# LEGOStore: A Linearizable Geo-Distributed Store Combining Replication and Erasure Coding

Hamidreza Zare
The Pennsylvania State University
hkz5146@psu.edu

Viveck Ramesh Cadambe
The Pennsylvania State University
viveck@psu.edu

Bhuvan Urgaonkar
The Pennsylvania State University
buu1@psu.edu

Nader Alfares
The Pennsylvania State University
nna5040@psu.edu

Praneet Soni
The Pennsylvania State University
praneetsoni.soni@gmail.com

Chetan Sharma
The Pennsylvania State University
chesharm@google.com

Arif A Merchant
Google
aamerchant@google.com

## ABSTRACT

We design and implement LEGOStore, an erasure coding (EC) based linearizable data store over geo-distributed public cloud data centers (DCs). For such a data store, the confluence of the following factors opens up opportunities for EC to be latency-competitive with replication: (a) the necessity of communicating with remote DCs to tolerate entire DC failures and implement linearizability; and (b) the emergence of DCs near most large population centers. LEGOStore employs an optimization framework that, for a given object, carefully chooses among replication and EC, as well as among various DC placements to minimize overall costs. To handle workload dynamism, LEGOStore employs a novel agile reconfiguration protocol. Our evaluation using a LEGOStore prototype spanning 9 Google Cloud Platform DCs demonstrates the efficacy of our ideas. We observe cost savings ranging from moderate (5-20%) to significant (60%) over baselines representing the state of the art while meeting tail latency SLOs. Our reconfiguration protocol is able to transition key placements in 3 to 4 inter-DC RTTs (< 1s in our experiments), allowing for agile adaptation to dynamic conditions.

**PVLDB Reference Format:**
Hamidreza Zare, Viveck Ramesh Cadambe, Bhuvan Urgaonkar, Nader Alfares, Praneet Soni, Chetan Sharma, and Arif A Merchant. LEGOStore: A Linearizable Geo-Distributed Store Combining Replication and Erasure Coding. PVLDB, 15(10): 2201 - 2215, 2022.
doi:10.14778/3547305.3547323

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/shahrooz1997/LEGOstore.

## 1 INTRODUCTION

Consistent geo-distributed key-value (KV) stores are crucial building blocks of modern Internet-scale services including databases and web applications. Strong consistency (i.e., linearizability) is especially preferred by users for the ease of development and testing it offers. As a case in point, the hugely popular S3 store from Amazon Web Services (AWS), in essence a KV store with a GET/PUT interface, was recently re-designed to switch its consistency model from eventual to linearizable [72]. However, because of the inherent lower bound in [12], linearizable KV stores inevitably incur significant latency and cost overheads compared to weaker consistency models such as causal and eventual consistency. These drawbacks are particularly pronounced in the geo-distributed setting because of the high inter-data center networking costs, and large network latencies (see Table 2). To further exacerbate the problem, dynamic phenomena such as shifts in arrival rates, appearance of clients in new locations far from where data is stored, increase in network delays, etc., can lead to gaps between predicted and actual performance both in terms of costs and tail latencies.

We design LEGOStore, a geo-distributed linearizable KV store (with the familiar GET/PUT or read/write API), meant for a global user-base. LEGOStore procures its resources from a cloud provider's fleet of data centers (DCs) like many storage service providers [24, 56, 58]. Since an entire DC may become unavailable [27, 34, 58, 71], LEGOStore employs redundancy across geo-distributed DCs to operate despite such events. LEGOStore's goal is to *offer tail latency service-level objectives (SLOs) that are predictable and robust in the face of myriad sources of dynamism at a low cost.* For achieving these properties, LEGOStore is built upon the following three pillars:

**1. Erasure Coding (EC)** is a generalization of replication that is more storage-efficient than replication for a given fault tolerance. A long line of research has helped establish EC's efficacy *within a DC* or for weaker consistency needs. However, EC's efficacy in the linearizable geo-distributed setting is relatively less well-understood. Two recent works Giza [19] and Pando [67] demonstrate some aspects of EC's promise in the geo-distributed context. In particular, because EC allows fragmenting the data and storing it in a fault-tolerant manner, it leads to smaller storage costs and (more importantly) smaller inter-DC networking costs. However, the smaller-sized fragments inevitably require contacting more DCs and, therefore, are thought to imply higher latencies in a first-order estimation. By comprehensively exploring a wide gamut of workload features and SLOs, and careful design of the data placement

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547323

**Table 1: Storage and VM prices for the 9 GCP data centers that our prototype spans. We use custom VMs with 1 core and 1 GB of RAM from General-purpose machine type family to run the LEGOStore's servers and Standard provisioned space for its storage [30].**

| | GCP data center location | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Tokyo | Sydney | Singapore | Frankfurt | London | Virginia | São Paulo | Los Angeles | Oregon |
| Storage ($/GB.Month) | 0.052 | 0.054 | 0.044 | 0.048 | 0.048 | 0.044 | 0.06 | 0.048 | 0.04 |
| Virtual machine ($/hour) | 0.0261 | 0.0283 | 0.0253 | 0.0262 | 0.0262 | 0.0226 | 0.0310 | 0.0248 | 0.0215 |

**Table 2: Diverse RTTs and network pricing for 9 chosen GCP data centers. The RTTs are measured between VMs placed within the DCs; they would be higher if one of the VMs were external to GCP but not by enough to change the outcome of our optimizer. For the same general recipient location, the outbound network prices are sometimes higher if the recipient is located outside of GCP (egress pricing) but these prices exhibit a similar geographical diversity [31].**

| | | User location | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Tokyo | | Sydney | | Singapore | | Frankfurt | | London | | Virginia | | São Paulo | | Los Angeles | | Oregon | |
| | | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) | n/w price ($/GB) | Latency (ms) |
| GCP data center | Tokyo | - | 2 | 0.15 | 115 | 0.12 | 70 | 0.12 | 226 | 0.12 | 218 | 0.12 | 148 | 0.12 | 253 | 0.12 | 100 | 0.12 | 90 |
| | Sydney | 0.15 | 115 | - | 2 | 0.15 | 94 | 0.15 | 289 | 0.15 | 277 | 0.15 | 204 | 0.15 | 291 | 0.15 | 139 | 0.15 | 162 |
| | Singapore | 0.09 | 72 | 0.15 | 94 | - | 2 | 0.09 | 202 | 0.09 | 203 | 0.09 | 214 | 0.09 | 319 | 0.09 | 165 | 0.09 | 166 |
| | Frankfurt | 0.08 | 229 | 0.15 | 289 | 0.08 | 201 | - | 2 | 0.08 | 15 | 0.08 | 89 | 0.08 | 202 | 0.08 | 153 | 0.08 | 139 |
| | London | 0.08 | 222 | 0.15 | 280 | 0.08 | 204 | 0.08 | 15 | - | 2 | 0.08 | 79 | 0.08 | 192 | 0.08 | 141 | 0.08 | 131 |
| | Virginia | 0.08 | 146 | 0.15 | 204 | 0.08 | 214 | 0.08 | 90 | 0.08 | 79 | - | 2 | 0.08 | 116 | 0.08 | 68 | 0.08 | 58 |
| | São Paulo | 0.08 | 252 | 0.15 | 292 | 0.08 | 317 | 0.08 | 202 | 0.08 | 192 | 0.08 | 117 | - | 1 | 0.08 | 155 | 0.08 | 172 |
| | Los Angeles | 0.08 | 101 | 0.15 | 139 | 0.08 | 180 | 0.08 | 153 | 0.08 | 142 | 0.08 | 67 | 0.08 | 155 | - | 2 | 0.08 | 26 |
| | Oregon | 0.08 | 95 | 0.15 | 164 | 0.08 | 165 | 0.08 | 142 | 0.08 | 131 | 0.08 | 58 | 0.08 | 173 | 0.08 | 26 | - | 2 |

and EC parameters through an optimization framework, LEGOStore brings out the full potential of EC in the geo-distributed setting.

**2.** LEGOStore adapts **non blocking, leaderless, quorum based linearizability protocols** for both EC [16] and replication [9, 10]. When used with carefully optimized quorums, these protocols help realize LEGOStore's goal of predictable performance (i.e., meeting latency SLOs with a very high likelihood as long as workload features match their predicted values considered by our optimization). A second important reason is that, when used in conjunction with a well-designed resource autoscaling strategy, the latency resulting from non-blocking protocols depends primarily on its inter-DC latency and data transfer time components. This is in contrast with the leader-based consensus used in Giza and Pando (see Figure 4 ). While these works may be able to offer lower latencies for certain workloads, they are susceptible to severe performance degradation under concurrency-induced contention.

**3.** To offer robust SLOs in the face of dynamism, LEGOStore implements an **agile reconfiguration mechanism**. LEGOStore continually weighs the pros and cons of changing the *configuration*[1] of a key or a group of keys via a cost-benefit analysis rooted in its cost/performance modeling. If prompted by this analysis, it uses a novel reconfiguration protocol to safely switch the configurations of concerned keys without breaking linearizability. We have designed this reconfiguration protocol carefully and specifically to work alongside our GET/PUT protocols to keep its execution time small which, in turn, allows us to limit the performance degradation experienced by user requests issued during a reconfiguration.

[1]By the configuration of a key, we mean the following aspects of its placement: (i) whether EC or replication is being used ; (ii) the EC parameters or replication degree being used; and (iii) the DCs that comprise various quorums.

**Contributions:** We design LEGOStore, a cost-effective KV store with predictable tail latency that adapts to dynamism. We develop an optimization framework that, for a group of keys with similar suitable workload features, takes as input these features, and public cloud characteristics and determines configurations that satisfy SLOs at minimal cost. The configuration choice involves selecting one from a family of linearizability protocols, and the protocol parameters that can include the degree of redundancy, DC placement, quorum sizes, and EC parameters. In this manner it selectively uses EC for its better storage/networking costs when allowed by latency goals, and uses replication otherwise. A key's storage method may change over time based on changes in workload features. We develop a safe reconfiguration protocol, and an accompanying heuristic cost/benefit analysis that allows LEGOStore to control costs by dynamically adapting configurations.

We build a prototype LEGOStore system. We carry out extensive evaluations using our optimizer and prototype spanning 9 Google Cloud Platform (GCP) DCs. We get insights from our evaluation to make suitable modeling and protocol design choices. Because of our design, our prototype has close match with the performance predicted by the optimizer. Potential cost savings over baselines based on state-of-the-art works such as SPANStore [75] and Pando [67] range from moderate (5-20%) to significant (up to 60%). The most significant cost savings emerge from carefully avoiding the use of DCs with high outbound network prices. Our work offers a number of general trends and insights relating workload and infrastructure properties to cost-effective realization of linearizability. Some of our findings are perhaps non-intuitive: (i) smaller EC-fragments do not always lead to lower costs (Section 4.2.4); (ii) there exists a significant asymmetry between GETs and PUTs in terms of costs, and read-heavy workloads lead to different choices than write-heavy

workloads (Section 4.2.3); (iii) we find scenarios where the optimizer is able to exploit the lower costs of EC without a latency penalty (Section 4.2.5); and (iv) even when a majority of the requests to a key arise at a particular location, the DC near that location may not necessarily be used for this key in our optimizer's solution.

