Using RDMA for Efficient Index Replication in LSM Key-Value Stores

Michalis Vardoulakis
Institute of Computer Science, FORTH
mvard@ics.forth.gr

Giorgos Saloustros
Institute of Computer Science, FORTH
gesalous@ics.forth.gr

Pilar González-Férez
Department of Computer Engineering,
University of Murcia
pilargf@um.es

Angelos Bilas
Institute of Computer Science, FORTH
bilas@ics.forth.gr

1 Also with the Computer Science Department, University of Crete

Abstract

Log-Structured Merge tree (LSM tree) Key-Value (KV) stores have become a foundational layer in the storage stacks of datacenter and cloud services. Current approaches for achieving reliability and availability avoid replication at the KV store level and instead perform these operations at higher layers, e.g., the DB layer that runs on top of the KV store. The main reason is that past designs for replicated KV stores favor reducing network traffic and increasing I/O size. Therefore, they perform costly compactions to reorganize data in both the primary and backup nodes. Since all nodes in a rack-scale KV store function both as primary and backup nodes for different data shards (regions), this approach eventually hurts overall system performance.

In this paper, we design and implement Talos, an efficient rack-scale LSM-based KV store that aims to significantly reduce the I/O amplification and CPU overhead in backup nodes and make replication in the KV store practical. We rely on two observations: (a) the increased use of RDMA in the datacenter, which reduces CPU overhead for communication, and (b) the use of KV separation that is becoming prevalent in modern KV stores. We use a primary-backup replication scheme that performs compactions only on the primary nodes and sends the pre-built index to the backup nodes of the region, avoiding all compactions in backups. Our approach includes an efficient mechanism to deal with pointer translation across nodes in the region index. Our results show that Talos reduces in the backup nodes, I/O amplification by up to 3×, CPU overhead by up to 1.6×, and memory size needed for the write path by up to 2×, without increasing network bandwidth excessively, and by up to 1.3×. Overall, we show that our approach has benefits even when small KV pairs dominate in a workload (80%-90%). Finally, it enables KV stores to operate with larger growth factors (from 10 to 16) to reduce space amplification without sacrificing precious CPU cycles.

1 Introduction

Replicated persistent key-value (KV) stores are the heart of modern datacenter storage stacks [2, 11, 13, 26, 30]. These systems typically use an LSM tree [32] index structure because of its 1) fast data ingestion capability for small and variable size data items while maintaining good read and scan performance and 2) its low space overhead on the storage devices [14]. However, LSM-based KV stores suffer from high compaction costs (I/O amplification and CPU overhead) for reorganizing the multi-level index.

To provide reliability and availability, state-of-the-art KV stores [11, 26] provide replication of KV pairs in multiple, typically two or three [6], nodes. Current designs for replication optimize network traffic and favor sequential I/O to the storage devices both in the primary and backup nodes. Essentially, these systems perform costly compactions to reorganize data in both the primary and backup nodes to ensure: (a) minimal network traffic by moving user data across nodes, and (b) sequential device access by performing only large I/Os. However, this approach comes at a significant increase in device traffic (I/O amplification) and CPU utilization at the backups. Given that all nodes in a distributed design function both as primaries and backups at the same time, eventually this approach hurts overall system performance. For this reason, in many cases, current approaches for reliability and availability avoid replication at the KV store level and instead perform these operations at higher layers, e.g. the DB layer that runs on top of the KV store [11, 26].

Nowadays, state-of-the-art KV stores [11, 26] adopt the eager approach [11, 26, 36] which minimizes network traffic and recovery time at the expense of I/O amplification, CPU, and memory overhead at the secondaries. This approach is appropriate for systems designed for TCP/IP networks and Hard Disk Drives (HDDs).

In our work, we rely on two key observations: (a) the increased use of RDMA in the datacenter [18, 38] which reduces CPU overhead for communication and (b) the use of KV separation that is becoming prevalent in modern KV
stores [3, 16, 27, 29, 34, 41]. Network I/O overhead is not a significant constraint due to the use of RDMA, especially at the rack level [15, 22, 23, 31]. Also, LSM-based KV stores tend to increasingly employ KV separation to reduce I/O amplification [3, 10, 16, 25, 29, 34]. KV separation places KV pairs in a value log and keeps an LSM index where values point to locations in the value log. This technique introduces small and random read I/Os, which fast storage devices can handle, and reduces I/O amplification by up to 10x [4]. Additionally, recent work [27, 41] has significantly improved garbage collection overhead for KV separation [10, 37] making it production ready.

We design and implement Talos, an efficient rack-scale LSM-based KV store. Talos significantly reduces I/O amplification and CPU overhead in secondary nodes and makes replication in the KV store practical. Talos uses the Kreon [9, 34] open-source storage engine, which is an LSM-based KV store with KV separation, and RDMA communication for data replication, client-server, and server-server communication. Talos’s main novelty lies in the fact that it eliminates compactions in the replicas and sends a pre-built index from the primaries.

The three main design challenges in Talos are the following. First, to efficiently replicate the data (value log) Talos uses an efficient RDMA-based primary-backup communication protocol. This protocol does not require the involvement of the replica CPU in communication operations [39].

