DEPART: Replica Decoupling for Distributed Key-Value Storage

Qiang Zhang[1], Yongkun Li[1], Patrick P. C. Lee[2], Yinlong Xu[1,3], Si Wu[1]
[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*

Abstract

Modern distributed key-value (KV) stores adopt replication for fault tolerance by distributing replicas of KV pairs across nodes. However, existing distributed KV stores often manage all replicas in the same index structure, thereby leading to significant I/O costs beyond the replication redundancy. We propose a notion called *replica decoupling*, which decouples the storage management of the primary and redundant copies of replicas, so as to not only mitigate the I/O costs in indexing, but also provide tunable performance. In particular, we design a novel two-layer log that enables tunable ordering for the redundant copies to achieve balanced read/write performance. We implement a distributed KV store prototype, DEPART, atop Cassandra. Experiments show that DEPART outperforms Cassandra in all performance aspects under various consistency levels and parameter settings. Specifically, under the eventual consistency setting, DEPART achieves up to 1.43×, 2.43×, 2.68×, and 1.44× throughput for writes, reads, scans, and updates, respectively.

1 Introduction

Key-value (KV) stores serve as essential building blocks in the storage infrastructure of modern data-intensive applications, such as web search [14, 31], social networking [57], photo stores [10], and cloud storage [25,37]. To support large-scale usage, KV stores are often deployed in a *distributed* manner by storing the data objects (in the form of KV pairs) across multiple nodes. Examples of distributed KV stores include BigTable [14], HBase [3], Dynamo [25], HyperDex [28], Cassandra [37], TiKV [50], and Riak [54].

Failures become prevalent in any large-scale deployment, so providing fault tolerance for distributed KV stores is critical. Replication remains the commonly used fault tolerance mechanism in modern distributed KV stores (including the examples listed above [3, 14, 25, 28, 37, 50, 54]). Specifically, for each KV pair issued by a user write, replication makes multiple exact copies (called *replicas*) and distributes the replicas across different nodes, so as to tolerate any node failure.

One subtlety is that each node internally stores all replicas in the same index structure; we call such an approach *uniform indexing*. For example, we have examined the codebases of various open-source distributed KV stores, including HBase [3], HyperDex [28], Cassandra [37], TiKV [50], and ScyllaDB [60], and they all adopt uniform indexing for replica management. In particular, they keep all replicas originated from different nodes in a *log-structured-merged tree (LSM-tree)* [48], a multi-level tree structure that supports efficient reads and writes of KV pairs and maintains sorted KV pairs in each level for efficient scans (or range queries) to consecutive ranges of KV pairs. They either build on local LSM-tree KV stores (e.g., HyperDex uses HyperLevelDB [27] and TiKV uses RocksDB [29]), or implement their own LSM-tree structures (e.g., in HBase and Cassandra).

Uniform indexing is simple to implement for replica management, but it also significantly degrades both the write and read performance. First, the LSM-tree performs frequent compaction operations that rewrite the currently stored KV pairs to maintain their sorted order in each level. Storing all replicas in the same LSM-tree exacerbates the write amplification beyond the replication redundancy. For example, when replication is disabled, the write amplifications of Cassandra and TiKV are 6.5× and 13.8×, respectively; however, under triple replication, the write amplifications reach 25.7× and 50.9× in Cassandra and TiKV, respectively, incurring more than three times in write amplification (§3.1). Also, as reading a KV pair needs to search multiple levels in the LSM-tree, uniform indexing amplifies the search space and exacerbates the read amplification as well. For example, under triple replication, the read amplification of Cassandra reaches 34.6× (§3.1).

Our insight is that instead of putting all replicas in the same index structure, if we use different index structures for managing the storage of different types of replicas, we not only mitigate the read/write amplifications by reducing the size of the index structure for each type of replicas, but also enable flexible storage management to adapt to different design trade-offs. We make a case by proposing *replica decoupling*, which decouples the storage management of the replicas of each KV pair based on the *primary copy* (i.e., the main replica of the KV pair) and the *redundant copies* (i.e., the remaining replicas of the KV pair aside the primary copy). We use the LSM-tree to manage the primary copies only, so as to preserve the design features of the LSM-tree but in a more lightweight manner; meanwhile, we use simpler but tunable index structures for the redundant copies to balance the read and write performance depending on the performance requirements.

In this paper, we design replica decoupling in DEPART, a novel distributed KV store that decouples the storage management of primary and redundant copies for fault tolerance. DEPART builds on Cassandra [37]. It supports lightweight

differentiation of the primary and redundant copies of replicas on the critical I/O path, while keeping the existing data organization and configurable consistency features of Cassandra. While managing the primary copies in the LSM-tree, DEPART proposes a novel *two-layer log* to manage the redundant copies with tunable ordering for balanced read and write performance. Its idea is to issue batched writes for the redundant copies into an append-only *global log* for high write performance. It further splits the global log into multiple *local logs*. In particular, the ordering of KV pairs in each local log is tunable by a *single* parameter to balance the read and write performance for the redundant copies; for example, given a high read (or write) consistency level (i.e., the number of replicas to be read (or written) in a successful operation; see §2.3), the two-layer log can be tuned to favor for high read (or write) performance. The two-layer log also improves failure recovery performance, by organizing the KV pairs by different key ranges and limiting a recovery operation to access only the relevant range of KV pairs. Our contributions are summarized as follows.

- We analyze two state-of-the-art distributed KV stores, Cassandra and TiKV, and reveal their performance limitations due to uniform indexing for replicas.
- We design DEPART, which realizes replica decoupling and has several key design features: (i) lightweight differentiation of primary and redundant copies, (ii) a two-layer log design with tunable ordering of redundant copies, and (iii) a fast failure recovery implementation via parallelization.
- We implement DEPART atop Cassandra v3.11.4 [2]. Experiments show that DEPART outperforms Cassandra in various settings. For example, for the case of eventual consistency, DEPART achieves 1.43×, 2.43×, 2.68×, and 1.44× throughput gains over Cassandra in writes, reads, scans, and updates, respectively. DEPART also maintains its read and write performance gains under various consistency level configurations.

The source code of our DEPART prototype is available at: `https://github.com/ustcadsl/depart`.

2 Background

We use Cassandra [37] (which serves as the baseline for our DEPART design) as an example to describe the background of a distributed KV store, including its storage architecture, I/O workflows, and consistency management.

2.1 Storage Architecture

Data organization. A distributed KV store partitions KV pairs across a cluster of nodes. In Cassandra, the KV pairs are partitioned based on *consistent hashing* [33], which has also been adopted by other production distributed KV stores [25, 41, 54, 60]. Consistent hashing associates the locations of all nodes with a *hash ring* and maps each KV pair deterministically to a node. Specifically, we consider a distributed

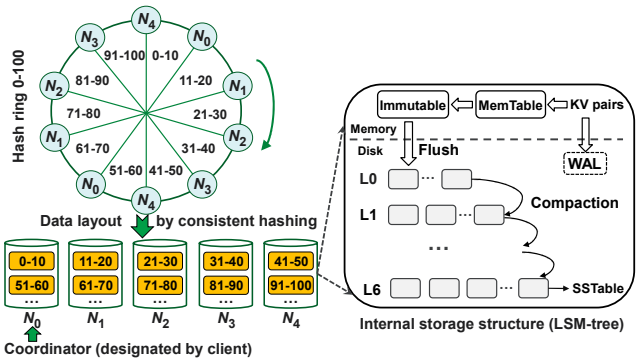


Figure 1: Storage architecture in Cassandra.

KV store with *n physical nodes*, each of which is associated with *v virtual nodes*. It divides the hash ring into $n \times v$ *ranges*, each of which covers one of the virtual nodes. For example, as shown in Figure 1, there are $n = 5$ physical nodes (i.e., N_0 to N_4) with $v = 2$ virtual nodes each. The hash ring contains $2 \times 5 = 10$ ranges, say $(0-10)$, $(11-20)$, $\cdots$, $(91-100)$. Each of the ranges is associated with the nearest virtual node in the clockwise direction in the hash ring and the corresponding physical node; for example, both ranges $(0-10)$ and $(51-60)$ are assigned to N_0. For each KV pair, consistent hashing hashes the key to a location in the hash ring (e.g., using MurmurHash [6] in Cassandra). The KV pair is then stored in the corresponding physical node that is associated with the range.

Replication is commonly used in modern distributed KV stores [3, 14, 25, 28, 37, 50] for fault tolerance, by distributing the replicas of each KV pair across different nodes to protect against node failures. In Cassandra, replicas are stored in a sequence of nodes along the clockwise direction in the hash ring denoted by $N_i, N_{(i+1) \bmod n}, N_{(i+2) \bmod n}, \cdots$, where $0 \le i \le n-1$ and N_i (i.e., the first node in the node sequence) is the node to which the KV pair is hashed based on consistent hashing. We refer to the replica that is stored in N_i as the *primary copy*, while referring to the remaining replicas that are stored in the successive physical nodes along the clockwise direction in the hash ring as the *redundant copies*.