## 2 BACKGROUND

**Public Cloud Latencies and Pricing:** The lower bound of [11, 12] implies that *both* GET and PUT operations in LEGOStore necessarily involve inter-DC latencies and data transfers unlike with weaker consistency models. The latencies between users and various DCs of a public cloud provider span a large range. In Table 2 we depict our measurements of round-trip times (RTTs) between pairs of DCs out of a set of 9 Google Cloud Platform (GCP) data centers we use. The smallest RTTs are 15-20 msec while the largest exceed 300 msec; RTTs between nodes within the same DC are 1-2 msec and pale in comparison. Similarly, the prices for storage, computational, and network resources across DCs also exhibit geographical diversity as shown in Tables 1 and 2. This diversity is the most prominent for data transfers—the cheapest per-byte transfer is $0.08/GB (e.g., London to Tokyo), the costliest is $0.15/GB (e.g., Sydney to Tokyo). LEGOStore's design must carefully navigate these sources of diversity to meet latency SLOs at minimum cost.

**Our Choice of Consistent Storage Algorithms:** Due to the lower bound mentioned above, in leader-based protocols (e.g., Raft [53]), for the geo-distributed scenario of interest to us, one round trip time to the leader is inevitable. Using such leader-based protocols can drive up latency when the workload is distributed over a wide geographical area, and there is no leader node that is sufficiently close to all DCs so as to satisfy the SLO requirements. Such a design can also place excessive load on DCs that are more centrally located. Therefore, as such, we choose algorithms with leaderless, quorum-based protocols in our design and implementation. We describe these protocols (ABD for replication and CAS for EC) next.

**The ABD Algorithm:**[2] Let $N$ denote the degree of replication (specifically, spanning $N$ separate DCs) being used for the key under discussion. In the ABD algorithm, for a given key, each of the nodes (i.e., DCs) stores a (tag, value) pair, where the tag is a (logical timestamp, client ID) pair. The node replaces this tuple when it receives a value with a higher tag from a client operation. The PUT operation consists of two phases. The first phase, which involves a logical-time query, requires responses from a quorum of $q_1$ nodes, and the second phase, which involves sending the new (tag, value) pair requires acknowledgements from a quorum of $q_2$ nodes. The GET operation also consists of two phases. The first phase again involves logical-time queries from a quorum of $q_1$ nodes and determining the highest of these tags. The second "write-back" phase sends the (tag, value) pair chosen from the first-phase responses to a quorum of $q_2$ nodes. If $q_1 + q_2 > N$, then ABD guarantees linearizability. If $q_1, q_2 \leq N - f$, then operations terminate so long as the number of node failures is at most $f$. Note that this is a stronger liveness guarantee as compared to Paxos; ABD circumvents FLP impossibility [26] because it implements a data type (read/write memory) that is weaker than consensus (see also [52] and Theorems 17.5, 17.9 in [47]). This liveness property translates into excellent robustness of operation latency (see Section 4.3). See formal description of ABD in Appendix A of [82].

[2] The name "ABD" comes from the authors, Attya, Bar-Noy and Dolev [9, 10].

**Table 3: Coarse cost comparison of replication (ABD) versus erasure coding (CAS). Costs reported are per GET/PUT operation, and the per-server storage cost. We assume that each value has $B$ bits and the metadata size is negligible. All quorum sizes are assumed to be $(N+k)/2$ for CAS and $(N+1)/2$ for ABD; $N, k$ are assumed to be odd to ignore integer rounding. Latency is reported as the number of round trips of the protocol.**

| | PUT cost | PUT latency | GET cost | GET latency | Storage cost |
|---|---|---|---|---|---|
| CAS[a] | $\frac{NB}{k}$ | 3 rounds | $\frac{(N-K)B}{2K}$ | 2 rounds | $\delta \frac{B}{k}$ |
| ABD[b] | $NB$ | 2 rounds | $(N-1)B$ | 2 rounds | 1 |

[a] With efficient garbage collection, $\delta$ can be kept small; it is equal to 1 for keys with sufficiently low arrival rates.
[b] ABD has a higher communication cost vs. CAS for GETs, even if $k = 1$, since it propagates values in the write-back phase, whereas CAS only propagates metadata.

Whereas the above vanilla ABD requires two phases for all its GET operations, a slight enhancement allows some (potentially many) GET operations to complete in only one phase; we refer to such a scenario as an "Optimized GET," see details in [82].

**Erasure Coding:** Erasure coding (EC) is a generalization of replication that is attractive for modern storage systems because of its potential cost savings over replication. An $(N, K)$ Reed Solomon Code stores a value over $N$ nodes, with each node storing a codeword symbol of size $1/K$ of the original value, unlike replication where each node stores the entire value. The value can be decoded from *any* $K$ of the $N$ nodes, so the code tolerates up to $f = N - K$ failures. On the other hand, replication duplicates the data $N = f+1$ times to tolerate $f$ failures. For a fixed value of $N$, EC leads to a $K$-fold reduction in storage cost compared to $N$-way replication for the same fault-tolerance. It also leads to a $K$-fold reduction in communication cost for PUTs, which can be significant because of the inter-DC network transfer pricing. While this suggests that costs reduce with increasing $K$, we will see that the actual dependence of costs on $K$ in LEGOStore is far more complex (see Section 4.2.4).

In EC-based protocols, GET operations require $K > 1$ nodes to respond with codeword symbols for the *same* version of the key. However, due to asynchrony, different nodes may store different versions at a given time. Reconciling the different versions incurs additional communication overheads for EC-based algorithms.

**The CAS Algorithm:** We use the *coded atomic storage* (CAS) algorithm[3] of [16, 17], described in Appendix B of [82]. In CAS, servers store a list of triples, each consisting of a tag, a codeword symbol, and a label that can be 'pre' or 'fin'. The GET protocol operates in two phases like ABD; however, PUT operates in *three* phases. Similar to ABD, the first phase of PUT acquires the latest tag. The second phase sends an encoded value to servers, and servers store this symbol with a 'pre' label. The third phase propagates the 'fin' label to servers, and servers which receive it update the label for that tag. The three phases of PUT require quorums of $q_1, q_2, q_3$, resp., responses to complete. Servers respond to queries from GETs/PUTs only with latest tag labeled 'fin' in their lists. A GET operates in two phases, the first phase to acquire the highest tag labeled 'fin' and the second to acquire the chunks for that tag, decode the value and do a write-back. The two phases of GET require responses

[3] The algorithm in Appendix B of [82] is a modification the algorithm in [16, 17] to allow for flexible quorum sizes, which in turn this exposes more cost-saving opportunities.

from quorums of size $q_1, q_4$, resp. In the write-back phase, CAS only sends a 'fin' label with the tag, unlike ABD which sends the entire value. In fact, the structural differences between ABD and CAS protocols translates to lower communication costs for CAS even if $k = 1$ (i.e., replication) is used as compared to ABD, with the penalty of incurring higher PUT latency due to the additional phase (See Table 3). This variation between ABD and CAS offers LEGOStore further flexibility in adapting to workloads as demonstrated in Section 4.2. Similar to ABD, we also employ an "Optimized GET" for CAS that enables some (potentially many) GET operations to complete in only one phase. This optimized GET is based on a client-side cache (recall a client is different from a user, cf. Section 3) for the value computed in second phase of GET. LEGOStore exploits these differences to reduce costs based on if the workload is read- or write-intensive. On the server-side protocol, unlike ABD where a server simply replaces a value with a higher tagged value, CAS requires servers to store a history of the codeword symbols corresponding to multiple versions, and then garbage collect (GC) older versions at a later point. However, in practice, the overhead is negligible for the workloads we study (see Appendix F in [82], also remarks in Table 3.). The preliminary cost comparison of ABD and CAS in Table 3 ignores several important aspects of practical key-value stores, in particular the spatial diversity of pricing and latency, flexibility of choosing quorum sizes and locations, and the impact of arrival rates. Our paper refines the insights of Table 3 in the context of LEGOStore (See Section 3.2 for details).

# 3 LEGOSTORE SYSTEM DESIGN

## 3.1 Interface and Components

LEGOStore is a linearizable key-value store spanning a set $\mathcal{D}$ of $D$ DCs of a public cloud provider.Applications using LEGOStore ("users") link the LEGOStore library that offers them an API comprising the following linearizable operations:

- CREATE(k,v): creates the key `k` using default configuration `c` (we will define a configuration momentarily) if it doesn't already exist and stores `(k,c)` in the local meta-data server (MDS); returns an error if the key already exists.[4]
- GET(k): returns value for `k` if `k` exists; else returns an error.
- PUT(k,v): sets value of `k` to `v`; returns error if `k` doesn't exist.
- DELETE(k): removes `k`; returns error if `k` doesn't exist.

To service these operations, the library issues RPCs to a LEGOStore "client" within a DC in $\mathcal{D}$. A LEGOStore client implements the client-end of LEGOStore's consistency protocols. A user resident within a DC in $\mathcal{D}$ would be assisted by a client within the same DC. For users outside of $\mathcal{D}$, a natural choice would be a client in the nearest DC. The client assisting a user may change over time (e.g., due to user movement) but only across operations. Since the user-client delay is negligible compared to other RTTs involved in request servicing (recall Table 2), we will ignore it in our modeling.

In order to service a `GET` or a `PUT` request for a key `k`, a client first determines the "configuration" for `k` which consists of the following elements: (i) replication or erasure coding to be used (and, correspondingly, ABD or CAS); (ii) the DCs that comprise relevant quorums; and (iii) the identities of the LEGOStore "proxies" within each of these DCs. Having obtained the configuration, a client issues protocol-specific Remote Procedure Calls (RPCs) to proxies in relevant quorums to service the user request. Each DC's proxy serves as the intermediary between the client and the compute/storage servers that (a) implement the server-end of our consistency protocols and (b) store actual data (replicas for ABD, EC chunks for CAS) along with appropriate tags.

[4]A default configuration uses the nearest DCs for various quorums in terms of their RTTs from the client.

