ELECT: Enabling Erasure Coding Tiering for LSM-tree-based Storage

Yanjing Ren[1], Yuanming Ren[1], Xiaolu Li[2], Yuchong Hu[2], Jingwei Li[3]*, and Patrick P. C. Lee[1]

[1]*The Chinese University of Hong Kong* [2]*Huazhong University of Science and Technology*

[3]*University of Electronic Science and Technology of China*

Abstract

Given the skewed nature of practical key-value (KV) storage workloads, distributed KV stores can adopt a tiered approach to support fast data access in a hot tier and persistent storage in a cold tier. To provide data availability guarantees for the hot tier, existing distributed KV stores often rely on replication and incur prohibitively high redundancy overhead. Erasure coding provides a low-cost redundancy alternative, but incurs high access performance overhead. We present ELECT, a distributed KV store that enables erasure coding tiering based on the log-structured merge tree (LSM-tree), by adopting a hybrid redundancy approach that carefully combines replication and erasure coding with respect to the LSM-tree layout. ELECT incorporates hotness awareness and selectively converts data from replication to erasure coding in the hot tier and offloads data from the hot tier to the cold tier. It also provides a tunable approach to balance the trade-off between storage savings and access performance through a single user-configurable parameter. We implemented ELECT atop Cassandra, which is replication-based. Experiments on Alibaba Cloud show that ELECT achieves significant storage savings in the hot tier, while maintaining high performance and data availability guarantees, compared with Cassandra.

1 Introduction

Storage tiering provides a storage paradigm for balancing the trade-off between access performance and storage persistence in large-scale storage. In particular, for distributed key-value (KV) storage, practical KV workloads are known to have skewed access patterns [6, 9, 16, 62], in which a small fraction of KV pairs are frequently accessed (i.e., hot) and the remaining large fraction of KV pairs are rarely accessed (i.e., cold). Thus, it is natural for distributed KV stores to adopt storage tiering, in which a *hot tier* provides fast data access for hot KV pairs, while a *cold tier* provides persistent storage with less-demanding performance requirements for cold KV pairs.

A primary use case for storage tiering is *edge-cloud storage*. Our motivation is that Internet-of-things (IoT) applications are forecast to generate over 79.4 ZB of data in the wild by 2025 [51]. Since the cloud access performance is bottlenecked by the constrained Internet bandwidth, IoT applications are often coupled with the edge computing paradigm, in which edge nodes, the lightweight instances (e.g., micro-datacenters) provisioned with limited computation and storage resources, are deployed in close proximity to IoT devices [52, 53]. From a storage perspective, we can deploy a distributed KV store as the high-performance hot tier in the edge, while the cloud forms the persistent cold tier. Such an edge-cloud storage architecture has been used in virtualized network functions [40], multi-tier computing [42], multimedia management [21], etc. In addition to edge-cloud storage, storage tiering is also applicable to other storage architectures, such as content-delivery networks and cloud block storage (e.g., the combination of Amazon's Elastic Block Store (EBS) [1] and Simple Storage Service (S3) [2]).

Providing data availability guarantees for hot-tier storage is necessary, especially when the hot tier is deployed in distributed storage environments where failures are prevalent [24]. For example, in edge-cloud storage, edge nodes have limited storage resources and are also prone to failures [53]. While the cloud provides abundant persistent storage resources, reconstructing any lost data for failed edge nodes from the cloud is inefficient due to the high edge-cloud latencies [12, 67]. Thus, to ensure data availability against edge node failures, the edge can introduce storage redundancy, so that any lost data in the edge can be directly reconstructed through the redundant data from other available edge nodes, without the need for retrieving data from the cloud for reconstruction. However, modern distributed KV stores [18, 22, 34, 46] are commonly designed for cloud data centers with sufficient resources, and adopt *replication* to distribute exact redundant copies for individual KV pairs across multiple nodes to provide fault tolerance against node failures. Replication multiplies storage overhead, which is prohibitive for resource-constrained edge nodes, or generally, the high-performance hot tier with limited storage resources.

Erasure coding provides a low-cost redundancy alternative to achieve data availability with much lower storage overhead compared with replication (see §2.3 for details). It has been extensively studied in the literature, especially for distributed KV storage in data centers (§7). However, there exists a fundamental storage-performance trade-off for replication and erasure coding: replication incurs high storage overhead, yet it supports not only load balancing of reads across redundant copies, but also simple reconstruction of any lost data from another available redundant copy; in contrast, erasure coding significantly reduces storage overhead, but it does not keep

*Jingwei Li is the corresponding author.

redundant copies for load balancing and is known to incur higher bandwidth and I/Os in reconstructing lost data when failures happen [19, 26]. It is thus critical to mitigate the storage overhead, while maintaining high access performance as if replication were used, in the hot tier.

We present ELECT, a distributed KV store that enables erasure coding tiering. ELECT builds on the log-structured merge tree (LSM-tree) [45]. Given the skewed nature of practical KV storage workloads [6, 9, 16, 62], ELECT extends the LSM-tree with a *hybrid redundancy* approach by storing limited amounts of hot KV pairs with replication in the hot tier for high access performance, while still achieving significant storage savings by storing large amounts of cold KV pairs with erasure coding in the hot tier. In addition, it can offload cold KV pairs from the hot tier to the cold tier to further alleviate the storage overhead in the hot tier.

Enabling hybrid redundancy in ELECT, however, is non-trivial. ELECT should decide *how* (e.g., at what granularities), *when* (e.g., on or off the write path), and *what* (e.g., differentiating hot and cold KV pairs) to convert replicated KV pairs into erasure-coded KV pairs. Most importantly, ELECT should provide a mechanism to balance the trade-off between storage savings and access performance; such a mechanism should be adaptive to various user requirements. Note that erasure coding has also been proposed for caching [49] and content delivery networks [61] in the context of storage tiering. The novelty of ELECT lies in the careful combination of replication and erasure coding for LSM-tree-based storage with several new design techniques (see §7 for details).

Our contributions are summarized as follows.

- We design ELECT to make a case for enabling erasure coding tiering for distributed KV storage. ELECT has several design features: (i) a redundancy transitioning approach for the conversion from replication to erasure coding based on the LSM-tree; (ii) a hotness-aware approach for both redundancy transitioning and the data offloading from the hot tier to the cold tier; and (iii) a tunable approach, with only a single user-specified parameter based on a storage saving target for simple deployment, for configuring how much data to be erasure-coded and offloaded.
- We implemented ELECT atop Cassandra v4.1.0 [3]. Cassandra [34] is a distributed KV store that uses consistent hashing [31] for data partitioning (§2.1) and the LSM-tree for internal storage management (§2.2). We choose Cassandra due to its decentralized, high-performance, and fault-tolerant nature. Cassandra supports only replication, and ELECT extends Cassandra with erasure coding tiering.
- We conduct experiments in an edge-cloud setting on Alibaba Cloud [38]. Compared with (replication-based) Cassandra, ELECT achieves 56.1% edge storage savings, with similar performance in normal read/write operations.

We now open-source our ELECT prototype at **https://github.com/adslabcuhk/elect**.

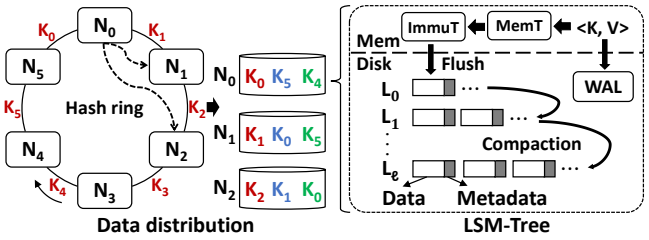


Figure 1: Triple replication in a distributed KV store, in which each node uses an LSM-tree for internal storage management.

2 Background

2.1 Distributed KV Stores

Modern distributed KV stores partition KV pairs across multiple nodes by either consistent-hashing-based [31] distribution [17, 18, 22, 34] or range-based distribution [5, 11, 46, 57]. Consistent hashing arranges nodes in a hash ring, in which each node is associated with a range of the hash ring and stores the KV pairs whose keys are hashed to the range. For load balancing, each node can be further associated with multiple virtual nodes that are associated with different ranges of the hash ring. In contrast, range-based distribution divides the entire key space into non-overlapping ranges, in which each node stores the KV pairs in one of the ranges. ELECT builds on Cassandra [34], which uses consistent hashing, yet its design is compatible with both distribution approaches.

Replication is commonly used in distributed KV stores for fault tolerance [5,11,18,22,34,46,57]. Given a replication factor R, replication distributes R copies of each KV pair across a set of nodes, which collectively form a *replication group*. Suppose that there are M nodes (denoted by $N_0, N_1, \cdots, N_{R-1}$) arranged in a clockwise direction of a hash ring. Each node N_i is associated with a non-overlapping key range K_i, where $0 \leq i \leq R-1$, that corresponds to the keys that precede N_i in the hash ring. Suppose that a KV pair is mapped to N_i. The first copy of the KV pair (called the *primary replica*) is stored in N_i, and the $R-1$ additional copies (called the *secondary replicas*) are stored in the following nodes along the clockwise direction of the hash ring (i.e., $N_{i+1 \bmod M}$, $N_{i+2 \bmod M}$, $\cdots$, $N_{i+R-1 \bmod M}$). Thus, each node N_i now manages the primary replicas of its associated key range as well as the secondary replicas of the associated key ranges of the $R-1$ preceding nodes in the anti-clockwise direction of the hash ring. Figure 1 shows an example of triple replication (i.e., $R=3$) and $M=6$ nodes. For example, N_0 stores the KV pairs for K_0, K_5, and K_4.

2.2 Log-Structured Merge Trees (LSM-Trees)

Each node in Cassandra [34] manages its internal storage based on an LSM-tree [45], as also used in state-of-the-art local [9] and other distributed [5, 11, 34, 46] KV stores. An LSM-tree is a data structure designed for efficient writes (i.e., `Put` requests), reads (i.e., `Get` requests), and scans (i.e., `Scan` requests). Figure 1 shows how an LSM-tree is deployed in

a distributed KV store. An LSM-tree organizes KV pairs in immutable fixed-size files, called *SSTables*, across $\ell+1$ levels, denoted by $L_0, L_1, \cdots, L_\ell$; L_0 is the lowest level and L_ℓ is the highest level. Each SSTable stores multiple KV pairs in a *sorted* manner in units of *data blocks* of size several KiB each. To support fast reads, each SSTable maintains an *index block* to track the key ranges and offsets of all data blocks as well as a *Bloom filter* [7] to track its currently stored keys with a small false positive rate. We refer to the data blocks as the *data component*, and collectively refer to the index block, Bloom filter, and SSTable metadata as the *metadata component*, for an SSTable. Inside the LSM-tree, the number of SSTables in each level increases from the lower to higher levels, while the KV pairs of an SSTable do not overlap with those of other SSTables in the same level except L_0 (note that some advanced LSM-trees may have overlapping KV pairs across SSTables in the same level [48]).