Second, since the index of the primary contains pointers to its value log, Talos implements an efficient rewrite mechanism at the backups. More precisely, it takes advantage of the property of Kreon that allocates its space aligned in segments (currently set to 2 MB) and creates mappings between primary and backup segments. Then, it uses these mappings to rewrite pointers at the backups efficiently. This approach allows Talos to operate at larger growth factors to save space without significant CPU overhead.

Finally, to reduce CPU overhead for client-server communication, Talos implements an RDMA protocol with one-sided RDMA write operations. Talos’s protocol supports variable size messages that are essential for KV stores. Since these messages are sent in a single round trip, Talos is able to reduce the processing overhead at the server.

We evaluate Talos’s performance using a modified version of the Yahoo Cloud Service Benchmark (YCSB) [12] that supports variable key-value sizes for all YCSB workloads, similar to Facebook’s [8] production workloads. Our results show that our index replication method compared to a baseline implementation that performs compactions at the backups spends 10× fewer CPU cycles per operation to replicate its index. Furthermore, it has 1.1 − 1.7× higher throughput, reduces I/O amplification by 1.1 − 2.3×, and increases CPU efficiency by 1.2 − 1.6×. Overall, Talos technique of sending and rewriting a pre-built index gives KV stores the ability to operate at larger growth factors and save space without spending precious CPU cycles [4, 14].

2 Background

**LSM tree** LSM tree [32] is a write-optimized data structure that organizes its data in multiple hierarchical levels. These levels grow exponentially by a constant growth factor $f$. The first level ($L_0$) is kept in memory, whereas the rest of the levels are on the storage device. There are different ways to organize data across LSM levels [21, 32]. However, in this work, we focus on leveled LSM KV stores that organize each level in non-overlapping ranges.

In LSM KV stores, the growth factor $f$ determines how much the levels grow. The minimum I/O amplification is for $f = 4$ [4]. However, systems choose larger growth factors (typically 8-10) because a larger $f$ reduces space amplification at the expense of increased I/O amplification for high update ratios, assuming that intermediate levels contain only update and delete operations. For example, assuming an $L_0$ size of 64 MB and a device of 4 TB capacity an increase of the growth factor from 10 to 16 results in 6% space savings.

**KV separation** Current KV store designs [10, 16, 25, 29, 34] use the ability of fast storage devices to operate at a high (close to 80% [34]) percentage of their maximum read throughput under small and random I/Os to reduce I/O amplification. The main techniques are KV separation [3, 10, 16, 25, 29, 34] and hybrid KV placement [27, 41]. KV separation appends values in a value log instead of storing values with the keys in the index. As a result, they only re-organize the keys (and pointers) in the multi-level structure, which, depending on the KV pairs sizes, can reduce I/O amplification by up to 10x [4]. Hybrid KV placement [27, 41] is a technique that extends KV separation and solves problem of the garbage collection overhead introduced by the garbage collection in the value log, especially for medium $\geq 100$ B and large $\geq 1000$ B KV pairs.

**RDMA** RDMA verbs is a low-level API for RDMA-enabled applications. The verbs API operates atop of various lossless transport protocols such as Infiniband or RDMA over Converged Ethernet (RoCE). The verbs API supports two-sided send/receive message operations and one-sided RDMA read/write operations. In send and receive operations, both the sender and the receiver actively participate in the communication, consuming CPU cycles. RDMA read and write operations allow one peer to directly read or write the memory of a remote peer without the remote one having to post an operation, hence bypassing the remote node CPU and consuming CPU cycles only in the originating node.

**Kreon** Kreon [9, 34] is an open-source persistent LSM-based KV store designed for fast storage devices (NVMe).
Kreon increases CPU efficiency and reduces I/O amplification using (a) KV separation, and (b) Memory-mapped I/O for its I/O cache and direct I/O for writing data to the device. To perform KV separation [1,25,29,33], Kreon stores key-value pairs in a log and keeps an index with pointers to the key-value log. This technique reduces I/O amplification up to $10 \times$ [4]. A multilevel LSM structure is used to organize its index. The first level $L_0$ lies in memory, whereas the rest of the levels are on the device. Kreon organizes each level internally as a B+-tree where its leaves contain the <key_prefix, value_location> pairs. Finally, all logical structures (level’s indexes and value log) are represented as a list of segments on the device. The segment is the basic allocation unit in Kreon and is currently set to 2 MB. All segment allocations in Kreon are segment aligned.

Kreon uses two different I/O paths: memory-mapped I/O to manage its I/O cache, access the storage devices during read and scan operations, and to write its value log. Furthermore, it uses direct I/O to read and write the levels during compactions.

3 Design

3.1 Overview

Talos is a persistent rack-scale KV store that increases CPU efficiency in backup regions for data replication purposes. Talos uses a primary-backup protocol [7] for replicating the data via RDMA writes without involving the CPU of the backups [39] for efficiency purposes. To reduce the overhead of keeping an up-to-date index at the backups, we design and implement the Send Index operation for systems that use KV-separation [3,16,29,34] or hybrid KV placement [27,41].