## 3.2 Finding Cost-Effective Configurations

We develop an optimization that determines cost-effective configurations assuming perfect knowledge of workload and system properties. Since our protocols operate at a per-key granularity due to the composability of linearizability [36]—notice how the ABD and CAS algorithms in Appendix A and B of [82] are described for a generic key—we can decompose our datastore-wide optimization into smaller optimization problems, one per key.[5]

**Inputs (See Table 4):** We assume that LEGOStore spans $D$ geo-distributed DCs numbered 1, ..., $D$. We assume that the following are available at a per-key granularity: (i) overall request arrival rate; (ii) geographical distribution of requests (specifically, fractions of the overall arrival rate emerging in/near each DC); (iii) fraction of requests that are GET operations; (iv) average object size and meta-data[6] size; (v) GET and PUT latency SLOs. We assume that SLOs are in terms of $99^{th}$ percentile latencies. We assume that the availability requirement is expressed via the single parameter $f >$ 0; LEGOStore must continue servicing requests despite up to $f$ DC failures. The system properties considered in our formulation are: (i) inter-DC latencies and prices for network traffic between clients and servers; (ii) storage; (iii) computational resources in the form of virtual machines (VMs).

**Decision Variables:** Our decision variables, as described at the bottom of Table 4, help capture all aspects of a valid configuration. These include: (i) boolean variable $e_g$ whether this key would be served using ABD, and (ii) variables $\mathbf{iq}_g^k$ which DCs constitute various quorums that the chosen algorithm (2 and 4 quorums, resp., for ABD and CAS) requires (see variable $iq_g^k$ in Table 4).

**Optimization:** Our optimization tries to minimize the cost of operating key $g \in G$ in the next *epoch* – a period of relative stability in workload features. Our objective for key $g \in G$, which is cost per unit time during the epoch being considered, is expressed as:

$$\begin{aligned} \text{minimize } & \left(C_{g,get} + C_{g,put} + C_{g,Storage} + C_{g,VM}\right) \\ & \text{s.t. (11)} - \text{(29) in Appendix C of [82].} \end{aligned} \tag{1}$$

The first two components of the objective with *put* and *get* in their subscripts denote the networking costs per unit time of PUT and GET operations, resp., for key $g$ while the last two denote costs per unit time spent towards storage and computation, resp. The details of the optimization are in Appendix C of [82], we reproduce some representative constraints and equations here.

[5]Although we design our optimizer for individual keys, aggregating keys with similar workload features and considering such a "group" of keys in the optimizer may be useful (perhaps even necessary) for LEGOStore to scale to large number of keys.

[6]Meta-data transferred over the network can have non-negligible cost/latency implications and that is what we explicitly capture. On the other hand, the storage of meta-data contributes relatively negligibly to costs and we do not consider those costs.

Table 4: Input and decision variables used by LEGOStore's optimization.

| Input | Interpretation | Type |
|---|---|---|
| $D$ | Number of data centers | integer |
| $\mathcal{D}$ | Set of data centers numbered 1, ..., $D$ | set |
| $l_{ij}$ | Latency from DC $i$ to DC $j$ (RTT/2) | real |
| $B_{ij}$ | Bandwidth between DC $i$ and DC $j$ | real |
| $\mathcal{G}$ | Set of keys | set |
| $\lambda_g$ | Aggregate request arrival rate for key $g \in \mathcal{G}$ | integer |
| $\rho_g$ | Read-write ratio for $g$ | real [0,1] |
| $\alpha_{ig}$ | Fraction of requests originating at/near DC $i$ for key $g$ | real |
| $o_g$ | Average object size, *including* protocol-specific metadata exchanged between a client and a server | integer |
| $o_m$ | Average protocol-specific metadata exchanged between a client and a server for each phase | integer |
| $l_{get}$ | GET latency SLO | real |
| $l_{put}$ | PUT latency SLO | real |
| $f$ | Availability requirement (i.e., number of failed DCs to tolerate) | integer |
| $p_i^s$ | Storage price (per byte per unit time) for DC $i \in \mathcal{D}$ | real |
| $p_{ij}^n$ | Network price per byte from location $i$ to location $j$ | real |
| $p_i^v$ | VM price at DC $i$ (simplifying assumption: all VMs of a single size) | real |
| $\theta^v$ | This quantity multiplied by the request arrival rate at DC $i$ captures the VM capacity required at $i$ | real |
| **Var.** | **Interpretation** | **Type** |
| $e_g$ | Protocol (0 for ABD, 1 for CAS) for key $g$ | boolean |
| $m_g$ | Length of code (replication factor for ABD) | integer |
| $k_g$ | Dimension of code (equals 1 for replication) | integer |
| $q_{i,g}$ | Quorum size for $i^{\text{th}}$ quorum of key $g$ | integer |
| $v_i$ | Capacity of VMs at DC $i$ | real |
| $\mathbf{iq}_g^k$ | Indicator for data placement for $k^{\text{th}}$ quorum of key $g$. $iq_{ijg}^k = 1$ iff DC $j$ in $k^{\text{th}}$ quorum of clients in/near DC $i$ | boolean |

The networking cost per unit time of PUTs for key $g$ must be represented differently based on whether ABD or CAS is used: $C_{g,put} = \underbrace{e_g \cdot C_{g,put,CAS}}_{\text{n/w cost if CAS chosen}} + \underbrace{(1-e_g) \cdot C_{g,put,ABD}}_{\text{n/w cost if ABD chosen}}$, where,

$$C_{g,put,ABD} = (1-\rho_g) \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \Big( \underbrace{o_m \sum_{j=1}^{D} p_{ji}^n \cdot iq_{ijg}^1}_{\text{n/w cost for phase 1}} + \underbrace{o_g \sum_{k=1}^{D} p_{ik}^n \cdot iq_{ikg}^2}_{\text{n/w cost for phase 2}} \Big).$$

$$C_{g,put,CAS} = (1-\rho_g) \cdot \lambda_g \sum_{i=1}^{D} \alpha_{ig} \Big( o_m \Big( \underbrace{\sum_{j=1}^{D} p_{ji}^n \cdot iq_{ijg}^1}_{\text{phase 1}} + \underbrace{\sum_{k=1}^{D} p_{ik}^n \cdot iq_{ikg}^3}_{phase3} \Big) + \underbrace{\frac{o_g}{k_g} \sum_{m=1}^{D} p_{im}^n \cdot iq_{img}^2}_{phase2} \Big). \quad (2)$$

Note the role played by the key boolean decision variable $iq_{ijg}^k$ whose interpretation is: $iq_{ijg}^k$=1 iff data center $j$ is in the $k^{th}$ quorum for clients in/near data center $i$. In the above expressions, $(1-\rho_g) \cdot \lambda_g \cdot \alpha_{ig}$ captures the PUT request rate arising at/near data center $i$ and the $o_m$ and $o_g$ multipliers convert this into bytes per unit time. The terms within the braces model the per-byte network transfer prices. The first term represents network transfer prices that apply to the first phase of the ABD PUT protocol whereas the second term does the same for ABD PUT's second phase. The term $p_{ji}^n \cdot iq_{ijg}^1$ should be understood as follows: since ABD's first phase involves clients in/near data center $i$ sending relatively small-sized *write-query* messages to all servers in their quorum (i.e., quorum 1, hence the 1 in the superscript of $iq$) followed by these servers responding with their (tag, value) pairs, the subscript in $p_{ji}^n$ is selected to denote the price of data transfer from $j$ (for the server at data center $j$) to $i$ (for clients located in/near data center $i$).The network cost per unit time for CAS is similar, with the number of phases being 3 and the network cost savings offered by CAS reflected in phase 3, where the value size $o_g$ is divided by the parameter $k_g$. The networking costs for GET are presented in Appendix C in [82].

The storage cost is modeled as: $C_{g,storage} = p^s \cdot \big(e_g \cdot m_g \cdot \frac{o_g}{k_g} + (1-e_g) \cdot m_g \cdot o_g\big)$, see [82] for explanations. Finally, we consider the VM costs per unit time for key $g$. Our assumptions on modeling VM costs include: ability of procurement of VMs at fine granularity (see, e.g., [29]) and VM autoscaling [13, 32] to ensure satisfactory provisioning of VM capacity at each DC. We assume that this suitable VM capacity chosen by such an autoscaling policy is proportional to the total request arrival rate at data center $i$ for key $g$. With these assumptions, the VM cost for key $g$ at data center $i$ is: $C_{g,VM} = \theta^v \cdot \sum_{j=1}^{D} p_j^v \cdot \lambda_g + \sum_{i=1}^{D} \alpha_{ig} + \sum_{k=1}^{4} iq_{ijg}^k$, where $\theta^v$ is an empirically determined multiplier that estimates VM capacity needed to serve the the request rate arriving at data center $j$ for $g$.

**Constraints:** Our optimization needs to capture the 3 types of constraints related to: (i) ensuring linearizability; (ii) meeting availability guarantees corresponding to the parameter $f$; and (iii) meeting latency SLOs. The key modeling choices we make are: (i) to use worst-case latency as a "proxy" for tail latency; and (ii) ignore latency contributors within a data center other than data transfer time (e.g., queuing delays, encoding/ decoding time). For PUT operations in CAS, the constraints are $\forall i, j, k \in \mathcal{D}$,:

$$\underbrace{iq_{ijg}^1 \cdot \Big(l_{ij} + l_{ji} + \frac{o_m}{B_{ji}}\Big)}_{\text{Latency of first phase of PUT}} + \underbrace{iq_{img}^2 \cdot \Big(l_{im} + \frac{o_g/k_g}{B_{im}} + l_{mi}\Big)}_{\text{Latency of second phase of PUT}} + \underbrace{iq_{ikg}^3 \cdot \Big(l_{ik} + \frac{o_m}{B_{ik}} + l_{ki}\Big)}_{\text{Latency of third phase of PUT}} \leq l_{put}. \quad (3)$$

See Appendix C in [82] for explanations, and for constraints for PUT operations of ABD, and for GETs. Linearizability and availability targets manifest as constraints on quorum sizes.

### 3.3 How to Reconfigure?

LEGOStore uses a reconfiguration protocol that transitions chosen keys from their old configurations to their new configurations without violating linearizability. Unlike our approach, consensus-based protocols such as Raft and Viewstamped Replication [44, 55],

implement the key-value store as a log of commands applied sequentially to a replicated state machine. These solutions implement reconfiguration by adding it as a special command to this log. Thus, when a reconfiguration request is issued, the commands that are issued before the reconfiguration request are first applied to the state machine before executing the reconfiguration. To execute the reconfiguration, the leader transfers the state to the new configuration. After the transfer, it resumes handling of client commands that are serialized after the reconfiguration request, but replicating the state machine in the new configuration. While our approach does not involve a replicated log[7], it is possible to develop an approach that inherits the essential idea of consensus-based reconfiguration as follows: (i) On receiving a reconfiguration request, wait for all ongoing operations to complete, and pause all new operations; (ii) Perform the reconfiguration by transferring state from the old to the new configurations; (iii) Resume all operations.

Under the reasonable assumption that reconfigurations *of a given key are performed relatively infrequently*,[8] our design goal is to ensure that user performance is not degraded in the common case where the configurations remain static. For this, it is crucial that user operations that are not concurrent with reconfigurations follow the baseline static protocols without requiring additional steps/phases (such as contacting a controller). Our protocol does not assume any special relation between the old and new configurations. It can handle all types transitions, including changing of the replication factor, EC parameters, quorum structure, and the protocol itself.

