau_{max}, e_i\rangle : \langle \tau_{max}, e_i\rangle \in List\}$
            $List \leftarrow List \cup \{\langle \tau, e_i\rangle, \langle \tau_{max}, \perp\rangle\}$
16:     else
            $List \leftarrow List \cup \{\langle \tau, \perp\rangle\}$
18:     Send ACK to $q$
    end receive
```

$t_{\sigma^*} \in Tags_{dec}^{\geq k}$, the set of decodable tag, i.e., the value $v_{\sigma^*}$ can be decoded by $c_\pi$ in $\pi$, 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 $c_\pi$. Note that since $k > n/3$ hence $|S_{\sigma^*} \cap S_\pi| \geq \left\lceil \frac{n+k}{2} \right\rceil + \left\lceil \frac{n+1}{3} \right\rceil \geq 1$, i.e., $S_{\sigma^*} \cap S_\pi \neq \emptyset$. Then $t$ must be in some servers in $S_{\sigma^*}$ 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 $\Lambda$) 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. □

## V. ONE-ROUND ERASURE CODING ATOMIC STORAGE

In this section, we describe our EC-based storage protocol – One-Round Erasure coding Atomic Storage (OREAS). One desirable property of atomicity [20] (or linearizability [14]) is composability [14]. That is, a collection of atomic storage units used together in an arbitrary way still satisfy atomicity. Note that some other forms of strong consistency might not have such a property. This is also why we are interested in providing atomicity. Due to composability, we describe OREAS in terms of a single key-value pair (KV-pair), i.e., the key field is omitted in our discussion. Our implementation supports multiple clients accessing multiple KV-pairs. In our evaluation, we test CassandrEAS with multiple keys. Due to space constraint, we briefly describe our protocol at a high-level along with the pseudo-code. Most technical details (including proofs) are presented in the accompanied technical report.[¶] The pseudo-code for readers and writers is presented in Algorithm **??**, and pseudo-code for servers is in Algorithm **??**.

*Meta-data:* OREAS uses tags, where a tag $\tau$ is defined as a pair $(z, w)$ such that $w$ is an unique ID of the writer, and $z$ is the local machine time (when the writer issues a specific write operation). Every pair of tags can be compared in the lexicographical order. It is reasonable to use machine time in our context, since the "client" is a proxy server (or a coordinator), which has a synchronized clock with other servers. In fact, Cassandra uses machine time to identify new values as well. For the case that time synchronization is difficult, we have another protocol, called TREAS, which uses a logical timestamp to replace the machine time. For brevity, we focus on OREAS in this paper.

[¶]Our report is not currently available on web due to anonymous clause. If the paper is accepted, we will upload it to arxiv.

*Write Operation:* Suppose $v$ is the value that the writer intends to update the KV-pair with. The writer (or writer proxy) encodes $v$ to coded elements $c_1, c_2, \cdots c_n$, along with tag $\tau$, and sends the corresponding element to each server. Every server that receives a coded element stores the coded element in its $List$ and performs garbage collection (i.e., replacing elements with smaller tags or storing a pair with $\perp$ value field). Finally, it sends an acknowledgment message back to the writer. Once the writer receives acknowledgments from $\lceil \frac{n+k}{2} \rceil$ servers, it completes the write operation.

*Read Operation:* The reader (or reader proxy) requests all servers for the lists of values. Upon receiving the request, a server sends its local variable $List$ containing $(t, c_i)$ or $(t, \perp)$ to the reader. Once the reader receives responses from $\lceil \frac{n+k}{2} \rceil$ of the servers, it decodes the value $v$ corresponding to the tag $t_{max}$. Our protocol ensures that as long as the number of crashed servers is $\leq f$ and the number of concurrent writes is $\leq \delta$, a value is always decodable, and atomicity is satisfied. Next the reader computes the coded elements of $v$, and together with the servers executes the steps as in the write operation (i.e., write-back phase), and then completes the read operation.