Primary servers, after performing a compaction from level $L_i$ to $L_{i+1}$, send the resulting $L_{i+1}$ to their backups in order to eliminate compactions in backup regions. Because $L_{i+1}$ contains references to the primary’s storage address space, backups use a lightweight rewrite mechanism to convert the primary’s $L_{i+1}$ into a valid index for their own storage address space. During the Send Index operation, the backup uses metadata (hundreds of KB) retrieved during the replication of the KV log to translate pointers of the primary’s KV log into its own storage space.

We design and implement an RDMA Write-based protocol for both its server-server and client-server communication. We build our protocol using one-sided RDMA write operations because they reduce the network processing CPU overhead at the server [24] due to the absence of network interrupts. Furthermore, Talos, as a KV store, must support variable size messages. We design our protocol to complete all KV operations in a single round trip to reduce the messages processed per operation by the servers.

Talos uses Kreon [9,34] KV store for efficiently managing the index over its local devices. We modify Kreon to use direct I/O to write its KV log to further reduce CPU overhead for consecutive write page faults, since write I/Os are always large. Direct I/O also avoids polluting the buffer cache from compaction traffic.

Finally, Talos partitions the key-value space into non-overlapping key ranges named regions and offers clients a CRUD API (Create, Read, Update, Delete) as well as range (scan) queries. Talos consists of the three following entities, as shown in Figure 1:

1. Zookeeper [20], a highly available service that keeps the metadata of Talos highly available and strongly consistent, and checks for the health of Talos region servers.
2. Region servers, which keep a subset of regions for which they either have the primary or the backup role.
3. Talos master, which is responsible for assigning regions to region servers and orchestrating recovery operations after a failure.

3.2 Primary-backup Value Log Replication

We design and implement a primary-backup replication protocol to remain available and avoid data loss in case of failures. Each region server stores a set of regions and has either the primary or backup role for any region in its set. The main design challenge that Talos addresses is to replicate data and keep full indexes at the backups with low CPU overhead. Having an up-to-date index at each backup is necessary to provide a fast recovery time in case of a failure.

Talos implements a primary-backup protocol over RDMA for replication [7,39]. On initialization, the primary sends a message to each backup to request an RDMA buffer of the same size as Kreon’s value log segment (2 MB). When a client issues an insert or update KV operation, the primary replicates this operation in its set of backup servers. The primary completes the client’s operation in the following three steps:

1. Inserts the KV pair in Kreon, which returns the offset of the KV pair in the value log, as shown in step 1 in Figure 2a).
2. Appends (via RDMA write operation) the KV pair to the RDMA buffer of each replica at the corresponding offset, as shown in step 2 in Figure 2a).

3. Sends a reply to the client after receiving the completion event from all backups for the above RDMA write operation.

The backup’s CPU is not involved in any of the above steps due to the use of RDMA write operations. When a client receives an acknowledgment it means that its operations have been replicated to all the memories of the replica set.

When the last log segment of the primary becomes full, the primary writes this log segment to persistent storage and sends a flush message to each backup requesting them to persist their RDMA buffer, as shown in step 3 in Figure 2a. Backup servers then copy their RDMA buffer to the last log segment of the corresponding Kreon region and write that log segment to their persistent storage. Backup servers also update their log segment map, as shown in step 4 in Figure 2a.

The log segment map contains entries of the form <primary value log segment, replica value log segment>. Each backup server keeps this map and updates it after each flush message. Backups use this map to rewrite the primary index. We will discuss this index rewriting mechanism in more detail in Section 3.3.

Each backup keeps the log segment map per backup region in memory. The log segment map has a small memory footprint; for a 1 TB device capacity and two replicas, the value log will be 512 GB in the worst case. With the segment size set to 2 MB, the memory size of the log segment map across all regions will be at most 4 MB. In case of primary failure, the new primary informs its backups about the new mappings.

### 3.3 Efficient Backup Index Construction

**Talos** instead of repeating the compaction process at each server to reduce network traffic, takes a radical approach. Primary executes the heavy, in terms of CPU, compaction process of $L_i$ and $L_i+1$ and sends the resulting $L_i'_{i+1}$ to the backups. This approach has the following advantages. First, servers do not need to keep an $L_0$ level for their backup regions, reducing the memory budget for $L_0$ by $2 \times$ when keeping one replica per region or by $3 \times$ when keeping two replicas. Second, backups save device I/O and CPU since they do not perform compactions for their backup regions.

Essentially, this approach trades network I/O traffic for CPU, device I/O, and memory at servers since network bulk transfers are CPU efficient due to RDMA. The main design challenge to address is sending the primary level index to the backup in a format that backups can rewrite with low CPU overhead. **Talos** implements the rewriting process at the backup as follows.

When level $L_i$ of a region in a primary is full, **Talos** starts a compaction process to compact $L_i$ with $L_{i+1}$ into $L_{i+1}'$. The primary reads $L_i$ and $L_{i+1}$ and builds $L_{i+1}'_B$-tree index. $L_{i+1}'$ is represented in the device as a list of segments (currently set to 2 MB) which contains either leaf or index nodes of the $B$-tree. Leaf nodes contain pairs of the form <key prefix, pointer to value log> whereas index nodes contain pairs of the form <pivot, pointer to node>.
To transfer the $L_{i+1}'$ index, the primary initially requests from each backup to register an RDMA buffer of segment size. Talos only uses these buffers during the $L_{i+1}'$ index transfer and deregisters and frees them once the compaction is completed.