We wish to keep the number of communication phases as well as the number of operations affected as small as possible. Towards this, LEGOStore's reconfiguration protocol improves upon steps (i)-(iii) above. Reconfigurations are conducted by a controller that reads data from the old configuration and transfers it to the new configuration. On detecting a workload or system change (See Section 3.4 for details), the controller immediately performs the reconfiguration without having to wait for all ongoing operations to complete, i.e., without having to perform step (i). This enables LEGOStore to adapt quickly to workload changes. Furthermore, steps (ii),(iii) are conducted jointly through a single round of messaging. In particular, LEGOStore's protocol integrates with the underlying protocols of CAS and ABD and piggybacks the reconfiguration requests from the controller along with the actions that read or transfer the data. LEGOStore's algorithm is provably linearizable (Appendix D of [82]), and therefore, preserves the correctness of the overall data store. The algorithm blocks certain concurrent operations and then resumes them on completing the reconfiguration.

The reconfiguration protocol is described formally in Appendix D of [82]. We assume that reconfigurations are applied sequentially by the reconfiguration controller (or simply controller). The controller sends a `reconfig_query` message to all the servers in the old configuration. The `reconfig_query` message serves to both signal a change in configuration, as well as an internal 'get' request for the controller to read a consistent value in order to transfer it to the new configuration. On receiving this message, the servers pause all the ongoing operations and respond with the latest value if the old configuration is performing ABD, or the highest tag labeled 'fin' if it is performing CAS. The controller waits for a quorum to respond from the old configuration. If the old configuration is performing CAS, then the reconfiguration controller sends a `reconfiguration_get` message to servers in the old configuration with the highest tag among the messages received from the quorum (the quorum size is $N - q_2 + 1$ if the old configuration is performing ABD and $N - \min(q_3, q_4) + 1$ if it is performing CAS). A server that receives the `reconfiguration_get` message with tag $t$ sends a codeword symbol corresponding to that tag, if it is available locally; else it responds with an acknowledgement. The reconfiguration controller then obtains responses from a quorum of $q_4$ messages and decodes the value from the responses.The controller then proceeds to write the $(tag, value)$ pair to the new configuration, performing encoding if the new configuration involves CAS. On completing writing the value to the new configuration, the controller sends a `finish_reconfig` message to servers in the old configuration. On receiving these messages, the servers complete all operations with tag less than or equal to $t_{highest}$ and send `operation_fail` messages along with information of new configuration to other pending operations that were paused. On receiving `operation_fail` messages, the clients restart the operation in the new configuration.

## 3.4 When and What to Reconfigure?

At the heart of our strategy are these questions: (i) is a key configured poorly for its current or upcoming workload? (ii) if yes, should it be stored using a different configuration? While our discussion focuses on workload dynamism, our ideas also apply to changes in system/infrastructure properties.

**Is a Key Configured Sub-Optimally?** Some workload changes can be predicted (e.g., cyclical temporal patterns or domain-specific insights from users) while others can only be determined after they have occurred. Generally speaking, a system such as LEGOStore would employ a combination of predictive and reactive approaches for detecting workload changes [13, 63, 69]. In this paper, we pursue a purely reactive approach. We employ two types of reactive rules that are indicative of a key being configured poorly:

• *SLO violations:* If SLO violations for a key are observed for more than a threshold duration or during a window containing a threshold number of requests, LEGOStore chooses to reconfigure it. Section 4.4, 4.5 show that LEGOStore's reconfiguration occurs within 1 sec; the threshold should be set sufficiently larger than this to avoid harmful oscillatory behavior over-optimizing for transient phenomena. If, in addition to SLO violations, some quorum members are suspected of being slow or having failed, these nodes are removed from consideration when determining the next configuration.

• *Cost sub-optimality:* Alternatively, a key's configuration might meet the SLO but the estimated running cost might exceed the expected cost. We consider such sub-optimality to have occurred if the observed cost exceeds modeled cost by more than a threshold percentage as assessed over a window of a certain duration.[9] Having determined the need for a change, LEGOStore reconfigures the key based on our *cost-benefit analysis* described below.

[7] Rather than a replicated log, we simply have a replicated single read/write variable per key, which is updated on receiving new values.

[8] More precisely, we assume that reconfigurations to a key are separated in time by periods that are much longer (several minutes to hours or even longer) than the time it takes to reconfigure (sub-second to a second, see measurements in Section 4.4).

[9] Detailed exploration of the impact of the thresholds on performance is future work.

**Should Such a Key be Reconfigured?** For the case of SLO violations, LEGOStore will reconfigure as we consider SLO maintenance to be sacrosanct. In the rest of the discussion, we focus on the case of cost sub-optimality. We assume that such a key's workload features can be predicted for the near term. In the absence of such predictability—and this applies more generally to any similar system—LEGOStore's options are limited to using a state-of-the-art latency-oriented optimization (see our baselines ABD Nearest and CAS Nearest in Section 4.1) that is likely to be able to meet the SLO; the impact of such a heuristic on cost is examined in Section 4.2.

Assuming predictability, LEGOStore computes the new configuration with the updated workload characteristics using the optimization framework from Section 3.2. For an illustrative instance of such decision-making, denote this newly computed configuration as $c_{new}$ and the existing configuration as $c_{exist}$. Let us denote by $\texttt{Cost}(c)$ the per time unit cost incurred when using configuration $c$. We assume an additional predicted feature $T_{new}$, the minimum duration for which the predicted workload properties will endure before changing. LEGOStore compares the cost involved in reconfiguring with potential cost savings arising due to it. Reconfiguring a key entails (a) *explicit costs* arising from the addtional data transfer; and (b) *implicit costs* resulting from requests that are slowed down or rejected. An evaluation (see Section 4.4 for details) of our reconfiguration protocol suggests that the number of operations experiencing slowdown is small. Therefore, we consider only (a).

LEGOStore's cost-benefit analysis is simple. A reconfiguration to $c_{new}$ is carried out if the potential (minimum) cost savings $T_{new} \cdot (\texttt{Cost}(c_{exist}) - \texttt{Cost}(c_{new}))$ significantly outweigh the explicit cost of reconfiguration as captured by $\texttt{ReCost}(c_{old}, c_{exist}) \cdot (1 + \alpha)$. $\texttt{ReCost}(.,.)$ is the cost of network transfer induced by our reconfiguration and its calculation involves ideas similar to those presented in our optimizer (see Appendix C of [82]). The $(1 + \alpha)$ multiplier ($\alpha > 0$) serves to capture how aggressive or conservative LEGOStore wishes to be. The efficacy of our heuristics depends on the predictability in workload features and the parameter $T_{new}$.

Algorithms 1, 2 of [82] show that the time to complete a reconfiguration (call it $T_{re}$) is largely dictated by the RTTs between the controller and the servers farthest from it in the quorums involved in various phases of the reconfiguration protocol. Figure 10 of [82] highlights this period when both the old and the new configurations use ABD.

## 4 EVALUATION

We implement a prototype LEGOStore on the Google Compute Engine (GCE) public cloud and make the code for our prototype as well as the optimizer available under the Apache license 2.0 at github.com/shahrooz1997/LEGOstore. Details of our prototype implementation may be found in Appendix H of [82]. We evaluate LEGOStore in terms of its ability to (i) lower costs compared to the state-of-the-art; and (ii) meet latency SLOs. We use Porcupine [8] for verifying linearizability of execution histories of our prototype. To reproduce the artifacts, please refer to the LEGOStore repository.

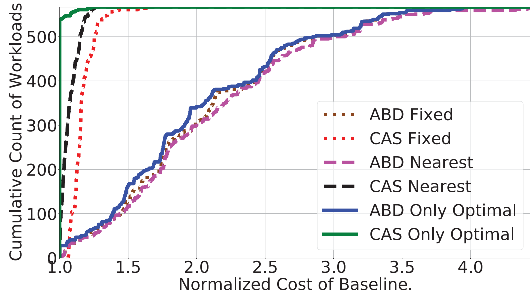
### 4.1 Experimental Setup

**Prototype Setup:** We deploy our LEGOStore prototype across 9 Google Cloud Platform (GCP) DCs with locations, pairwise RTTs, and resource pricing shown in Tables 1 and 2. We locate our users within these data centers as well for the experiments. We conduct extensive validation of the efficacy of the latency and cost modeling underlying our optimizer (Appendix G.1 of [82]).
**Workloads:** We employ a custom-built workload[10] generator which emulates a user application with an assumption that it sends requests as per a Poisson process. We explore a large workload space by systematically varying our workload parameters as follows.

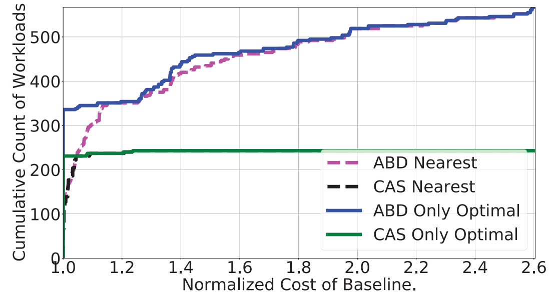
- 3 per-key sizes in KB: (i) 1, (ii) 10, and (iii) 100;
- 3 per-key read ratios for high-read (HR), read-write (RW), and high-write (HW) workloads, resp.: (i) 30:1, (ii) 1:1, and (iii) 1:30;
- 3 per-key arrival rates in requests/sec: (i) 50, (ii) 200, and (iii) 500;
- 3 sizes for the overall data: (i) 100 GB, (ii) 1 TB, and (iii) 10 TB.
- 7 different client distributions: (i) Oregon, (ii) Los Angeles, (iii) Tokyo, (iv) Sydney, (v) Los Angeles and Oregon, (vi) Sydney and Singapore, and (vii) Sydney and Tokyo.

