Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores

Michalis Vardoulakis[*]
Institute of Computer Science, FORTH, Heraklion, Greece
mvard@ics.forth.gr

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

Pilar González-Férez
Dept. of Computer Engineering, Univ. of Murcia, Spain
pilargf@um.es

Angelos Bilas[*]
Institute of Computer Science, FORTH, Heraklion Greece
bilas@ics.forth.gr

Abstract

Key-value (KV) stores based on LSM tree have become a foundational layer in the storage stack of datacenters and cloud services. Current approaches for achieving reliability and availability favor reducing network traffic and send to replicas only new KV pairs. As a result, they perform costly compactions to reorganize data in both the primary and backup nodes, which increases device I/O traffic and CPU overhead, and eventually hurts overall system performance. In this paper we describe *Tebis*, an efficient LSM-based KV store that reduces I/O amplification and CPU overhead for maintaining the replica index. We use a primary-backup replication scheme that performs compactions only on the primary nodes and sends pre-built indexes to backup nodes, avoiding all compactions in backup nodes. Our approach includes an efficient mechanism to deal with pointer translation across nodes in the pre-built region index. Our results show that *Tebis* reduces pressure on backup nodes compared to performing full compactions: Throughput is increased by $1.1 - 1.48\times$, CPU efficiency is increased by $1.06 - 1.54\times$, and I/O amplification is reduced by $1.13 - 1.81\times$, without increasing server to server network traffic excessively (by up to $1.09 - 1.82\times$).

CCS Concepts: **• Information systems → Key-value stores**; **B-trees**; **Flash memory**; **• Networks → Network design principles**.

[*]Also with the Department of Computer Science, University of Crete, Greece.

EuroSys '22, April 5–8, 2022, RENNES, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9162-7/22/04...$15.00
https://doi.org/10.1145/3492321.3519572

Keywords: Key Value stores, LSM tree, B+ tree, Flash, RDMA

ACM Reference Format:
Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2022. Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3492321.3519572

1 Introduction

Key-value (KV) stores are the heart of modern datacenter storage stacks [2, 11, 13, 27, 30]. These systems typically use an LSM tree [32] index structure because it achieves: 1) fast data ingestion capability for small and variable size data items while maintaining good read and scan performance, and 2) low space overhead on the storage devices [14]. However, LSM-based KV stores suffer from high compaction costs for reorganizing the multi-level index [29, 34], including both I/O amplification and CPU overhead.

To provide reliability and availability, state-of-the-art KV stores [11, 27] replicate KV pairs in multiple, typically two or three [7], nodes. Current designs [11, 27, 37] perform costly compactions to reorganize data in the primary and backup nodes to ensure: (a) minimal network traffic by moving only user data across nodes and (b) sequential device accesses by performing only large I/Os. However, this approach comes at a significant increase in read I/O traffic, CPU utilization, and memory use at the backups. Given that all nodes function both as primaries and backups at the same time, eventually this approach hurts overall system performance.

In our work, we rely on two observations: 1) The increased use of RDMA in the datacenter [19, 39] which increases available throughput at low CPU utilization. This makes it viable to trade network traffic for CPU and device I/O. 2) The use of KV separation in state-of-the-art KV stores [4, 16, 28, 29, 34, 45] that reduces, depening on the KV pair sizes, the size of the index.

Instead of storing values and keys in the LSM tree, KV separation places values in a separate log and uses additional metadata to point in the value log [10, 16, 26, 29, 34]. This technique reduces I/O amplification by up to 10x [5] by introducing small and random read I/Os, which are not as harmful for fast storage devices. Additionally, recent work [28, 45] significantly improves garbage collection overhead for KV separation [10, 38].

In this work, we design and implement *Tebis*, an efficient replicated LSM-based KV store. The main novelty in *Tebis* is that it reduces compaction overhead at the *backups* by shipping a pre-built index from the *primary*. This approach reduces read I/O amplification, CPU overhead, and memory utilization in *backup* nodes.

The design of *Tebis* addresses a main and two secondary challenges. The main challenge is an efficient rewrite mechanism of the index at the *backup* nodes: The index received at the *backups* contains segment offsets of the device in the *primary*. *Tebis* creates mappings between aligned *primary* and *backup* segments. Then, it uses these mappings to rewrite device locations at the *backups* efficiently.

Tebis replicates the data value log, using an efficient RDMA-based primary backup communication protocol that does not require the involvement of the *backup* CPUs in communication operations [40]. In addition, to reduce CPU overhead for client server communication, *Tebis* uses one-sided RDMA write operations. The protocol of *Tebis* supports variable size messages that are essential for KV stores using a single round trip to reduce the processing overhead at the server.

We evaluate *Tebis* 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 [9] production workloads. Our results show that our index shipping method spends $1.06 - 1.54\times$ fewer CPU cycles per operation than a baseline implementation that performs compactions at the *backups*. Furthermore, it has $1.13 - 1.48\times$ higher throughput, and reduces I/O amplification by $1.13 - 1.81\times$. Overall, our technique of sending and rewriting a pre-built index trades CPU, memory, and read I/O amplification for increased network traffic.

2 Background

In this section we briefly discuss LSM tree index structure and KV separation, the Kreon [18, 34] key-value store, and the basic RDMA operations.

LSM tree: LSM tree [32] is a write-optimized data structure that organizes its data in multiple levels whose sizes grow by a constant growth factor f. The first level size is in the order of hundreds of MB and stored fully in memory, whereas the rest of the levels are stored in the device. Although the in-memory level is named memtable in some systems [17], for simplicity, we use the LSM terminology and refer to it as L_0. There are different ways to organize data in levels [23, 32]. In this work, we focus on leveled KV stores that organize each level in non-overlapping ranges, which is also the most broadly used approach.

In LSM tree [32], a grotwh factor $f = 4$ results in the minimum I/O amplification [5]. However, KV stores in production use larger growth factors, typically 8-12 [14], which increase overall I/O amplification but reduce the number of levels. Fewer levels result in less space usage on the device for high update ratios, assuming intermediate levels contain only update and delete operations.

Current KV store designs [10, 16, 26, 29, 34] use the ability of fast storage devices to operate at a high percentage (close to 80% [34]) of their maximum read throughput under small and random I/Os to reduce I/O amplification. The main techniques are KV separation [4, 10, 16, 26, 29, 34] and hybrid KV placement [28, 45]. KV separation appends keys and values in a separate value log, instead of storing values with the keys in the index. The index keeps metadata to the corresponding value in the log. As a result, they only re-organize keys and value pointers in the multi-level structure. This approach, depending on the KV-pair sizes, reduces I/O amplification by up to 10x [5]. Hybrid KV placement [28, 45] is a technique that extends KV separation and reduces garbage collection overhead, especially for medium ($\geq$ 100 B) and large ($\geq$ 1000 B) KV pairs [9]. Hybrid KV placement also places large KV pairs in a separte log, small KV pairs in place within each LSM tree level, and medium KV pairs in seperate value logs until the last, one or two, level(s) where it reclaims the medium value log.