On receiving a leaf segment, each backup region server parses it and performs the following steps. Leaf segments contain key prefixes that work across both indexes. The backup has to rewrite pointers to the value log before using them. Talos’s storage engine performs all allocations in 2 MB aligned segments. As a result, the first high 22 bits of a device offset refer to the segment’s start device offset. The remaining bits are an offset within that segment. To rewrite the value log pointers of the primary, the backup first calculates the segment start offset of each KV pair. Since all segments are aligned, it does this through a modulo operation with segment size. Then it issues a lookup in the log map and replaces the primary segment address with its local segment address.

For index segments, Talos keeps in-memory an index map for the duration of $L_{i+1}'$ compaction. Backups add entries to this map whenever they receive an index segment from the primary. This map contains entries using as the primary’s index segment as the key and the corresponding backup’s index segment as the value, as shown in Figure 2b. This mechanism translates pointers to index or leaf nodes within the segment the same way as it does for value log pointers in leaves. Finally, on compaction completion, the primary sends the root node offset of $L_{i+1}$ to each backup, which each backup translates to its storage space.

3.4 Failure Detection

Talos uses Zookeeper’s ephemeral nodes to detect failures. An ephemeral node is a node in Zookeeper that gets automatically deleted when its creator stops responding to heartbeats of Zookeeper. Every region server creates an ephemeral node during its initialization. In case of a failure, the Talos master gets notified about the failure and runs the corresponding recovery operation. In case of Talos master failure, all region servers get notified about its failure through the ephemeral node mechanism. Then, they run an election process through Zookeeper and decide which node takes over as Talos master.

3.5 Failure Recovery

Talos uses Zookeeper, similar to other systems [2,28], to store its region map. Each entry in the region map consists of the range of the region <start key, end key>, the primary server responsible for it, and the list of its backup servers. The region map is infrequently updated when a region is created, deleted after a failure, or during load-balancing operations. Therefore, in Talos Zookeeper operations are not in the common path of data access.

The Talos master reads the region map during its initialization and issues open region commands to each region server in the Talos cluster, assigning them a primary or a backup role. After initialization, the role of the Talos master is to orchestrate the recovery process in case of failures and to perform load balancing operations.

Clients read and cache the region map during their initialization. Before each KV operation, clients look up their local copy of the region map to determine the primary region server where they should send their request. Clients cache the region map since each region entry is 64 B, meaning just 640 KB are enough for a region map with 10,000 regions, and changes to it are infrequent. When a client issues a KV operation to a region server that is not currently responsible for the corresponding range, the region server instructs it to update their region map.

Talos has to handle three distinct failure cases: 1) backup failure, 2) primary failure, and 3) Talos master failure. Since each Talos region server is part of multiple region groups, a single node failure results in numerous primary and backup failures, which the Talos master handles concurrently. First, we discuss how we handle backup failures.

In case of a backup failure, the Talos master replaces the crashed region server with another one that is not already part of the corresponding region’s group. The Talos master then instructs the rest of the region servers in the group to transfer the region data to the new member of the region group. The region experiencing the backup failure will remain available throughout the whole process since its primary is unaffected. However, during the reconstruction of the new backup, the region data are more susceptible to future failures, since there’s one less backup copy.

In case of a primary failure, the Talos master first promotes one of the existing backup region servers in that region group to the primary role, and updates the region map. The new primary already has a complete KV log and an index for levels $L_i$, where $i \geq 1$. The new primary region server replays the last few segments of its value log in order to construct an $L_0$ index in its memory before being able to server client requests. Now that a new primary region server exists for the group, the Talos master handles this failure as if it were a backup failure. During the primary reconstruction process, Talos cannot server client requests from the affected region.

When the Talos master crashes, the rest of the region servers in the Talos cluster will be notified through Zookeeper’s ephemeral node mechanism, as discussed in Section 3.4. They will then use Zookeeper in order to elect a new Talos master. During the Talos master downtime, Talos cannot handle any region failures, meaning that any region that has suffered a primary failure will remain unavailable until a new Talos master is elected and initiates the recovery process for any regions that suffered a failure.
3.6 RDMA Write-based Communication Protocol

Talos performs all client server communication via one-sided RDMA write operations [24] to avoid network interrupts and thus reduce the CPU overhead in the server’s receive path [23, 24]. Furthermore, to avoid the overhead of registering and deregistering RDMA memory buffers per KV operation, the server and client allocate a pair of buffers during queue pair (QP) creation. Their size is dynamic within a range (256 KB currently) set by the client on QP creation. region server frees this buffer when a client disconnects or suffers a failure. The client manages these buffers to improve CPU efficiency in the server.

Clients allocate a pair of messages for each KV operation; one for their request and one for the server’s reply. All buffers sizes are multiples of a size unit named message segment size (currently set to 128 bytes). Clients put in the header of each request the offset at their remote buffer where region server can write its reply. Upon completion of a request, the region server prepares the request’s reply in the corresponding local circular buffer at the offset supplied by the client. Then it issues an RDMA write operation to the client’s remote circular buffer at the exact offset. Figure 3 shows a visual representation of these steps. As a result, the region server avoids expensive synchronization operations between its workers to allocate space in the remote client buffers and update buffer state metadata (free or reserved). If the client allocates a reply of insufficient size, the region server sends part of the reply and informs the client to retrieve the rest of the data.