This gives us a total of 567 diverse *basic* workloads for a given availability target and latency SLO. Finally, we also vary the availability target ($f$=1 in this section and $f$=2 in Appendix G of [82]) and latency SLO (in the range 200 ms—1 sec) in our experiments. Additionally, we use the following customized workloads to explore particular performance related phenomena: (i) a uniform client distribution across all 9 locations; (ii) workloads related to Figures 3-5; we describe these in the text accompanying these figures. Finally, while our exact metadata size varies slightly between ABD and CAS, we round it up to an overestimated 100 B.
**Baselines:** We would like to compare LEGOStore's efficacy to the most important state-of-the-art approaches. To enable such comparison, we construct the following baselines:
• **ABD Fixed and CAS Fixed:** These baselines use only ABD or only CAS, respectively. The baseline employs either a fixed degree of replication or a fixed set of CAS parameters. These parameter values (3 for ABD and $(5, 3)$ for CAS) are the ones chosen most frequently by our optimizer across our large set of experiments described in Section 4.2. For these fixed parameters, these baselines pick the DCs with the smallest average network prices for their quorums, where the average for a DC $i$ is calculated over the price of transferring data to all user locations. A comparison with these baselines demonstrates that merely knowing the right parameters does not suffice—one must pick the actual DCs judiciously.
• **ABD Nearest and CAS Nearest:** These baselines also use only ABD or only CAS. However, they do not a priori fix the degree of replication or the EC parameters. Instead, we pick the optimized value for each parameter and choose quorums that result in the smallest latencies for the GET/PUT operations ignoring cost concerns. They solve a variant of our optimizer where the objective is latency minimization, expressions involving costs are not considered, and all other constraints are the same. These baselines serve as representatives of existing works (e.g., Volley [4] and [62]) that primarily focus on latency reduction. While the baselines are admittedly not as sophisticated as Volley, our results demonstrate that unbridled focus on latency can lead to high costs.
• **ABD Only Optimal and CAS Only Optimal:** These are our most sophisticated baselines meant to represent state-of-the-art

[10]We do not use an existing workload generator such as YCSB [79] because we wish to explore a wider workload feature space than covered by available tools.




**(a) Latency SLO=1 sec.**




**(b) Latency SLO=200 msec.**

**Figure 1: Cumulative count of the normalized cost of our baselines (for our 567 basic workloads) with $f = 1$ and two extreme latency SLOs.**

approaches. ABD Only Optimal and CAS Only Optimal are representative of works that optimize replication-based systems such as SPANStore [75] or EC-based systems such as Pando [67].

It is instructive to note that our baselines are quite powerful. In fact, our optimizer picks the lower cost feasible solution among ABD Only Optimal and CAS Only Optimal, so we expect at least one of these baselines to be competitive for any given workload. Yet, we will demonstrate that: (i) these baselines *individually* perform poorly for many of our workloads; and (ii) the choice of which of the two is better for a particular workload is highly non-trivial.

## 4.2 Insights from Our Optimization

*4.2.1 The Extent and Nature of Cost Savings.* In Figure 1, we express our optimizer's cost savings over our baselines via each baseline's *normalized cost* (cost offered by baseline / cost offered by our optimizer); note that our optimizer has the lowest cost among all the baselines, so this ratio is at least 1. We consider our collection of 567 basic workloads with $f$=1 and (a) a relaxed SLO of 1 sec and (b) a more stringent SLO of 200 msec. At least one of the baselines - and consequently, our optimizer - meet the SLOs for all the 567 workloads chosen. We begin by contrasting the first-order strengths and weaknesses of ABD and CAS as they are understood in conventional wisdom. For the relatively relaxed latency SLO of 1 sec in Figure 1(a), we find that ABD Only Optimal (and other ABD-based variants) have more than twice the cost of our optimizer for more than 300 (i.e, more than half) of our workloads. On the other hand, CAS Only Optimal closely tracks our optimizer's cost.That is, as widely held, if high latencies are tolerable, EC can save storage and networking costs. Figure 1(b), with its far more stringent SLO of 200 msec, confirms another aspect of conventional wisdom. CAS only optimal is now simply unable to meet the SLO for many workloads (324 out of 567). This is expected given its 3-phase PUT operations and larger quorums. (See similar results with $f$=2 in Figure 12 of [82])). What is surprising, however, is that when we focus on the subset of 243 workloads for which CAS Only Optimal is feasible, it proves to be the cost-effective choice—nearly all workloads for which CAS Only Optimal is feasible have a normalized cost of 1. So, even for stringent SLOs, EC does hold the potential of saving costs. Unlike in Figure 1(a), however, CAS Fixed or CAS Nearest are nowhere close to being as effective as CAS Only Optimal. That is, while EC can be cost-effective for these workloads, its quorums need to be chosen carefully rather than via simple greedy heuristics.[11] Replication tends to be the less preferred choice for more relaxed SLOs but, again, there are exceptions.

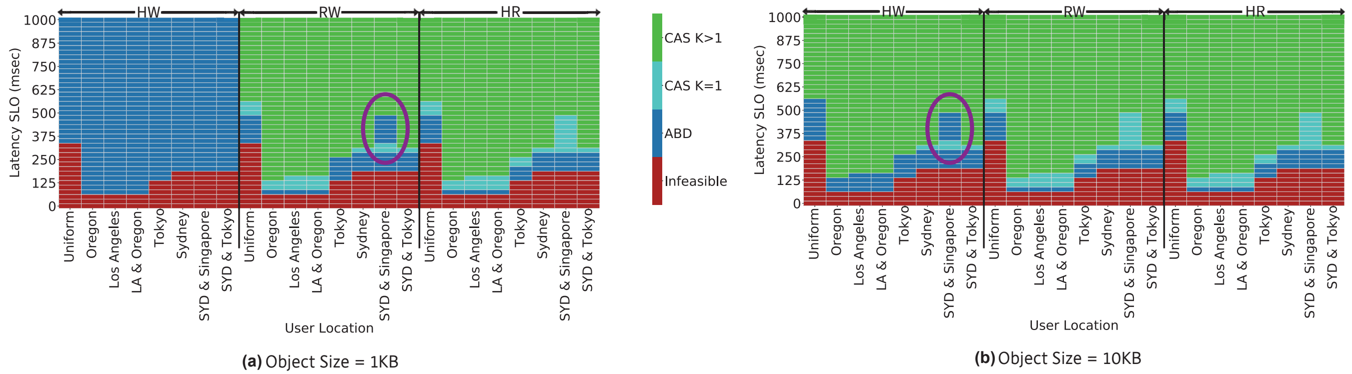
*4.2.2 Sensitivity to Latency SLO.* We focus on how the cost-efficacy of ABD vs. CAS depends on the latency SLO by examining the entire range of latencies from 50 msec to 1 sec. Furthermore, we separate out this dependence based on read ratio, availability target and object size. Our selected results are shown in Figure 2. As expected, as one moves towards more relaxed SLOs, the optimizer's choice tends to shift from ABD to CAS (recall the 3-phase PUTs and larger quorums in CAS). The complexity that the figures bring out is *when* this transition from replication to EC occurs—we see that, depending on workload features, this transition may never occur (e.g., HW in Figure 2(a)) or may occur at a relatively high latency (e.g., at 575 msec for the uniform user distribution for RW/HR).[12] In particular, more spatially distributed workloads correspond to a tendency to choose replication over EC; for instance, for workloads with uniformly distributed users, SLOs smaller than 300 msec are infeasible due to a natural lower bound implied by the inter-DC latencies. We find that $f$ also has a complex impact on the optimizer's choice; see results with $f$=2 in Figure 13, Appendix G in [82].

*4.2.3 Read- vs. Write-Intensive Workloads.* One phenomenon that visibly stands out in Figure 2 is how write-intensive workloads for the relatively small object sizes (HW in Figure 2(a)) prefer ABD even for the more relaxed SLOs. This preference of ABD over CAS becomes less pronounced when we increase the object size to 10KB in Figure 2(b). Finally, we also observe that read-intensive/moderate workloads tend to prefer CAS ($K$=1) over ABD, even when replication is used. To understand this asymmetry, note the following:

- **Reads:** Whereas both ABD and CAS have a "write-back" phase for read operations, ABD's write-back phase carries data, while CAS's only carries metadata, and thereby incurs much lower network cost. Thus, our optimizer tends to prefer CAS for HR workloads.
- **Writes:** For writes, CAS involves 3 phases whereas ABD only requires 2. Since each phase incurs an additional overhead in terms of metadata, the metadata costs for write operations are higher for

[11] As a further nuance, only 3 workloads out of 243 use CAS with $k$=1.

[12] The reader might be intrigued by the portions of Figure 2 highlighted using ovals. Here, our optimizer's choice shifts from ABD to CAS as the latency SLO is relaxed (as expected) but then it shifts back to ABD! We consider this to be a quirk of the heuristics embedded in our optimizer rather than a fundamental property of the optimal solution.




**(a)** Object Size = 1KB

**(b)** Object Size = 10KB

**Figure 2: Sensitivity of the optimizer's choice to the latency SLO. We consider 2 object sizes (1KB and 10KB), 8 different client distributions, arrival rate=500 req/sec, and $f$=1. We consider 3 different read ratios (HW, RW, HR defined in Section 4.1).**

CAS. Therefore, especially for small object sizes (Figure 2(a)) and write-heavy workloads, our optimizer will tend to prefer ABD.

Collectively, the results in Figure 2 convey the significant complexity of choosing between ABD and CAS. Within CAS-friendly workloads, there is further substantial complexity in how the parameter $K$ depends on workload features.

*4.2.4 Factors Affecting Optimal Code Dimension $K$.* We illustrate our findings using the representative results in Figure 3(a)-(c) based on a workload with the following features for which CAS is the cost-effective choice: object size=1KB; datastore size=1TB; arrival rate=200 req/sec; read ratio= RW (50%); user locations are Sydney and Tokyo; latency SLO=1 sec. To understand the effects in Figure 3(a)-(c), we develop a simple analytical model based on the empirical results (details in Appendix E, [82]). Our model relates cost to $K$, object size ($o$), arrival rate ($\lambda$), and $f$ as follows:

$$cost = \left(c_1 \cdot \lambda \cdot K + c_2 \cdot o \cdot \lambda \cdot \frac{f}{K} + c_3 \cdot o \cdot \frac{2f}{K} + \overline{c}_4\right). \tag{4}$$

Here, $c_1, c_2, c_3$ are system-specific constants VM cost, network cost, and storage cost, respectively[13]. Our model captures and helps understand the *non-monotonicity of cost in K* seen in Figure 3(a). This behavior emerges because the following cost components move in opposite directions with growing $K$: network and storage costs decrease due to reduction in object size, while VM costs increase due to increase in quorum sizes. Fundamentally, this implies that even under very relaxed latency constraints, the highest value of $K$ is not necessarily optimal (contrary to the coarse analysis of Table 3). Our model yields the following optimal value of $K$: $K_{opt} = \sqrt{\frac{o \cdot f \cdot (c_2 \cdot \lambda + 2c_3)}{c_1 \cdot \lambda}}$. Observe that $K_{opt}$ increases with object size[14], which is in agreement with Figure 3(b). We observe a similar qualitative match between our model-predicted dependence of $K_{opt}$ on arrival rate and that in Figure 3(c). $K_{opt}$ is a decreasing function of the arrival rate $\lambda$, and saturates to a constant $K^*$ when $\lambda \longrightarrow \infty$, i.e., when the storage cost becomes a negligible component of the overall cost. Interestingly, even for $\lambda \longrightarrow \infty$, the system does not revert to replication, i.e., $K^*$ is not necessarily 1.