Writes and reads of KV pairs are issued to an LSM-tree as follows. Each write appends a newly written KV pair to an on-disk *write-ahead log (WAL)* for crash consistency and then inserts it into a mutable in-memory structure called a *MemTable*. When the MemTable is full, it is turned into an *Immutable MemTable*, which is then flushed to the lowest level L_0 as an SSTable. Each level has a capacity limit, increasing from lower to higher levels. When a lower level reaches its capacity limit, it triggers *compaction* to merge the KV pairs in the lower level into its next higher level. Specifically, a compaction operation selects an SSTable in the lower level, reads all SSTables in the higher level that have overlapping key ranges with the selected SSTable, sorts all the latest KV pairs (while all stale KV pairs are discarded), and re-generates and stores the non-overlapping SSTables in the higher level. On the other hand, each read for a key searches the MemTable and then the SSTables from L_0 to L_ℓ. It returns the KV pair if the key is found, or null otherwise.

2.3 Erasure Coding

Erasure coding provides low-redundancy fault tolerance for distributed storage. In this work, we focus on Reed-Solomon (RS) codes [50], configured by two parameters n and k (where $k<n$), as our erasure code construction. We choose RS codes for three reasons: (i) they support general coding parameters n and k (provided $k<n$); (ii) they have the minimum storage redundancy for tolerating against any $n-k$ node failures (a.k.a. the Maximum Distance Separable (MDS) property); and (iii) they have been popularly deployed in production [24, 43].

An (n,k) RS code encodes k original (uncoded) fixed-size *data chunks* into $n-k$ (coded) *parity chunks* of the same size, and the collection of n data and parity chunks forms a *coding group*; the (n,k) RS code considered here is *systematic*, meaning that the coding group keeps the k data chunks. It ensures that any k out of the n chunks of a coding group can reconstruct all k original data chunks. Large-scale storage systems comprise multiple coding groups that are independently encoded/decoded, and the n chunks of each coding group are distributed across n nodes to tolerate any $n-k$ node failures with $n/k\times$ storage overhead. Compared with replication, RS codes incur much lower storage overhead with higher fault tolerance; for example, Facebook [43] uses the $(14,10)$ RS code for four-node fault tolerance with $1.4\times$ storage overhead only, while traditional triple replication [25] only provides two-node fault tolerance and incurs $3\times$ storage overhead.

Erasure coding is known to have reconstruction penalty. For example, for any lost chunk, an (n,k) RS code needs to retrieve k available chunks from other alive nodes in the same coding group so as to decode the lost chunk. Reconstruction is common in practice due to the prevalence of failures [24, 26, 43], and there are two major reconstruction operations: degraded reads (i.e., reads issued to lost chunks) and full-node recovery (i.e., all data stored in a node is lost).

There are code constructions that reduce the reconstruction bandwidth of RS codes (e.g., regenerating codes [19] and locally repairable codes [26]). However, they still retrieve more data for reconstruction than the amount of lost data, and the trade-off between storage savings and reconstruction bandwidth in erasure coding is fundamental [19].

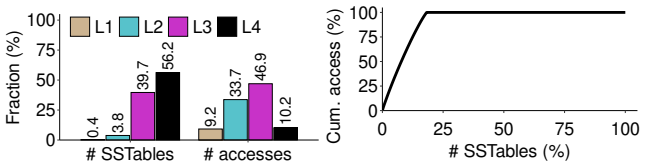
3 Design Considerations

Before we present the design of ELECT, we pose five design questions that need to be addressed.

Q1: At what granularity should KV pairs be encoded? In the context of KV stores, there are two approaches to encode KV pairs at different granularities: (i) *self-encoding* [8, 32, 33, 44], which splits a KV pair into k fixed-size data chunks for encoding, and (ii) *cross-encoding* [14, 37, 65, 66], which aggregates multiple KV pairs into individual data chunks and performs encoding on each group of k different data chunks. Self-encoding improves the parallelism of data access, but incurs significant metadata overhead for indexing all chunks of individual KV pairs [65], especially when KV services are dominated by small KV pairs [9]. In contrast, cross-encoding mitigates such metadata overhead, but the degraded read to a KV pair during a node failure needs to retrieve k surviving chunks of the same coding group for reconstruction, thereby leading to amplified I/Os and bandwidth.

ELECT opts for cross-encoding to reduce the metadata overhead; if we only encode cold KV pairs that are rarely accessed (see Q3 below), the degraded read overhead should be limited. Also, since LSM-trees organize KV pairs in units of SSTables, ELECT opts for cross-encoding across multiple SSTables (i.e., each SSTable is treated as a chunk) to align with the LSM-tree-based storage management.

Q2: Should erasure coding be performed on or off the write path? Erasure coding for KV pairs can be performed *inline* [8, 20, 44], in which KV pairs are encoded on the write path, or *offline* [23, 36, 58], in which KV pairs are first written and later encoded in the background. Offline encoding has the flexibility of first storing hot KV pairs with replication



(a) Statistics across levels (b) Access distributions in L_4

Figure 2: Storage and access patterns in Cassandra.

for high access performance (see Q3 below). Thus, ELECT opts for offline encoding, in which KV pairs are first written with replication, and later performs erasure coding (across SSTables) in the background.

Q3: How should skewed access patterns be addressed? Practical KV workloads have skewed access patterns [6, 9, 16, 62], in which few KV pairs are frequently accessed (hot) and the majority of KV pairs are rarely accessed (cold). ELECT opts to apply erasure coding to cold SSTables to mitigate the degraded read overhead in cross-encoding (see Q1), while storing hot SSTables with replication for high-performance accesses with limited additional storage overhead. The idea of applying replication for hot data and erasure coding for cold data has been studied in prior studies [23,36,58], yet they target different deployment environments and how to adapt this idea into LSM-tree-based KV stores remains unexplored.

We motivate our design by examining the storage and access patterns of SSTables in different LSM-tree levels by generating realistic KV workloads using the benchmarking tool YCSB [16], which has been extensively used for KV storage evaluation in the literature. Specifically, we load 100 M 1-KiB KV pairs with a key size of 24 bytes and a value size of 1000 bytes into Cassandra (v4.1.0) via YCSB in our testbed (see §6.1 for testbed details). Also, using the `nodetool` command in Cassandra, we flush the MemTable of each LSM-tree to disk and force the compaction on all SSTables to keep all nodes in a stable state. We find that the last level (i.e., the highest level) is L_4, while L_0 is empty as the SSTables originally in L_0 are merged to L_1 after the forced compaction. We then issue 10 M reads to the stored KV pairs, where the keys are accessed under the Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). Note that we set the replication factor as one to mitigate the impact of replication, and disable the key cache and row cache in Cassandra to have all KV pairs read from on-disk SSTables.

Figure 2(a) shows the distributions of numbers of SSTables and accesses to SSTables in each level. The intermediate levels L_2 and L_3 have high read frequencies. However, L_4 stores the most SSTables (56.2% of all SSTables), but only accounts for 10.2% of accesses. This motivates us to perform erasure coding only for the SSTables in the last level (e.g., L_4 in this example) and replication for the SSTables in the lower levels, so that we still achieve significant storage savings and limit the degraded read overhead caused by erasure coding.

Figure 2(b) further shows the cumulative distribution of access frequencies versus the SSTables in L_4. Only 18.2% of SSTables in L_4 are accessed. This suggests that we can apply erasure coding in a more fine-grained manner by selecting only the SSTables that are rarely accessed for erasure coding (i.e., with negligible degraded read overhead).

Q4: How should the access overhead in the cold tier be mitigated? It is expected that the cold tier has worse access performance than the hot tier. For example, in edge-cloud storage, while the cloud provides much more abundant storage resources than the edge, it is also limited by the high edge-cloud latency over the Internet (e.g., 30 ms for client-to-cloud communication versus 5 ms for client-to-edge communication [12, 67]). ELECT should selectively offload data that is rarely accessed from the hot tier to the cold tier, so as to avoid frequently retrieving the data back from the cold tier.

Q5: How should ELECT address the trade-off between storage savings and access performance? Both redundancy transitioning from replication to erasure coding and the data offloading from the hot tier to the cold tier in essence trade access performance for storage savings. ELECT should provide a tunable mechanism that allows users to balance the trade-off depending on their requirements.

4 ELECT Design

ELECT extends Cassandra [34], which uses replication for fault tolerance, with erasure coding tiering. It is deployed across multiple nodes in the hot tier and is backed by the cold tier with persistent storage. It proposes several design elements to address the questions in §3.

- *LSM-tree-based redundancy transitioning* (§4.1). ELECT applies cross-encoding (see Q1) across SSTables in an offline manner (see Q2), by converting SSTables from replication into erasure coding (called *redundancy transitioning*). It decouples the replicas originating from different nodes into multiple LSM-trees, such that it applies cross-encoding to the primary replicas and removes the secondary replicas after encoding. One subtlety is that it should maintain the correctness of redundancy transitioning even under LSM-tree compaction, which changes the content of SSTables.
- *Hotness awareness* (§4.2). ELECT applies cross-encoding to only SSTables in the last LSM-tree level (see Q3). It also offloads SSTables that tend to be rarely accessed from the hot tier to the cold tier to mitigate the access overhead in the cold tier (see Q4).
- *Balancing storage-performance trade-off* (§4.3). ELECT provides a user-configurable parameter, namely the storage saving target, to balance the trade-off between storage savings and access performance (see Q5).

4.1 LSM-tree-based Redundancy Transitioning

ELECT decomposes redundancy transitioning into four steps: LSM-tree management (§4.1.1), parity node selection (§4.1.2), cross-SSTable encoding (§4.1.3), and secondary replica removal (§4.1.4). Figure 3 shows the overall redundancy transitioning workflow in ELECT.