Kreon: *Tebis* uses Kreon [18, 34] to manage data within each server. Kreon is a persistent LSM-based KV store designed for fast storage devices (NVMe SSDs). 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. A multilevel LSM structure is used to organize its index. The first level L_0 resides in memory, whereas the rest of the levels are on the device. Kreon organizes each level internally as a B+ tree. Leaves contain <key_prefix, value_location> pairs. All level indexes and the value log are represented as a list of fixed-size segments on the device (currently 2 MB).

Kreon uses two different I/O paths: (a) It uses *memory-mapped I/O* to manage its I/O cache and access the storage devices during read and scan operations. (b) It uses direct I/O to read and write the levels during compactions and avoid polluting the I/O cache. We modify Kreon to use explicit I/O system calls, that bypass the buffer cache, for writing its KV log to further reduce CPU overhead for consecutive write page faults.

Remote Direct Memory Access: RDMA supports *two-sided* send/receive operations and *one-sided* read/write operations [3]. In send and receive, both the sender and the receiver actively participate in the communication, consuming CPU cycles. Read and write operations allow one peer to directly read or write the memory of a remote peer without involving the remote CPU, consuming CPU cycles only in

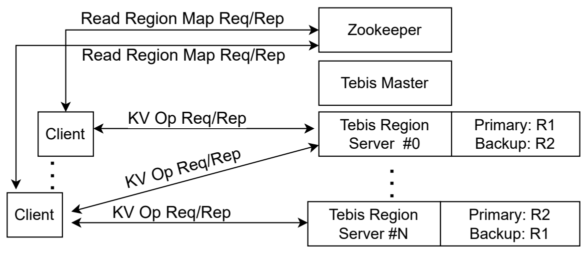


Figure 1. *Tebis* overview.

the originating node. In *Tebis*, we use one-sided RDMA write operations.

3 Design

3.1 Overview

Tebis partitions the key-value space into non-overlapping key ranges, named *regions*. *Tebis* assigns each region to multiple servers with either the *primary* or *backup* role. Each region stores and organizes data in an LSM tree with KV separation. *Tebis* consists of three main entities (Figure 1):

1. *Tebis Region servers*, which host the regions with either a *primary* or *backup* role.
2. The *master* which orchestrates the recovery process in case of failures and performs load balancing operations. The *master* reads the region map during initialization and issues *open region* commands to each *region server* in the *Tebis* cluster, assigning a *primary* or a *backup* role.
3. Zookeeper [22] stores information about the metadata of each region. Zookeeper is not in the common path of client operations in *Tebis* since changes in regions are triggered either by membership changes due to failures or load balancing operations. Furthermore, the *master* of *Tebis* uses the membership service of Zookeeper to detect changes in server status (join or fail) and trigger appropriate action. *Tebis* can make use of similar systems [1, 35, 43] that provide strongly consistent metadata replication and notifications services.

During initialization, clients read and cache the region map, without consuming significant memory and CPU overhead. The region map size is small and in the order of hundreds of KB. Changes to the region map incur only after a failure or load balancing operation. Prior to 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. When a client issues a KV operation to a *region server* that is not currently responsible for the corresponding range due to a system reconfiguration, the *region server* instructs the client to update its region map.

Tebis implements a primary-backup protocol over RDMA [8, 40]. Next, we discuss how *Tebis* replicates its log (Section 3.2) and index (Section 3.3).

3.2 Primary-Backup Value Log Replication

Tebis replicates its log without involving the CPU of *backups*. When it receives updates and inserts from clients, the *primary* replicates each operation to its set of *backup* servers in three steps (Figure 2). It inserts the KV pair in Kreon which returns the offset of the KV pair in the value log tail segment. Then, it appends (via an RDMA write operation) the KV pair to the RDMA buffer of each *backup* at the corresponding offset (step 1 in Figure 2).*Tebis* waits for an acknowledgment that all RDMA write operations have been replicated in the memory of *backup* nodes. To detect that the RDMA write operations are complete and the data are written in the remote memory of the *backups*, *Tebis* uses the work completion events of reliable queue pairs [3]. The CPU of the *backup* server is not involved in any of these steps. When a client receives an acknowledgment it means that its operation has been replicated in the replica set.

On the other hand, persisting the tail segment involves the CPU of both the *primary* and *backups*. When the tail segment of the log in the *primary* becomes full, the *primary* flushes the segment to persistent storage (step 2a in Figure 2) and sends a *flush tail* message to each *backup* to persist their RDMA buffer (step 2b in Figure 2). Upon receiving a flush request, *Backups* write the corresponding RDMA buffer to persistent storage (step 2c in Figure 2). Finally, they send an acknowledgment to the *primary* (step 3 in Figure 2).

Each *backup* region maintains a log map with entries <primary segment number, backup segment number>, specifying the location of each segment on the storage device in the *primary* and the *backup*. *Backups* use these to rewrite the primary pointers in the Send-Index method, as described in Section 3.3. The log map has a small memory footprint in the order of MB. Each entry in the map is 16 B and a value log of 1 TB with a segment size of 2 MB requires a log map of 8 MB.

For this purpose, the *primary* piggybacks flush message with the tail segment number in its storage device. *Backup* servers, after persisting their value log tail segment, use this information to create the corresponding entry in their log map (step 2d in Figure 2). Note that the log map in the *backups* is valid until a *primary* changes due to a failure or load balance operation. In these cases, *Tebis* promotes a *backup* as the new *primary* and the rest of the *backups* need to update their log map. This procedure is also an in-memory operation without requiring I/O. The new *primary* sends its log map to the rest of the *backups*. The *backups* iterate over the map and replace the segment numbers of the previous with the segment numbers of the new *primary*.

3.3 Index Shipping and Rewrite at the *Backup*

Tebis avoids the full compaction process at the *backup* regions to save device read I/O throughput, CPU, and memory. Instead, after each compaction the *primary* ships a new index

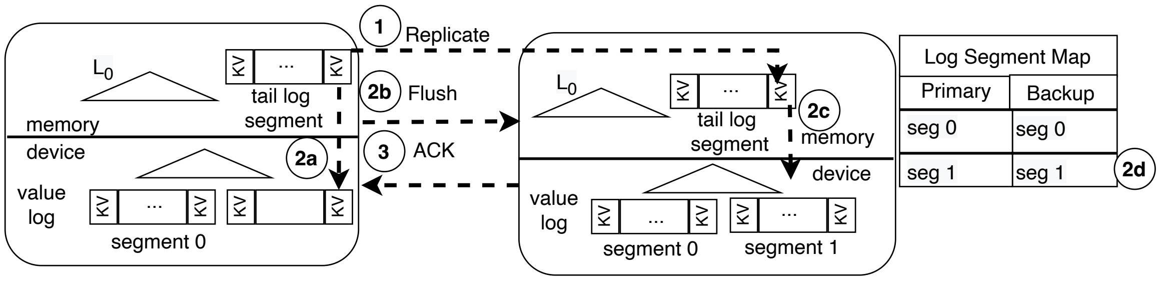


Figure 2. Value log replication in *Tebis*.

