Tebis: Efficient Replica Index Construction for Persistent LSM-based Key-Value Stores

Anonymous Author(s)
Submission Id: 182

ACM Reference Format:
Anonymous Author(s). 2022. Tebis: Efficient Replica Index Construction for Persistent LSM-based Key-Value Stores . In *ACM European Conference on Systems (Eurosys '22), April 5–8, 2022, Rennes, France.* ACM, New York, NY, USA, 13 pages.

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 *Tebis*, 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 [27, 29, 34, 41]. 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 *Tebis* 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 ontop 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 *Tebis*, an efficient rack-scale LSM-based KV store. *Tebis* significantly reduces I/O amplification and CPU overhead in secondary nodes and makes replication in the KV store practical. *Tebis* 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. *Tebis*'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 *Tebis* are the following. First, to efficiently replicate the data (value log) *Tebis* 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, *Tebis* 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 *Tebis* to operate at larger growth factors to save space without significant CPU overhead.

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

We evaluate *Tebis*'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 $1.1 - 1.7\times$ higher throughput, reduces I/O amplification by $1.1 - 2.3\times$, and increases CPU efficiency by $1.2 - 1.6\times$. Overall, *Tebis* 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 pair 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× [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

Tebis is a persistent rack-scale KV store that increases CPU efficiency in *backup* regions for data replication purposes. *Tebis* 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, *Tebis*, 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.

Tebis 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, *Tebis* 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. *Tebis* consists of the three following entities, as shown in Figure 1:

1. *Zookeeper* [20], a highly available service that keeps the metadata of *Tebis* highly available and strongly consistent, and checks for the health of *Tebis region servers*.

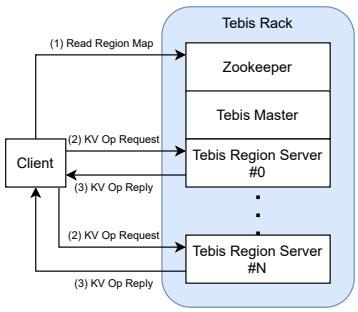


Figure 1. *Tebis* overview.

2. *Region servers*, which keep a subset of regions for which they either have the *primary* or the *backup* role.
3. *Tebis-Master*, which is responsible for assigning regions to *region servers* and orchestrating recovery operations after a failure.

3.2 Primary-backup Value Log Replication

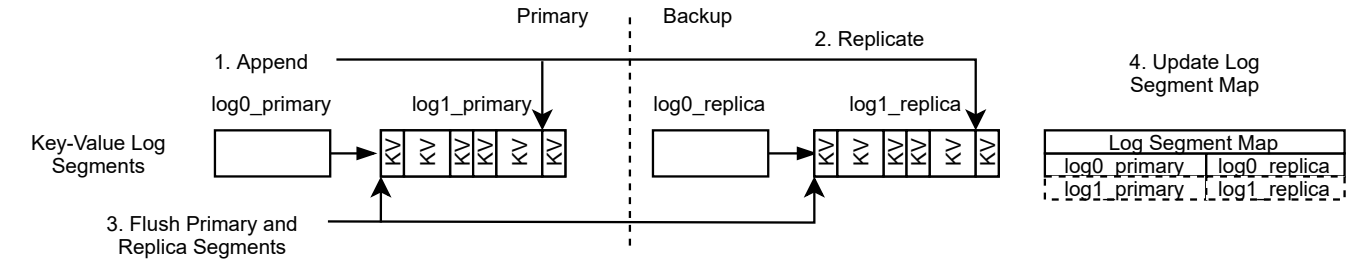
We design and implement a primary-backup replication protocol to remain available and avoid data loss in case of failures. Each *region server* stores a set of regions and has either the *primary* or *backup* role for any region in its set. The main design challenge that *Tebis* addresses is to replicate 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.