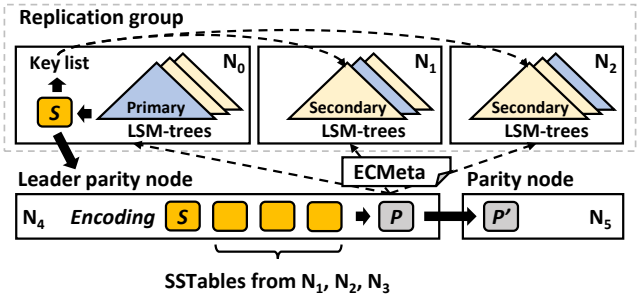


Figure 3: LSM-tree-based redundancy transitioning in ELECT for a coding group of (6,4) RS coding, in which N_0, N_1, N_2, N_3 are data nodes, while N_4 (leader parity node) and N_5 are parity nodes.

4.1.1 LSM-tree Management

Decoupled replication management. In Cassandra, all replicas stored in a node are managed in a single LSM-tree. To facilitate cross-SSTable encoding across nodes and subsequent removals of replicas, ELECT borrows the idea of decoupled replication management [57,68] (originally designed for reducing I/O amplification) by separating the replicas into multiple LSM-trees in each node. Recall that for a replication factor R, each node maintains the primary replicas originating from the node itself and the secondary replicas originating from the $R-1$ preceding nodes in the hash ring (§2.1). ELECT now lets each node maintain R LSM-trees, comprising one *primary LSM-tree* for the primary replicas and $R-1$ *secondary LSM-trees* for the $R-1$ respective sets of secondary replicas. For example, in Figure 1 with $R=3$, N_0 writes the primary replicas in key range K_0 to the primary LSM-tree and the secondary replicas in key ranges K_5 and K_4 to two other secondary LSM-trees.

LSM-tree level generation. The original LSM-tree creates a new level L_ℓ when the current last level $L_{\ell-1}$ is full. However, depending on the current storage usage, the last level L_ℓ may only contain a small number of SSTables. This compromises the effectiveness of ELECT, which applies erasure coding only to the SSTables in the last level. To address this issue, ELECT modifies the current LSM-tree design and creates the new level L_ℓ in the LSM-tree only when the size of the current last level $L_{\ell-1}$ reaches the capacity limit of the next level L_ℓ. For example, the LSM-tree in Cassandra currently sets the size limits of $L_{\ell-1}$ and L_ℓ as T and $10T$, respectively, where T is some capacity limit and the default capacity difference across adjacent levels is $10\times$. ELECT now keeps adding SSTables to $L_{\ell-1}$ even though the size exceeds T. It only creates L_ℓ when the size of $L_{\ell-1}$ exceeds $10T$. It then still keeps a size T of SSTables in $L_{\ell-1}$ and moves at least $9T$ of SSTables to L_ℓ. In this case, ELECT ensures that the last level L_ℓ always contains a sufficiently large number of SSTables (almost 90% of all SSTables across all levels in our case) and maintains the effectiveness of redundancy transitioning. Note that if the LSM-tree grows and adds a new level, the current erasure-coded SSTables in the previous last level will be moved to the new last level.

4.1.2 Parity Node Selection

ELECT applies cross-SSTable encoding on k uncoded SSTables (called *data SSTables*) from k nodes (called *data nodes*) and generates $n-k$ coded SSTables (called *parity SSTables*) that are stored in $n-k$ nodes (called *parity nodes*). Before encoding, ELECT first selects the set of parity nodes to which the parity SSTables are distributed. The selection process should satisfy the following requirements: (i) for fault tolerance, the parity nodes should be distinct from the data nodes; (ii) for load balancing, the parity SSTables are evenly distributed across all nodes after parity node selection; and (iii) for scalability, the parity nodes can be deterministically selected by individual nodes without centralized coordination.

To satisfy the above requirements, ELECT forms each coding group over n consecutive nodes in the hash ring, say $N_{i \bmod M}, N_{(i+1) \bmod M}, \cdots, N_{(i+n-1) \bmod M}$ for $0 \le i < M$, where M is the total number of nodes, the first k nodes are the data nodes, and the following $n-k$ nodes are the parity nodes. Also, each node locally maintains a monotonic sequence number Q (initialized as zero). Specifically, for each SSTable in the primary LSM-tree that is selected by N_i ($0 \le i < M$) for erasure coding, N_i selects a *leader parity node* N_p, which will be responsible for computing and sending the parity SSTables (§4.1.3) to $n-k-1$ other parity nodes. It computes p as:

$$p = (i + (Q \bmod k) + 1) \bmod M, \qquad (1)$$

and increments the sequence number Q by one for each SSTable being selected for erasure coding. Note that the selection of SSTables is based on their priorities (§4.2).

We explain via an example the idea behind Equation (1). From Figure 1 (where $M=6$), we use $(6,4)$ RS coding. Thus, N_0 (deterministically) selects a leader parity node from N_1, N_2, N_3, and N_4 in a round-robin fashion for encoding its SSTables as Q increases; similarly, N_1 selects a leader parity node from N_2, N_3, N_4, and N_5, and so forth. Inversely, each node N_i ($0 \le i < M$) in the whole system will be selected as a leader parity node by k nodes $N_{(i-1) \bmod M}, N_{(i-2) \bmod M}, \cdots, N_{(i-k) \bmod M}$, which now become the k data nodes of a coding group; for example, in Figure 3, N_4 serves as a leader parity node for N_0, N_1, N_2, and N_3 in a coding group under $(6,4)$ RS coding. Since a large-scale storage system typically contains multiple coding groups, ELECT ensures that all coding groups are distributed across different sequences of n consecutive nodes. Finally, each leader parity node (say N_p) will be the first parity node of a coding group, and the remaining $n-k-1$ parity nodes of the coding group are the $n-k-1$ succeeding nodes of N_p along the clockwise direction of the hash ring.

4.1.3 Cross-SSTable Encoding

Encoding workflow. Each of the k data nodes of a coding group sends an SSTable to the leader parity node, which is determined by the sequence number Q according to Equation (1) (§4.1.2). Upon receiving the k data SSTables, the

leader parity node encodes them into $n-k$ parity SSTables, stores one of the parity SSTables locally, and sends the remaining parity SSTables to the other $n-k-1$ parity nodes. The parity nodes store the parity SSTables as separate files outside of their LSM-trees. For example, from Figure 3, consider a coding group for $(6,4)$ RS coding with $k=4$ data nodes N_0, N_1, N_2, and N_3, all of which share the same leader parity node N_4. N_0 sends an SSTable (say S) to N_4. Then, N_4 encodes S together with other SSTables (from N_1, N_2, and N_3, respectively) to generate the parity SSTables (say P and P'). N_4 stores P locally and sends P' to another parity node N_5.

The leader parity node also generates a metadata structure, called `ECMeta`, for the coding group to support failure reconstruction (§5). The `ECMeta` contains n 4-tuples, each of which describes a data/parity SSTable in the coding group, including: (i) the cryptographic hash (e.g., SHA-256) of the content of the data component of the SSTable (32 bytes), (ii) the SSTable size (4 bytes), (iii) the identifier of the node that stores the SSTable (4 bytes), and (iv) the position in the coding group (indexed from 0 to $n-1$) (4 bytes). In particular, we borrow the idea from deduplication [47] and use the SSTable hash as the unique identifier to search for the SSTable during reconstruction, assuming that the hash collisions for distinct SSTables are practically unlikely. The leader parity node then sends the `ECMeta` of each data SSTable to the corresponding R nodes in the replication group. Upon receiving the `ECMeta`, each of the R nodes includes the `ECMeta` in the metadata component of the corresponding SSTable.

Compaction-triggered parity updates. When a primary LSM-tree undergoes compaction, its data SSTables (in the last level) may be updated. ELECT needs to update the parity SSTables of the same coding group to maintain consistency.

Consider a primary LSM-tree that undergoes compaction. Let $S_0, S_1, \cdots, S_u$ be the old data SSTables before compaction, and $S'_0, S'_1, \cdots, S'_v$ be the new data SSTables after compaction, where u and v are the numbers of old data SSTables and new data SSTables, respectively. Without loss of generality, the two sequences of data SSTables $(S_0, S_1, \cdots, S_u)$ and $(S'_0, S'_1, \cdots, S'_v)$ are ordered by (non-overlapping) key ranges. ELECT pairs each of the old and new data SSTables as (S_0, S'_0), (S_1, S'_1), and so forth. If $u < v$ (i.e., there exist more new data SSTables), the extra new data SSTables are simply added as regular SSTables to the primary LSM-tree without erasure coding; if $u > v$ (i.e., there exist fewer new data SSTables due to deleted KV pairs), ELECT pairs each extra old data SSTables with zero-filled dummy SSTables. The dummy SSTables can later be replaced by the new data SSTables that correspond to the same leader parity node.

For each pair of old and new SSTables, ELECT reads the `ECMeta` of the old data SSTable to identify the corresponding leader parity node. It sends the pair to the leader parity node, which updates the parity SSTables of the same coding group based on delta-based parity updates (similar to read-modify-writes in RAID) [10] and sends out the updated `ECMeta`.

4.1.4 Secondary Replica Removal

ELECT removes the secondary replicas from the secondary LSM-trees after cross-SSTable encoding to reclaim storage space. Since the LSM-trees of different nodes perform compaction asynchronously, they may have distinct SSTables. It is important to avoid incorrectly removing KV pairs from the secondary replicas, especially for the KV pairs that are updated after cross-SSTable encoding.

After cross-SSTable encoding, for each primary LSM-tree, ELECT generates a *key list* for each data SSTable, where the key list includes the keys in the SSTables and the corresponding written timestamps; note that the timestamps are already provided by Cassandra to identify the latest versions of KV pairs among replicas. It sends the key list to the other secondary LSM-trees that contain the secondary replicas of the data SSTable. For each secondary LSM-tree, ELECT finds all SSTables in the last level whose KV pairs are covered by the key list. It removes only the KV pairs that are either the current or older versions with respect to the timestamps specified in the key list. Note that ELECT creates new SSTables and tracks the key ranges of the removed KV pairs in the metadata components of the new SSTables, so as to be compatible with the LSM-tree management under replication. Finally, it reconstructs the SSTables for the remaining KV pairs. If the current versions of all KV pairs indicated by the key list are removed, the key list is also removed.