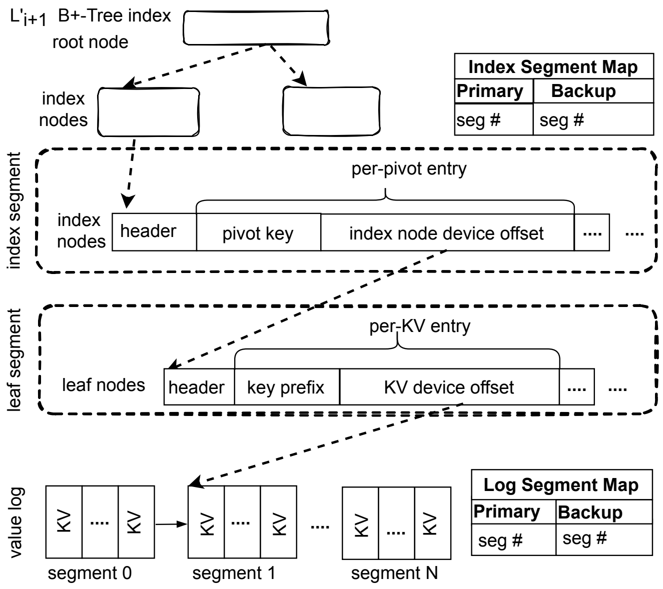


Figure 3. *Tebis* B+ tree index and value log organization on the storage devices.

to the *backups*. The main challenge in *Tebis* is to rewrite the index at the *backups* to contain valid device addresses since servers do not share a global storage name space [2, 6, 21].

In the Send-Index method, when L_i becomes full, the *primary* region executes the heavy, in terms of CPU and device I/O, compaction process of L_i and L_{i+1}. Then, the *primary* sends the resulting index L'_{i+1} to the *backup* regions. This method reduces in each *backup* (a) device read I/O traffic from reading L_i and L_{i+1}, (b) CPU since it avoids in-memory sorting, and (c) memory for L_0. *Backup* regions do not need to keep an in-memory L_0. L_0 is used to amortize I/O cost during compaction with L_1 by keeping KV pairs sorted in-memory. For L_0 to L_1 compactions, *backup* regions do not need to read L_0 and L_1. Instead, they receive and rewrite the *primary* L'_1 index. Omitting L_0 in *backup* regions results in reducing the memory budget for L_0 by 2× for one replica per region or by 3× for two replicas.

A consequence of the Send-Index method is that it increases network traffic. Essentially, Send-Index sends over the network the reorganized indexes. This increased traffic uses network throughput instead of device read I/O. In addition, the CPU required for RDMA communication is reduced compared to the CPU required for merge-sort and read I/O.

In *Tebis*, the main device structures are the value log and the B+ tree indexes of the levels. *Tebis* stores both the value log and the B+ tree indexes as a list of fixed segments. Similar to log segments, each segment is 2 MB and its starting device offset is segment aligned. During rewriting, *Tebis* replaces the high order bits of the *primary* segment with the new segment number in the *backup* device.

The index of a region (Figure 3) consists of leaf and index nodes. For each KV pair, leaf nodes (bottom in Figure 3) contain a key prefix, which reduces I/O operations to the value log [34], and a device offset which points to the device location of the KV pair in the value log. Index nodes store variable size pivot keys and pointers to device locations of their successor, index or leaf, nodes. *Backups* need to rewrite the device offset of KV pairs in leaf nodes and index nodes (dashed arrows in Figure 3).

Backups keep track of two mappings for segments: the log map and the index map. The log map is updated during flush operation of the log (Section 3.2). The index map is updated dynamically during the Send-Index method and it is valid only during compaction from L_i to L_{i+1}. During compaction, the *primary* builds its index bottom-up and left to right. As a result, the *primary* can send the new index incrementally as it is being build, segment by segment.

After producing an index segment for L'_{i+1}, the *primary* sends it to its *backups*. The *backup* region allocates a new local segment and adds a new entry to its index map. Then, it parses and rewrites the index segment by modifying device offsets for all pivot (index nodes) and KV pairs (leaf nodes). For each source device offset it replaces the high order bits with the local segment from the segment map.

Finally, on compaction completion, the *primary* sends the offset of the root node in L'_{i+1}, which is the entry point of the index, to each *backup*. Then, each *backup* translates to the root offset of its storage space using its index map.

It is important to note that the index shipping and rewriting technique can be applied to KV stores that perform

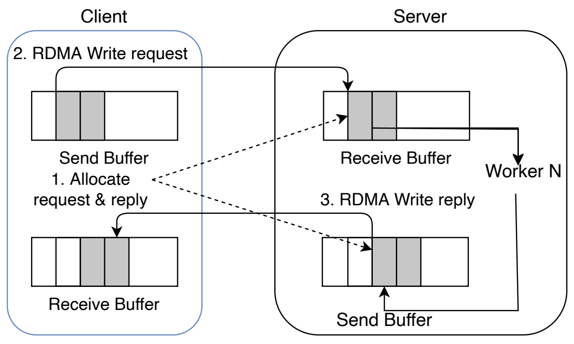


Figure 4. Allocation and request-reply flow of *Tebis* RDMA Write-based communication protocol.

full compactions such as RocksDB [17] or use KV separation [16, 28, 29, 45]. In these systems SSTs may contain device offsets of the *primary* to its value log or an internal SST index which need rewriting similar to *Tebis*.

3.4 RDMA Write-based Communication Protocol

The main design points that *Tebis* addresses are the management of RDMA buffers without synchronization at the server and support for variable size messages.

3.4.1 RDMA Buffer Management. *Tebis* performs client-server communication via one-sided RDMA write operations [25] to avoid network interrupts and reduce the CPU overhead in the server [24, 25]. After connection establishment, the server and the client allocate a pair of buffers with configurable size (currently 256 KB). The *region server* frees these buffers when a client disconnects or fails. A thread monitors inactive queue pairs and checks if the queue pair is still in valid state. Currently, this thread spins on RDMA buffer but could also use a sleep-wakeup approach.

Clients manage both request and reply buffers to avoid synchronization among workers in the server. Clients allocate a pair of messages for each KV operation; one for their request and one for the server reply (step 1 in Figure 4). Each request header includes the buffer offset where the *region server* can write its reply. Workers complete requests asynchronously and respond to the client out of order.