3.6.1 Receive Path

To detect incoming messages, in the absence of network interrupts, each region server has a spinning thread which spins on predefined memory locations in its corresponding clients’ remote circular buffers, named rendezvous points. The spinning thread detects a new message by checking for a magic number in the last field of the message header, called the receive field, at the next rendezvous location. After it detects a new message header, it reads the payload size from the message header to determine the location of the message’s tail. Upon successful arrival of the tail, it assigns the new client request to one of its workers and advances to the next rendezvous location of the circular buffer.

To support variable size messages Talos adds appropriate padding so that their size is message segment aligned. This quantization has two benefits: 1) Possible rendezvous points are at the start of each segment, offset by the size of a message header minus the size of the header’s receive field. Upon reception of a message the region server advances its rendezvous point by adding the current message size. 2) The region server doesn’t have to zero the whole message upon each request completion; it only zeros the possible rendezvous points in the message segments where the request was written.

3.6.2 Reset Operation in Circular Buffers

There are two ways to reset the rendezvous point to the start of the circular buffer: 1) When the last message received in the circular buffer takes up its whole space, the server will pick the start of the circular buffer as the next rendezvous location, and 2) When the remaining space in the circular buffer is not enough for the client to send their next message, they will have to circle back to the start of the buffer. In this case, they will send a reset rendezvous message to inform the server that the next rendezvous location is now at the start of the circular buffer.

3.6.3 Task Scheduling

To limit the max number of threads, Talos uses a configurable number of workers. Each worker has a private task queue to avoid CPU overhead associated with contention on shared data. In this queue, the spinning thread places new tasks, as shown in Figure 4. Workers poll their queue to retrieve a new request and sleep if no new task is retrieved within a set period of time (currently 100 µs). The primary goal of Talos’s task scheduling policy is to limit the number of wake-up operations, since they include crossings between user and kernel space. The spinning thread assigns a new task to the current worker unless its task queue has more than a
set amount of tasks already enqueued. In the latter case, the spinning thread selects a running worker with less than that set amount of queued tasks and assigns to it the new task. If all running workers already exceed the task queue limit, the spinning thread wakes up a sleeping worker and enqueues this task to their task queue.

4 Evaluation Methodology

Our testbed consists of two servers where we run the KV store. The servers are identical and are equipped with an AMD EPYC 7551P processor running at 2 GHz with 32 cores and 128 GB of DDR3 DRAM. Each server uses as a storage device a 1.5 TB Samsung NVMe from the PM173X series and a Mellanox ConnectX 3 Pro RDMA network card with a bandwidth of 56 Gbps. We limit the buffer cache used by Talos’s storage engine (Kreon) using cgroups to be a quarter of the dataset in all cases.

In our experiments, we run the YCSB benchmark [12] workloads Load A and Run A – Run D. Table 1 summarizes the operations run during each workload. We use a C++ version of YCSB [35] and we modify it to produce different values according to the KV pair size distribution we study. We run Talos with a total of 32 regions across both servers. Each server serves as primary for the 16 and as backup for the other 16. Furthermore, each server has 2 spinning threads and 8 worker threads in all experiments. The remaining cores in the server are used for compactions.

In our evaluation, we also vary the KV pair sizes according to the KV sizes proposed by Facebook [8], as shown in Table 2. We first evaluate the following workloads where all KV pairs have the same size: Small (S), Medium (M), and Large (L). Then, we evaluate workloads that use mixes of S, M, and L KV pairs. We use small-dominated (SD) KV size distribution proposed by Facebook [8], as well as two new mixed workloads: MD (medium dominated) and LD (large dominated). We summarize these KV size distributions in Table 2.

We examine the throughput (KOperations/s), efficiency (KCycles/operation), I/O amplification, and network amplification of Talos for the three following setups: (1) without replication (No Replication), (2) with replication, using our mechanism for sending the index to the backups (Send Index), and (3) with replication, where the backups perform compactions to build their index (Build Index), which serves as a baseline. In Build Index, servers keeps additionally an $L_0$ level in memory for their backup regions, whereas in Send Index, they do not. For these two setups two be equal, and since we always use the same number of regions, in Build Index we configure each region $L_0$ size to be half of the $L_0$ size used in the other two setups.

We measure efficiency in cycles/op and define it as:

$$\text{efficiency} = \frac{\text{CPU utilization \times cycles \times cores}}{\text{average ops/s}} \text{ cycles/op},$$

where $\text{CPU utilization}$ is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by mpstat. As $\text{cycles/s}$ we use the per-core clock frequency. $\text{average ops/s}$ is the throughput reported by YCSB, and $\text{cores}$ is the number of system cores including hyperthreads.

I/O amplification measures the excess device traffic generated due to compactions (for primary and backup regions) by Talos, and we define it as:

$$\text{IO amplification} = \frac{\text{device traffic}}{\text{dataset size}},$$

where $\text{device traffic}$ is the total number of bytes read from or written to the storage device and $\text{dataset size}$ is the total size of all key-value requests issued during the experiment.