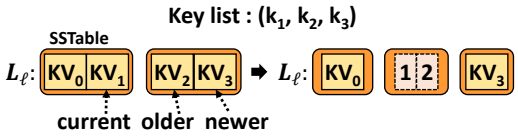
Figure 4(a) shows the replica removal workflow in the last level of a secondary LSM-tree. Suppose that the secondary LSM-tree has four KV pairs in the last level L_ℓ, denoted by KV_0, KV_1, KV_2, and KV_3, whose keys are k_0, k_1, k_2, and k_3, respectively, while the four KV pairs are stored in two SSTables (KV_0, KV_1) and (KV_2, KV_3). Now, suppose that the secondary LSM-tree receives a key list (k_1, k_2, k_3), whose timestamps indicate that KV_1 is the current version (i.e., same timestamp), KV_2 is older, and KV_3 is newer. Thus, ELECT removes KV_1 and KV_2. It also creates an SSTable that tracks the key ranges for the deleted KV_1 and KV_2.

Note that the secondary LSM-tree may not remove the current version of a KV pair (e.g., KV_2) as indicated in the key list, as the KV pair to be removed is not yet moved to the last level due to asynchronous compaction of different nodes. Figure 4(b) shows a case where the second last level $L_{\ell-1}$ has a newer version KV_1 and the current version KV_2 with respect to the timestamps in the key list. During compaction, ELECT removes KV_2 and adds KV_1 to the last level L_ℓ.

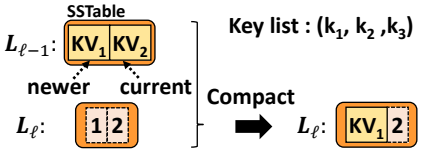
4.2 Hotness Awareness

ELECT incorporates hotness awareness into redundancy transitioning and data offloading.

Hotness-aware redundancy transitioning. ELECT monitors the hotness of each SSTable based on two metrics: (i) the access frequency, which refers to the number of reads issued to the SSTable as measured by Cassandra, and (ii) the lifetime, which refers to the elapsed time since the SSTable



(a) Removal in the last level



(b) Removal during compaction

Figure 4: Secondary replica removal in ELECT.

creation. Among the SSTables in the last level of the primary LSM-tree in each node, an SSTable is said to have a higher priority to be selected for encoding (§4.1.3) if it has a lower access frequency and (if a tie exists) a longer lifetime. ELECT selects the SSTables with higher priorities for encoding based on a storage saving target (§4.3).

Cold-data offloading. ELECT dynamically offloads SSTables from the hot tier to the cold tier based on the hotness of SSTables, so as to further mitigate the storage overhead in the hot tier. First, it offloads parity SSTables with long lifetimes as they only affect parity updates (§4.1.3) and failure reconstruction. Second, after all parity SSTables are offloaded, it selectively offloads the data SSTables with higher priorities. The exact numbers of data and parity SSTables being offloaded depend on a storage saving target (§4.3). Note that if an SSTable is selected to be offloaded, only the SSTable's data component is moved, while its metadata component remains in the hot tier to serve read and compaction operations. Furthermore, when a read or compaction operation touches an SSTable in the cold tier, ELECT retrieves the SSTable's data component from the cold tier to the hot tier.

4.3 Balancing Storage-Performance Trade-Off

Redundancy transitioning and data offloading alleviate the storage overhead in the hot tier, yet they also incur performance overhead compared with replicating all data in the hot tier. To balance the trade-off between the storage savings and access performance, ELECT introduces a configurable *storage saving target* α with respect to when all SSTables are replicated, so as to control the number of SSTables involved in redundancy transitioning and data offloading. Specifically, α is a fractional value between zero and one, such that a larger α implies that more SSTables are erasure-coded and offloaded from the hot tier to the cold tier, and vice versa.

Quantifying storage overhead. We approximate the storage overhead in the hot tier based on the number of SSTables in a single primary LSM-tree in a node. Let C_{all} be the total number of SSTables in the primary LSM-tree, C_{last} be the number of SSTables in the last level of the LSM-tree, C_{rt} be the number of data SSTables in the last level being converted from replication to erasure coding, and C_{pm} and C_{dm} be the numbers of parity SSTables and data SSTables being offloaded to the cold tier, respectively.

We now quantify the actual storage size of a replication group (in terms of the number of SSTables), assuming that the storage load is balanced (i.e., all nodes have the same number of SSTables in their respective primary LSM-trees). ELECT replicates $C_{all}-C_{rt}$ SSTables and encodes C_{rt} SSTables, so their storage usage is $(C_{all}-C_{rt})\cdot R+C_{rt}\cdot\frac{n}{k}$. It also offloads $C_{pm}+C_{dm}$ SSTables to the cold tier. Thus, the actual storage size of a replication group is:

$$(C_{all}-C_{rt})\cdot R+C_{rt}\cdot\tfrac{n}{k}-C_{pm}-C_{dm}. \quad (2)$$

When all SSTables are replicated, the actual storage size of a replication group is $C_{all}\cdot R$. To achieve the storage saving target α, our goal is to configure C_{rt}, C_{pm}, and C_{dm}, so that

$$1-\tfrac{1}{C_{all}\cdot R}[(C_{all}-C_{rt})\cdot R+C_{rt}\cdot\tfrac{n}{k}-C_{pm}-C_{dm}]\geq\alpha. \quad (3)$$

In ELECT, each node computes C_{rt}, C_{pm}, and C_{dm} independently (without centralized coordination) given C_{all}, C_{last}, and α. Assuming balanced storage loads, the respective values of C_{all} and C_{last} across nodes have very small differences.

Balancing trade-off. ELECT starts with redundancy transitioning to keep all SSTables (replicated or erasure-coded) in the hot tier. If α is not met, it offloads parity SSTables to the cold tier, so that all SSTables remain accessible from the hot tier when no failure occurs; in case of a node failure, parity SSTables are needed for recovery. If α is still not met, ELECT offloads data SSTables to the cold tier. Note that ELECT only offloads erasure-coded SSTables to the cold tier, so α may not be achievable if it is too large.

- *Case 1 (Redundancy transitioning):* ELECT sets $C_{pm}=C_{dm}=0$ and chooses the largest possible C_{rt} to maximize storage savings. Note that $C_{rt}\leq C_{last}$. From Equation (3), C_{rt} is given by:
$$C_{rt}=\min\{\tfrac{R\cdot C_{all}\cdot\alpha}{R-n/k},\ C_{last}\}. \quad (4)$$
- *Case 2 (Offloading of parity SSTables):* ELECT proceeds to Case 2 if α is not met, i.e., $C_{rt}=C_{last}$. It sets $C_{dm}=0$ and chooses the largest possible C_{pm}. Note that the number of parity SSTables is at most $\frac{n-k}{k}\cdot C_{last}$. From Equation (3), C_{pm} is given by:
$$C_{pm}=\min\{R\cdot C_{all}\cdot\alpha-(R-\tfrac{n}{k})\cdot C_{last},\ \tfrac{n-k}{k}\cdot C_{last}\}. \quad (5)$$
- *Case 3 (Offloading of data SSTables):* ELECT proceeds to Case 3 if α is still not met, i.e., $C_{pm}=\frac{n-k}{k}\cdot C_{last}$. It chooses the largest possible C_{dm}. Note that $C_{dm}\leq C_{last}$. From Equation (3), C_{dm} is given by:
$$C_{dm}=\min\{C_{all}\cdot R\cdot\alpha-(R-1)\cdot C_{last},\ C_{last}\}. \quad (6)$$

5 Implementation

We implement ELECT in Java based on Cassandra v4.1.0 [3], with around 27 K lines of code of modifications to Cassandra's codebase (which consists of 1.25 M lines of code), by adding redundancy transitioning, hotness monitoring, data offloading, full-node recovery, and degraded reads/writes.

We implement the erasure coding operations based on Intel's Intelligent Storage Acceleration Library [27] and link the operations with Cassandra through the Java Native Interface.

Consistent reads/writes. Under replication, Cassandra supports consistent reads/writes based on the configurable read/write consistency levels, which specify the number of nodes in a replication group that need to acknowledge a read/write request. ELECT maintains the same read/write workflows for replicated KV pairs as in Cassandra. For writes, ELECT performs the same consistent writes as in Cassandra since it always writes KV pairs via replication. For reads, after receiving enough acknowledgments according to the read consistency level, if a KV pair is replicated, ELECT follows the same consistent read path as in Cassandra; if a KV pair is erasure-coded, ELECT always returns the KV pair from the primary LSM-tree or issues degraded reads (see below) if the KV pair is unavailable. ELECT currently does not verify reads for erasure-coded KV pairs.

Full-node recovery. Suppose that a node crashes and all its LSM-trees are lost. ELECT performs recovery in a new node on a per-LSM-tree basis. To recover a primary LSM-tree, ELECT retrieves the secondary LSM-tree from another alive node to the new node. The SSTables from the lowest to the second last level in the LSM-tree are replicated and can be directly recovered from their replicas. For the SSTables in the last level being erasure-coded, ELECT retrieves k data or parity SSTables of the same coding group based on the `ECMeta` from the other alive nodes or the cloud to decode the lost SSTables. To recover a secondary LSM-tree, ELECT retrieves a primary LSM-tree or a secondary LSM-tree from the other nodes in the same replication group; if a primary LSM-tree is retrieved, ELECT removes the data components of SSTables that are erasure-coded, as the secondary LSM-tree only keeps their metadata components (§4.1.4).

Degraded reads. Suppose that ELECT receives a degraded read to an unavailable KV pair. ELECT relays the read request to another alive node in the same replication group, with a flag indicating the KV pair is unavailable. If the KV pair is stored with replication, the alive node directly returns the KV pair; otherwise, if the KV pair is stored with erasure coding, the alive node decodes the SSTable containing the KV pair by retrieving k data or parity SSTables of the same coding group from the other alive nodes according to the `ECMeta`.

Degraded writes. Suppose that ELECT receives a degraded write to a failed node. It follows Cassandra to apply the *hinted handoff* mechanism [4], which allows the replay of a write to a failed node that returns online.

Limitations. ELECT does not support incremental recovery for individual SSTables as in Cassandra. Under replication, Cassandra builds a Merkle tree [41] in each node to detect inconsistencies among replicas for any failed SSTable recovery. Since ELECT includes erasure-coded SSTables in LSM-trees, it needs a revised Merkle tree that addresses both replication and erasure coding.

ELECT also does not currently support dynamic topology changes. We consider a possible approach for supporting topology changes in ELECT as follows. For replicated KV pairs, ELECT can relocate replicas when the nodes join or leave as in Cassandra. For erasure-coded KV pairs, ELECT can relocate some of the erasure-coded SSTables to keep them in consecutive nodes in the hash ring (§4.1.2). As in consistent hashing, ELECT should only relocate the KV pairs stored in the adjacent nodes of each joining/leaving node in the hash ring instead of all SSTables of the whole storage system, so as to mitigate the relocation overhead.