For put requests, the reply is of fixed size, so the client allocates the exact amount of memory needed prior to the operation. On the other hand, for get and scan requests, the reply size is variable and unknown a priori to the client. If the value size is larger than the buffer size of the reply, the *region server* sends part of the reply and informs the client to increase its allocation size for reply buffers to avoid similar cases in subsequent requests. Then, it retrieves the rest of the value from an offset provided by the server. As a result, the penalty, in this case, is a round trip with a small impact on overall latency.

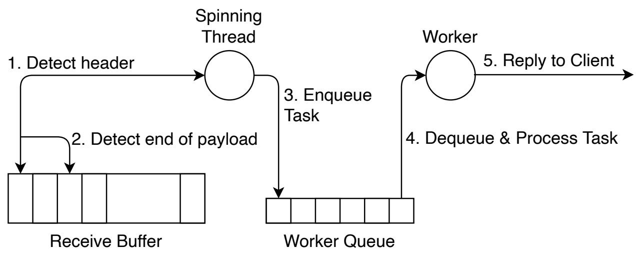


Figure 5. Message detection and task processing pipeline in *Tebis*. For simplicity, we only draw one circular buffer and a single worker.

Scaling the RDMA protocol of *Tebis* to large numbers of clients requires using more memory for RDMA buffers and polling for new messages in more rendezvous points. To limit the required memory for RDMA buffers, *Tebis* could divide this memory elastically between more and less active clients. Also, other approaches such as LITE [41] could be appropriate for persistent LSM KV stores since the 90-percentile tail latency of LSM KV stores is in the order of hundreds of μs. Polling a large number of rendezvous points can be mitigated by adjusting the number spinning threads in *Tebis* and distinguishing hot from cold clients to reduce the polling frequency. We leave these as extensions for future work.

3.4.2 Variable Size Messages and Task Scheduling. The main design challenge with variable size messages is how to detect their arrival at the *region server* without using network interrupts.

All messages in *Tebis* consist of a *message header* of size 128 B and a variable size payload. To support variable size payloads, *Tebis* pads the payload to a multiple of the message header size. To detect incoming messages each *region server* uses a *spinning thread*, as shown in Figure 5. The spinning thread polls a fixed memory location in each RDMA buffer it shares with a client. The spinning thread detects a new message by checking for a rendezvous magic number at the last four bytes of the current message header. Then, it reads the payload size from the message header to determine the end of the variable size message and the next header. A second *rendezvous point* is used at the end of the payload to check that the whole message has arrived. Upon receiving a message, the spinning thread creates a new client request for one of its workers, zeroes of the message in the RDMA buffer, and advances its rendezvous point to the next message header. The fact that all messages are multiple of message header size has the benefit that the spinning thread does not have to zero the whole message memory area. Instead, it only zeroes the possible locations of message header size in the area where future message headers may arrive.

When clients reach the end of the RDMA buffer, the client informs the server spinning thread to reset the rendezvous points at the start of the buffer. There are two possible cases:

(a) When the last message received reaches the end of the buffer, the spinning thread sets automatically the rendezvous point without any communication with the client. (b) When the remaining space in the circular buffer is not enough for the current message, the client sends a NOOP request message to the server with a size equal to the remaining space in the buffer. The spinning thread detects it and assigns it to a worker. The worker then sends a NOOP reply. When the clients detect the NOOP reply, it proceeds as in case a.

Tebis uses a configurable number of workers. Each worker has a private *task queue* where the spinning thread places new tasks (Figure 5). Workers poll their queue to retrieve a new request and sleep if the spinning thread does not assign a new task within a period (currently 100 μs). To limit the number of wake-up operations, the spinning thread assigns a new task to the same worker as long as its task queue has fewer pending tasks than a threshold (currently set to 64). Then, the spinning thread selects the next running worker with fewer than threshold tasks. If none exists, it proceeds to wake-up a sleeping worker.

3.5 Failure Detection and Recovery

Tebis uses the ephemeral nodes mechanism of Zookeeper to detect failures. Zookeeper automatically deletes an ephemeral node when the node stops responding to heartbeats. Every *region server* creates and registers an ephemeral node with its hostname during initialization.

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

In case of a *backup* failure, the *master* replaces the crashed *region server* with a new node that is not already part of the region. In this case, the new node has *backup* role and the *master* instructs the rest of the *region servers* in the group to transfer their region data to the new *backup*. The region experiencing the *backup* failure will remain available throughout the whole process since its *primary* is unaffected.

In case of a *primary* failure, the *master* first promotes one of the existing *backups* in the 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 to construct L_0 in its memory before starting to serve client requests. The L_0 size that *Tebis* has to replay is in the order of tens or hundreds of MB. When the new *primary region server* is ready, the *master* handles this failure as a *backup* failure. During *primary* reconstruction, *Tebis* cannot serve client requests for the affected region.

When the *master* fails, Zookeeper notifies the rest of the *region servers* through the ephemeral node mechanism. Then, the *region servers* use Zookeeper to elect a new *master*. During downtime, *Tebis* can serve requests from existing primaries but will not handle any additional failure. If a primary or backup region fails, the respective region become unavailable until a new *master* is elected and it handle the primary or backup failure as before.

4 Evaluation Methodology

Our experimental setup consists of three identical servers equipped with two Intel(R) Xeon(R) CPU E5-2630 running at 2.4 GHz, with 16 physical cores for a total of 32 hyper-threads and with 256 GB of DDR4 DRAM. All servers run CentOS 7.3 with Linux kernel 4.4.159. Each server has a 1.5 TB Samsung PM173X NVMe SSD and a 56 Gbps Mellanox ConnectX 3 Pro RDMA network card. To ensure our experiments exhibit significant I/O activity, we use *cgroups* to limit the buffer cache used by *memory-mapped I/O* to 25% of the dataset size in all cases as shown in Table 2.

In our experiments, we run the YCSB benchmark [12] and its workloads Load A and Run A to Run D. Table 1 summarizes the operations run during each workload. We run *Tebis* with 32 regions equally distributed across all servers. Furthermore, each server has two spinning threads and eight worker threads in all experiments. Servers use the remaining cores to perform compactions.

In all experiments, we use two separate servers to run the clients. In each server, we run four client processes with four threads per process. To generate enough outstanding requests for each server, each client process uses four queue pairs per server which are shared among each client's threads. Clients send requests asynchronously to all 32 regions as long as there is space in the RDMA buffers of the channel to each server, therefore, the outstanding number of requests is limited by RDMA buffer size. Each client generates the same number of operations. The total number of operations is 100 million requests for Load A and 50 million operations for each of the Run A – Run D phases in YCSB.

Similar to other KV stores that use KV separation [10, 29] Kreon uses garbage collection to reclaim free space from the value log. It moves valid values from the head of the log to the tail and trims space. In *Tebis*, the primary informs *backups* for this operation and they only perform the trim. In all our experiments the log is not garbage collected in the primary (both Build-Index and Send-Index) because we wanted to focus on compaction.