Internal storage with the LSM-tree. Each node internally manages KV pairs with some index structure. In particular, the LSM-tree [48] is one of the most commonly used index structures in distributed KV stores, including Cassandra and others [3, 20, 28, 41, 50, 60]. An LSM-tree KV store organizes KV pairs in a multi-level tree and keeps KV pairs sorted by keys in each level, so as to support efficient reads, writes, and scans. As shown in Figure 1, the LSM-tree KV store maintains a tree-based index structure with multiple levels (denoted by $L_0, L_1, \cdots$) with an increasing capacity, in which each level stores the KV pairs in units of files called *SSTables*. It first appends the written KV pairs into an on-disk write-ahead log (WAL), and inserts them into an in-memory MemTable. When the MemTable is full, the LSM-tree KV store turns the MemTable to an immutable MemTable, which

is flushed to the lowest level L_0 as an SSTable. When a lower level reaches a capacity limit, the LSM-tree KV store merges the KV pairs at the lower level into the next higher level via *compaction*. To keep the KV pairs in each level sorted, a compaction operation first reads the KV pairs from both levels, merges the sorted KV pairs, and writes back the sorted KV pairs. Thus, compaction incurs extra I/Os during writes, leading to *write amplification*. Also, since KV pairs are not sorted across different levels, reading a KV pair needs to search from the lowest level L_0 to the higher levels, leading to *read amplification*. Both write and read amplifications are shown to cause significant performance degradations in LSM-tree KV stores [12, 43, 51].

2.2 I/O Workflows

Write workflow. Writing a KV pair in Cassandra works as follows. A client first randomly selects and connects to one of the nodes, called the *coordinator* and sends it the KV pair. The coordinator determines the nodes in which the primary and redundant copies are stored, based on consistent hashing. It then forwards the KV pairs to the nodes.

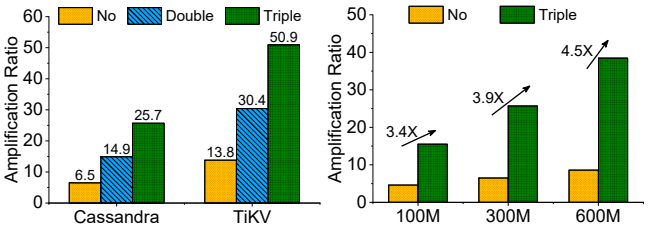
Read workflow. Reading a KV pair in Cassandra is similar to writing a KV pair and works as follows. The client first selects and contacts a coordinator. It issues the read request to the coordinator, which finds the nodes in which the replicas (regardless of primary and redundant copies) of the KV pair are stored. For load balancing, the coordinator prefers to read the KV pair from the nodes with low latencies, determined by the dynamic snitching module [5]. It then returns the KV pair to the client.

2.3 Consistency Management

Cassandra supports different consistency modes, e.g., strong consistency and eventual consistency. They are configured by tuning the *replication factor* (denoted by k), as well as the *read consistency level (RCL)* and the *write consistency level (WCL)*. The replication factor k is defined as the total number of replicas for fault tolerance. RCL and WCL are defined as the minimum numbers of replicas (regardless of primary or redundant copies) to be read and written by the coordinator to acknowledge the successful read and write operations, respectively. Both RCL and WCL are set as an integer from one to k. If WCL+RCL$> k$, then strong consistency is provided; if WCL+RCL $\leq k$, then eventual consistency is provided. By default, both WCL and RCL are set to one in Cassandra.

3 Replica Decoupling

To motivate replica decoupling, we describe the limitations of uniform indexing for managing all replicas in internal storage management (§3.1). We also describe the naïve replica decoupling designs to motivate our DEPART design (§3.2).



(a) Impact of replication in Cassandra and TiKV

(b) Impact of KV store size in Cassandra

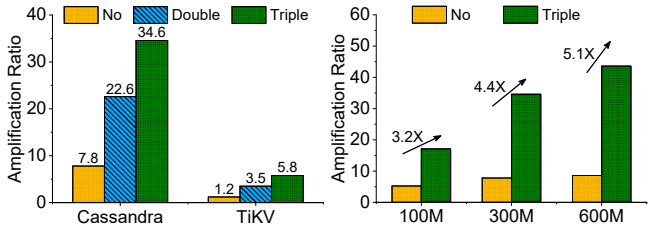
Figure 2: Write amplifications of no-replication ("No"), double replication ("double"), and triple replication ("triple") in Cassandra and TiKV.

3.1 Uniform Indexing and its Limitations

Recall from §1 that existing distributed KV stores (e.g., [3, 28, 37, 50]) mainly adopt uniform indexing, in which all replicas (including all primary and redundant copies) designated for each node are managed under the same index structure. We show that uniform indexing, rather than the extra writes from replication, is the main cause of significantly exacerbating both the write and read amplifications of the LSM-tree.

Limitation #1: Write amplification aggravation. With uniform indexing, each node treats all replicas as the regular KV pairs and stores them in the same LSM-tree without distinction (Figure 1). To show how it exacerbates the write amplification, we evaluate the write amplifications of two open-source distributed KV stores, Cassandra (v3.11.4) [2] and TiKV (release 4.0) [50]. Specifically, we deploy Cassandra and TiKV on a 5-node cluster with their default settings (detailed in §5). We configure a client machine to issue the writes of 300 M KV pairs of size 1 KiB each to the cluster that initially has empty storage. We consider no-replication ($k = 1$), double replication ($k = 2$), and triple replication ($k = 3$). Figure 2(a) shows that no-replication incurs a write amplification of 6.5× for Cassandra, due to the compaction overhead caused by the LSM-tree. However, for triple replication, the write amplification increases to 25.7×, which is around 4× the write amplification of no-replication. We also observe a similar trend for TiKV, where the write amplification increases from 13.8× to 50.9× (i.e., 3.7× increase).

Also, as the KV store size increases, the write amplification increases more significantly and shows a super-linear trend. The reason is that a larger KV store size increases the number of levels in the LSM-tree, leading to higher compaction overhead and a larger write amplification. We configure a client machine to issue the writes of 100 M, 300 M, and 600 M KV pairs of size 1 KiB each to the initially empty cluster. Here, we focus on Cassandra. Figure 2(b) shows the write amplifications of Cassandra for different KV store sizes. For a larger data store, the increase of the write amplification under triple replication compared to no replication also becomes larger. For example, triple replication has 3.4× write amplification compared to no replication under 100 M KV pairs, and becomes 4.5× under 600 M KV pairs. This super-linear trend



(a) Impact of replication in Cassandra and TiKV

(b) Impact of KV store size in Cassandra

Figure 3: Read amplifications of no-replication ("No"), double replication ("double"), and triple replication ("triple") in Cassandra and TiKV.

also implies that a larger KV store size can limit the scalability of uniform indexing.

Limitation #2: Read amplification aggravation. Uniform indexing also severely exacerbates the read amplification. The main reason is that all replicas are stored in the same LSM-tree, thereby enlarging the search space of KV pairs. We evaluate the read amplifications of Cassandra and TiKV as in the above settings, while a client machine issues 30 M reads to the existing 300 M KV pairs of size 1 KiB each that have already been stored. Figure 3(a) shows that for Cassandra, the read amplification increases from 7.8× in no-replication to 34.6× in triple replication (i.e., 4.4× increase). We observe a similar trend for TiKV.

In addition, we study the impact of the KV store size on the read amplification. Here, we focus on Cassandra. We first issue the writes of 100 M, 300 M, and 600 M KV pairs of size 1 KiB each to the initially empty cluster, followed by issuing 30 M reads to the existing KV pairs. Figure 3(b) shows a super-linear increase for the read amplification as the KV store size increases for Cassandra.

3.2 Motivation

Our analysis in §3.1 shows that uniform indexing exacerbates both write and read amplifications, as it is costly to manage all replicas within a single LSM-tree. This motivates us to explore the potentials of replica decoupling, which decouples the primary and redundant copies of replicas and manage them in separate index structures. We first consider two naïve replica decoupling approaches, and then motivate our design.

Naïve approaches. A simple replica decoupling approach is to deploy two LSM-trees, one for primary copies and one for all redundant copies. However, the LSM-tree for redundant copies still has a large size (especially for a large replication factor), while not all redundant copies are accessed in each I/O operation. For example, to recover a single-node failure under triple replication, only half of the redundant copies on average are accessed. Thus, there are extra I/Os for searching the whole LSM-tree for a subset of redundant copies.

Another simple replica decoupling approach is to manage k LSM-trees (k is the replication factor) for k replicas derived from each KV pair. For example, for Cassandra with triple replication, node N_i receives the redundant copies whose corresponding primary copies are stored in nodes $N_{(i-1) \bmod n}$ and $N_{(i-2) \bmod n}$ (where $0 \leq i \leq n-1$ and n is the number of physical nodes). Then we use three LSM-trees in node N_i, one of which stores the primary copies and the other two store the redundant copies from nodes $N_{(i-1) \bmod n}$ and $N_{(i-2) \bmod n}$, respectively.