*4.2.5 Does EC Necessarily Have Higher Latency Than Replication?* Conventional wisdom dictates that EC has lower costs than replication but suffers from higher latency. We show that perhaps surprisingly, this insight does not always lead to the right choices in the geo-distributed setting. Note that for a linearizable store, requests cannot be local [12], and so even with replication, requests need to contact multiple DCs and the overall latency corresponds to the response time of the farthest DC. Thus, in a geo-distributed scenario where there are multiple DCs at similar distances as the farthest DC in a replication-based system, EC can offer comparable latency at a lower cost. Our optimizer corroborates this insight. Consider a workload where requests to a million objects of 1 KB come from users in Tokyo. The workload is HR (read ratio of 97%) with an arrival rate of 500 req/sec. To tolerate $f$=1 failure, the lowest GET latency achievable via ABD is 139 msec at a cost of \$1.057 per hour, whereas using CAS achieves a GET latency of 160 msec at a cost of \$0.704 per hour - a cost saving of 33% for a mere 21 msec of latency gap. To tolerate $f$=2 failures for the same workload, the lowest GET latency with ABD is 180 msec at a cost of \$1.254 per hour, whereas CAS offers a GET latency of 190 msec at a cost of \$0.773 per hour - 38% lower cost for a mere 10 msec latency increase.

*4.2.6 Are Nearest DCs Always the Right Choice?* Our optimizer reveals that, perhaps surprisingly, the naturally appealing approach of using DCs nearest to user locations [4] can lead to wasted costs. We describe one such finding in Appendix G.2 of [82].

## 4.3 Scalable Concurrency Handling

A distinguishing feature of LEGOStore is that it is designed to provide reliable tail latency even in the face of highly concurrent access to a key. For consensus-based protocols that apply operations sequentially to a replicated state machines one after another, even in the optimistic case where all operations are issued at a leader that does not fail, the latency is expected to grow linearly with the amount of concurrency. Furthermore, in distributed consensus, due to FLP impossibility [26], concurrent operations may endure several (in theory, unbounded) rounds of communication.

[13] $\overline{c}_4$ is a constant and does not affect $K_{opt}$.
[14] A qualification to note is that the phenomenon is connected to our modeling choice of having VM cost independent of the object size $o$. E.g., if the VM cost were chosen as an affine function of $o$, then the dependence of $K_{opt}$ on $o$ would diminish.

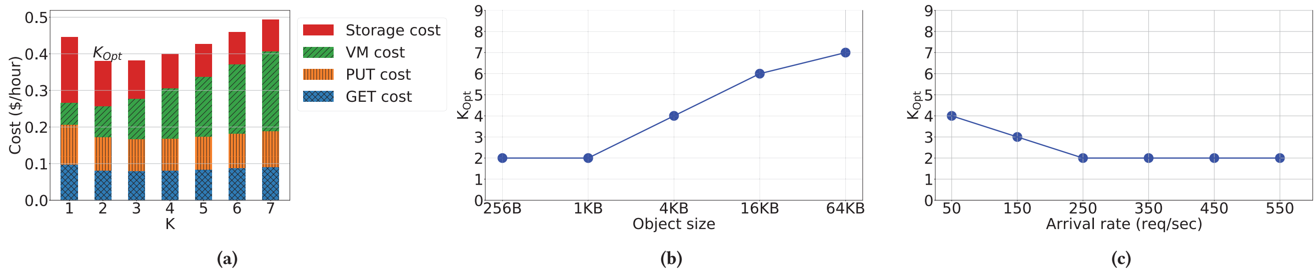
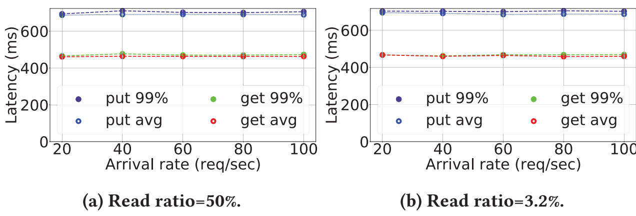



**Figure 3: For CAS-based solutions, cost is non-monotonic in $K$ and $K_{opt}$ has a complex relation with object size and arrival rate. Latency SLO is 1 sec.**




**(a) Read ratio=50%.** **(b) Read ratio=3.2%.**

**Figure 4: LEGOStore is able to ensure that latency offered to a key is robust even at for highly concurrent accesses. Here we plot the latency experienced by clients at the Tokyo location for arrival rates in [20-100] req/sec.**

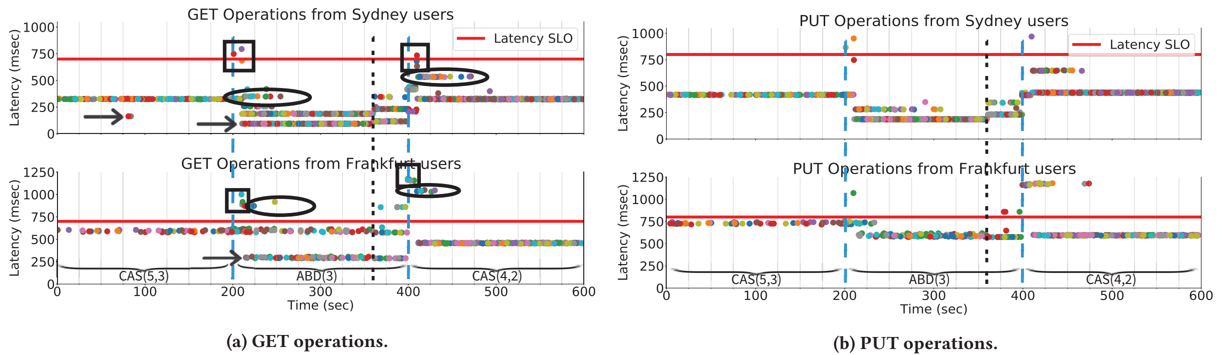
To validate our expectation of robust tail latency even under high concurrency, we increase the arrival rate for the *same key* with object size 1 KB. The object is configured to use CAS(5, 3) with DCs in Singapore, Frankfurt, Virginia, California, and Oregon. In particular, requests from uniformly-distributed user locations come to the single key. We run the experiments for both HW and RW for a period of 1 minute for each arrival rate. We plot the latency experienced by clients at the Tokyo location against arrival rate in Figure 4. LEGOStore demonstrates a remarkable robustness of the latency of operations. Even for an arrival rate of 100 req/sec to the same key, every operation completes and we see no degradation in performance for the average and tail latencies. We recorded a maximum concurrency of 142 write operations on one key for an arrival rate of 100 req/sec and 30:1 write ratio. Little's law suggests an average concurrency of around 60 operations for this experiment. Note the contrast with consensus-based protocols, where the tail latency is crucially dependent on limited concurrency for a given key. E.g., in [67] Figure 13, even with somewhat limited concurrency, the latency of only "successful" writes can grow up to 30s without leader fallback, and at least doubles with leader fallback.

The similarity of the latency in Figure 4(b) which has a HW workload as compared with the RW workload in Figure 4(a) indicates that our latencies remain robust even if the workload is write heavy. It is also worth noting that Figure 4 is a further corroboration of the robustness of our modeling in Section 3.2. Specifically, our model ignores intra-DC phenomena such as queuing, and the robustness of latency despite a high arrival rate shows the overwhelming significance of the inter-DC RTTs in determining response times.

## 4.4 Reconfiguration to Handle Load Change

In this subsection and the next, we explore LEGOStore's ability to perform fast reconfiguration in line with the expectations set in Section 3.3. We consider a set of 20 keys with similar workloads, each with an object size of 1 KB and $f$=1 to which RW (i.e., read ratio of 50%) requests arrive from 4 locations with the following distribution: Tokyo (30%), Sydney (30%), Singapore (30%), and Frankfurt (10%). Each user issues one request every 2 seconds on average. Our latency SLOs are 700 msec and 800 msec for GETs and PUTs, respectively. As seen in Figure 5, till $t$=200 sec, requests arrive at a total rate of 100 req/sec (i.e., 200 users) from the 4 locations. LEGOStore employs configurations with CAS(5,3) for our keys with DCs in Tokyo, Sydney, Singapore, Virginia, and Oregon. The figure plots the latency experienced by users at Sydney and Frankfurt; users at Singapore and Tokyo experience similar SLO adherence. LEGOStore successfully meets SLOs. In fact, a small number of GET requests (shown using a right-facing arrow) see superior performance as they are "optimized" GETs (recall Section 2). At $t$=200 sec, the collective request arrival rate increases 4-fold to 400 req/sec (i.e., 800 users) while all other workload features remain unchanged. We assume that the controller located at LA issues a reconfiguration without delay on detecting this workload change. For the new workload, LEGOStore's optimizer recommends a new configuration performing ABD with replication factor of 3 over DCs in Tokyo, Sydney, and Singapore. Across multiple measurements, we find that reconfiguration concludes in less than 1 sec. The breakdown of overall reconfiguration for a sample instance that takes 717 msec is: (i) reconfig query=68 msec; (ii) reconfig finalize=208 msec; (iii) reconfig write=139 msec; (iv) updating metadata=163; and (v) reconfig finish=139 msec.

We examine user experience during and in the immediate aftermath of reconfiguration. We show the latencies experienced by all the users each at the Sydney and Frankfurt locations to isolate the performance degradation experienced at each user location more clearly.A user request experiences one of two types of degradation which mainly depends on when it arrives in relation to the reconfiguration. **Type (i)** A small number of requests (small due to how quick the reconfiguration is) is blocked at the old configuration servers with the possibility of either getting eventually serviced by these old servers or having to restart in the new configuration (see Section 3.3). These are the requests experiencing latencies in the 750 msec - 1 sec range and highlighted using boxes for the GET requests. **Type (ii)** A second possibility applies to all other requests that do not get blocked at the old configuration servers. These requests incur an additional delay of about 200 msec (users in Sydney) and 250 msec (Frankfurt) to acquire the new configuration from LA and are shown using an ovals for the GET requests. This




(a) GET operations.

(b) PUT operations.

**Figure 5: The efficacy and performance impact of two reconfigurations is shown for one of 20 keys with similar workloads. The first reconfiguration occurs in response to a 4-fold increase in request arrival rate at $t$=200 sec. The second reconfiguration occurs at $t$=400 sec in response to the Singapore DC failing at $t$=360 sec. The arrows show the optimized GET operations while the squares and ovals respectively highlight two types of performance degradation associated with reconfiguration: (i) requests blocked in the old configuration, and (ii) first request issued by a user after the reconfiguration which needs to acquire the new configuration from the controller at LA. The different colors for the latency dots represent different users.**

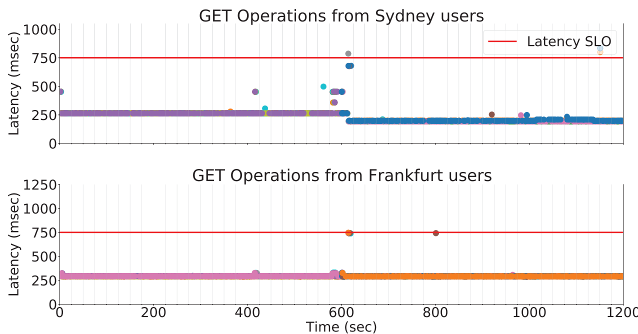



**Figure 6: The efficacy and performance impact of a reconfiguration (at t=10 min) is shown for a key that we derive from the Wikipedia dataset.**

increase in latency happens because the users do not know that a reconfiguration has occurred and try to do an operation with the old configuration, e.g., see requests at $t \sim 200$ sec experiencing a slight degradation among GET operations from Sydney users.