Tebis 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:

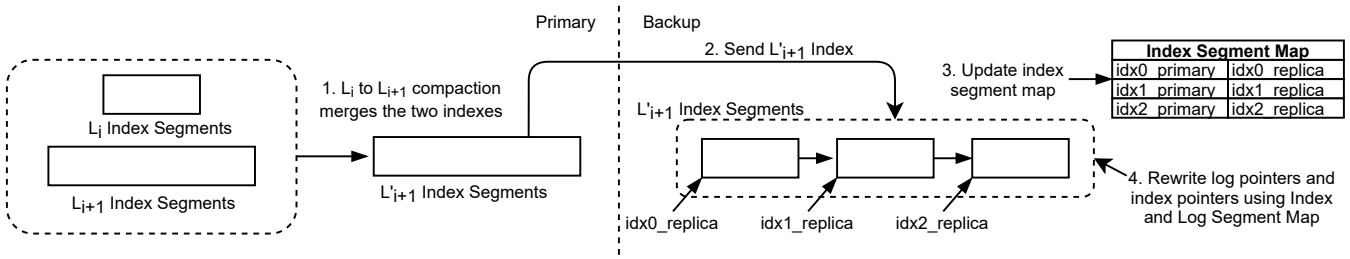
1. Inserts the KV pair in Kreon, which returns the offset of the KV pair in the value log, as shown in step 1 in Figure 2a).
2. Appends (via RDMA write operation) the KV pair to the RDMA buffer of each replica at the corresponding offset, as shown in step 2 in Figure 2a).
3. Sends a reply to the client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



(a) Key-Value log replication process



(b) Key-Value index replication process

Figure 2. Replication in *Tebis*.

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

Tebis 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+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× when keeping one replica per region or by 3× 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. *Tebis* implements the rewriting process at the *backup* as follows.

When level L_i of a region in a *primary* is full, *Tebis* 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 contains 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. *Tebis* 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. *Tebis*'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, *Tebis* 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

Tebis 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 *Tebis-Master* gets notified about the failure and runs the corresponding recovery operation. In case of *Tebis-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 *Tebis-Master*.

3.5 Failure Recovery

Tebis 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 *Tebis* Zookeeper operations are not in the common path of data access.

The *Tebis-Master* reads the region map during its initialization and issues *open region* commands to each *region server* in the *Tebis* cluster, assigning them a *primary* or a *backup* role. After initialization, the role of the *Tebis-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.

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

In case of a *backup* failure, the *Tebis-Master* replaces the crashed *region server* with another one that is not already part of the corresponding region's group. The *Tebis-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 *Tebis-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 *Tebis-Master* handles this failure as if it were a *backup* failure. During the *primary* reconstruction process, *Tebis* cannot server client requests from the affected region.

When the *Tebis-Master* crashes, the rest of the *region servers* in the *Tebis* 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 *Tebis-Master*. During the *Tebis-Master* downtime, *Tebis* cannot handle any region failures, meaning that any region that has suffered a *primary* failure will remain unavailable until a new *Tebis-Master* is elected and initiates the recovery process for any regions that suffered a failure.

3.6 RDMA Write-based Communication Protocol

Tebis 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

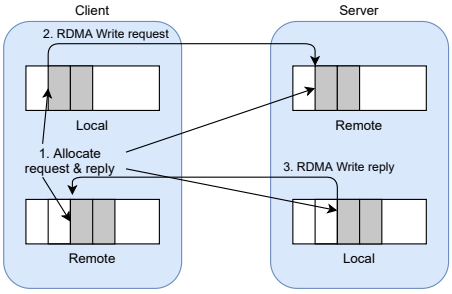


Figure 3. Allocation and request-reply flow of our RDMA Write-based communication protocol.

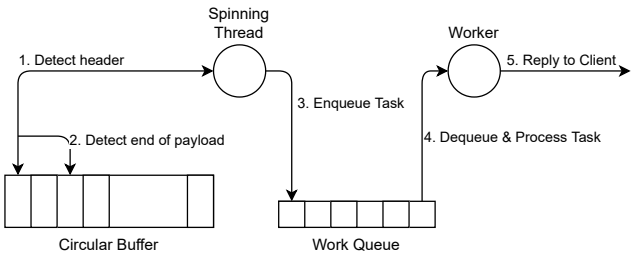


Figure 4. Message detection and task processing pipeline in *Tebis*. For simplicity, we only draw one circular buffer and a single worker.

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 *Tebis* 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, *Tebis* 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 *Tebis*'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 *Tebis*'s storage engine (Kreon) using *cgroups* to be a quarter of the dataset in all cases.

	Workload
Load A	100% inserts
Run A	50% reads, 50% updates
Run B	95% reads, 5% updates
Run C	100% reads
Run D	95% reads, 5% inserts

Table 1. Workloads evaluated with YCSB. All workloads use a Zipfian distribution except for Run D that use latest distribution.