However, maintaining multiple LSM-trees incurs both significant memory and I/O overheads. Since each LSM-tree has its own MemTable and immutable MemTable, the memory overhead amplifies by k times for the replication factor k. Specifically, if the MemTable size is m MiB and the cluster size is n, the memory cost of Cassandra is $m \times n$ MiB as each node maintains a single LSM-tree. However, when using k LSM-trees in each node, the memory cost becomes $k \times m \times n$ MiB, which is k times that in Cassandra. Note that if we reduce the MemTable size for each LSM-tree to limit the memory overhead, it degrades the efficiency of flushing the MemTable to disk, thereby degrading the user write performance [7, 8].

Also, each LSM-tree incurs its own compaction overhead for maintaining the fully-sorted ordering in each level. Thus, the compaction overhead is still significant and the compaction operations of multiple LSM-trees in the same node compete for the disk bandwidth, and hence the overall I/O performance is compromised. Our evaluation (Exp#1 in §5.2) shows that replica decoupling with multiple LSM-trees only brings limited performance gains over uniform indexing, even though the replicas are managed by different LSM-trees.

Our approach. Recall that the LSM-tree always maintains the fully-sorted ordering in each level. Using a single LSM-tree for all replicas in uniform indexing, or using multiple LSM-trees for replica decoupling, may favor high read performance, but both of them incur substantial high compaction overhead that degrades write performance. In particular, different consistency levels imply different performance requirements for the reads and writes issued to the replicas, such that a high read (or write) consistency level requires high read (or write) performance for the replicas. This motivates us to design a new storage management solution that supports *tunable ordering* for replica decoupling, so as to balance the read and write performance.

4 DEPART Design

We present DEPART, a distributed KV store that builds on Cassandra to realize replica decoupling by separating the storage management of primary and redundant copies. We introduce its architecture (§4.1) and elaborate its design techniques (§4.2-§4.5).

4.1 Overall Architecture

DEPART decouples the storage management of primary and redundant copies to achieve high performance. It manages the primary copies in the LSM-tree, while managing the re-

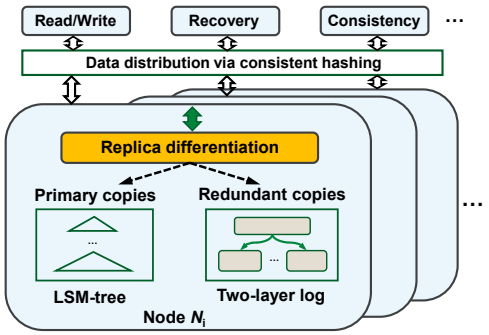


Figure 4: DEPART architecture.

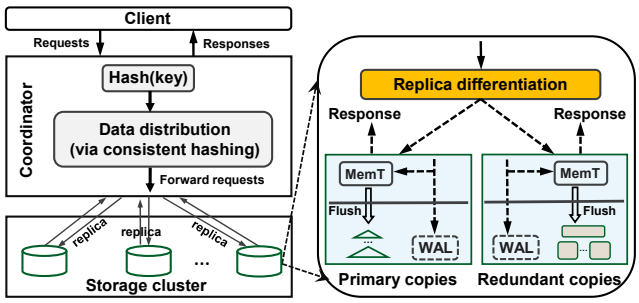


Figure 5: Replica differentiation in DEPART.

dundant copies in a novel *two-layer log*, whose ordering of redundant copies is tunable depending on the performance requirements. Keeping only the primary copies in the LSM-tree maintains the design features of the LSM-tree for reads, writes, and scans, but in a more lightweight manner as the LSM-tree size is now significantly smaller without the redundant copies. Also, the tunable ordering of the two-layer log allows balanced read and write performance for different settings of consistency levels.

Figure 4 depicts the architecture of DEPART. Note that DEPART only modifies the internal storage module of each Cassandra node, but preserves the inter-node management in Cassandra (e.g., consistent hashing for data organization and consistency management). In summary, DEPART addresses several design challenges via a number of techniques.

- **Lightweight replica differentiation.** DEPART differentiates the primary and redundant copies in the storage module of each node for separate management. Its replica differentiation is lightweight based on simple hash computations, and incurs limited overhead on the critical I/O path (§4.2).
- **Two-layer log design.** DEPART manages the redundant copies with a two-layer log, so as to achieve fast writes and efficient recovery. It first appends redundant copies to a *global log* as sequential batched writes. It then splits the global log into multiple *local logs* in background (§4.3).
- **Tunable ordering.** DEPART further provides a tunable ordering scheme for the two-layer log design to adjust the degree of ordering of the redundant copies with a single parameter, so as to balance the read and write performance for accessing the redundant copies (§4.4).
- **Parallel recovery.** DEPART uses a parallel recovery scheme that reads and writes the primary and redundant copies in parallel during recovery, so as to achieve high recovery performance (§4.5).

4.2 Replica Differentiation

DEPART differentiates the written KV pairs in the storage module of each node as primary or redundant copies. Figure 5 depicts the replica differentiation workflow. Recall that the coordinator forwards k replicas of a KV pair to a sequence of k nodes along the clockwise direction in the hash ring, where the key of the KV pair is hashed to the first node in the node sequence (§2). When a node, say N, receives one of the replicas of the KV pair from the coordinator, it performs the same hash computation (i.e., MurmurHash [6] in Cassandra) on the key of the replica and determines the node to which the key is hashed. If the resulting node is the same as N itself, then N is the first node in the node sequence and we refer to the replica as a primary copy; otherwise, we refer to the replica as a redundant copy.

Each node maintains a write-ahead log (WAL) and a MemTable for the LSM-tree (for primary copies) and the two-layer log (for redundant copies). After a node differentiates whether the KV pair is a primary copy or a redundant copy, it writes the KV pair to the corresponding WAL and MemTable and acknowledges the coordinator. When the MemTable is full and becomes immutable, the node flushes the immutable MemTable to either the LSM-tree or the two-layer log.

The logic of replica differentiation is lightweight, as it requires one extra hash computation in each storage node in the critical I/O path (and k extra computations in total for the replication factor k). Our experiments show that the differentiation time is less than 0.4% of the total write time (Exp#5 in §5.2).

4.3 Two-layer Log Design

Each node maintains a *two-layer log*, which is designed for the management of redundant copies with the following design features. First, it supports fast writes for the redundant copies, even though the number of redundant copies is much larger than that of primary copies and increases with the replication factor. Second, it supports tunable ordering to adapt to different consistency levels (§4.4). Third, it supports efficient parallel recovery of any failed nodes by allowing fast reads to the redundant copies in parallel.

Figure 6 shows the architecture of the two-layer log in each node. Upon receiving the replicas, a node first issues sequential batched writes for the redundant copies into a *global log*. A background thread continuously retrieves the redundant copies from the global log and splits them into multiple *local logs*. We elaborate the global log and local log designs below.

Append-only global log. To enable fast writes, each node writes all redundant copies of KV pairs (flushed from the

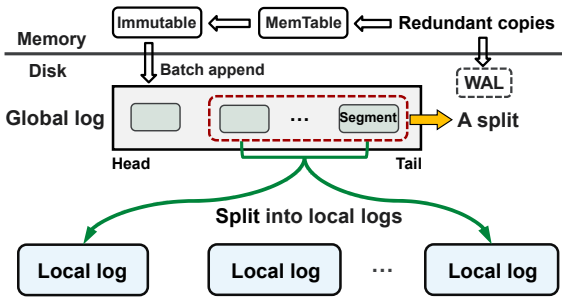


Figure 6: Architecture of the two-layer log design.

immutable MemTable) to an append-only global log. All redundant copies are grouped in units of *segments* and appended to the head of the global log as sequential batched writes. Note that the global log only stores all redundant copies without maintaining any extra index structure. Thus, it achieves high write performance for the redundant copies.

Keeping all redundant copies in the global log achieves high write performance, but poses two issues. First, the recovery performance degrades. For the redundant copies in the global log, their corresponding primary copies may reside in different nodes. When a node failure happens, only part of the redundant copies in the global log (i.e., the redundant copies whose corresponding primary copies reside in the failed node) are needed for recovery. Thus, recovery incurs only partial access to the global log, thereby incurring lots of random I/Os. Second, the garbage collection cost increases. As new KV pairs are appended to the log head, invalid (or stale) KV pairs cannot be overwritten and hence they occupy lots of space. This incurs large storage overhead, especially in update-intensive workloads. Garbage collection can be used to reduce the storage cost by continuously reclaiming the free space of invalid KV pairs from the log tail, but it inevitably introduces large amount of extra I/Os to read segments from the log tail and write back the valid KV pairs to the log head.

Splitting into local logs. To enable fast recovery, DEPART maintains a background thread to continuously split the global log into multiple local logs, each of which keeps only the redundant copies whose corresponding primary copies are stored in the same node. This allows the recovery of any failed node to access only the local log associated with the failed node. Note that each node only needs to maintain $k-1$ local logs (recall that k is the replication factor), since consistent hashing distributes the replicas in a sequence of nodes along the clockwise direction in the hash ring and each node only stores a redundant copy from up to $k-1$ nodes.