6 Evaluation

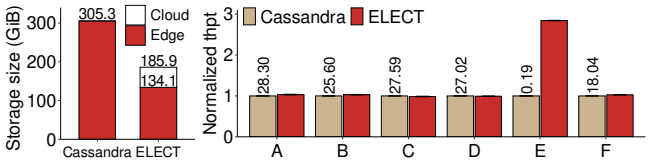
We show via evaluation that ELECT reduces the storage overhead of Cassandra and maintains high performance.

6.1 Methodology

Testbed. We consider an *edge-cloud setting*, where the edge serves as the hot tier and the cloud serves as the cold tier. Specifically, we conduct evaluation on Alibaba Cloud [38]. We set up $M = 10$ edge nodes and multiple (up to 32) client nodes in the same geographical region. Each node is deployed on an `ecs.i3g.2xlarge` instance with eight 2.5 GHz vCPUs, 32 GiB RAM, 447 GiB SSD, and Ubuntu 22.04 LTS. All nodes are connected with a 3 Gbps network, with a network latency of no more than 1 ms. We also deploy the cloud on the Alibaba Object Storage Service in a different geographical region. Our measurement shows that the network latency between the two regions is at least 45 ms.

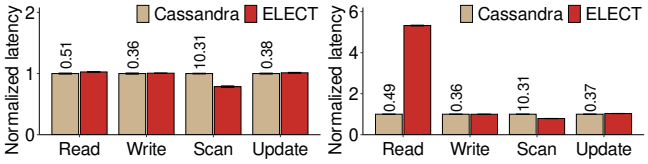
Default settings. We compare ELECT with the vanilla Cassandra. We configure Cassandra with triple replication ($R = 3$) and store all replicas in the edge nodes. We also configure ELECT with triple replication and the $(n,k) = (6,4)$ RS code, and enable all features under a storage saving target $\alpha = 0.6$. For both systems, we fix the SSTable size as 4 MiB [28,64]. We disable SSTable compression for accurate storage size calculation. We set the read and write consistency levels as one and three, respectively (i.e., each of the reads and writes needs to be acknowledged by one node and all three replica nodes, respectively) for strong consistency. All other parameters remain the same as the defaults in Cassandra.

We use the benchmarking tool YCSB [16] to generate different types of workloads. By default, in Exp#1, we load 100 M 1-KiB KV pairs with 24-byte keys and 1000-byte values into storage before each experiment and generate 10 M requests; in the subsequent experiments that focus on examining the performance of individual KV operations, we load 10 M 1-KiB KV pairs and generate 1 M requests, while the performance trends remain stable as in Exp#1 even we use smaller workloads. In all experiments, the requests by default follow a Zipf distribution with a Zipfian constant of 0.99 (default in YCSB). Also, we run YCSB clients in two client nodes, each of which has eight YCSB client threads to simulate concurrent requests from multiple clients.



(a) Storage size (b) Throughput

Figure 5: Exp#1: YCSB core workloads. Throughput results are normalized by Cassandra's throughput (above each bar) in KOPS.



(a) Normal mode (b) Degraded mode

Figure 6: Exp#2: Benchmarking of KV operations. Results are normalized by Cassandra's latencies (above each bar) in milliseconds.

Our experiments consider normal (without failures) and degraded (with failures) modes. For degraded mode, we crash two edge nodes via the `kill -9` command.

We plot the average results over five runs, with error bars showing the 95% confidence intervals under the Student's t-distribution (note that some error bars may be invisible due to small deviations).

6.2 Overall Analysis

Exp#1 (YCSB core workloads). We first compare the overall storage overhead and performance of Cassandra and ELECT using the six YCSB core workloads [16], namely A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), E (95% scans, 5% writes), and F (50% reads, 50% read-modify-writes). Each workload (except D) follows a Zipf distribution, while Workload D reads the latest written KV pairs. For ELECT, we measure both edge-only and overall edge-cloud storage sizes.

Figure 5 shows the storage size and throughput results. ELECT achieves 56.1% storage savings (in the edge only) and 39.1% overall storage savings (in both the edge and cloud) compared with Cassandra. The actual edge storage savings of ELECT are slightly less than $\alpha = 0.6$, as it also maintains metadata components for deleted KV pairs in the secondary LSM-trees after redundancy transitioning. The metadata components of LSM-trees in ELECT account for 6.9% of its edge storage size (not shown in the figure); note that no metadata components are offloaded to the cloud (§4.2).

In terms of performance, both Cassandra and ELECT have similar throughput (with up to 3% differences) in all workloads except E. For Workload E, which is scan-intensive, ELECT achieves a 2.84× throughput gain over Cassandra. The reason is that ELECT reduces individual LSM-tree sizes and hence I/O amplification through decoupled replication management, so the number of SSTables being accessed is also reduced [57,68]. Such reduced access overhead has more prominent performance improvements to scans, which read a range of KV pairs.

Exp#2 (Benchmarking of KV operations). We evaluate the average latencies of specific KV operations, including reads, writes, scans, and updates. We load 10 M 1-KiB KV pairs and issue 1 M requests for each type of KV operations (§6.1). We consider both normal and degraded modes. For degraded mode, we measure the performance of all requests (including normal and degraded requests) when the system is in degraded mode with edge node failures. For example, if a read encounters a non-failed node, it is a normal read; if it encounters a failed node, it becomes a degraded read and ELECT recovers the SSTable that contains the requested key.

Figure 6 shows the results. Note that Cassandra keeps almost identical performance in both normal and degraded modes as it keeps all replicated storage in the edge. In normal mode (Figure 6(a)), ELECT maintains similar performance as in Cassandra in reads, writes, and updates with up to 2.7% of higher average latencies; it reduces the average scan latency of Cassandra by 21.5% (see Exp#1). In degraded mode (Figure 6(b)), ELECT still has similar performance of Cassandra in writes and updates with up to 3.3% higher average latencies and reduces the average scan latency of Cassandra by 21.1%. However, ELECT incurs a latency increase of 5.32× in reads over Cassandra, mainly due to the retrieval of SSTables from the cloud to the edge for recovery if the degraded reads are issued to the KV pairs in the last LSM-tree level.

We further examine the read and scan results in degraded mode. For reads, we observe that both Cassandra and ELECT have very similar 99th-percentile latencies at about 1.7 ms (not shown in the figure), meaning that most reads can be served in the edge and have small latency differences. Some degraded reads need to retrieve SSTables from the cloud, and such requests increase the average read latency in degraded mode. Unlike reads, ELECT still shows performance gains in scans (which include normal and degraded reads to a range of KV pairs) as in normal mode. The reason is that most unavailable SSTables are recovered in the early stage of scans, so the overall adverse impact on scans is much mitigated as opposed to reads.

6.3 System-level Analysis

Exp#3 (Performance breakdown). We break down the performance of writes, reads in normal mode, and reads in degraded mode. Each write comprises (i) writing to the WAL, (ii) writing to the MemTable, (iii) flushing the MemTable, (iv) compaction, (v) redundancy transitioning, and (vi) data offloading. Each read in normal or degraded mode comprises (i) reading from the MemTable, (ii) reading from the block cache, (iii) reading from SSTables, and (iv) recovery (for degraded reads). We measure the time of each step across all nodes and obtain the average results on processing 1 MiB of writes/reads based on the workloads as described in §6.1. Since the steps are performed in parallel, the actual time spent

Steps	Cassandra	ELECT
Write		
WAL	21.32 ± 0.76 ms	21.84 ± 0.28 ms
MemTable	37.98 ± 1.73 ms	40.84 ± 0.13 ms
Flushing	16.95 ± 0.29 ms	17.70 ± 0.18 ms
Compaction	205.87± 2.21 ms	169.03 ± 3.23 ms
Transitioning	-	239.05 ± 2.69 ms
Offloading	-	162.84 ± 12.05 ms
Read in normal mode		
Cache	17.05 ± 0.27 ms	18.35 ± 0.34 ms
MemTable	20.78 ± 0.95 ms	23.20 ± 0.61 ms
SSTables	182.69 ± 2.53 ms	177.55 ± 0.60 ms
Read in degraded mode		
Cache	17.41 ± 0.33 ms	18.75 ± 0.18 ms
MemTable	21.54 ± 0.66 ms	23.38 ± 0.46 ms
SSTables	184.39 ± 1.67 ms	184.14 ± 2.35 ms
Recovery	-	1957.64 ± 34.16 ms

Table 1: Exp#3: Performance breakdown. We show the average latency of each step for processing 1 MiB of writes/reads and the corresponding 95% confidence interval.

on an operation is less than the sum of times of all steps.

Table 1 shows the performance breakdown. Most common steps in Cassandra and ELECT have similar latencies, except for compaction in writes, ELECT has a smaller latency by 17.9%. The reason is that ELECT decouples replication management into multiple LSM-trees and reduces the I/O amplifications in compaction [57, 68]. The redundancy transitioning has the highest average latency among all steps, while the offloading has a low average latency since only a fraction of data is transmitted from the edge to the cloud. However, both redundancy transitioning and data offloading are performed in the background and incur limited overhead to writes, so ELECT has similar write performance as Cassandra (see Exp#1 and Exp#2).

For reads in normal mode, both Cassandra and ELECT have similar performance in each step. For reads in degraded mode, ELECT is bottlenecked by the recovery step (with 1957.64 ms per MiB).

Exp#4 (Full-node recovery). We evaluate full-node recovery in Cassandra and ELECT. We crash one edge node, delete all its data, start a new edge node, and use the `nodetool` command to recover all lost data into the new edge node. We evaluate the recovery performance of different loaded data sizes, including 10 GiB, 20 GiB, and 30 GiB (i.e., 10 M, 20 M, and 30 M 1-KiB KV pairs). For fair comparisons, we disable the Merkle tree operation in Cassandra (which is not supported in ELECT (§5)). Figure 7 shows the full-node recovery times. The recovery times of Cassandra and ELECT increase almost linearly. ELECT incurs about 50% higher recovery time than Cassandra since it needs to retrieve data and parity SSTables from other edge nodes or the cloud to decode the lost SSTables in the primary LSM-tree.