	KV Size Mix S%-M%-L%	#KV Pairs	Cache per Server (GB)	Dataset Size (GB)
S	100-0-0	100M	0.38	3
M	0-100-0	100M	1.4	11.4
L	0-0-100	100M	11.9	95.2
SD	60-20-20	100M	2.8	23.2
MD	20-60-20	100M	3.3	26.5
LD	20-20-60	100M	7.5	60

Table 2. KV size distributions we use for our YCSB evaluation. Small KV pairs are 33 B, medium KV pairs are 123 B, and large KV pairs are 1023 B. We report the record count, cache size per server, and dataset size used with each KV size distribution.

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 *Tebis* 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 *Tebis* 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:

$$efficiency = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}} \; cycles/op,$$

where *CPU_utilization* is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by *mpstat*. As $cycles/s$ we use the per-core clock frequency. $average\_ops/s$ is the throughput reported by YCSB, and *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 *Tebis*, and we define it as:

$$IO\_amplification = \frac{device\_traffic}{dataset\_size},$$

where *device_traffic* is the total number of bytes read from or written to the storage device and *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 *Tebis*, and we define it as:

$$network\_amplification = \frac{network\_traffic}{dataset\_size},$$

where *network_traffic* is the total number of bytes sent by and received from the servers' network cards.

5 Experimental Evaluation

In our evaluation of *Tebis* 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 *Tebis* 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 *Tebis*?
4. Does Send Index improve performance and efficiency in small-dominated workloads, where KV separation gains diminish?

5.1 *Tebis* Performance and Efficiency

In Figure 5, we evaluate *Tebis* 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

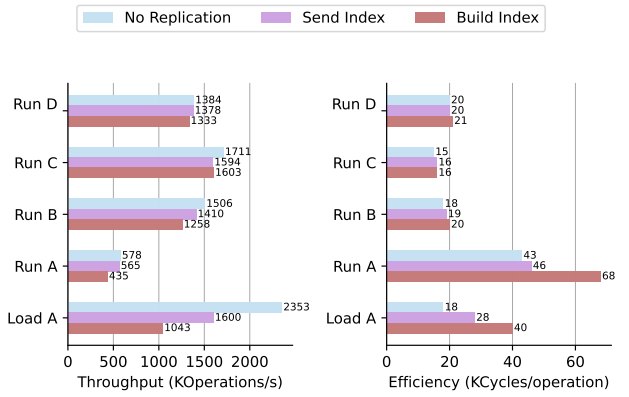


Figure 5. Performance and efficiency of *Tebis* for YCSB workloads Load A, Run A – Run D with the SD KV size distribution.

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× for all KV size distributions, increases CPU efficiency by 1.2 – 1.6×, and reduces I/O amplification by 1.1 – 3.0×. Also, it is crucial to notice that compared to No Replication, Build Index increases I/O amplification by 1.6 – 3.4× while Send Index only increases I/O amplification by 1.4 – 1.5×, since eliminating compactions in *backup* regions means no additional read traffic for replication. Furthermore, *Tebis* 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×. This trade-off favors *Tebis* 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 to 1.5× 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 *Tebis* using *perf* with call graph tracking enabled. We profile *Tebis* while using Send Index and Build Index configurations. We use the call graph profiles generated to measure where CPU cycles are spent in *Tebis* 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.

Tebis'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, *Tebis region servers* spend 10× fewer cycles on constructing *backup* indexes and 1.36× fewer cycles overall when compared to using the Build Index method.

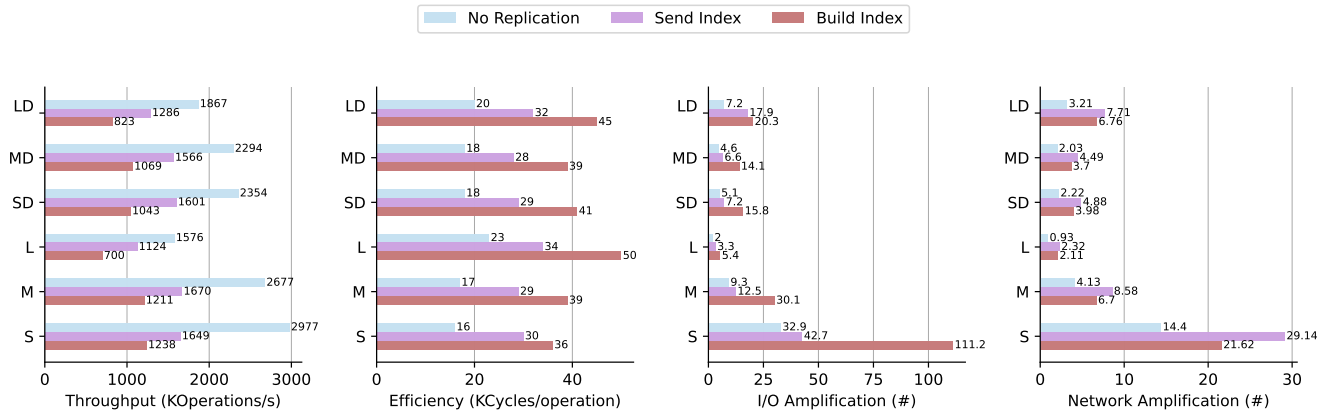
Sending the *primary* index to *backups* eliminates compactions for backup regions resulting in increased CPU efficiency during *backup* index construction. While *backup region servers* have to rewrite these index segments, the rewriting process only involves hash table lookups without requiring any read I/O, resulting in a more efficient *backup* index construction technique.