## 4.5 Reconfiguration to Handle DC Failure

When a DC in one of the quorums fails, LEGOStore will send the request to all other DCs participating in the configuration that are not in the quorum. This will in general be sub-optimal cost-wise and may also fail to meet the SLO. In Appendix G.3 Figure 11 in [82], we show a sample result where a DC failure results in such SLO violation. To alleviate this, upon detecting a failure,[15] LEGOStore invokes its optimizer to determine a new cost-effective configuration that discounts the failed DC and then transitions to this new configuration. Figure 5 depicts a scenario where the Singapore DC, a member of both ABD quorums, fails at $t$=360 sec. We assume that this failure is detected and remediated via a transition to a new configuration using CAS(4,2) at $t$=400 sec. Again, we find that the transition occurs within a second and has a small adverse impact on request latency— most requests whose latency exceeds the SLO are of the unavoidable Type (ii).

[15]LEGOStore can work with any existing approach for failure detection.

## 4.6 LEGOStore for a Real-World Workload

In this subsection, we construct our workload using a publicly available dataset collected from Wikipedia's web server [68]. This is a read-mostly workload with a highly skewed popularity distribution. We extract arrival time and request size information from the dataset and interpret each request as a GET or a PUT based on its type. We sample a set of 1550 distinct objects (each interpreted as a key) from the dataset whose aggregate arrival rate can be accommodated by our prototype. We consider workload features for this set of keys over two 1-hour long periods (call these $T_1$ and $T_2$). Since the workload itself does not reveal a distribution of clients, we assume a uniform distribution of clients among 5 of our DCs (Tokyo, Sydney, Singapore, Frankfurt, London) for $T_1$ and a uniform distribution among all 9 DCs for $T_2$. We use our optimizer to determine cost-effective configurations for each of these keys for both of our 1-hour periods. We choose a latency SLO of 750 msec.

Our findings demonstrate that LEGOStore offers cost savings over baselines for the Wikipedia workload. We compare the cost offered by the optimizer against our various baselines for all of the 1550 keys in Figure 15 of [82]. Even for a fixed duration, the results highlight the importance of the optimizer as a variety of different configurations are chosen for different objects - this includes both replication and CAS and different parameters for CAS. Further, with change in client distribution for a given key, LEGOStore's reconfiguration and optimizer couple to ensure sustained cost effectiveness and improvement over baselines. In Figure 6, we highlight an illustrative key's performance in our LEGOStore prototype over a 20 minute period with the first 10 minutes from $T_1$ and the second 10 minutes from $T_2$. The arrival rate to this key changes from 16 to 35 req/sec.Our optimizer chooses CAS (m=5, k=1) for $T_1$ and CAS (m=8, k=1) for $T_2$. The latter yields a 20% cost reduction over the former and triggers a reconfiguration. Figure 6 is centered around the reconfiguration LEGOStore carries out for this key at t=10 min. Similar to earlier experiments, our prototype accomplishes the reconfiguration within 1.96 seconds with an increase in response times for a small number of requests during the reconfiguration.

We find that our optimizer, itself executed on public cloud VMs, contributes negligibly to the operational cost of LEGOStore. As an

illustrative calculation, for the workload in this section we consider extremely frequent reconfigurations occurring once every 5 minutes for the key with the highest arrival rate of 20.16 req/sec. Each invocation of our optimizer costs about $0.0001 on average (the average optimizer execution time is 18 sec, see Appendix H). This turns out to be a mere 0.48% the overall costs for this key.

## 5 RELATED WORK

**EC Based Data Storage:** Several papers have studied the design of in-memory KV stores [2, 18, 22, 45, 57, 60, 74, 77, 84]. A significant body of work focuses on minimizing repair costs and encoding/decoding [14, 22, 41–43, 49, 65, 70, 73, 76, 78, 80]. The cost savings offered by EC have motivated its use particularly in production archival (i.e., write-once/rarely) systems [38, 50]. These papers do not focus on consistency aspects that are relevant to workloads with both reads and writes, nor do they study the geo-distributed setting; therefore, the key factors governing their performance are different from us. Strongly consistent EC-based algorithms and KV stores are developed in [1, 21, 23, 35, 59]; however, none of these works study the geo-distributed setting or the public cloud.

**Strongly Consistent Geo-Distributed Storage:** There are several strongly consistent geo-distributed KV stores [19, 20, 33, 67, 75, 83]. SpanStore [75] develops an optimization to minimize costs while satisfying latencies for a strongly consistent geo-distributed store on the public cloud. While there are several technical differences (e.g., SpanStore uses a blocking protocol via locks), the most important advance made by LEGOStore is its integration of EC into the picture. Besides tuning EC parameters, LEGOStore integrates the constraints of structurally more complex EC-based protocols to enable cost savings. Most closely related to our work are Giza [19] and Pando [67], which are both strongly consistent EC-based geo-distributed data stores. Both data stores modify consensus protocols (Paxos and Fast Paxos) to utilize EC and minimize latency. The most notable difference between these works and LEGOStore is that LEGOStore is designed to keep *tail* latency predictable and robust and keep costs low in the face of dynamism. Since Giza and Pando are based on consensus, they will tend to have higher latency under concurrent writes, e.g., for hot objects with high arrival rates. Furthermore, neither Pando nor Giza have an explicit reconfiguration algorithm. On the other hand, since Giza and Pando use consensus, they offer more complex primitives such as Read-Modify-Writes and versioned objects. A noteworthy comparison point vs. Giza is that it does not operate in the public cloud and does not contain an optimization framework for cost minimization.

**Reconfiguration:** There is a growing body of work that develops *non-blocking* algorithms for reconfiguration [6, 51]. Algorithms in [6, 51] require an additional phase of a client to contact a controller/configuration service in the critical path of *every* operation. In LEGOStore, for the common case of operations that are not concurrent with a reconfiguration, the number of phases (and therefore the latency, costs) are identical to the baseline static protocol. Our algorithm has a resemblance to an adaptation of [46] in the tutorial [5]. That algorithm works mainly for replication and requires clients to propagate values to the new configuration rather than the controller, which can incur larger costs. Our reconfiguration algorithm utilizes concepts/structures that appear in previous algorithms; our main contribution is to adapt the existing algorithms specifically to ABD and CAS in order to keep the reconfiguration latency/costs low and predictable. In particular, our algorithm piggybacks read/write requests for the reconfiguration along with messages that are sent to block ongoing operations, and makes careful choices on operations that can be completed in older configurations to provably ensure linearizability. Several works [4, 7, 62] design heuristics to determine when and which objects to reconfigure. Sharov et. al [62] give a method for optimizing the configuration of quorum-based replication schemes, including the placement of the leaders and replica locations for read and write operations as well as transactions. The paper shares conceptual similarities with LEGOStore's optimization, but was limited to replication-based schemes and focused solely on minimizing latency for placement, whereas we focus on erasure-coded schemes and include costs in our placement decisions; on the other hand, our methods do not readily apply to systems that support transactions. A similar comparison applies to the replication-oriented optimizers described in [3, 28, 81]. Volley [4] describes techniques for dynamically migrating data among Microsoft's geo-distributed data centers to keep content closer to users and keeping server loads well-balanced.

**Data Placement and Optimization for Public Cloud:** There is a rich area of data placement and tuning of consistency parameters for replication based geo-distributed stores [2, 7, 39, 48, 61, 64, 66, 75]. These works expose the role of diverse workloads and costs in system design, and our optimization framework is inspired by this body of work. However, most of these references [7, 39, 61, 66, 75] only consider replication. Reference [2] studied placement and parameter optimization for EC *within* a DC; while some insights are qualitatively similar, our geo-distributed setting along with its diversity makes the salient factors that govern performance different. From an optimization viewpoint, closest are [48, 64] which study EC over geo-distributed public clouds; however, they do not consider consistency and related quorums constraints and costs.

## 6 CONCLUSION

We developed LEGOStore, a linearizable geo-distributed key-value store which procured resources from a public cloud provider. LEGOStore's goal was to offer tail latency SLOs that were predictable and robust in the face of dynamism. We focused on salient aspects of EC's benefits for LEGOStore. Several additional aspects of key-value store design constitute interesting future directions. For instance, LEGOStore's effectiveness depends on a module that detects workload change and then reconfigures based on the detected changes. Additionally, we focused on read/write operations and have not implemented read/modify/write (RMW) operations [40], which inevitably suffer from less robust tail-latency due to FLP impossibility. The recent paper Gryff [15] designs a provably strongly consistent data store with RMW operations and yet provides the favorable tail-latency properties of ABD for read and write operations. Gryff is based on replication, and the development of a similar system that uses EC is an interesting area of future research.

## ACKNOWLEDGEMENTS

This work was supported in part by a Google Faculty Award, and the NSF under grants CCF- 1553248 and CNS-1717571. We thank Raj Pandey for his help in Section 4.6.

## REFERENCES


[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-Scalable Byzantine Fault-Tolerant Services. *SIGOPS Oper. Syst. Rev.* 39, 5 (oct 2005), 59–74. https://doi.org/10.1145/1095809.1095817

[2] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian. 2018. EC-Store: Bridging the Gap between Storage and Latency in Distributed Erasure Coded Systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 255–266. https://doi.org/10.1109/ICDCS.2018.00034

[3] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *Proc. VLDB Endow.* 13, 13 (sep 2020), 3573–3587. https://doi.org/10.14778/3424573.3424578

[4] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 2.

[5] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. 2010. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS: The Distributed Computing Column* 102 (October 2010), 84–108. https://www.microsoft.com/en-us/research/publication/reconfiguring-replicated-atomic-storage-a-tutorial/

[6] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. 2011. Dynamic Atomic Storage without Consensus. *J. ACM* 58, 2, Article 7 (apr 2011), 32 pages. https://doi.org/10.1145/1944345.1944348

[7] Masoud Saeida Ardekani and Douglas B. Terry. 2014. A Self-Configurable Geo-Replicated Cloud Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 367–381. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/ardekani

[8] Anish Athalye. 2017. Porcupine: A fast linearizability checker in Go. https://github.com/anishathalye/porcupine (branch: master), (gathered data dates: 11/01/2021 and 04/14/2022).