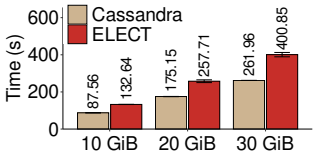


Figure 7: Exp#4: Full-node recovery time.

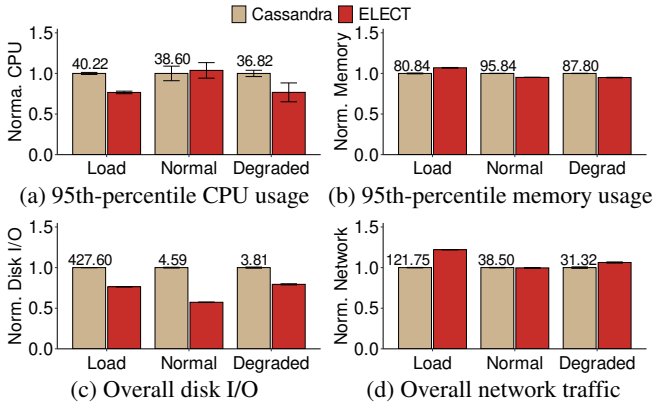
Steps	Time
Copy	13.54± 0.22 s
Retrieve	373.98± 11.61 s
Decode	13.34 ± 0.48 s

Table 2: Exp#4: Recovery time breakdown for 30 GiB data.

We further provide a breakdown of the full-node recovery time of ELECT into three steps: (i) copying replicated SSTables from other edge nodes; (ii) retrieving data and parity SSTables for decoding; and (iii) decoding the lost SSTables. Table 2 shows the full-node recovery time breakdown of ELECT for 30 GiB data. The recovery performance is network-bound, in which retrieving the data and parity SSTables occupies 93.3% of the total recovery time.

Exp#5 (Resource usage). We compare the CPU usage, memory usage, disk I/O size, and network traffic of Cassandra and ELECT. We consider three settings: (i) loading KV pairs until the SSTables reach a stable state, (ii) running in normal mode without failures, and (iii) running in degraded mode with two node failures. After loading KV pairs, we issue 1 M requests, including reads, writes, updates, and scans, with one-fourth of all requests each. We measure the CPU usage, memory usage, disk I/O, and network usage in all alive edge nodes through the Linux system tools, namely `top`, `free`, `iostat`, and `iftop`, respectively. We collect the resource usage data every 1 s. We show the 95th-percentile CPU usage and memory usage as well as the overall disk I/O size and network traffic size.

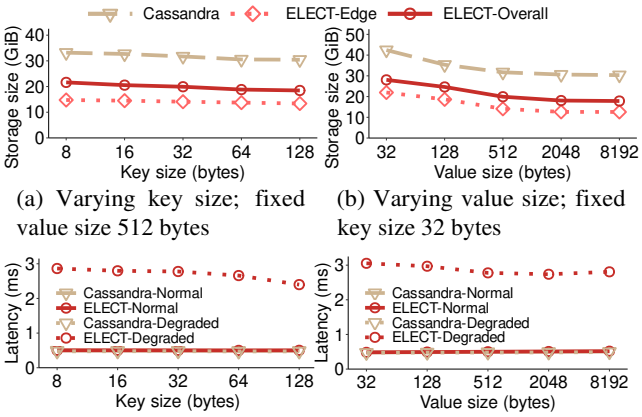
Figure 8 shows the results. Figure 8(a) shows that ELECT has 23.4% and 23.3% less 95th-percentile CPU usage than Cassandra in load and degraded operations, respectively. ELECT mainly performs network transmissions in redundancy transitioning and data offloading, both of which involve less CPU usage (in each 1-second interval). However, ELECT has long durations in both redundancy transitioning and data offloading (Table 1), so its total CPU time is still higher than Cassandra's. Figure 8(b) shows that ELECT slightly increases the 95th-percentile memory usage by 7.0% during the load operation, since it needs extra memory space for erasure coding. It also slightly reduces the memory usage in normal and degraded modes by 4.9% and 5.1%, respectively, as it reduces the LSM-tree size through decoupled replication management. Figure 8(c) shows that ELECT reduces the disk I/O size of Cassandra by 23.6% in the load operation, as it reduces the LSM-tree size and hence I/O amplifications through decoupled replication management, and also reduces the compaction overhead of the secondary LSM-trees by writing fewer KV pairs to the last LSM-tree level. Note that ELECT still has higher disk I/O size in degraded mode than in normal mode due to the reconstruction overhead in erasure coding. Figure 8(d) shows that ELECT has similar network traffic to Cassandra in normal mode, while



(a) 95th-percentile CPU usage (b) 95th-percentile memory usage

(c) Overall disk I/O (d) Overall network traffic

Figure 8: Exp#5: Resource usage. All results are normalized by Cassandra's actual resource usage results (numbered atop the bars), including CPU usage (%), memory usage (GiB), disk I/O size (GiB), and network traffic (GiB).



(a) Varying key size; fixed value size 512 bytes (b) Varying value size; fixed key size 32 bytes

(c) Reads versus key size (d) Reads versus value size

Figure 9: Exp#6: Impact of key and value sizes.

it incurs 22.1% and 6.3% higher network traffic than Cassandra in load and degraded operations, respectively. In the load operation, ELECT distributes SSTables for redundancy transitioning and offloads SSTables to the cloud; in degraded mode, it retrieves SSTables to recover unavailable SSTables.

6.4 Parameter Sensitivity Analysis

Exp#6 (Impact of key and value sizes). We evaluate the impact of different key and value sizes on Cassandra and ELECT to show that ELECT still maintains storage savings for different key/value sizes. Note that we fix the total size of KV pairs loaded to storage as 10 GiB, so the total number of KV pairs decreases as the key size or value size increases.

Figures 9(a) and 9(b) show the actual storage sizes of Cassandra and ELECT for various key sizes with a fixed value size 512 bytes and various value sizes with a fixed key size 32 bytes, respectively. The actual storage sizes of both systems decrease as the key and value sizes increase. The edge and overall storage savings of ELECT over Cassandra increase from 55.5% to 56.0% and from 34.9% to 39.3%, respectively, as the key size increases from 8 bytes to 128 bytes, and from 48.2% to 58.5% and from 33.9% to 41.1%, respectively, as the value size increases from 32 bytes to 8 KiB. The reason is that larger key and value sizes reduce the amount of metadata for SSTable maintenance. This reduces the storage overhead after redundancy transitioning in ELECT.

Figures 9(c) and 9(d) show the average read latencies in normal and degraded modes for various key and value sizes. The read latencies of Cassandra and ELECT in normal mode and the read latency of Cassandra in degraded mode are similar as the KV pairs can be directly accessed. However, ELECT has much higher average read latencies in degraded mode than Cassandra, especially for small key and value sizes (e.g., 6.4× when the key and value sizes are both 32 bytes), since smaller key and value sizes increase the number of KV pairs stored in an SSTable and hence the query overhead.

Exp#7 (Impact of storage saving target). We evaluate the impact of target storage saving α on ELECT, by varying α from 0.1 to 0.9. Our results demonstrate the trade-off between storage savings and access performance.

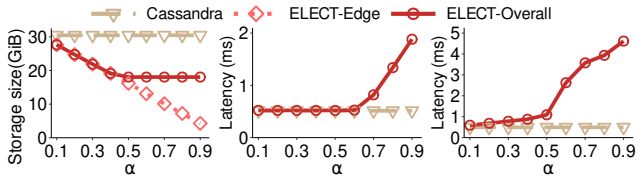
Figure 10(a) shows the edge-only and overall storage sizes of ELECT. As α increases, the edge storage savings over Cassandra increase from 9.2% to 86.0%, and differ from α by no more than 4%. ELECT offloads SSTables from the edge to the cloud when $\alpha \geq 0.5$, yet the overall storage size (in both the edge and cloud) of ELECT remains unchanged when $\alpha \geq 0.5$ and its savings over Cassandra stay at 40.8%. The reason is that redundancy transitioning only applies to the SSTables in the last LSM-tree level and cannot further reduce the storage sizes of replicated SSTables in the lower LSM-tree levels.

Figure 10(b) shows the average read latencies in normal mode. The read latency of ELECT remains stable as α increases from 0.1 to 0.6, but increases significantly from 0.53 ms to 1.89 ms when α increases from 0.6 to 0.9. The reason is that after offloading data SSTables to the cloud (Case 3 in §4.3), reads to the primary LSM-tree retrieve data SSTables back from the cloud and are slowed down by edge-cloud communication. We pose the parameter sensitivity analysis for 99th-percentile latencies as future work.

Figure 10(c) shows the average read latencies in degraded mode. As α increases from 0.1 to 0.5, the average latency of ELECT increases from 0.59 ms to 1.09 ms, as more SSTables are involved in redundancy transitioning and degraded reads trigger more decoding operations. As α further increases, ELECT starts to offload data SSTables to the cloud, and the degraded reads are further slowed down due to the retrieval of data SSTables from the cloud for decoding. When $\alpha = 0.9$, the average latency increases to 4.62 ms.

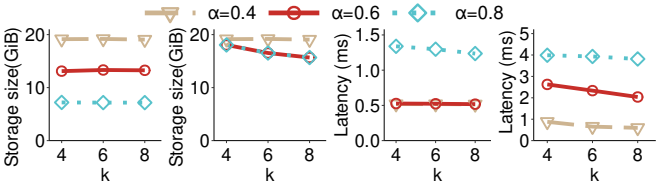
Exp#8 (Impact of coding parameters). We study the impact of coding parameters on ELECT by varying k and fixing $n = k+2$. We also consider different values of α. We focus on the edge and overall storage sizes as well as the average read latencies in normal and degraded modes.

Figure 11 shows the results. For a fixed α, even with



(a) Storage size (b) Reads in normal mode (c) Reads in degraded mode

Figure 10: Exp#7: Impact of storage saving target. Note that the results for Cassandra remain unaffected by α but are included in the plots for comparisons.



(a) Edge storage size (b) Overall storage size (c) Reads in normal mode (d) Reads in degraded mode

Figure 11: Exp#8: Impact of coding parameters.

different values of k, ELECT still maintains similar edge storage sizes with no more than 1.7% differences (Figure 11(a)). Although a larger k (with the fixed $n-k$) implies smaller redundancy, the storage saving target α also determines the actual edge storage size. Thus, the edge storage size remains almost unaffected by different values of k for a fixed α.