In comparison with Send Index, Build Index also spends 1.16x cycles on network and runtime processing. This is due to additional queuing effects which are a result of the increased load due to *backup* compactions.

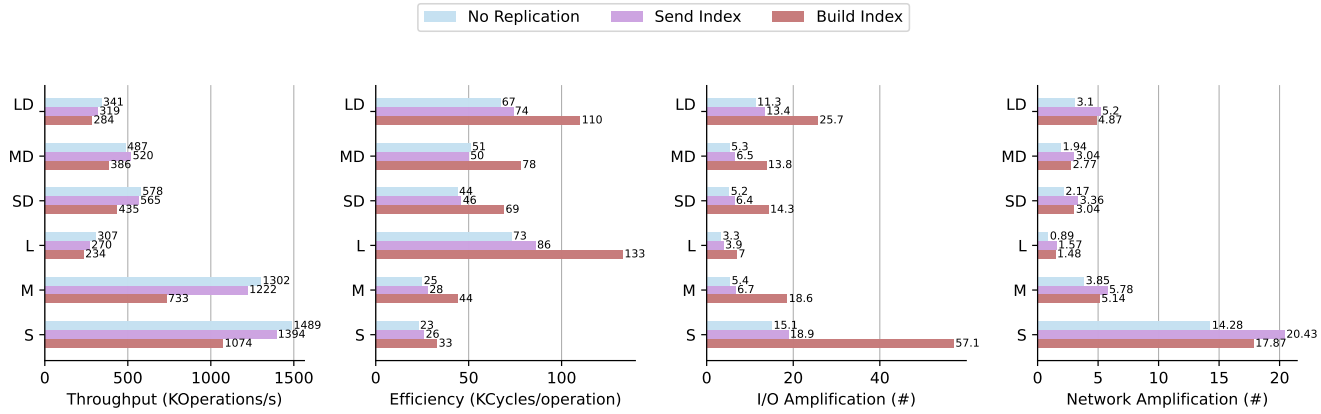
5.3 Impact of Growth Factor

In Figure 9, we show that the gains in performance, efficiency, and 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



(a) Load A YCSB workload



(b) Run A YCSB workload

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.

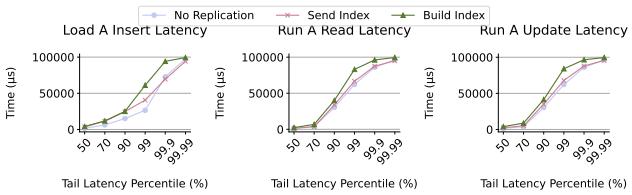


Figure 7. Tail latency for YCSB Load A and Run A workload operations using the SD key-value size distribution.

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.

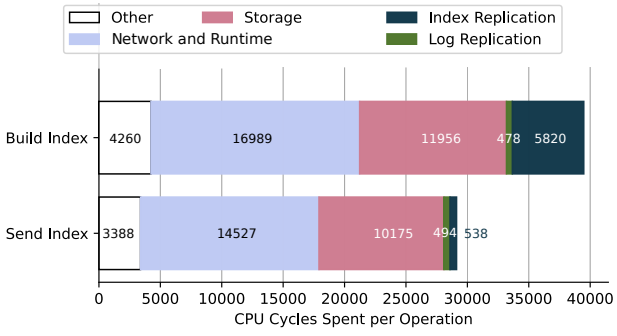


Figure 8. Breakdown of cycles spent per operation on network, storage, log replication and index replication.