[9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1990. Sharing Memory Robustly in Message-Passing Systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Quebec, Canada) *(PODC '90)*. Association for Computing Machinery, New York, NY, USA, 363–375. https://doi.org/10.1145/93385.93441

[10] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (Jan 1995), 124–142. https://doi.org/10.1145/200836.200869

[11] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

[12] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (may 1994), 91–122. https://doi.org/10.1145/176575.176576

[13] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 126–138. https://doi.org/10.1145/3357223.3362706

[14] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. 2019. Fast Recovery Techniques for Erasure-Coded Clusters in Non-Uniform Traffic Network. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 61, 10 pages. https://doi.org/10.1145/3337821.3337831

[15] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 591–617. https://www.usenix.org/conference/nsdi20/presentation/burke

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

[17] Viveck R. Cadambe, Nancy Lynch, Muriel Médard, and Peter Musial. 2014. A Coded Shared Atomic Memory Algorithm for Message Passing Architectures. In *2014 IEEE 13th International Symposium on Network Computing and Applications (NCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 253–260. https://doi.org/10.1109/NCA.2014.44

[18] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. 2017. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Trans. Storage* 13, 3, Article 25 (sep 2017), 30 pages. https://doi.org/10.1145/3129900

[19] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 539–551. https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[21] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. 2013. PoWerStore: Proofs of Writing for Efficient and Robust Storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 285–298. https://doi.org/10.1145/2508859.2516750

[22] S. Duan, P. Subedi, K. Teranishi, P. Davis, H. Kolla, M. Gamell, and M. Parashar. 2018. Scalable Data Resilience for In-memory Data Staging. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 105–115. https://doi.org/10.1109/IPDPS.2018.00021

[23] Partha Dutta, Rachid Guerraoui, and Ron R. Levy. 2008. Optimistic Erasure-Coded Distributed Storage. In *Distributed Computing*, Gadi Taubenfeld (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 182–196.

[24] Evernote. 2021. Evernote. https://evernote.com.

[25] Facebook. 2021. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (apr 1985), 374–382. https://doi.org/10.1145/3149.214121

[27] Mary Jo Foley. 2020. Microsoft's March 3 Azure East US outage: What went wrong (or right)? https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/ (visited on 05/05/2021).

[28] Brad Glasbergen, Michael Abebe, and Khuzaima Daudjee. 2018. Tutorial: Adaptive Replication and Partitioning in Data Systems. In *Proceedings of the 19th International Middleware Conference Tutorials* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 5 pages. https://doi.org/10.1145/3279945.3279946

[29] Google. 2021. Machine types | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/machine-types (visited on 05/05/2021).

[30] Google. 2021. VM instances pricing | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/vm-instance-pricing (visited on 05/05/2021).

[31] Google. 2021. VM instances pricing | Compute Engine Documentation | Google Cloud. https://cloud.google.com/vpc/network-pricing (visited on 05/05/2021).

[32] Y. Guo, A. L. Stolyar, and A. Walid. 2020. Online VM Auto-Scaling Algorithms for Application Hosting in a Cloud. *IEEE Transactions on Cloud Computing* 8, 03 (jul 2020), 889–898. https://doi.org/10.1109/TCC.2018.2830793

[33] Harshit Gupta and Umakishore Ramachandran. 2018. FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems* (Hamilton, New Zealand) *(DEBS '18)*. Association for Computing Machinery, New York, NY, USA, 148–159. https://doi.org/10.1145/3210284.3210297

[34] Nick Heath. 2018. Azure outage: Microsoft working to restore key services after US regional disruption. https://www.techrepublic.com/article/azure-outage-microsoft-working-to-restore-key-services-after-us-regional-outage/ (visited on 05/05/2021).

[35] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 73–86.

[36] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[37] Eben Hewitt. 2010. *Cassandra: The Definitive Guide* (1st ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[38] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2342821.2342823

[39] A. Jonathan, M. Uluyol, A. Chandra, and J. Weissman. 2017. Ensuring reliability in geo-distributed edge cloud. In *2017 Resilience Week (RWS)*. IEEE, Wilmington, DE, USA, 127–132. https://doi.org/10.1109/RWEEK.2017.8088660

[40] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[41] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. 2017. PARIX: Speculative Partial Writes in Erasure-Coded Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 581–587. https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-huiba

[42] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. 2017. BCStore: Bandwidth-Efficient In-memory KV-Store with Batch Coding. *International Conference on Massive Storage Systems and Technology (MSST)* 33 (2017), 13.

[43] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 331–344. https://www.usenix.org/conference/fast19/presentation/li

[44] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT.

[45] Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. 2017. Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached. *IEEE Data Eng. Bull.* 40, 1 (2017), 50–61.

[46] Nancy Lynch and Alexander Shvartsman. 2002. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *International Symposium on Distributed Computing*. Springer, Toulouse, France, 173–190. https://doi.org/10.1007/3-540-36108-1_12

[47] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[48] J. Matt, P. Waibel, and S. Schulte. 2017. Cost- and Latency-Efficient Redundant Data Storage in the Cloud. In *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, Kanazawa, Japan, 164–172. https://doi.org/10.1109/SOCA.2017.30

[49] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 30, 16 pages. https://doi.org/10.1145/2901318.2901328

[50] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, Broomfield, CO, 383–398.

[51] Nicolas Nicolaou, Viveck Cadambe, N Prakash, Kishori Konwar, Muriel Medard, and Nancy Lynch. 2019. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Dallas, Texas, USA, 2195–2205.

[52] European Association of Theoretical Computer Science. 2011. Edsger W. Dijkstra Prize in Distributed Computing: 2011.

[53] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[54] OpenStack. 2021. Erasure Code API library written in C with pluggable Erasure Code backends. https://github.com/openstack/liberasurecode. (branch: master), (gathered data dates: 11/01/2021 and 04/14/2022).

[55] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. https://doi.org/10.1145/2517349.2522716

[56] Overleaf. 2021. Overleaf. https://www.overleaf.com.

[57] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-cache: load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, USA, 401–417.

[58] Ben Rossi. 2015. Amazon investigating major cloud outage, GitHub and Heroku report issues. https://www.information-age.com/amazon-investigating-major-cloud-outage-github-and-heroku-report-issues-123459971/ (visited on 05/05/2021).

[59] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. 2004. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. Association for Computing Machinery, New York, NY, USA, 48–58. https://doi.org/10.1145/1024393.1024400

[60] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. High-performance and resilient key-value store with online erasure coding for big data workloads. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Atlanta, GA, USA, 527–537.

[61] PN Shankaranarayanan, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. 2014. Performance sensitive replication in geo-distributed cloud datastores. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Atlanta, GA, USA, 240–251.

[62] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take Me to Your Leader!: Online Optimization of Distributed Storage Configurations. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1490–1501. https://doi.org/10.14778/2824032.2824047

[63] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 5, 14 pages. https://doi.org/10.1145/2038916.2038921

[64] Maomeng Su, Lei Zhang, Yongwei Wu, Kang Chen, and Keqin Li. 2016. Systematic data placement optimization in multi-cloud storage for complex requirements. *IEEE Trans. Comput.* 65, 6 (2016), 1964–1977.

[65] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2018. Fast and Strongly-consistent Per-item Resilience in Key-value Stores. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 39, 14 pages. https://doi.org/10.1145/3190508.3190536

[66] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Farmington, PA, USA, 309–324.

[67] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 157–180. https://www.usenix.org/conference/nsdi20/presentation/uluyol

[68] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845. http://www.globule.org/publi/WWADH_comnet2009.html, http://www.wikibench.eu/wiki/2007-09/ (visited on 03/10/2022).

[69] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE Computer Society, Seattle, WA, USA, 217–228. https://doi.org/10.1109/ICAC.2005.27

[70] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. 2018. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 139–154. https://www.usenix.org/conference/fast18/presentation/vajha

[71] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. 2018. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 373–389. https://www.usenix.org/conference/osdi18/presentation/veeraraghavan

[72] Werner Vogels. 2021. Diving Deep on S3 Consistency. https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html (visited on 05/05/2021).

[73] Fang Wang, Yingjie Tang, Yanwen Xie, and Xuehai Tang. 2019. XORInc: Optimizing Data Repair and Update for Erasure-Coded Systems with XOR-Based In-Network Computation. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Santa Clara, CA, USA, 244–256.

[74] Shuang Wang, Jianzhong Huang, Xiao Qin, Qiang Cao, and Changsheng Xie. 2017. Wps: A workload-aware placement scheme for erasure-coded in-memory stores. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, Shenzhen, China, 1–10.

[75] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 292–308. https://doi.org/10.1145/2517349.2522730

[76] Jie Xia, Jianzhong Huang, Xiao Qin, Qiang Cao, and Changsheng Xie. 2017. Revisiting Updating Schemes for Erasure-Coded In-Memory Stores. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–6. https://doi.org/10.1109/NAS.2017.8026875

[77] Yu Xiang, Tian Lan, Vaneet Aggarwal, Yih-Farn R Chen, Yu Xiang, Tian Lan, Vaneet Aggarwal, and Yih-Farn R Chen. 2016. Joint latency and cost optimization for erasure-coded data center storage. *IEEE/ACM Transactions on Networking (TON)* 24, 4 (2016), 2443–2457.

[78] Xin Xie, Chentao Wu, Junqing Gu, Han Qiu, Jie Li, Minyi Guo, Xubin He, Yuanyuan Dong, and Yafei Zhao. 2019. AZ-Code: An Efficient Availability Zone Level Erasure Code to Provide High Fault Tolerance in Cloud Storage Systems. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE Computer Society, Los Alamitos, CA, USA, 230–243. https://doi.org/10.1109/MSST.2019.00004

[79] Yahoo. 2010. Yahoo Cloud Serving Benchmark (YCSB). https://research.yahoo.com/news/yahoo-cloud-serving-benchmark.

[80] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. 2017. Erasure Coding for Small Objects in In-Memory KV Storage. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '17)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3078468.3078470

[81] Victor Zakhary, Faisal Nawab, Divy Agrawal, and Amr El Abbadi. 2018. Global-Scale Placement of Transactional Data Stores. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, Vienna, Austria, 385–396. https://doi.org/10.5441/002/edbt.2018.34

[82] Hamidreza Zare, Viveck R. Cadambe, Bhuvan Urgaonkar, Chetan Sharma, Praneet Soni, Nader Alfares, and Arif Merchant. 2021. LEGOStore: A Linearizable Geo-Distributed Store Combining Replication and Erasure Coding. Arxiv preprint available at https://arxiv.org/abs/2111.12009.

[83] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/2517349.2522729

[84] Dongfang Zhao, Ke Wang, Kan Qiao, Tonglin Li, Iman Sadooghi, and Ioan Raicu. 2016. Toward high-performance key-value stores through GPU encoding and locality-aware encoding. *J. Parallel Distributed Comput.* 96 (2016), 27–37.