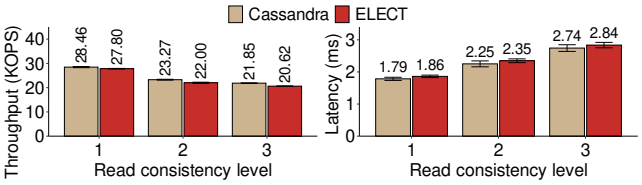
The overall storage size (Figure 11(b)) of ELECT drops from 18.0 GiB to 15.7 GiB when k increases from 4 to 8, for both $\alpha = 0.6$ and $\alpha = 0.8$, as a larger k generates fewer parity SSTables and reduces the overall storage size for a large α; note that the overall storage size remains unaffected for different values of k for $\alpha = 0.4$.

For a fixed α, the reads latencies in normal and degraded modes are also similar for different values of k, albeit slight latency decreases in degraded mode as k increases (Figures 11(c) and 11(d)). Intuitively, a larger k implies higher reconstruction overhead, as k SSTables are retrieved for reconstruction in RS codes (§2.3). On the other hand, a larger k also implies that fewer parity SSTables are generated and offloaded to the cloud, and hence a degraded read retrieves fewer parity SSTables from the cloud. This leads to slightly improved read performance in degraded mode for a large k.

This experiment aims to show the applicability of ELECT for different values of k. A more detailed analysis on the trade-off between α and k is our future work.

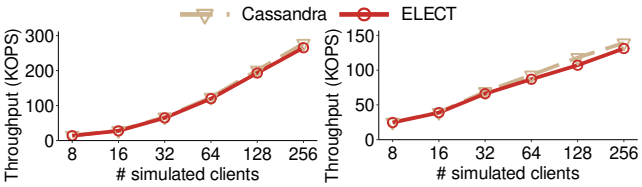
Exp#9 (Impact of read consistency level). We show how ELECT preserves consistent reads in Cassandra (§5). We vary the read consistency level from one to three, while the write consistency level remains three (under triple replication). We focus on the throughput and 99th-percentile latency of normal reads; for the latter, we show the impact of waiting for responses from multiple replicas.

Figure 12 shows the results versus the read consistency level for both Cassandra and ELECT. As the read consistency level increases from one to three (i.e., each read needs to wait for the responses from more replicas), the average read throughput of both systems (Figure 12(a)) decreases by 23.2% for Cassandra and 25.8% for ELECT, while the 99th-percentile read latencies for both systems (Figure 12(b)) increase by around 50%. The results suggest that ELECT maintains similar read performance as in Cassandra under consistent reads.



(a) Read throughput (b) 99th-percentile latency

Figure 12: Exp#9: Impact of read consistency level.



(a) Reads in normal mode (b) Writes in normal mode

Figure 13: Exp#10: Impact of number of clients.

Exp#10 (Impact of number of clients). We examine the read/write performance of ELECT in normal mode as the number of clients increases. We vary the number of client nodes (deployed in different instances) from 1 to 32, while each client node runs eight client threads, so the maximum number of simulated clients reaches 256. Each simulated client issues 100 K KV requests.

Figure 13 shows the throughput versus the number of simulated clients for both Cassandra and ELECT. Both systems show similar increasing throughput trends as the number of simulated clients increases. ELECT has slightly less throughput than Cassandra, by 4% in normal reads and 5.7% in writes when the number of simulated clients is 256, due to the redundancy transitioning overhead.

6.5 Discussion

We discuss the performance of ELECT in other aspects that are currently not explicitly evaluated.

Varying skewness in workloads. We currently focus on skewed workloads as observed in practice [6, 9, 16, 62]. With less skewed workloads, reads access larger portions of the key space. Thus, more reads are issued to the last LSM-tree level, and ELECT may see performance drops in degraded mode as it applies erasure coding to the last LSM-tree level. Note that ELECT does not directly determine the hotness of KV pairs by monitoring their access patterns, while the read and write patterns may have different distributions [6, 62]. Thus, the actual performance of ELECT may be greatly affected by the real-world access patterns.

Varying LSM-tree sizes. ELECT supports different LSM-tree sizes as it still encodes SSTables in the last LSM-tree level. It is expected to achieve higher storage savings for larger LSM-trees since the number of SSTables increases exponentially across LSM-tree levels and the last LSM-tree level contains more SSTables.

Impact on reliability. The increase in the recovery time of ELECT (Exp#4) degrades reliability (e.g., in terms of mean-time-to-data-loss (MTTDL)). On the other hand, since ELECT offloads some erasure-coded KV pairs to the cold tier, which is in general more reliable than the hot tier (e.g., when edge nodes serve as the hot tier versus the cloud serves as the cold tier in edge-cloud storage), the reliability can be improved. The actual impact on reliability due to redundancy transitioning remains an open issue.

Impact of LSM-tree compression. Our evaluation disables compression to fairly measure ELECT's storage savings. ELECT still works with compression enabled and is expected to achieve storage savings. Since Cassandra performs compression on SSTables, ELECT can collect k compressed data SSTables and pad them with zeroes to match the maximum size of the k compressed data SSTables, so as to generate parity SSTables via erasure coding. Note that such padded zeros are only for erasure coding compatibility. They need not be stored in data SSTables and will not add storage overheads.

Comparisons with CassandrEAS [8]. CassandrEAS also extends Cassandra with erasure coding, but does not consider redundancy transitioning. CassandrEAS reportedly has much higher read and write latencies than Cassandra [8]. Our evaluation also finds that CassandrEAS (based on its open-source version) incurs high storage overhead for small values due to extra metadata for erasure coding (e.g., 1.6× storage overhead for 24-byte keys and 64-byte values compared with 3-way replication in Cassandra).

7 Related Work

Replication in distributed KV stores. Replication is commonly used in modern distributed KV stores [5, 18, 34, 46]. Several studies propose new replica management mechanisms to support high-throughput and strongly consistent writes [56], reduce data loss rates [15], improve query performance [22, 54], and reduce I/O amplification in LSM-tree compaction [57, 68]. In particular, ELECT has the similar ideas of DEPART [68] and Tebis [57] to separate the LSM-tree management for replicas, but focuses on synchronizing the views of replicas for efficient redundancy transitioning. Both DEPART and Tebis do not consider erasure coding.

Erasure coding in distributed KV stores. Erasure coding has been extensively used in distributed KV stores. Some studies apply replication for keys and metadata as well as erasure coding for values for persistent [8, 32, 33, 44] and in-memory [13, 37] KV stores. Some approaches [14, 65] apply erasure coding across whole objects (including keys, values, and metadata) for further storage savings, but they are applied for in-memory KV stores and their performance is guaranteed with memory access. Erasure coding is also recently explored for disaggregated memory [35, 69].

In the context of storage tiering, EC-Cache [49] and C2DN [61] also apply erasure coding to caching and content delivery networks, respectively. EC-Cache performs self-encoding on large objects and keeps erasure-coded chunks across cache servers, while C2DN replicates small objects and applies erasure coding to large objects. We emphasize that ELECT differs from EC-Cache and C2DN in both problem formulation and design techniques. Regarding problem formulation, EC-Cache and C2DN aim for load balancing using erasure coding for high performance, while ELECT considers redundancy transitioning (from replication to erasure coding) for storage savings. Regarding design techniques, EC-Cache and C2DN are centralized (EC-Cache manages cache servers with a centralized coordinator, while C2DN uses a cluster-local load balancer), while ELECT performs decentralized parity node selection (§4.1.2). ELECT also addresses selective data offloading (§4.2) and configurable storage savings (§4.3), both of which are not addressed by EC-Cache and C2DN.

Redundancy transitioning. Earlier studies consider the transitioning between replication and erasure coding on fixed-size blocks in RAID [58] and distributed file systems [23, 36], while ELECT considers variable-size KV pairs. Furthermore, ELECT builds on Cassandra, a decentralized KV store, while the above studies [23, 36, 58] are centralized. Some studies consider the transitioning between erasure codes with different coding parameters to trade between performance and redundancy overhead [55, 59, 60, 63] or between reliability and redundancy overhead [29, 30]. Convertible codes [39] are new erasure codes that minimize the transitioning I/O. ELECT applies redundancy transitioning from replication to erasure coding to balance the storage-performance trade-off.

8 Conclusions

We design ELECT, a distributed KV store that enables erasure coding tiering, to make a case for storage-efficient, high-performance, and fault-tolerant KV storage. ELECT adopts a hybrid redundancy approach by replicating hot KV pairs and erasure-coding cold KV pairs. It also selectively offloads them from the hot tier to the cold tier. It is tunable with a single storage saving target parameter to balance the trade-off between storage savings and access performance. Experiments on Alibaba Cloud demonstrate the storage savings and performance efficiency of ELECT compared with Cassandra.

Acknowledgements. We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their comments. This work was supported in part by National Key R&D Program of China (2022YFB4501200), Key Research and Development Program of Hubei Province (2021BAA189), National Natural Science Foundation of China (61972073, 62302175, and 62332004), and Research Grants Council of Hong Kong (GRF 14214622 and AoE/P-404/18).

References

[1] Amazon. Amazon Elastic Block Store. https://aws.amazon.com/ebs/.

[2] Amazon. Amazon Simple Storage Service. https://aws.amazon.com/s3/.

[3] Apache. Cassandra 4.1.0. https://github.com/apache/cassandra/releases/tag/cassandra-4.1.0.

[4] Apache. Cassandra documentation - hints. https://cassandra.apache.org/doc/4.1/cassandra/operating/hints.html.

[5] Apache. HBase. https://hbase.apache.org/.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. of ACM SIGMETRICS*, 2012.

[7] Burton Howard Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 12(7):422--426, 1970.

[8] Viveck R. Cadambe, Kishori M. Konwar, Muriel Medard, Haochen Pan, Lewis Tseng, and Yingjian Wu. CassandrEAS: Highly available and storage-efficient distributed key-value store with erasure coding. In *Proc. of NCA*, 2020.

[9] Zhichao Cao and Siying Dong. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proc. of USENIX FAST*, 2020.

[10] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, 2014.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI*, 2006.

[12] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. In *Proc. of IEEE GLOBECOM*, 2020.

[13] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Trans. on Storage*, 13(3):25, 2017.

[14] Liangfeng Cheng, Yuchong Hu, and Patrick P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SOCC*, 2019.

[15] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of USENIX ATC*, 2013.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SOCC*, 2010.

[17] Couchbase. Couchbase automated data partitioning. https://www.couchbase.com/blog/what-exactly-membase/.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP*, 2007.

[19] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539--4551, 2010.