The splitting operation works as follows. It first retrieves a configurable number of segments, collectively called a *split*, from the tail of the global log. It then reorganizes a split of redundant copies into multiple sub-splits, each of which contains only the redundant copies whose corresponding primary copies reside in the same node. It finally writes back each sub-split into a separate local log in an append-only manner,

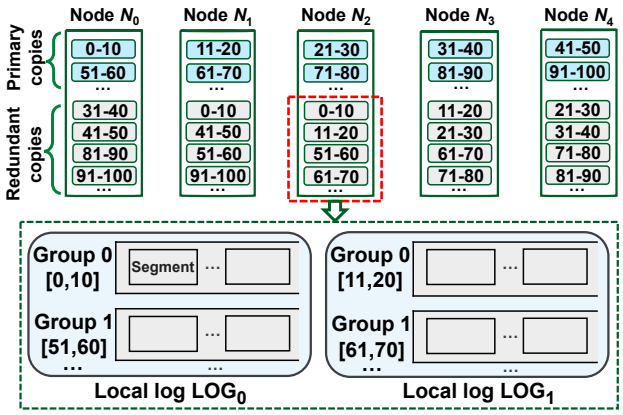


Figure 7: Range-based grouping within local logs.

and issues the writes to different local logs in parallel.

During splitting, DEPART also discards any invalid KV pairs in the selected segments. Thus, it does not trigger garbage collection explicitly; instead, it realizes garbage collection in the splitting operation to save the extra I/Os.

For each redundant copy, each node needs to determine the node in which its corresponding primary copy resides. It can be feasibly done locally within a node based on replica differentiation (§4.2).

Range-based grouping within local logs. While splitting the global log into multiple local logs alleviates the recovery and garbage collection overhead, the benefit remains limited since the ranges of a hash ring stored in each node are not necessarily contiguous (e.g., in Figure 1, Node N_0 stores ranges [0,10] and [51,60]). Recovering any range of KV pairs only needs to access the redundant copies for the range, so it still causes partial accesses to a local log and issues random I/Os.

We enhance each local log by managing KV pairs with range-based grouping. Figure 7 shows the idea of range-based grouping. Each local log is further divided into multiple *range groups*, each of which corresponds to a range in the hash ring. Note that different range groups within each local log have no overlaps in keys, so they can be managed independently. For example, for Node N_2 in Figure 7, the local log LOG_0 stores the redundant copies whose corresponding primary copies reside in Node N_0. As Node N_0 has two ranges, [0,10] and [51,60], LOG_0 now contains two range groups, each of which holds the redundant copies for [0,10] and [51,60], respectively. Range-based grouping can be realized by comparing the keys (or their hashes) with the boundary of each range in the hash ring based on consistent hashing (§2.1). It still ensures that the writes to each range group in a local log are performed in an append-only, batched manner. The number of range groups in a local log, and hence the number of ranges in Cassandra, are configurable by the parameter `num_tokens` [2]. Range-based grouping improves the recovery performance by accessing only the KV pairs in the corresponding range groups without accessing all KV pairs in the whole local log.

Under range-based grouping, when writing KV pairs from

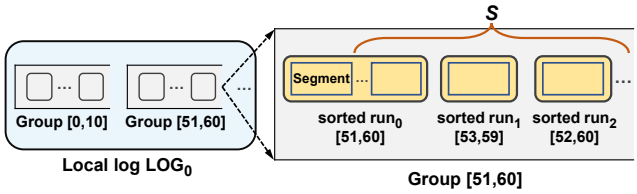


Figure 8: Tunable ordering.

the global log to the local logs during a splitting operation, DEPART further sorts all KV pairs by keys for each range before storing the KV pairs in a range group in the local logs; we call the sorted KV pairs for a range in a splitting operation a *sorted run*. Thus, each range group may store multiple sorted runs, while different sorted runs in the same range group may have overlaps in keys. Managing the range groups by sorted runs makes tunable ordering feasible (§4.4).

Reads from the two-layer log. To read a KV pair from the two-layer log, DEPART first checks the segments in the global log one by one, starting with the latest one. Note that the internal structure of each segment is similar to that of SSTables in the LSM-tree, so DEPART first reads the metadata from the segment and reads the corresponding KV pair according to the offset in the metadata. If the KV pair is not found in the global log, then DEPART searches the corresponding range group, located by comparing the key with the boundary keys of the range groups. Since each range group contains multiple sorted runs and KV pairs within the sorted run are fully sorted, DEPART searches from the latest to the oldest sorted run, and uses binary search to find the key within a sorted run.

4.4 Tunable Ordering

Recall that each range group in a local log may contain multiple sorted runs, and the KV pairs across the sorted runs within a range group are not fully sorted. If a range group contains too many sorted runs, the read performance for the redundant copies will degrade, especially for high read consistency levels where both primary and redundant copies are accessed in a read operation (§2.3). Thus, we extend the two-layer log with a *tunable ordering* scheme, in which users can configure a single parameter to adjust the degree of ordering of each range group for different consistency requirements.

DEPART adjusts the degree of ordering across multiple sorted runs with a user-configurable threshold S, which is a positive integer that controls the maximum number of sorted runs being allowed to exist in each range group. Figure 8 shows the idea of the tunable ordering scheme. For each new sorted run generated from a splitting operation, DEPART first checks if the existing number of sorted runs in a range group reaches the threshold. If not, it appends the new sorted run from the splitting operation directly to the range group; otherwise, it merge-sorts the new sorted run with the existing ones into a single sorted run. The merge-sort operation is similar to the compaction operation in the LSM-tree, including three

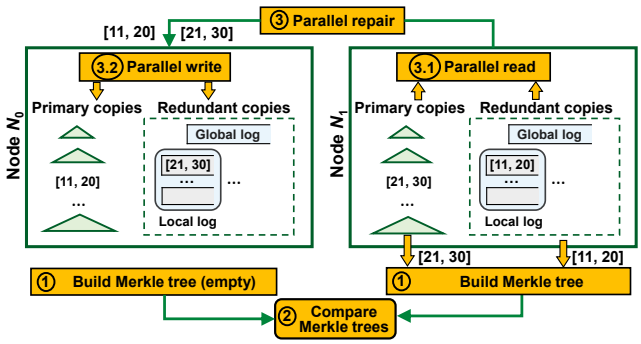


Figure 9: Parallel recovery in DEPART.

steps: (i) it reads all existing sorted runs from the range group; (ii) it merges all KV pairs in the new sorted run and the existing sorted runs, and if there exist multiple KV pairs with the same key in different sorted runs, it keeps only the KV pair in the latest sorted run and discards the older ones; and (iii) it writes back all merged KV pairs into the range group. We consider two special cases for different values of S.

- Case 1: $S = 1$. In this case, each range group always sorts the incoming sorted run with the currently stored sorted run, and hence all KV pairs are sorted. Thus, each local log resembles a single-level LSM-tree.
- Case 2: S approaches infinity. In this case, DEPART always appends the new sorted run into a range group without any merge-sorting with any existing sorted runs.

To set an appropriate value of S, we note that S determines the trade-off between the read and write performance. A small S favors the read performance by keeping a small number of sorted runs in a range group. It also maintains the storage efficiency by discarding the invalid KV pairs in merge-sort operations. However, it incurs a large merge-sort overhead that degrades the write performance. Thus, to support a system setting with the high read consistency level under read-dominant workloads, we should set a high degree of ordering with a small S to benefit reads; otherwise, we should decrease the degree of ordering by increasing the value of S to benefit writes. We also evaluate the impact of different values of S via experiments, and we recommend a default setting, $S = 20$, that can effectively balance the read and write performance under different consistency configurations (see Exp#8 in §5.2).

4.5 Parallel Recovery

For fast recovery of any failed node, DEPART proposes a *parallel recovery* scheme that exploits the benefit of decoupling the storage management of primary and redundant copies. We first review the recovery process in the current Cassandra implementation. Cassandra currently does not have a centralized node to monitor data loss and coordinate data recovery. Instead, it maintains a Merkle tree [4, 47] in each node to detect data inconsistency among multiple copies. Note that Merkle trees are also used by other consistent-hashing-based distributed KV stores, such as Dynamo [25] and Riak [54].

A Merkle tree is a binary hash tree, in which each leaf node stores the hash value of a range of KV pairs, while each non-leaf node stores the hash value of its child nodes. If a KV pair is lost, the replicas of the KV pair become inconsistent as detected by the Merkle trees across the nodes that store the replicas, so Cassandra triggers a recovery process. Specifically, the recovery process has three steps: (i) building a Merkle tree for each range of KV pairs in each node; (ii) comparing the Merkle trees of the same range of KV pairs in different nodes to identify any inconsistent range of KV pairs (which implies data loss); and (iii) reconstructing any inconsistent range by retrieving the range of KV pairs from a non-failed node and sending the range of KV pairs to the recovered node.

DEPART parallelizes the read and write processes for recovering multiple ranges of KV pairs for fast recovery. Its parallel recovery process is based on the recovery workflow in Cassandra, as shown in Figure 9. Suppose that we recover the lost data of a failed node at node N_0. First, each node in DEPART retrieves the KV pairs from the LSM-tree and the two-layer log, and builds its own Merkle tree (note that the Merkle tree in N_0 is initially empty) (Step 1). N_0 compares the Merkle trees and identifies the missing KV pairs (Step 2). To recover the lost KV pairs, each surviving node (e.g., node N_1 in Figure 9) issues parallel reads to the primary and redundant copies with two threads, and similarly the new node (i.e., N_0) retrieves the KV pairs from other surviving nodes and issues parallel writes for the primary and redundant copies with two threads. Such multi-threading is feasible as the primary and redundant copies are stored in different index structures.