Lastly, network amplification is a measure of the excess network traffic generated by Talos, and we define it as:

$$\text{network amplification} = \frac{\text{network traffic}}{\text{dataset size}},$$

where $\text{network traffic}$ is the total number of bytes sent by and received from the servers’ network cards.

5 Experimental Evaluation

In our evaluation of Talos we answer the following questions:
1. How does our backup index construction (Send Index) method compare to performing compactions in backup regions (Build Index) to construct the index?

2. Where does Talos spend its CPU cycles? How many cycles does Send Index save compared to Build Index for index construction?

3. How does increasing the growth factor affect Talos?

4. Does Send Index improve performance and efficiency in small-dominated workloads, where KV separation gains diminish?

5.1 Talos Performance and Efficiency

In Figure 5, we evaluate Talos using YCSB workloads Load A and Run A – Run D for the SD [8] workload. Since replication doesn’t impact read-dominated workloads, the performance in workloads Run B – Run D remains the same for all three deployments. We focus the rest of our evaluation on the insert and update heavy workloads Load A and Run A.

We run Load A and Run A workloads for all six KV size distributions and with growth factor 4 which minimizes I/O amplification (but not space amplification). We set the $L_0$ size to 64K keys for the No Replication and Send Index configurations and to 32K keys for the Build Index configuration, since Build Index has twice as many $L_0$ indexes. We measure throughput, efficiency, and I/O amplification for the three different deployments explained in Section 4. We summarize these results in Figure 6. We also report the tail latency in these workloads for the SD KV size distribution in Figure 7.

Compared to Build Index, Send Index increases throughput by $1.1 - 1.7 \times$ for all KV size distributions, increases CPU efficiency by $1.2 - 1.6 \times$, and reduces I/O amplification by $1.1 - 3.0 \times$. Also, it is crucial to notice that compared to No Replication, Build Index increases I/O amplification by $1.6 - 3.4 \times$ while Send Index only increases I/O amplification by $1.4 - 1.5 \times$, since eliminating compactions in backup regions means no additional read traffic for replication. Furthermore, Talos increases CPU efficiency by replacing expensive I/O operations and key comparisons during compactions with a single traversal of the new index segments and hash table accesses to rewrite them.

Sending the backup region indexes over the network increases network traffic up to $1.2 \times$. This trade-off favors Talos since it pays a slight increase in network traffic for increased efficiency and decreased I/O amplification.

We also measure the tail latency for YCSB workloads Load A and Run A using the SD KV size distribution. As shown in Figure 7, Send Index improves the 99, 99.9, and 99.99% tail latencies from $1.1 \times$ to $1.5 \times$ compared to Build Index for all Load A and Run A operations.

5.2 Cycles/Op Breakdown

We run YCSB workloads Load A and Run A and profile Talos using perf with call graph tracking enabled. We profile Talos while using Send Index and Build Index configurations. We use the call graph profiles generated to measure where CPU cycles are spent in Talos for Send Index and Build Index. We count the CPU cycles spent on four major parts of our system:

- **Storage**: Cycles spent in KV store operations, excluding replication

- **Network and Runtime**: Cycles spent on detecting, scheduling, and processing client requests, excluding storage and replication

- **Log Replication**: Cycles spent on replicating KV pairs in a backup’s value log

- **Index Replication**: Cycles spent to construct indexes for backup regions. Send index spends these cycles to rewrite the index segments they receive from primary region servers. Build Index spends these cycles on compactions and iterating KV value log segments to insert them into Kreon’s $L_0$ index

- **Other**: All cycles not spent in the above categories

Figure 8 summarizes the results of our profiling. Talos’s Send Index technique requires 28% fewer cycles than performing compactions to construct backup indexes. This 28% cycles amount to roughly 12K cycles per operation. They are divided into: 5.5K fewer cycles for replicating backup indexes, 2K fewer cycles spent on storage, 2K fewer cycles spent on network and runtime processing, and 2.5K fewer cycles spent on other parts of the system. With the Send Index method, Talos region servers spend $10 \times$ fewer cycles on constructing backup indexes and $1.36 \times$ fewer cycles overall when compared to using the Build Index method.
Figure 6: Throughput, efficiency, I/O amplification, and network amplification for the different key-value size distributions during the (a) YCSB Load A and (b) Run A workloads.
Network and Runtime

Time (s)

100000

1.16x cycles on network and runtime processing. This is due to 4 to 8-10) and pay the penalty of higher I/O amplification during Load A remain constant when increasing the growth factor. However, during Run A, the gains of our Send Index approach compared to Build Index increase. Most notably, with growth factors 12 and 16, the performance improvement is 2.1 and 1.7× respectively. Similarly, efficiency is improved by 2.4 and 1.9×, and I/O amplification is decreased by 60%.

KV stores intentionally increase growth factor [4, 14] (from 4 to 8-10) and pay the penalty of higher I/O amplification to reduce space. In the Build Index, this penalty is further amplified to two or three times according to the number of replicas per region. However, Send Index eliminates these redundant compactions and allows us to increase the growth factor and thus the space efficiency of LSM tree-based KV stores without sacrificing significantly performance or CPU efficiency.