[20] Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In *Proc. of DISC*, 2008.

[21] EdgeKV. Augment image and video manager with edge compute. https://techdocs.akamai.com/edgekv/docs/augment-image-and-video-manager-with-edge-compute.

[22] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM*, 2012.

[23] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth A. Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing. *Technical Report, CMU-PDL-11-112, Carnegie Mellon University Parallel Data Laboratory*, 2011.

[24] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[25] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of ACM SOSP*, 2003.

[26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX ATC*, 2012.

[27] Intel. Intelligent storage acceleration library. https://github.com/intel/isa-l.

[28] Jeeyoon Jung and Dongkun Shin. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *Proc. of ACM HotStorage*, 2022.

[29] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-adaptive redundancy without placement restrictions. In *Proc. of USENIX OSDI*, 2022.

[30] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *Proc. of USENIX OSDI*, 2020.

[31] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.

[32] Kishori M. Konwar, N. Prakash, Erez Kantor, Nancy Lynch, Muriel Médard, and Alexander A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In *Proc. of IEEE IPDPS*, 2016.

[33] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, 2015.

[34] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35--40, 2010.

[35] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra: Resilient and highly available remote memory. In *Proc. of USENIX FAST*, 2022.

[36] Runhui Li, Yuchong Hu, and Patrick P. C. Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Trans. on parallel and distributed systems*, 28(9):2500--2513, 2017.

[37] Witold Litwin, Rim Moussa, and Thomas Schwarz. LH*RS---a highly-available scalable distributed data structure. *ACM Trans. on Database System*, 30(3):769--811, 2005.

[38] Alibaba Cloud Computing Co. Ltd. Alibaba cloud. `https://www.alibabacloud.com/`.

[39] Francisco Maturana and K. V. Rashmi. Convertible codes: Enabling efficient conversion of coded data in distributed storage. *IEEE Trans. on Information Theory*, 68(7):4392 -- 4407, 2022.

[40] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. FogStore: Toward a distributed data store for fog computing. In *Proc. of IEEE FWC*, 2017.

[41] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, 1987.

[42] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal De Lara. CloudPath: A multi-tier cloud computing framework. In *Proc. of ACM/IEEE SEC*, 2017.

[43] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.

[44] Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori Konwar, Muriel Medard, and Nancy Lynch. ARES: Adaptive, reconfigurable, erasure coded, atomic storage. In *Proc. of IEEE ICDCS*, 2019.

[45] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351--385, 1996.

[46] PingCAP. TiKV. `https://tikv.org`.

[47] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, 2002.

[48] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[49] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, 2016.

[50] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300--304, 1960.

[51] David Reinsel. How you contribute to today's growing datasphere and its enterprise impact. `https://blogs.idc.com/2019/11/04/how-you-contribute-to-todays-growing-datasphere-and-its-enterprise-impact/`, Nov 2019.

[52] Mahadev Satyanarayanan. The emergence of edge computing. *IEEE Computer*, 50(1):30--39, 2017.

[53] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637 -- 646, 2016.

[54] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *Proc. of USENIX ATC*, 2016.

[55] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, 2018.

[56] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of USENIX OSDI*, 2004.

[57] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: index shipping for efficient replication in lsm key-value stores. In *Proc. of EuroSys*, 2022.

[58] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108--136, 1996.

[59] Si Wu, Zhirong Shen, and Patrick P. C. Lee. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed-Solomon codes. In *Proc. of IEEE SRDS*, 2020.

[60] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *Proc. of USENIX FAST*, 2015.

[61] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *Proc. of USENIX NSDI*, 2022.

[62] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[63] Qiaori Yao, Yuchong Hu, Liangfeng Cheng, Patrick P. C. Lee, Dan Feng, Weichun Wang, and Wei Chen. StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *Proc. of IEEE ICDCS*, 2021.

[64] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. GearDB: A GC-free Key-Value store on HM-SMR drives with gear compaction. In *Proc. of USENIX FAST*, 2019.

[65] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, 2017.

[66] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, 2016.

[67] Heng Zhang, Shaoyuan Huang, Mengwei Xu, Deke Guo, Xiaofei Wang, Victor C. M. Leung, and Wenyu Wang. How far have edge clouds gone? a spatial-temporal analysis of edge network latency in the wild. In *Proc. of IEEE/ACM IWQoS*, 2023.

[68] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, and Si Wu. DEPART: Replica decoupling for distributed key-value storage. In *Proc. of USENIX FAST*, 2022.

[69] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-tolerant far memory. In *Proc. of USENIX OSDI*, 2022.

A Artifact Appendix

Abstract

ELECT is a distributed KV store that enables erasure coding tiering based on the LSM-tree. It adopts a hybrid redundancy approach that carefully combines replication and erasure coding with respect to the LSM-tree layout. Its prototype builds on Cassandra.

Scope

Our artifact can be used to validate the concepts and designs of ELECT presented in the paper. It is a research-driven prototype and has several limitations, such as the inability to support dynamic topology changes and incremental recovery, that restrict its direct applicability in production.

Contents

The artifact consists of two sub-directories:

- `src/`, which includes both the implementation of the ELECT prototype and a simple object storage backend, such that ELECT can be connected with the object storage backend in a local cluster; and
- `scripts/`, which includes both the evaluation scripts and the YCSB benchmarking tool, such that the key and value sizes are configurable.

The artifact also contains a README file that specifies the prerequisites for the testbed and dependencies, steps for building and configuring the ELECT prototype and YCSB benchmark tool, and detailed instructions for artifact evaluation.

Hosting

The artifact is accessible from GitHub at `https://github.com/adslabcuhk/elect`. The version we provided for the artifact evaluation is marked with the `v1.0` tag.

Requirements

Hardware dependencies

To successfully run the end-to-end experiments with our prototype and evaluation scripts, a minimum of eight machines are recommended. These machines need to be connected via a network, such that they are reachable from each other. For each machine, we recommend quad-cores, 16 GiB of memory and above, and an SSD. We need at least six machines that form the distributed KV store ELECT and use the default erasure coding parameters (n,k)=$(6,4)$. In addition, we have one machine that acts as a server node for storing cold data in the cold tier, and one machine for running the YCSB benchmark tool.

Software dependencies

Our artifact is developed and tested on Ubuntu 22.04 LTS with the following software dependencies:

- The ELECT prototype and YCSB benchmark tool: `openjdk-11-jdk`, `openjdk-11-jre`, `ant`, `ant-optional`, `maven`.
- Erasure coding: `clang`, `llvm`, `libisal-dev`.
- Evaluation scripts: `ansible`, `bc`, `python3`, `python3-pip`, `cassandra-driver`, `numpy`, `scipy`.

Testbed Setup

Please follow the steps below:

- Download the artifact from the URL: `https://github.com/adslabcuhk/elect/releases`.
- Extract the files using `tar -zxvf elect-1.0.tar.gz` and navigate into the package directory with `cd`.
- Modify the `scripts/settings.sh` file according to the `AE_INSTRUCTION.md`.
- Set up the machines with the provided scripts via `bash scripts/setup.sh full` (the setup takes about 40 minutes, depending on the hardware configurations).

For the detailed setup, configuration instructions, and troubleshooting, please refer to the `README.md` in the artifact repository. The `README.md` file provides comprehensive instructions on the manual setup process and the solutions to some common issues.

Evaluation

Artifact Claims

The performance results may vary from those in our paper due to different factors, such as cluster sizes, machine specifications, operating systems, and software packages. However, we expect that ELECT still demonstrates comparable performance to Cassandra in regular operations, while significantly reducing hot-tier storage overhead.

Experiments

To reproduce the results presented in the paper, please refer to the `AE_INSTRUCTION.md` file and follow the instructions provided in the `Evaluation` section.

Exp#1 (YCSB core workloads). *Expected outcome:* Exp#1 produces the results as shown in Figure 5, which illustrates that ELECT achieves similar performance as Cassandra in YCSB core workloads while significantly reducing the hot-tier storage overhead. In addition, ELECT outperforms Cassandra in workload E, which consists of 95% scan operations, due to replication decoupling. *Approximate runtime:* 20 compute hours.

Exp#2 (Benchmarking of KV operations). *Expected outcome:* Exp#2 produces the results as shown in Figure 6, which illustrates that ELECT achieves similar performance as Cassandra in common KV operations. Similar to Exp#1, ELECT still outperforms Cassandra in the scan operations due to replication decoupling. *Approximate runtime:* 5 compute hours.

Exp#3 (Performance breakdown). *Expected outcome:* Exp#3 produces the results as shown in Table 1, which illustrates that ELECT has similar latencies in most common steps as in Cassandra. *Approximate runtime:* 5 compute hours.

Exp#4 (Full-node recovery). *Expected outcome:* Exp#4 produces the results as shown in Figure 7 and Table 2, which illustrates that ELECT incurs medium recovery overhead due to the need for retrieving data and parity SSTables from other nodes or the cold tier. *Approximate runtime:* 14 compute hours.

Exp#5 (Resource usage). *Expected outcome:* Exp#5 produces the results as shown in Figure 8, which illustrates that ELECT only increases the memory and network usage when loading data due to redundancy transitioning and cold-data offloading. In addition, for CPU usage, the 95%-percentile CPU utilization will be less than Cassandra since the redundancy transitioning and cold-data offloading consist of a large amount of network transmission with long duration. *Approximate runtime:* 5 compute hours.

Exp#6 (Impact of key and value sizes). *Expected outcome:* Exp#6 produces the results as shown in Figure 9, which illustrates that ELECT still maintains storage savings for different key/value sizes. *Approximate runtime:* 40 compute hours.

Exp#7 (Impact of storage saving target). *Expected outcome:* Exp#7 produces the results as shown in Figure 10, which illustrates that ELECT can balance the storage overhead and performance according to the storage saving target. *Approximate runtime:* 45 compute hours.

Exp#8 (Impact of coding parameters). *Expected outcome:* Exp#8 produces the results as shown in Figure 11, which illustrates that ELECT can adapt to different erasure coding parameters. *Approximate runtime:* 12 compute hours.

Exp#9 (Impact of read consistency level). *Expected outcome:* Exp#9 produces the results as shown in Figure 12, which illustrates that ELECT supports consistent reads. *Approximate runtime:* 5 compute hours.

Exp#10 (Impact of number of clients). *Expected outcome:* Exp#10 produces the results as shown in Figure 13, which illustrates that ELECT supports multi-client KV operations. *Approximate runtime:* 5 compute hours.