5 Evaluation

DEPART builds on the codebase of Cassandra v3.11.4 [2] by implementing replica decoupling in the storage module of each node. Our DEPART prototype itself contains 6.9 K LoC, while the modification to Cassandra contains 1.9 K LoC. Note that Cassandra v3.11.4 contains about 206.2 K LoC. To demonstrate the benefits of the two-layer log design in DEPART, we also implement the naïve replica decoupling approach that simply stores replicas in multiple LSM-trees, which we refer to as *mLSM* (§3.2).

We conduct testbed experiments to demonstrate the efficiency of DEPART. We compare our DEPART prototype with Cassandra (v3.11.4), which performs uniform indexing for all replicas, mLSM. We address the following questions.

- How is the overall performance of DEPART compared with Cassandra and mLSM under different settings, e.g., the microbenchmark performance in different types of KV operations, the performance under different consistency configurations and different replication factors, as well as the performance under YCSB core workloads [21, 22]? (Experiments 1-4)
- What are the performance breakdowns of DEPART and Cassandra? (Experiment 5)
- What is the performance of DEPART when a node failure occurs? (Experiments 6-7)
- How does the performance of DEPART vary across parameter settings, including the ordering degree S, the store sizes, and the numbers of storage nodes? (Experiments 8-10)

5.1 Setup

Testbed. We conduct all experiments on a local cluster of multiple machines, each of which has two 12-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz, 32 GiB RAM, and a 500 GiB Samsung 860 EVO SATA SSD. All machines are interconnected via a 10 Gb/s Ethernet switch. Each machine runs CentOS 7.6.1810, with the 64-bit Linux kernel 3.10.0 and the Ext4 file system. We use one machine to simulate multiple clients via a thread pool, while the remaining machines serve as storage nodes.

Workloads. We generate workloads with YCSB [21, 22], a general-purpose cloud system benchmark tool. By default, we focus on 1 KiB KV pairs with 24-byte keys, and generate requests based on the Zipf distribution with the default Zipfian constant 0.99. We deploy YCSB on the client machine and set the number of client threads as 50, while each client thread issues a workload from YCSB.

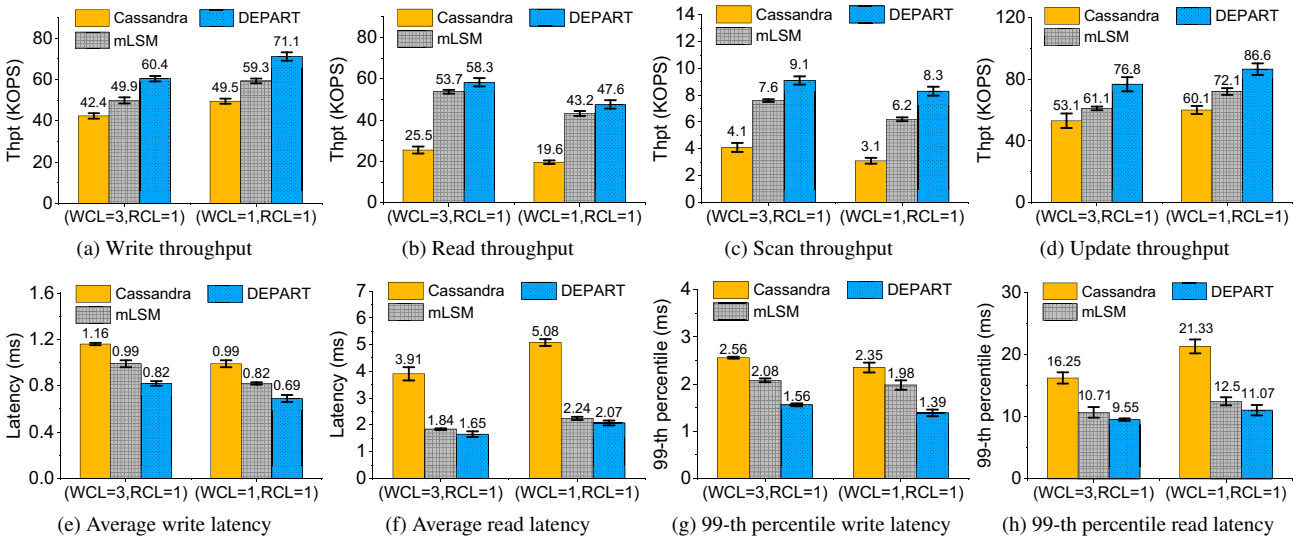
Default settings. We configure five storage nodes in the cluster and triple replication to deploy Cassandra and DEPART. Before each experiment, the cluster has empty storage. By default, we set (WCL=1, RCL=1) (i.e., the default setting in Cassandra), which corresponds to eventual consistency. We also study the impact of different consistency levels (Experiments 1 and 2). Both Cassandra and DEPART use the default `dynamic snitching` module [5] to choose the fastest nodes for serving reads, so as to load-balance reads across different replicas. For the parameter `num_tokens` [2], which determines the number of range groups, we use the default value 256 as in Cassandra.

For DEPART, we set the MemTable size to be the same as that of Cassandra (160 MiB by default), and the segment size in the global log to be the same as the MemTable size. Since DEPART keeps an extra MemTable for the two-layer log, we increase the `row_cache` size of Cassandra by 160 MiB for fair comparisons. For the two-layer log, we set the data size of each split operation as 20 segments (around 3 GiB) and set S as 20 to achieve balanced read and write performance. We keep the other parameter settings in Cassandra unchanged.

We plot the average results over five runs, with error bars showing the standard deviation.

5.2 Results

Experiment 1 (Performance in KV operations). We first compare the performance of Cassandra, mLSM, and DEPART in different KV operations, including writes (i.e., writing new KV pairs), reads (i.e., reading existing KV pairs), scans (i.e., reading existing consecutive KV pairs), and updates (i.e., updating existing KV pairs). We configure the client machine



(a) Write throughput (b) Read throughput (c) Scan throughput (d) Update throughput

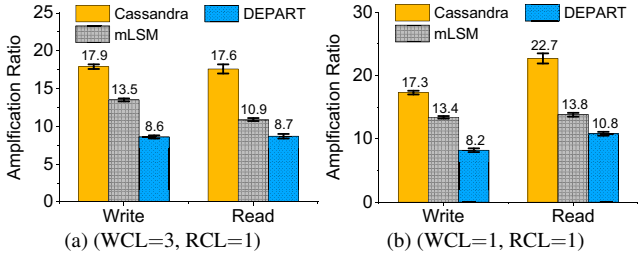
(e) Average write latency (f) Average read latency (g) 99-th percentile write latency (h) 99-th percentile read latency

Figure 10: Exp#1 (Performance in KV operations).

to first randomly write 200 M KV pairs. It then issues the following requests in order: (i) 20 M reads, (ii) 2 M scans (each scan contains one `seek()` to locate the first key and then iterates with 100 `next()`'s), and (iii) 200 M updates. We also consider two settings of consistency levels: (i) (WCL=3, RCL=1) (i.e., strong consistency) and (ii) (WCL=1, RCL=1) (i.e., eventual consistency).

Figure 10 shows the throughput and latency results. First, DEPART improves the overall performance over Cassandra in all cases. For (WCL=3, RCL=1), DEPART increases the throughput of writes, reads, scans, and updates to 1.42×, 2.29×, 2.22×, and 1.45×, respectively; it reduces the average write latency, average read latency, 99-th percentile write latency, and 99-th percentile read latency by 29%, 58%, 39%, and 41%, respectively. For (WCL=1, RCL=1), DEPART improves the throughput of writes, reads, scans, and updates to 1.43×, 2.43×, 2.68×, and 1.44×, respectively; it reduces the average write latency, average read latency, 99-th percentile write latency, and 99-th percentile read latency by 30%, 59%, 41%, and 48%, respectively. The latency results for scans and updates are similar and we omit the results here. The main reasons of the performance improvements of DEPART are two-fold. First, for reads, DEPART only searches the LSM-tree or the specific range group in the two-layer log within a node, thereby greatly reducing the search space. Second, for writes, DEPART mitigates the compaction overhead in the LSM-tree, which now keeps the primary copies only. The two-layer log also has limited merge-sort overhead by having a large value of the ordering degree S.