5.4 Small KVs Impact

The KV separation [10, 29, 34] and hybrid placement techniques [27, 41] gains in I/O amplification decrease for small ≤33 B KV pairs, which are important for internet-scale workloads [8]. This decrease is because the gains for KV separation of small KV pairs is around 2× [41]. However, if we include also the garbage collection overheads, the gains further diminish making KV separation identical to putting KVs in-place as RocksDB [17] does.

In this experiment, we investigate the impact that small KV pairs percentage has on the efficiency of Send Index method. We set the growth factor to 12 and examine four workloads (Run A, Load 1, Load 2 and Load 3) where we vary small KV pairs percentage to 40%, 60%, 80%, and 100%. In all four cases, we equally divide the remaining percentage between medium and large KV pairs.

As shown in Figure 10, Send Index has from 1.2 to 2.3× better throughput and efficiency than Build Index across all workloads. I/O amplification for Build Index increases from 7.4 to 9.3×. From the above, we conclude that the Send Index method has significant benefits even for workloads that consist of 80%-90% small KV pairs.

6 Related Work

In this section we group related work in the following categories: (a) LSM tree compaction offload techniques, (b) Log and index replication techniques, and (c) efficient RDMA protocols for KV stores:

Compaction offload: Acazoo [19] splits its data into shards and keeps replicas for each shard using the ZAB [20] replication protocol. Acazoo offloads compaction tasks from a shard’s primary to one of its replicas. Then, on compaction completion, it reconfigures the system through an election to make the server with the newly compacted data the primary.

Hailstorm [5] is a rack-scale persistent KV store. It uses a distributed filesystem to provide a global namespace and scatters SSTs across the rack (in a deterministic way). Thus it can scale its I/O subsystem similar to HBase/HDFS [2]. Unlike HBase, it schedules compaction tasks to other servers through the global namespace offered by the distributed filesystem substrate. Unlike these systems, Talos can efficiently keep both the primary and backup indexes up to date through the send index operation by using RDMA to perform efficient bulk network transfers.
Figure 9: Send Index improvement over Build Index for Load A, Run A and different growth factors.

Figure 10: Throughput, efficiency, I/O amplification, and network amplification for increasing percentages of small KVs during (a) YCSB Load A and (b) Run A workloads.
Log and index replication techniques: Rose [36] is a distributed replication engine that targets racks with hard disk drives and TCP/IP networking where device I/O is the bottleneck. In particular, it replicates data using a log and builds the replica index by applying mutations in an LSM tree index. The LSM tree removes random reads for updates and always performs large I/Os. Talos shares the idea of Rose to use the LSM tree to build an index at the replica. However, it adapts its design for racks that use fast storage devices and fast RDMA networks where the CPU is the bottleneck. It does this by sending and rewriting the index and removing redundant compactions at the backups.

Tailwind [39] is a replication protocol that uses RDMA writes for data movement, whereas for control operations, it uses conventional RPCs. The primary server transfers log records to buffers at the backup server by using one-sided RDMA writes. Backup servers are entirely passive; they flush their RDMA buffers to storage periodically when the primary requests it. They have implemented and evaluated their protocol on RAMCloud, a scale-out in-memory KV store. Tailwind improves throughput and latency compared to RAMCloud. Talos adopts Tailwind’s replication protocol for its value log but further proposes a method to keep a backup index efficiently.

Efficient RDMA protocols for KV stores: Kalia et al. [24] analyze different RDMA operations and show that one-sided RDMA write operations provide the best throughput and latency metrics. Talos uses one-sided RDMA write operations to build its protocol.

A second parameter is whether the KV store supports fixed or variable size KVs. For instance, HERD [23], a hash-based KV store, uses RDMA writes to send requests to the server, and RDMA send messages to send a reply back to the client. Send messages require a fixed maximum size for KVs. Talos uses only RDMA writes and appropriate buffer management to support arbitrary KV sizes. HERD uses unreliable connections for RDMA writes, and an unreliable datagram connection for RDMA sends. Note that they decide to use RDMA send messages and unreliable datagram connections because RDMA write performance does not scale with the number of outbound connections in their implementation. In addition, they show that unreliable and reliable connections provide almost the same performance. Talos uses reliable connections to reduce protocol complexity and examines their relative overhead in persistent KV stores. We have not detected scalability problems yet.

Other in-memory KV stores [15, 31, 40] use one-sided RDMA reads to offload read requests to the clients. Talos does not use RDMA reads since lookups in LSM tree-based systems are complex. Typically, lookups and scan queries consist of multiple accesses to the devices to fetch data. These data accesses must also be synchronized with compactions.

7 Conclusions

In this paper, we design Talos, a replicated persistent LSM-based KV store that targets racks with fast storage devices and fast network (RDMA). Talos implements an RDMA write-based client-server protocol and replicates its data using an efficient RDMA write-based primary-backup protocol. Talos takes a radical approach to keep an up-to-date index at the backups and avoid rebuilding it in case of a failure. Instead of performing compactions at the backup servers (Build Index) primary sends a pre-built index after each level compaction (Send Index), trading a slight increase in network traffic for increased CPU efficiency and decreased I/O amplification. Talos implements an efficient index rewrite mechanism at the backups, which is used to translate the primary index’s pointers into valid backup index pointers. Compared to Build Index, we find that Send Index increases throughput by up to 1.7×, CPU efficiency by up to 1.6×, decreases I/O amplification by up to 3.0×, and decreases tail latency by up to 1.5× during YCSB Load A and Run A workloads.