*Correctness:* We prove our protocols satisfy strong consistency (atomicity) in our technical report. We also develop a consistency checker based on the approach specified in [22] to ensure that our implementation also provides this guarantee. Validating strong consistency requires precise clock synchronization across all servers, since it needs to use global clock to ordering operations and events. This is impossible to achieve in a distributed system where clock drift is inevitable. To circumvent this, 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 of 20,000 operations under different configurations with potentially machine failures. The checker verifies that all traces we tested satisfy strong consistency.

*Storage Cost:* In OREAS, erasure coding is used, and each server stores a list of all the tags that it has received so far. If a server receives the coded element corresponding to a tag newer than the current KV-pair with timestamp $t$, the server would update the current KV-pairs to the form $(t, \perp)$. The key feature that is different from other EC-based algorithms (e.g., [7], [9], [13], [17], [18], [24], [28], [29]) is that each server stores exactly one coded element in $List$ and the rest of the elements in $List$ are of the form $(t', \perp)$, where $t'$ is some tag or timestamps that were garbage collected and $\perp$ is a place holder for null data. The cost of storing one value of unit size at each server is hence $\frac{1}{\lceil \frac{k}{\delta+1} \rceil}$. Recall that $\delta$ is the maximum number of writes concurrent with any read operation, and $k = n - 2f$.

**Storage and Communication Costs for** TREAS-OPT**.** We now briefly present the storage and communication costs associated with TREAS-OPT. Recall that by our assumption, the storage cost counts the size (in bits) of the coded elements stored in the $List$ variable at each server. We ignore the storage cost due to meta-data and temporary variables. Also, for the communication cost we measure the bits sent on the wire between the nodes.

**Theorem.** *The* TREAS-OPT *algorithm has: (i) storage cost* $(\delta + 1)\frac{n}{k}$*, (ii) communication cost for each write at most to* $\frac{n}{k}$*, and (iii) communication cost for each read at most* $(\delta + 2)\frac{n}{k}$.

## VI. 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 TREAS, which uses logical timestamp to deal with loosely synchronized clock. Our implementation of both algorithms consists of around 2,000 LoC.

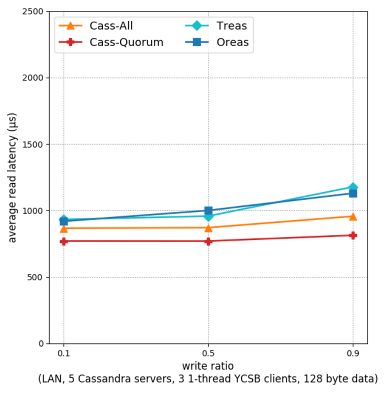
*Technical Details:* We mainly modify the following two files in the Cassandra codebase. In Cassandra's terminology, "mutation" is essentially a write operation that modifies the internal state of the servers. (i) *StorageProxy.java* is where the coordinator server handles the user's read or write operation. 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 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.‖ These details are presented in our technical report.

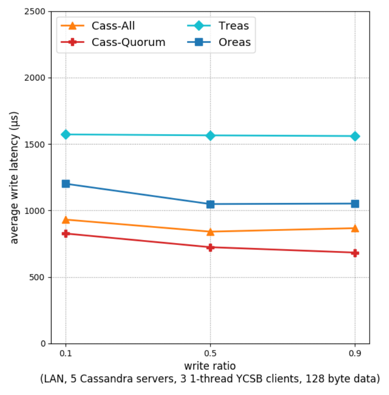
### B. Evaluation

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

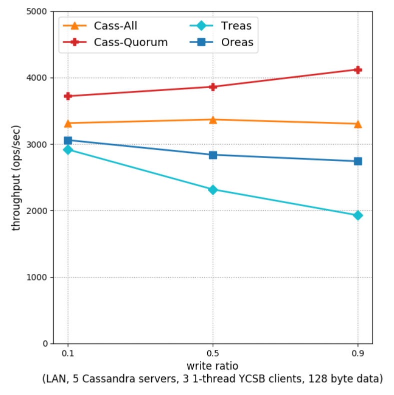
‖https://github.com/Backblaze/JavaReedSolomon



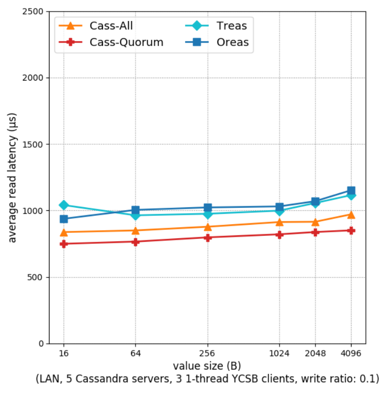
(a) average read latency vs. write ratio



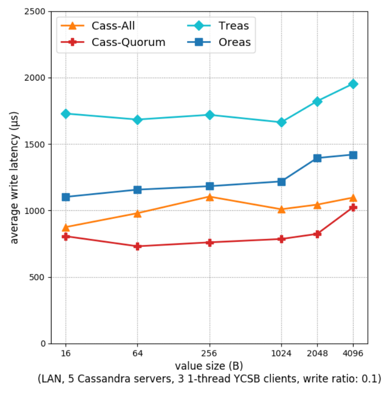
(b) average write latency vs. write ratio



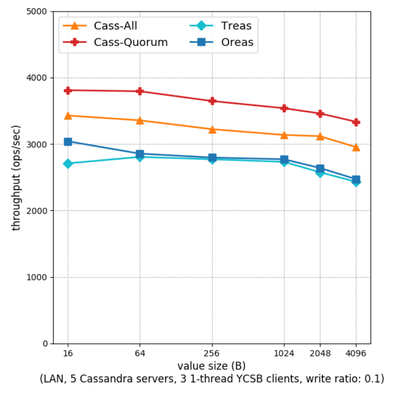
(c) throughput vs. write ratio