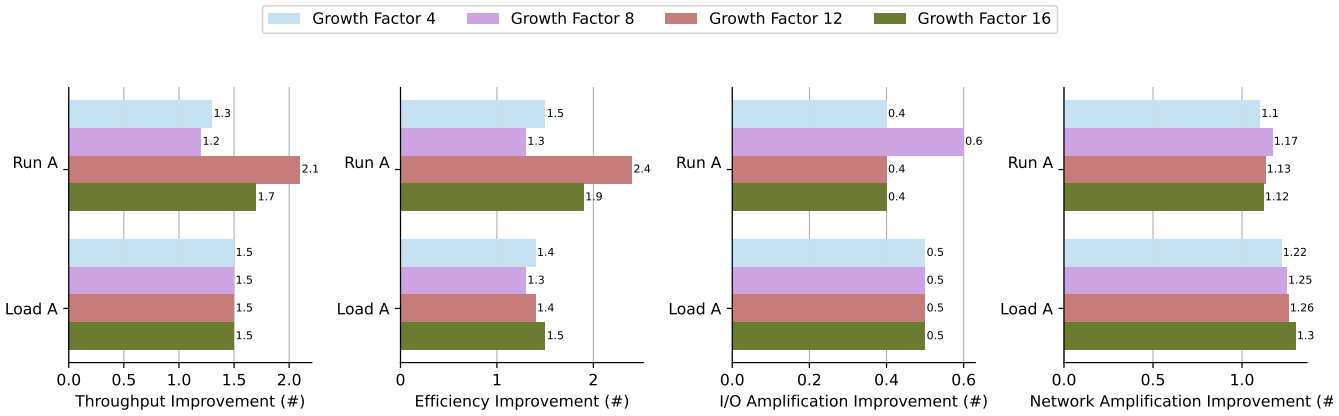
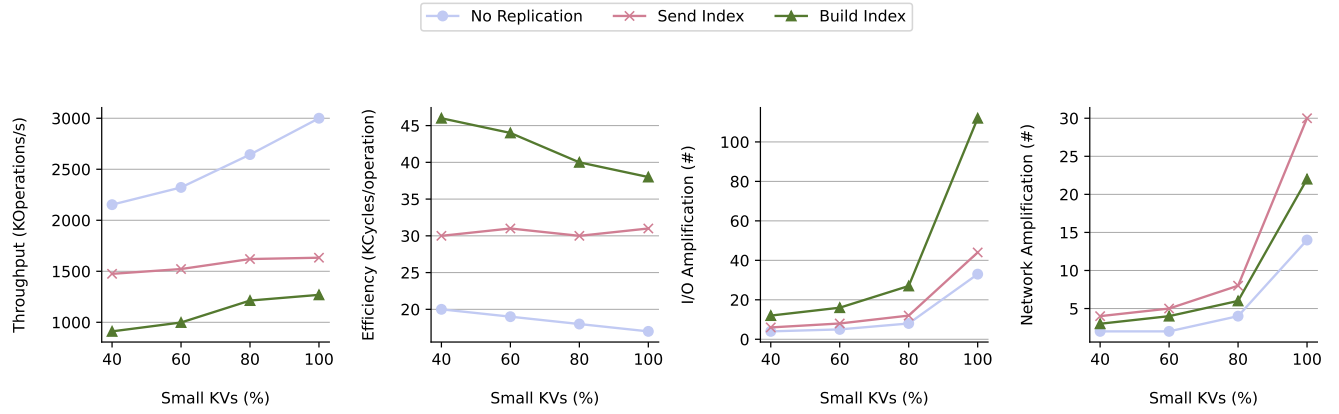
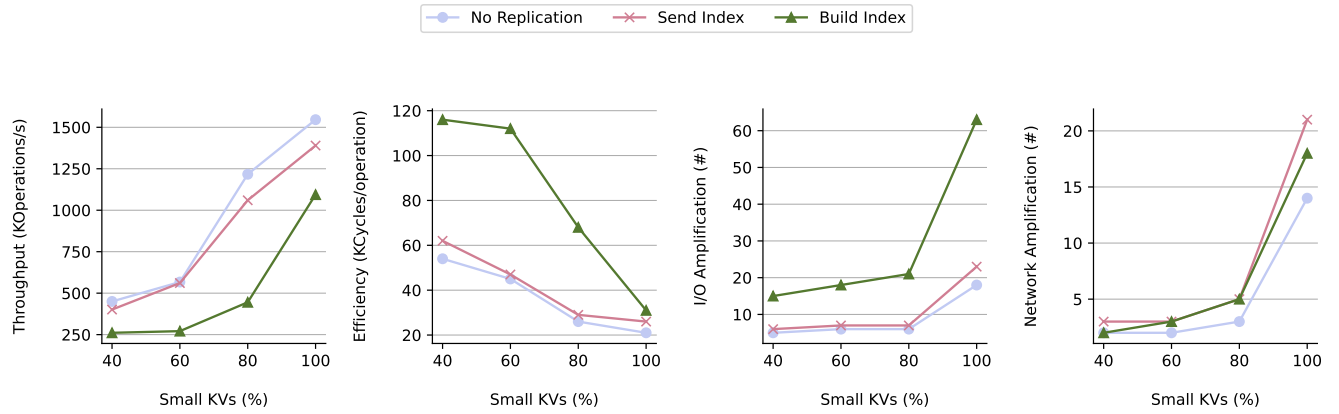