Our approach enables KV stores to operate with larger growth factors in order to save space (6% space saved by increasing the growth factor from 10 to 16), without inducing significant CPU overhead. Furthermore, we show that Send Index provides significant benefits even in workloads where small KVs account for as much as 90% of the total operations. We believe that the Send Index technique can be adopted by state-of-the-art replicated LSM-based KV stores to increase their CPU and space efficiency.

References

[1] J.-S. Ahn, M. A. Qader, W.-H. Kang, H. Nguyen, G. Zhang, and S. Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining lsm-tree and copy-on-write b+-tree. In Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems, HotStorage’19, page 9, USA, 2019. USENIX Association.

[2] Apache. Hbase. https://hbase.apache.org/, 2018.

[3] Aurelius. Titandb, June 2012.

[4] N. Batsaras, G. Saloustros, A. Papagiannis, P. Fatourou, and A. Bilas. Vat: Asymptotic cost analysis for multilevel key-value stores, 2020.

[5] L. Bindschaedler, A. Goel, and W. Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.
[6] D. Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[8] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies*, FAST ’16, pages 209–223, Santa Clara, CA, Feb. 2020. USENIX Association.

[9] CARV-ICS. Kreon. https://github.com/CARV-ICS-FORTH/kreon, 2021.

[10] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. Hashkv: Enabling efficient updates in kv storage via hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC ’18, pages 1007–1019, Berkeley, CA, USA, 2018. USENIX Association.

[11] K. Chodorow. *MongoDB: The Definitive Guide*. O’Reilly Media, second edition, 2013.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, pages 143–154, New York, NY, USA, 2010. ACM.

[13] 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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[14] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017.

[15] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[16] Facebook. Blobdb. http://rocksdb.org/, 2018. Accessed: October 20, 2021.

[17] Facebook. Rocksdb. http://rocksdb.org/, 2018.

[18] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, Apr. 2021.

[19] P. Garefalakis, P. Papadopoulos, and K. Magoutis. Acac zoo: A distributed key-value store based on replicated lsm-trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 211–220, 2014.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[21] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanegangi. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB ’97, pages 16–25, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[22] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, pages 743–752, 2011.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM ’14, pages 295–306, New York, NY, USA, 2014. ACM.

[24] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, pages 437–450, 2016.

[25] 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 *MSST*, pages 1–14. IEEE Computer Society, 2015.

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

[27] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui. Differentiated key-value storage management for balanced i/o performance. In *2021 USENIX Annual Technical Conference (USENIX ATC ’21)*, pages 673–687. USENIX Association, July 2021.
[28] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, et al. Kudu: Storage for fast analytics on fast data. Cloud-era, inc, 28, 2015.

[29] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In 14th USENIX Conference on File and Storage Technologies (FAST 16), pages 133–148, Santa Clara, CA, Feb. 2016. USENIX Association.

[30] Y. Matsunobu, S. Dong, and H. Lee. Myrocks: Lsm-tree database storage engine serving facebook’s social graph. Proc. VLDB Endow., 13(12):3217–3230, Aug. 2020.

[31] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference, USENIX ATC’13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[32] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. The log-structured merge-tree (lsm-tree). Acta Inf., 33(4):351–385, June 1996.

[33] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 537–550, Denver, CO, 2016. USENIX Association.

[34] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. An efficient memory-mapped key-value store for flash storage. In Proceedings of the ACM Symposium on Cloud Computing, SoCC ’18, pages 490–502, New York, NY, USA, 2018. ACM.

[35] J. Ren. Ycsb-c. https://github.com/basicthinker/YCSB-C, 2016.

[36] R. Sears, M. Callaghan, and E. Brewer. <i>rose</i>: Compressed, log-structured replication. Proc. VLDB Endow., 1(1):526–537, Aug. 2008.

[37] C. Shen, Y. Lu, F. Li, W. Liu, and J. Shu. Novkv: Efficient garbage collection for key-value separated lsm-stores. Oct. 2020.

[38] A. Singhvi, A. Akella, M. Anderson, R. Cauble, H. Deshmukh, D. Gibson, M. M. K. Martin, A. Strominger, T. F. Wenisch, and A. Vahdat. Cliquemaps: Productionizing an rma-based distributed caching system. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM ’21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic rdma-based replication. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’18, pages 851–863, Berkeley, CA, USA, 2018. USENIX Association.

[40] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In Proceedings of the 25th Symposium on Operating Systems Principles, pages 87–104, 2015.

[41] G. Xanthakis, G. Saloustros, N. Batsaras, P. Anastasios, and A. Bilas. Parallax: Hybrib key-value placement in lsm-based key-value stores. In Proceedings of the ACM Symposium on Cloud Computing, SoCC ’21, New York, NY, USA, 2021. ACM.