(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

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 (based on issuing command for 30 seconds), 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 Yahoo! Cloud Serving Benchmark (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. TREAS 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 $\delta$ 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.

## VII. DISCUSSION TOPICS

- *Future work*: The current implementation of CassandrEAS only supports KV-pairs, whereas Cassandra supports column family data model. We would like to discuss potential trade-offs and solutions to support various data model. Another interesting direction is the mechanism to support efficient geo-replication. Since each server stores a smaller amount of data, with proper design, the EC-based solution might be more efficient than the replication-based protocol (under certain workloads).

- *Importance/Applicability*: Some researchers warned us that it is *not* reasonable to sacrifice performance for storage space. However, in our views, there are certainly needs for reducing storage, as the amount of data is growing exponentially. Moreover, if we want to apply distributed storage systems on top of smaller machines (e.g., edge servers or even Internet-of-Things), then storage-efficient solutions are certainly appealing. We would love to learn new ideas on use cases and interesting trade-offs to explore.
- *Feasibility/Evaluation*: We only performed evaluation through YCSB. However, the use cases for EC-based storage systems might be very different from the normal cloud storage. It will be interesting to learn what workload is suitable for evaluating this type of systems.
- *Open theoretical problems*: Right now, CassandrEAS only supports read/write operations, but not transactions, whereas Cocytus [29] and Giza [9] supports transactions. Lightweight EC-based transaction mechanism is one interesting open problem. Another category of open problems is self-stabilization, reconfiguration, and recovery protocols discussed earlier.
- *Open engineering problems*: In our implementation of CassandrEAS, we did not introduce any optimization. The reason is that as the first step, we want to make a minimal modification to Cassandra's core functions to demonstrate feasibility. For further optimization, there are two possibilities: modifying the database engine and cache protocol so that it is more friendly to coded elements, and introducing more efficient data structure to store coded elements.
- *Limitations/Opportunity*: OREAS provides a knob to user to tune the performance and desired guarantees. Real-world systems also provide many different knobs. Unfortunately, knobs are usually difficult to tune in practice, because the trade-off is not always clear. Is it reasonable to expose the knob to users? What is the best way to help users (for selecting the most appropriate knob)?

## 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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. 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 Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Medard, and A. A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2016.

[18] Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. 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] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40, April 2010.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.