In our evaluation, we also vary the KV pair sizes according to the KV sizes proposed by Facebook [9], as shown in Table 2. We first evaluate the following workloads where all KV pairs have the same size: either Small (S), Medium (M), or Large (L). For this purpose, we use a C++ version of YCSB [36] and we modify it to produce different values according to the KV pair size distribution we study.

	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. Operation mix for YCSB. Workloads use Zipfian distribution and Run D uses the latest distribution.

	KV Size Mix S%-M%-L%	#KV Pairs	Dataset Size (GB)	Cache per Server (GB)
S	100-0-0	100M	3.0	0.38
M	0-100-0	100M	11.4	1.4
L	0-0-100	100M	95.2	11.9
SD	60-20-20	100M	23.2	2.8
MD	20-60-20	100M	26.5	3.3
LD	20-20-60	100M	60.0	7.5

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, dataset size, and cache size per server used with each KV size distribution.

In addition, we evaluate workloads that use mixes of small, medium, and large KV pairs. We use a small-dominated (SD) KV size distribution as proposed by Facebook [9], as well as a medium dominated (*MD*) and a large dominated (*LD*) workload. We summarize these KV distributions in Table 2.

We examine the throughput (ops/s), efficiency (cycles/op), I/O amplification, and network amplification of *Tebis* for the following three 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 all three configurations we use an L_0 size that stores 96K KV pairs. We note that Build-Index uses one L_0 for each replica, whereas Send-Index uses a single L_0 for the primary replica only. Thus, Send-Index is more memory-efficient than Build-Index

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

$$efficiency = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}} \; cycles/op, \tag{1}$$

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. Finally, $average\_ops/s$ 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)

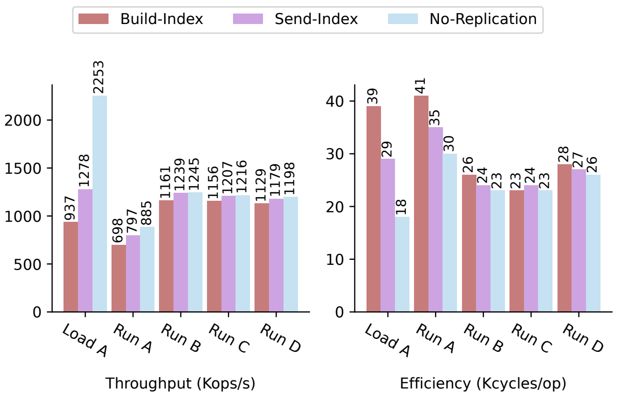


Figure 6. Performance and efficiency of *Tebis* for workloads Load A, Run A – Run D with the SD KV size distribution.

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.

We measure network amplification as traffic to all servers over application data written and read by the clients.

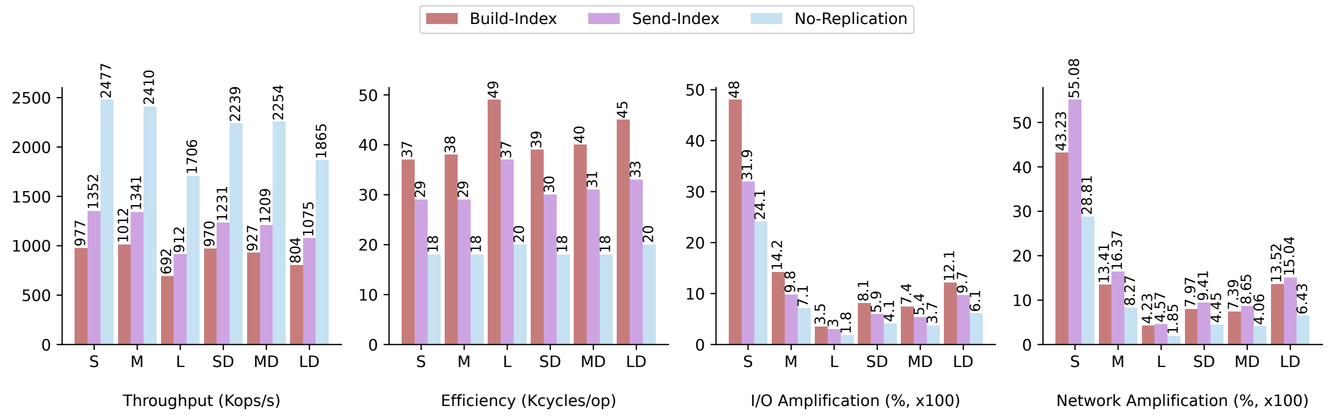
$$network\_amplification = \frac{network\_traffic}{dataset\_size},$$

where *network_traffic* is the total number of bytes sent and received by the server(s). Note that application data do not include network overhead (headers, acknowledgements), therefore, network traffic is always higher than application data. In addition, our RDMA client-server protocol uses a minimum payload of 256 B to reduce CPU use for detecting variable size messages in the servers, since for small messages the bottleneck is the packet rate in the NICs. This minimum payload is reflected in client-server network traffic for all experiments, including the No-Replication configuration.

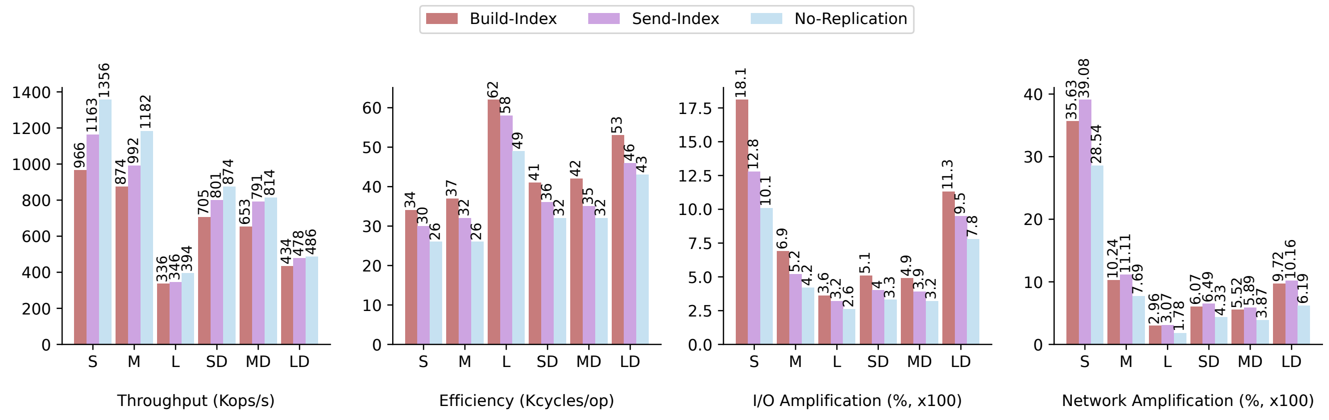
5 Experimental Evaluation

Our goal is to answer the following questions:

1. How does our *backup* index shipping method (Send-Index) 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 maintenance?
3. How does Send-Index improve performance and efficiency in small-dominated workloads?
4. What are the gains in throughput, efficiency, and I/O amplification for three-way replication?