Second, mLSM notably improves the read performance, but only shows marginal improvements on writes. Specifically, compared with Cassandra, it increases the throughput of writes, reads, scans, and updates to 1.18-1.20×, 2.11-2.20×, 1.85-2.0×, and 1.15-1.20×, respectively. It also reduces the average write latency, average read latency, 99-th percentile

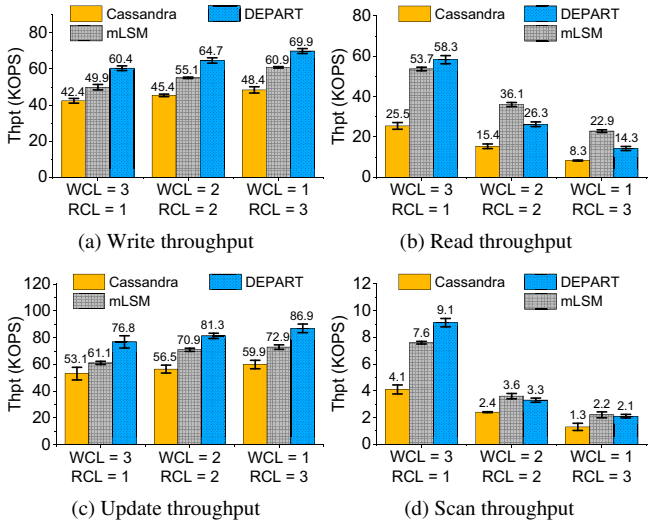


(a) (WCL=3, RCL=1) (b) (WCL=1, RCL=1)

Figure 11: Exp#1 (Read and write amplifications).

write latency, and 99-th percentile read latency by 15-17%, 53-56%, 16-18%, and 34-41%, respectively. The reason is that mLSM still triggers frequent compaction operations, which compete for disk bandwidth and degrade the write performance. For example, the total compaction sizes of Cassandra and mLSM are 3.46 TiB and 2.72 TiB, respectively, and the total compaction and merge-sort size of DEPART is 1.65 TiB (i.e., compared with Cassandra, mLSM only reduces the total compaction size by 21%, but DEPART reduces it by 52%).

We next compare the storage and memory costs of Cassandra, mLSM, and DEPART. After the end of the update phase, the KV store sizes are 613.5 GiB for Cassandra, 611.3 GiB for mLSM, and 654.8 GiB for DEPART. DEPART incurs 6.7% additional storage overhead compared with Cassandra, since each range group allows at most $S = 20$ sorted runs in our default setting and contains invalid KV pairs before being merge-sorted. To measure the memory overhead, we note that the MemTable size varies over time (up to the 160 MiB limit) as the KV pairs are continuously inserted into a MemTable and flushed to disk when the MemTable is full. Thus, we measure the total memory usage of the MemTables every five seconds and obtain the average results. The total memory usage of mLSM is 335.7 MiB, which is 3.7× that of Cassandra (90.4 MiB), as each LSM-tree maintains a MemTable. However, DEPART only costs 183.9 MiB, which is 2.0× that of Cassandra, since DEPART only maintains one extra



(a) Write throughput (b) Read throughput

(c) Update throughput (d) Scan throughput

Figure 12: Exp#2 (Performance under different consistency configurations).

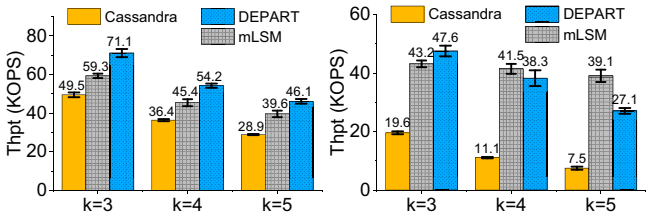
MemTable for the two-layer log. Note that the total memory usage of mLSM increases with the replication factor, while that of DEPART remains unaffected.

Finally, we compare the read/write amplifications (i.e., the ratios between the amounts of system reads/writes and the amounts of the user reads/writes) of Cassandra, mLSM, and DEPART. Figure 11 shows the results. Compared with Cassandra, mLSM reduces the read and write amplifications by up to 40% and 24%, respectively, while DEPART reduces the read and write amplifications by up to 53% and 52%, respectively. Note that the performance gain of DEPART is similar under both consistency levels, so we focus on (WCL=1, RCL=1) in the following experiments (except Exp#2 and 8).

Experiment 2 (Performance under different consistency configurations). We evaluate the performance under different consistency configurations. In particular, for strong consistency, we consider additional configurations for WCL and RCL under triple replication that satisfy the condition WCL+RCL>3, including (WCL=2, RCL=2) and (WCL=1, RCL=3).

Figure 12 shows the results. DEPART consistently improves the throughput of writes, reads, scans, and updates over Cassandra under different consistency configurations. Specifically, for (WCL=2, RCL=2), DEPART increases the throughput of writes, reads, scans, and updates to 1.43×, 1.70×, 1.38×, and 1.44×, respectively. For (WCL=1, RCL=3), DEPART increases the throughput of writes, reads, scans, and updates to 1.44×, 1.72×, 1.62×, 1.45×, respectively. DEPART also consistently improves the throughput of writes and updates over mLSM. Note that the write performance gains of DEPART over Cassandra stay nearly the same under different consistency configurations, since the index structures of both Cassandra and DEPART remain unchanged under triple replication.

However, the read performance gains of DEPART over



(a) Write throughput (b) Read throughput

Figure 13: Exp#3 (Performance under different replication factors).

Cassandra become smaller, and DEPART's read performance is worse than mLSM for RCL≥2. In this case, each read request needs to access at least two replicas successfully, so the redundant copies in the two-layer log must be searched. As the redundant copies in the two-layer log are not fully sorted, the performance is slower than reading the primary copies in the LSM-tree. Nevertheless, DEPART still achieves faster reads than Cassandra, as it searches for less data than Cassandra. Also, mLSM keeps redundant copies being fully sorted in each level, so it achieves higher read performance than DEPART. On the other hand, for RCL=1, each read only needs to access one replica for a successful operation. Most of the reads are routed to their primary copies, whose read latency is smaller than that of the redundant copies as determined by the dynamic snitching module (§2.2). Thus, the read performance gains under RCL=1 are higher than those under RCL≥2 in general.

Experiment 3 (Performance under different replication factors). We evaluate the performance of DEPART by varying the replication factor k from 3 to 5. We configure the client machine to first randomly write 200 M KV pairs and then issue 20 M reads.

Figure 13 shows the throughput results of writes and reads versus the replication factor. Compared with Cassandra, DEPART increases the throughput of writes and reads to 1.43-1.59× and 2.43-3.61×, respectively. Also, DEPART achieves a higher throughput gain for a larger replication factor. The main reasons are two-fold. First, for reads, DEPART either reads primary copies from the LSM-tree or reads redundant copies from the two-layer log; for the latter, it only searches the global log and the corresponding range group in the two-layer log. Thus, the read performance of DEPART is less affected by the number of replicas. However, Cassandra stores all replicas in a single LSM-tree and its reads need to traverse the whole LSM-tree. Its read performance drops significantly as the replication factor increases. Second, for writes, DEPART implements replica decoupling and manages the redundant copies in range groups. When the number of replicas increases, the compaction cost in the LSM-tree remains unchanged and the merge-sort cost in the two-layer log increases only slightly. However, Cassandra stores all replicas in the single LSM-tree and the compaction cost increases significantly as the number of replicas increases. Combining both reasons, the performance gain of DEPART becomes larger

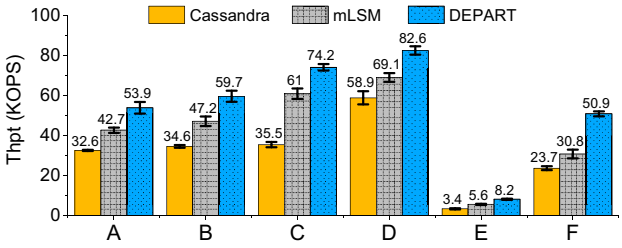
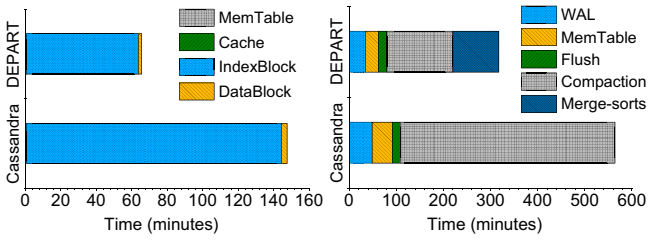
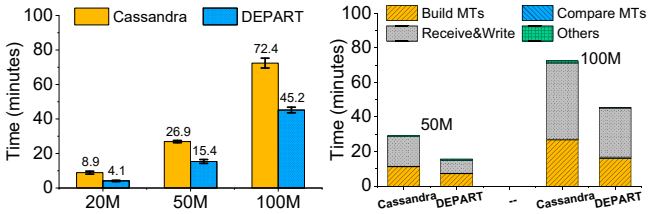


Figure 14: Exp#4 (YCSB performance).



(a) Read time breakdown (b) Write time breakdown

Figure 15: Exp#5 (Time breakdown for reads/writes).



(a) Recovery time (b) Time breakdown

Figure 16: Exp#6 (Recovery performance).

for a higher replication factor.

Similar to DEPART, mLSM also consistently improves the throughput of writes and reads over Cassandra under different replication factors. When the replication factor increases to $k = 4$ and $k = 5$, its read performance is even better than DEPART. The main reason is that mLSM only searches the corresponding LSM-tree that is fully sorted in each level regardless of the replication factor, but DEPART keeps redundant copies in the two-layer log that is not fully sorted under the default setting. Note that we can tune the degree of ordering of the two-layer log to further increase the read performance. Furthermore, the memory usage of DEPART remains 2× that of Cassandra, but that of mLSM increases to 5× when the replication factor increases to $k = 5$.