Figure 9. Send Index improvement over Build Index for Load A, Run A and different growth factors.



(a) Load A YCSB workload



(b) Run A YCSB workload

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.

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* $\leq$ 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 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, *Tebis* 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.

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. *Tebis* 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. *Tebis* 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. *Tebis* 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. *Tebis* 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. *Tebis* 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. *Tebis* 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 *Tebis*, a replicated persistent LSM-based KV store that targets racks with fast storage devices and fast network (RDMA). *Tebis* implements an RDMA write-based client-server protocol and replicates its data using an efficient RDMA write-based primary-backup protocol. *Tebis* 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. *Tebis* 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] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. 2019. 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* (Renton, WA, USA) *(HotStorage'19)*. USENIX Association, USA, 9.

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

[3] Aurelius. 2012. *TitanDB*. Retrieved September 30, 2021 from http://titan.thinkaurelius.com/

[4] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. 2020. VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. arXiv:2003.00103 [cs.DC]

[5] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 301–316. https://doi.org/10.1145/3373376.3378504

[6] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop apache project* 53, 1-13 (2008), 2.

[7] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. Distributed Systems (2Nd Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, Chapter The Primary-backup Approach, 199–216. http://dl.acm.org/citation.cfm?id=302430.302438

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST '16)*. USENIX Association, Santa Clara, CA, 209–223.

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

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 1007–1019. http://dl.acm.org/citation.cfm?id=3277355.3277451

[11] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide* (second ed.). O'Reilly Media. http://amazon.com/o/ASIN/1449344682/

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[14] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. 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. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[15] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[16] Facebook. 2018. BlobDB. http://rocksdb.org/. Accessed: October 9, 2021.

[17] Facebook. 2018. RocksDB. http://rocksdb.org/.

[18] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[19] Panagiotis Garefalakis, Panagiotis Papadopoulos, and Kostas Magoutis. 2014. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 211–220. https://doi.org/10.1109/SRDS.2014.43

[20] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[21] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 16–25.

[22] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*. 743–752.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. 437–450.

[25] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value

storage system for cloud data.. In *MSST*. IEEE Computer Society, 1–14. http://dblp.uni-trier.de/db/conf/mss/msst2015.html#LaiJYLSHCC15

[26] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[27] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association, 673–687.

[28] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. 2015. Kudu: Storage for fast analytics on fast data. *Cloudera, inc* 28 (2015).

[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[30] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3217–3230. https://doi.org/10.14778/3415478.3415546

[31] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. 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* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 103–114. http://dl.acm.org/citation.cfm?id=2535461.2535475

[32] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[33] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 537–550. https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

[34] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. ACM, New York, NY, USA, 490–502. https://doi.org/10.1145/3267809.3267824

[35] Jinglei Ren. 2016. YCSB-C. https://github.com/basicthinker/YCSB-C.

[36] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: Compressed, Log-Structured Replication. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 526–537. https://doi.org/10.14778/1453856.1453914

[37] Chen Shen, Youyou Lu, Fei Li, Weidong Liu, and Jiwu Shu. 2020. NovKV: Efficient Garbage Collection for Key-Value Separated LSM-Stores. (Oct. 2020), 8.

[38] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 93–105. https://doi.org/10.1145/3452296.3472934

[39] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 851–863. http://dl.acm.org/citation.cfm?id=3277355.3277438

[40] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 87–104.

[41] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Papagiannis Anastasios, and Angelos Bilas. 2021. Parallax: Hybrib Key-Value Placement in LSM-based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Hybrid Event) *(SoCC '21)*. ACM, New York, NY, USA.