(a) Load A



(b) Run A

Figure 7. 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.

5. Does using a smaller L_0 in Build-Index, to counterbalance the L_0 memory budget compared to Send-Index, has an impact on performance, efficiency, and I/O amplification?

5.1 *Tebis* Performance and Efficiency

In Figure 6, we evaluate *Tebis* for two-way replication using YCSB workloads Load A and Run A to Run D for the SD KV distribution [9]. Since replication does not have impact on read-dominated workloads, the performance in workloads Run B to Run D is similar for all three configurations. We focus the rest of our evaluation on the insert and update heavy workloads Load A and Run A, respectively.

We run Load A and Run A for all six KV distributions with a growth factor of 4, which minimizes I/O amplification [5]. Figure 7 shows that compared to Build-Index and for all KV size distributions, Send-Index increases throughput by $1.1 - 1.41\times$, CPU efficiency by $1.06 - 1.36\times$, and reduces I/O amplification by $1.13 - 1.45\times$. This happens because Send-Index 1) eliminates reads for L_i and L_{i+1} levels and 2) replaces in-memory sorting with index rewriting in *backup* regions. However, this trade-off favors *Tebis* since it uses available network throughput to reduce device I/O traffic and CPU usage.

We also measure the tail latency for YCSB workloads Load A and Run A using SD (Figure 8). Compared to Build-Index, Send-Index improves the 50, 70, 90, 99, 99.9, and 99.99% tail latency between $1.05 - 1.77\times$ for insert, $1.1 - 1.9\times$ for read, and $1.02 - 1.65\times$ for update. The reduction we observe in tail latency is due to the more efficient compactions in the *backup* regions. Send-Index releases device, CPU, and memory resources which the system uses for its *primary* regions and reduces stalls in L_0 where requests wait for compaction to finish.

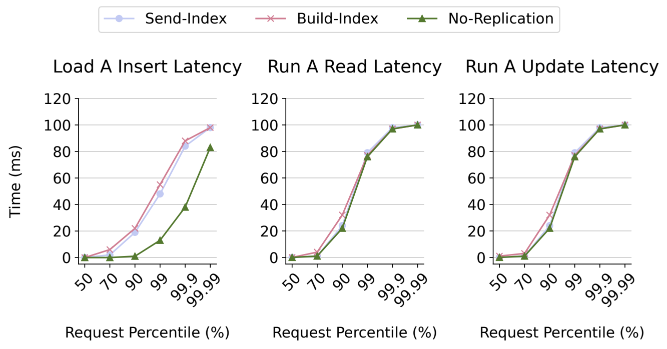


Figure 8. Tail latency for Load A and Run A using SD.

	cycles/op in Load A, SD, 100 M KV pairs		
	Build-index	Send-index	Reduction
Insert in L_0	5031	2730	45.7%
KV log replication	1170	1140	2.5%
Compaction	9640	2505	41.6%
Send index	0	1770	
Rewrite index	0	1350	
Server to client reply	740	740	0%
Other	22419	19765	11.1%
Total	39000	30000	23.1 %

Table 3. Breakdown of the cycles spent by all server threads in each component of *Tebis* for Send-Index and Build-Index.

5.2 Overhead Breakdown

To identify the percentage of time spent by each server thread in different parts of *Tebis*, we profile the system with *perf* and call graph tracking enabled for Load A. We use the call graph profiles generated for Send-Index and Build-Index to produce the corresponding flamegraphs and extract the percentage of time spent in each system component. We convert this utilization to CPU cycles using Equation 1. We break execution time into the following components:

Insert in L_0: Cost for inserting KV pairs in the L_0 B+ tree index of Kreon, when there is space in L_0. This stage also includes write I/O operations to persist the value log on the device.

KV log replication: Cost for replicating KV pairs in the value log of *backups*.

Compaction: Cost for compacting all *primary* and *backup* regions. This stage includes read and write I/O operations as well as in-memory sorting.

Send index: Cost of sending the index from the *primary* region to all *backups*. Note that this cost is zero in the case of Build-Index.

Rewrite index: Cost to traverse and rewrite the index in all *backups*. Note that this cost is zero in the case of Build-Index.

Other: The rest of the cost in *Tebis* such as polling and scheduling for incoming requests.

Table 3 summarizes our results. Compared to Build-Index, Send-Index requires 23.1% fewer CPU cycles per operation, overall. To replicate the L_0 B+ tree index, Send-Index requires 45.7% fewer CPU cycles. Also, Send-Index does not build in memory an L_0 index for its *backup* regions, which results in 2× fewer L_0 indexes than Build-Index. Compactions in Send-Index include also the send index to replicas and rewrite index stages. Overall, the total cost of compaction in Send-Index requires 41.6% less CPU cycles. This reduction is due to 1) less CPU for read I/O from the device and 2) replacing in-memory sorting with index rewrite.

5.3 Impact on Small KV Pairs

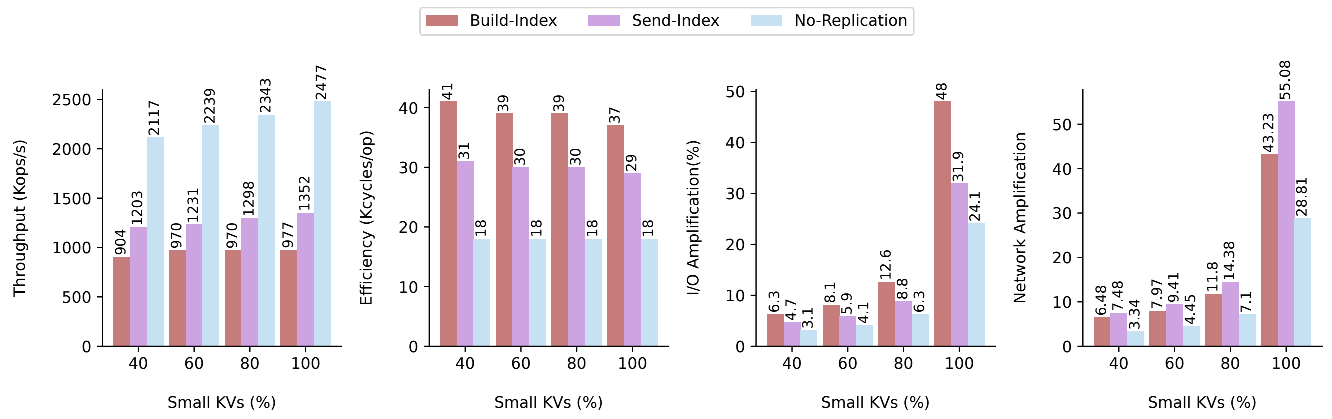
In this experiment, we investigate the improvement of Send-Index specifically for small KV pairs. We use a growth factor of 4 and we keep one replica per region (two-way replication). We examine four workloads where we vary the percentage of small KV pairs to 40%, 60%, 80%, and 100%. We equally divide the remaining percentage between medium and large KV pairs in all four cases.