Experiment 4 (YCSB performance). We compare Cassandra, mLSM, and DEPART using the six YCSB core workloads [21, 22], 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). The client machine first randomly writes 200 M KV pairs to the cluster before running each of the six YCSB core workloads. Each workload consists of 100 M operations, except for Workload E, which contains 10 M operations with each scan involving 100 `next()`'s.

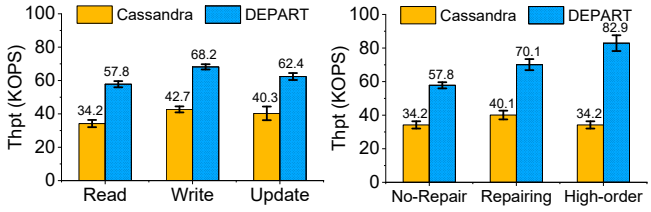
Figure 14 shows the results. DEPART outperforms Cassandra under all workloads. Specifically, it increases the throughput to 1.4-2.1× under read-dominant Workloads B-D, 1.6-2.2× under write-dominant Workloads A and F, and 2.4× under scan-dominant Workload E. With replica decoupling and the two-layer log design, DEPART reduces the compaction overhead of the LSM-tree during writes and reduces the search space during reads, so it improves both read and write performance simultaneously. On the other hand, mLSM also outperforms Cassandra under all workloads due to replica decoupling. However, DEPART further improves the performance of mLSM, as the latter incurs large compaction overhead.

Experiment 5 (Time breakdown for reads/writes). We show the time breakdown for both read and write processes in Cassandra and DEPART. We configure the client machine to first load 200 M KV pairs, followed by issuing 20 M reads. The read process comprises the reads to the MemTable, the cache (including the `row_cache` and `key_cache`), the index block of an SSTable (e.g., the Bloom filters and offsets), and the data block in the SSTable. Note that each segment in the two-layer log is treated as an SSTable here. The write process comprises writes to the WAL, writes to the MemTable, flushing the MemTable, the compaction of the LSM-tree, and the merge-sorts of the two-layer log (in DEPART only).

Figure 15 shows the time breakdown for reads and writes. For reads, most of the read time is for reading the index blocks of SSTables in both Cassandra and DEPART, since reading a KV pair needs to check the Bloom filter in the index block in each LSM-tree level to determine if the KV pair exists, and reads the data block from the SSTable according to the offset in the index block only if it does. Overall, DEPART reduces the time costs of reading the index blocks and the data blocks of SSTables by 56% and 45%, respectively. The reasons are two-fold. First, DEPART stores only the primary copies in the LSM-tree, so the number of SSTables in the LSM-tree greatly decreases. Also, DEPART manages the redundant copies in range groups, so the number of reads for locating a KV pair decreases as well.

For writes, DEPART reduces the time costs of writes to the WAL and writes to the MemTable by 28% and 37%, respectively, as DEPART writes primary and redundant copies in parallel. DEPART also greatly reduces the compaction overhead by reducing the LSM-tree size; for example, its compaction time is only 30.7% of Cassandra's. Furthermore, the merge-sort time of the two-layer log in DEPART is only 21.4% of the compaction time in Cassandra. Thus, the total time of compaction and merge-sorts in DEPART is only 52.1% of the compaction time in Cassandra.

Experiment 6 (Recovery performance). We evaluate the recovery performance on recovering a failed node. We consider different write sizes, by configuring the client machine to randomly write 20 M, 50 M, and 100 M KV pairs to the cluster. We then crash one node, by killing the KV store process with



(a) Throughput under failure (b) Degraded read

Figure 17: Exp#7 (Performance when a node crashes).

the "`kill -s processID`" command and removing all its data with the "`rm -r data`" command. Finally, we restart the KV store process on the same node and call the "`nodetool repair -full keyspacename`" command for recovery.

Figure 16(a) shows the total recovery time. Cassandra takes 8.9, 26.9, and 72.4 minutes to recover 20 M, 50 M, and 100 M KV pairs, respectively, while DEPART only takes 4.1, 15.4, and 45.2 minutes, respectively. Overall, DEPART reduces the recovery time of Cassandra by 38-54%. The main reason is that DEPART repairs primary and redundant copies in parallel, and scans much less data during recovery due to replica decoupling.

Figure 16(b) also shows the breakdown results of the recovery time for repairing 50 M and 100 M KV pairs. We consider different steps: Build MTs (i.e., building Merkle trees for all nodes), Compare MTs (i.e., comparing all Merkle trees), Receive&Write (i.e., receiving repaired data from other nodes and writing data to disk), and Others (i.e., other operations in recovery). DEPART reduces the time costs of Build MTs and Receive&Write by nearly a half compared to Cassandra through parallelizing the read/write processes to the primary and redundant copies.

Experiment 7 (Performance when a node crashes). We evaluate the read, write and update performance when a node crashes and before it is repaired. The client machine first randomly writes 100 M KV pairs to the cluster and we manually crash one node as in Experiment 6. We then issue 20 M reads, 100 M writes, and 100 M updates.

Figure 17(a) shows the throughput under a node failure. Compared with Cassandra, DEPART increases the throughput of reads, writes, and updates to 1.69×, 1.59×, and 1.55×, respectively. The main reason is that DEPART always searches much less data than Cassandra, even though it reads the redundant copies from the two-layer log. Also, DEPART always improves write performance under both normal and failure modes.

We also evaluate the degraded read performance in different cases: (i) the node repair is not yet triggered, (ii) the node repair is in progress, and (iii) the two-layer log for redundant copies in each node supports a higher degree of ordering by decreasing the threshold S (§4.4) from the default value 20 to 5. Figure 17(b) shows the degraded read throughput. When node repair is not triggered or is in progress, DEPART improves the throughput of degraded reads to 1.69× and 1.75×, respectively, since DEPART always searches much less data compared to uniform indexing in Cassandra. Also, when the two-layer log has a higher degree of ordering (e.g., with a smaller threshold S), the degraded read performance gains of DEPART become larger, because it is more efficient to read the KV pairs in the two-layer log that with a high degree of ordering.

S	Write thpt (KOPS)	Read thpt (KOPS)
1	37.2	42.3
10	57.2	31.5
20	64.7	23.1
$\rightarrow \infty$	78.4	7.6
Cassandra	45.4	15.4

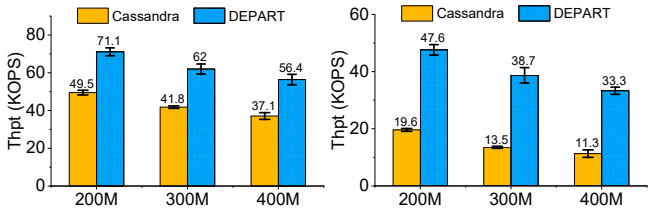
Table 1: Exp#8 (Impact of the ordering degree S).

Experiment 8 (Impact of the ordering degree S). We evaluate the write and read performance under different settings of the ordering degree S in DEPART, so as to show how DEPART can balance the read and write performance gains for the redundant copies by tuning the value of S. The client machine first randomly writes 200 M KV pairs to the cluster, followed by issuing 20 M reads. Here, we use the consistency configuration (WCL=2, RCL=2), so that the redundant copies must be accessed for each successful read.

Table 1 shows the results. For DEPART, if $S = 1$, the two-layer log reduces to a two-level LSM-tree, so it achieves the highest read throughput as the KV pairs are fully sorted, but the write throughput is the least due to the frequent merge-sorts for maintaining a single sorted run in each range group in the local logs. As we increase S (e.g., S is 10 or 20), the ordering of the two-layer log is relaxed and hence the merge-sort overhead becomes smaller, so the write throughput increases. As we set S to be a sufficiently large value, the two-layer log reduces to the append-only log, so the write throughput is the highest, but the read throughput is the least. Note that when S is 1, 10, or 20, DEPART still maintains higher throughput in both writes and reads than Cassandra, even though there exists a performance trade-off between writes and reads in DEPART.

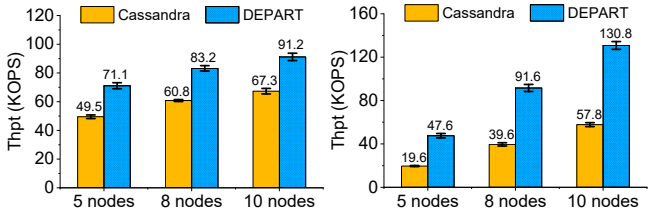
Experiment 9 (Impact of different KV store sizes). We now evaluate the impact of different KV store sizes. We vary the data size written by the client from 200 M to 400 M KV pairs (i.e., the total amount of primary and redundant copies increases from 600 GiB to 1200 GiB under triple replication). Figure 18 shows the throughput of writes and reads under different KV store sizes. Compared with Cassandra, DEPART improves the write and read throughput by 1.43-1.52× and 2.43-2.95×, respectively. Also, the performance gains of DEPART increase as the KV store size increases, as DEPART alleviates the write and read amplifications via replica decoupling, but Cassandra aggravates the write and read amplifications via uniform indexing.