Figure 9 shows that compared to Build-Index, Send-Index increases throughput by $1.13 - 1.38\times$, increases CPU efficiency by $1.13 - 1.39\times$, and reduces I/O amplification by $1.23 - 1.5\times$ across Load A and Run A. We see that the Send-Index benefits increase as small KV pairs increase. KV separation technique benefits in I/O amplification decrease as the percentage of small KV pairs increases since metadata in the LSM index are comparable in size with the KV pairs in the value log. As a result, I/O amplification from compaction increases which consumes device throughput. The Send-Index method benefits overall system performance because it offloads read I/O traffic from the device to the network.

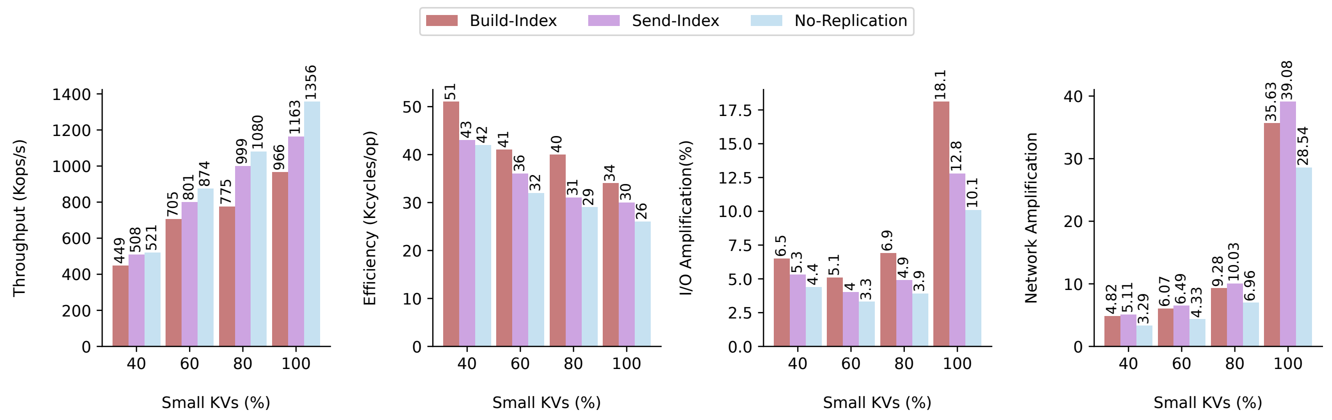
5.4 Three-way Replication

We run Load A and Run A for all six KV distributions with a growth factor of 4. In this experiment, we keep two replicas per region, in addition to the *primary* copy. We set the L_0 size to 96K keys for the No-Replication, Build-Index, and Send-Index configurations.

Figure 10 shows that for Load A, compared to Build-Index, Send-Index improves throughput by $1.1 - 1.48\times$, increases CPU efficiency by $1.16 - 1.54\times$, and decreases I/O amplification by $1.23 - 1.81\times$. Compared to two-way replication we see that the gains increase for throughput from $1.1 - 1.38\times$ to $1.1 - 1.48\times$, for efficiency from $1.06 - 1.36\times$ to $1.16 - 1.54\times$, and for I/O amplification from $1.13 - 1.45\times$ to $1.09 - 1.82\times$. Compared to two-way replication, in three-way replication we observe this relative increase in throughput, efficiency, and I/O amplification because we have more compactions that compete for device I/O throughput.



(a) Load A



(b) Run A

Figure 9. Throughput, efficiency, I/O amplification, and network amplification for increasing percentages of small KVs for (a) YCSB Load A and (b) Run A.

5.5 L_0 Memory Usage

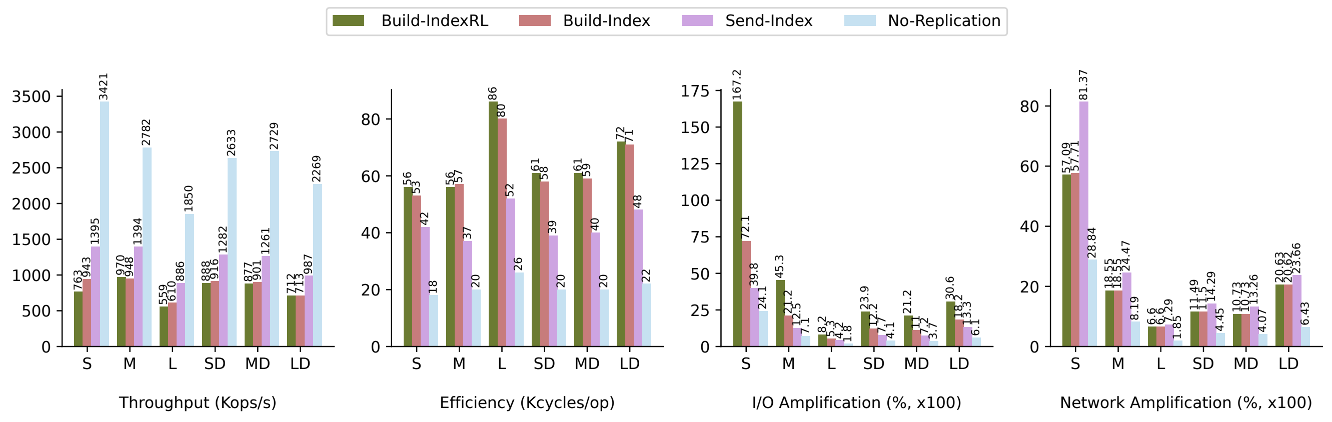
It is important to note that compared to Send-Index, Build-Index uses 2× more memory for L_0 when keeping two replicas and 3× more memory for three replicas. A server may host hundreds of regions, especially with increasing device capacities, for concurrency and load balancing purposes. As a result, the additional memory budget for Build-Index is in the order of tens of GB, e.g. assuming an L_0 size of 64 MB. In the Send-Index configuration the excess DRAM may be used for other purposes, such as RDMA communication buffers or a larger I/O cache. To show the impact of higher memory use, we use the configuration Build-Index Reduced L_0 (Build-IndexRL) which uses the same total memory budget for L_0 as Send-Index, by setting L_0 to 32K KV pairs for all *primary* and *backup* regions.

Compared to Build-IndexRL, Send-Index improves throughput by 1.2−1.32×, increases CPU efficiency by 1.17−1.53×, and decreases I/O amplification by 1.95 − 5.48×. Compared to Build-Index, we observe that the 3× smaller L_0 size of Build-IndexRL increases I/O amplification proportional to the number of small KV pairs which results in drop of throughput and efficiency.