To better show the scalability of the two-layer log design under different KV store sizes, we also evaluate the com-



(a) Write throughput (b) Read throughput

Figure 18: Exp#9 (Impact of different KV store sizes).



(a) Write throughput (b) Read throughput

Figure 19: Exp#10 (Impact of different numbers of nodes).

paction time of the LSM-tree and the merge-sort time of the two-layer log in DEPART, and compare them with the compaction time in Cassandra. When the KV store size increases from 200 M to 400 M KV pairs, the compaction time in Cassandra increases 2.3×, while the total time of compaction and merge-sort operations in DEPART only increases 1.9×. In particular, the ratio of the merge-sort time in DEPART to the compaction time in Cassandra drops from 21.4% to 18.7%, and the ratio of the compaction time in DEPART to the compaction time in Cassandra drops from 30.7% to 25.2%. Thus, DEPART scales well as the KV store grows.

Experiment 10 (Impact of different numbers of nodes). We evaluate the performance of DEPART when the cluster contains more nodes. We vary the number of nodes as 5, 8, and 10. We configure the client machine to issue the writes of 200 M, 320 M, and 400 M KV pairs, respectively, so that each node contains the same amount of data. After the writes, we configure the client machine to issue 20 M, 32 M, and 40 M reads, respectively.

Figure 19 shows the throughput results of write and read operations. Compared with Cassandra, DEPART increases the throughput of writes and reads to 1.35-1.43× and 2.26-2.43×, respectively. DEPART maintains its performance gains over Cassandra via replica decoupling, regardless of the cluster size. Thus, DEPART achieves good scalability as the cluster size increases.

6 Related Work

Local LSM-tree KV stores. A number of studies optimize the read and write performance of local LSM-tree KV stores that run on single machines. Read performance can be improved by Bloom filter optimization [23,38], adaptive caching [65], and scan optimization with succinct tries [68], while write performance can be improved by compaction optimization [24, 32, 55, 56], the fragmented LSM-tree [51], KV separation [12, 36, 43], I/O scheduling optimization [8], memory structure optimization [9], and a mix of optimization techniques for memory-disk-log components [7]. Our work focuses on the replica management in distributed KV stores, and is compatible with the above optimization techniques for local LSM-tree KV stores in individual nodes.

Distributed KV stores. Distributed KV stores can be classified into in-memory KV caches [42, 53] and persistent stores [1–3, 41, 50]. Optimization efforts for in-memory KV stores include lock-free and cache-friendly designs for high concurrency and throughput [13, 30, 40], erasure coding designs for memory efficiency [66, 67], self-tuning data placement [49], size-aware sharding for tail latency reduction [26], adaptive load balancing [16], secondary indexing [34], stretched Reed-Solomon coding [35], as well as hot spot optimization [15]. For distributed persistent KV stores, prior studies propose offline index construction for bulk loading [57], adaptive replica selection [52], multi-get scheduling [58], auto-tuning of tail latency optimization [39], load balancing [11], performance optimization via cost-benefit analysis with workload prediction [45], and optimizations of data placement and controlled migration [63, 64]. Persistent KV stores mostly adopt the LSM-tree in the storage layer to store all KV pairs. In contrast, DEPART proposes replica decoupling in distributed LSM-tree KV stores for efficient replica management.

Replica management. Prior studies improve the replication of distributed KV stores via efficient replica placement. Early studies include chain replication [61, 62] and its extension [44], low-cost wide-area replication [17], and dynamic hierarchical replication for data grids [46]. Copyset [19] and tiered replication [18] focus on maintaining high storage reliability. Replex [59] supports efficient queries on multiple keys. Our work focuses on the replica management within each storage node, while being compatible with the upper-layer replica placement policies.

7 Conclusion

We propose DEPART, which builds on a novel replica management scheme, replica decoupling, for distributed KV stores. DEPART uses a novel two-layer log design with tunable ordering to efficiently manage the redundant copies for different read and write performance requirements. Our DEPART prototype significantly outperforms Cassandra in different types of KV operations and maintains its performance gains for different consistency levels and parameter settings.

Acknowledgments

We thank our shepherd, Abutalib Aghayev, and the anonymous reviewers for their comments. This work is supported in part by NSFC (61772484, 61832011, 61772486), Youth Innovation Promotion Association CAS (No. 2019445), and USTC Research Funds of the Double First-Class Initiative (YD2150002003). Yongkun Li is USTC Tang Scholar, and he is the corresponding author.

References

[1] Amazon. DynamoDB. https://aws.amazon.com/dynamodb.

[2] Apache. Cassandra-3.11.4. https://github.com/apache/cassandra/tree/cassandra-3.11.4.

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

[4] Apache. Manual repair in Cassandra. https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/operations/opsRepairNodesManualRepair.html.

[5] Apache. Dynamic snitching. https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present_-and-future, 2021.

[6] A. Appleby. Murmurhash. https://sites.google.com/site/murmurhash/.

[7] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proc. of USENIX ATC*, 2017.

[8] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proc. of USENIX ATC*, 2019.

[9] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking memory in persistent key-value stores. In *Proc. of ACM EuroSys*, 2017.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *Proc. of USENIX OSDI*, 2010.

[11] K. L. Bogdanov, W. Reda, G. Q. Maguire, D. Kostić, and M. Canini. Fast and accurate load balancing for geo-distributed storage systems. In *Proc. of ACM SoCC*, 2018.

[12] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. of USENIX ATC*, 2018.

[13] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proc. of ACM SIGMOD*, 2018.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI*, 2006.

[15] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. HotRing: A hotspot-aware in-memory key-value store. In *Proc. of USENIX FAST*, 2020.

[16] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of ACM EuroSys*, 2015.

[17] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of USENIX NSDI*, 2006.

[18] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of USENIX ATC*, 2015.

[19] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of USENIX ATC*, 2013.

[20] F. Community. Fauna. https://docs.fauna.com/fauna/current/.

[21] B. F. Cooper. YCSB-0.15.0. https://github.com/brianfrankcooper/YCSB.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[23] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proc. of ACM SIGMOD*, 2017.

[24] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proc. of ACM SIGMOD*, 2018.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of of ACM SOSP*, 2007.

[26] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proc. of USENIX NSDI*, 2019.

[27] R. Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB/.

[28] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM*, 2012.

[29] Facebook. RocksDB. https://rocksdb.org.

[30] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. of USENIX NSDI*, 2013.

[31] S. Ghemawat and J. Dean. LevelDB. https://leveldb.org.

[32] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proc. of ACM EuroSys*, 2015.

[33] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.

[34] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *Proc. of USENIX ATC*, 2016.

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

[36] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, 2015.

[37] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[38] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proc. of USENIX ATC*, 2019.

[39] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In *Proc. of USENIX ATC*, 2018.

[40] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. of USENIX NSDI*, 2014.

[41] LinkedIn. Voldemort. `http://project-voldemort.com/`.

[42] LiveJournal. Memcached. `https://memcached.org/`.

[43] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, 2016.

[44] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, and L. Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Trans. Storage*, 4(4):11:1–11:28, Feb. 2009.

[45] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi. SOPHIA: Online reconfiguration of clustered NoSQL databases for time-varying workloads. In *Proc. of USENIX ATC*, 2019.

[46] N. Mansouri and G. H. Dastghaibyfard. A dynamic replica management strategy in data grid. *Journal of Network and Computer Applications*, 35(4):1297–1303, 2012.

[47] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques*, 1987.

[48] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. *Acta Informatica*, 33(4):351–385, 1996.

[49] J. a. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AutoPlacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 9(4):19.1–19.30, Dec. 2015.

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

[51] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[52] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proc. of ACM EuroSys*, 2017.

[53] Redislab. Redis. `https://redis.io`, 2017.

[54] Riak Community. Riak. `https://riak.com`.

[55] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proc. of ACM SIGMOD*, 2012.

[56] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-Trees. In *Proc. of USENIX FAST*, 2013.

[57] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proc. of USENIX FAST*, 2012.

[58] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. of USENIX NSDI*, 2015.

[59] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi. Replex: A scalable, highly available multi-index data store. In *Proc. of USENIX ATC*, 2016.

[60] S. Team. Scylladb. `https://www.scylladb.com`.

[61] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. of USENIX ATC*, 2009.

[62] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of USENIX OSDI*, 2004.

[63] R. Vilaça, R. Oliveira, and J. Pereira. *A Correlation-Aware Data Placement Strategy for Key-Value Stores*, pages 214–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[64] L. Wang, Y. Zhang, J. Xu, and G. Xue. MAPX: Controlled data migration in the expansion of decentralized object-based storage systems. In *Proc. of USENIX FAST*, 2020.

[65] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du. AC-Key: Adaptive caching for LSM-based key-value stores. In *Proc. of USENIX ATC*, 2020.

[66] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14. ACM, 2017.

[67] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, 2016.

[68] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. of ACM SIGMOD*, 2018.