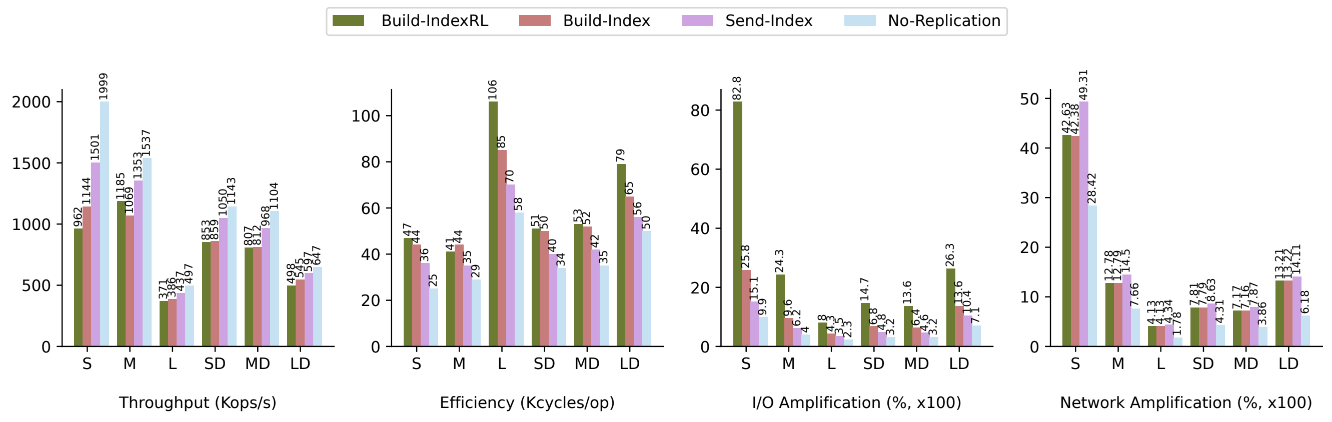
6 Related Work

We group related work in the following categories: (a) distributed persistent KV stores and (b) efficient RDMA protocols for KV stores.

Distributed persistent LSM KV stores: Acazoo [20] splits its dataset into shards and keeps replicas for each shard. To prevent write stalls due to compactions of the large LSM levels, Acazoo applies the following technique. When a primary has to execute a heavy compaction task, it changes one of the backup servers as primary. Then, the previous



(a) Load A



(b) Run A

Figure 10. Throughput, efficiency, I/O amplification, and network amplification for three-way replication with different KV size distributions for (a) Load A and (b) Run A.

primary performs the compaction task while the new primary serves new requests. Then, on compaction completion, it reconfigures the system to set the server with the newly compacted data as primary. DEPART [46] proposes a two-level log approach in which each *backup* first appends all of its KV pairs in a log. Then, periodically it groups KV pairs in a per primary server log. It does this in order to insert only the KV pairs of the *primary* in its LSM index during a failure and reduce recovery time. Rose [37] is a distributed key-value store, which replicates data using a log and builds the replica index by applying write operations in an LSM tree index. Furthermore, it uses compression to reduce I/O amplification and increase replication throughput. Contrary to these systems, *Tebis* performs the full compaction only at the *primary* and ships the index to the *backups*.

RAMCloud [33] is a scale-out, distributed in-memory KV store. It supports large-scale datasets by combining the main memories of thousands of servers. RAMCloud provides durability and availability using a primary-backup approach for data replication. A single (primary) copy of each object is kept in DRAM, and multiple backup copies are kept on persistent storage. RAMCloud achieves high availability by recovering data quickly in parallel from hundreds of devices after a crash.

Efficient RDMA protocols for KV stores: Tailwind [40] improves the performance of the replication protocol of RAMCloud by using RDMA writes for data replication. 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. *Tebis* adopts Tailwind's replication protocol for its value log but further proposes index shipping to keep a full index at the *backups* efficiently.

Kalia *et al.* [25] 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 [24], 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.

Other in-memory KV stores [15, 31, 44] 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 KV store that targets fast storage devices and fast RDMA-based networks. *Tebis* proposes a Send-Index method to keep efficiently an up-to-date index at the *backups*. Instead of performing compactions at the *backup* servers, the *primary* in *Tebis* sends its pre-built index of L'_{i+1} after each level compaction of L_i with L_{i+1} to all *backups*. As a result, *backup* regions incur less I/O amplification since they do not read L_i and L_{i+1}. In addition they incur less CPU overhead because they replace in-memory sorting with a lightweight index rewrite operation.

We find that in all setups where Send-Index has the same L_0 size with Build-Index, Send-Index increases throughput by $1.13 - 1.48\times$, CPU efficiency by up to $1.06 - 1.54\times$, decreases I/O amplification by $1.13 - 1.81\times$, and decreases tail latency by up to $1.5\times$ for Load A and Run A. On the other hand, it increases server to server network traffic $1.09 - 1.82\times$. Overall, we show that Send-Index benefits are analogous to the percentage of small KVs.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the project EVOLVE (Grant Agreement ID: 825061) and HiPEAC (Grant No. 871174). Pilar González-Férez was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds (grant "RTI2018-098156-B-C53"). We are thankful to Giorgos Xanthakis for his helpful comments. Finally, we thank the anonymous reviewers for their insightful comments and our shepherd John Ousterhout for his help with preparing the final version of the paper.

A Artifact Appendix

We have released *Tebis* under Apache2.0 License in github (https://github.com/CARV-ICS-FORTH/tebis), where we continue the development of our prototype. Furthermore, we have archived through Zenodo the version of *Tebis* used in this work [42]. The repository includes instructions on how to configure and run *Tebis*.

References

[1] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. *Microsecond Consensus for Microsecond Applications.* USENIX Association, USA.

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

[3] INFINIBAND TRADE ASSOCIATION. 2015. IB Specification Vol 1, 03,2015. Release-1.3. (2015).

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

[5] 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]

[6] 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

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

[8] 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

[9] 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 '20).* USENIX Association, Santa Clara, CA, 209–223.

[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: March 15, 2022.

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

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

[19] 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

[20] 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

[21] Haoyu Huang and Shahram Ghandeharizadeh. 2021. *Nova-LSM: A Distributed, Component-Based LSM-Tree Key-Value Store*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3448016.3457297

[22] 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

[23] 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.

[24] 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

[25] 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.

[26] 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

[27] 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

[28] 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.

[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] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (aug 2015), 55 pages. https://doi.org/10.1145/2806887

[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] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/2749246.2749267

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

[37] 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

[38] 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 pages.

[39] 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

[40] 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

[41] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 306–324. https://doi.org/10.1145/3132747.3132762

[42] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2022. *Tebis software*. https://doi.org/10.5281/zenodo.6349594

[43] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3127479.3128609

[44] 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.

[45] 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.

[46] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, and Si Wu. 2022. DEPART: Replica Decoupling for Distributed Key-Value Storage. In *20th USENIX Conference on File and Storage Technologies (FAST'22)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/fast22/presentation/zhang-qiang