Index Shipping for Efficient Replication in LSM Key-Value Stores with Hybrid KV Placement

GIORGOS STILIANAKIS[*†], FORTH, Heraklion, Greece and University of Crete, Heraklion, Greece
GIORGOS SALOUSTROS[*], FORTH, Heraklion, Greece
ORESTIS CHIOTAKIS[*†], FORTH, Heraklion, Greece and University of Crete, Heraklion, Greece
GIORGOS XANTHAKIS[*†], FORTH, Heraklion, Greece and University of Crete, Heraklion, Greece
ANGELOS BILAS[*†], FORTH, Heraklion, Greece and University of Crete, Heraklion, Greece

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 resource utilization on backup nodes compared to performing full compactions: Throughput is increased by 1.06 − 2.90×, CPU efficiency is increased by 1.21 − 2.78×, and I/O amplification is reduced by 1.7 − 3.27×, while network traffic increases by up to 1.32 − 3.76×.

CCS Concepts: • **Information systems** → **Key-value stores**; **B-tree indexes**; **Flash storage**; • **High-speed Network** → **RDMA**.

Additional Key Words and Phrases: Key Value stores, LSM tree, B+ tree, Flash, RDMA

1 INTRODUCTION

Key-value (KV) stores are the heart of modern datacenter storage stacks [2, 13, 15, 28, 32]. These systems typically use an LSM tree [34] 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 [16]. However, LSM-based KV stores suffer from high compaction costs for reorganizing the multi-level index [31, 36], including both I/O amplification and CPU overhead.

Furthermore, to provide reliability and availability, state-of-the-art KV stores [13, 28] replicate KV pairs in multiple, typically two or three [9], nodes. Current designs [13, 28, 41] perform costly compactions to reorganize

[*]Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH), Greece.
[†]Also with the Department of Computer Science, University of Crete, Greece.

Authors' addresses: Giorgos Stilianakis, geostyl@ics.forth.gr, Institue of Computer Science, FORTH, Heraklion, Crete, Greece , Computer Science and University of Crete, Heraklion, Crete, Greece; Giorgos Saloustros, gesalous@ics.forth.gr, Institute of Computer Science, FORTH, Heraklion, Crete, Greece; Orestis Chiotakis, orchiot@ics.forth.gr, Institute of Computer Science, FORTH, Heraklion, Crete, Greece , Computer Science and University of Crete, Heraklion, Crete, Greece; Giorgos Xanthakis, gxanth@ics.forth.gr, Institute of Computer Science, FORTH, Heraklion, Crete, Greece , Computer Science and University of Crete, Heraklion, Crete, Greece; Angelos Bilas, bilas@ics.forth.gr, Institute of Computer Science, FORTH, Heraklion, Crete, Greece , Computer Science and University of Crete, Heraklion, Crete, Greece.

ACM 1553-3077/2024/4-ART
https://doi.org/10.1145/3658672

	BlobDBnoGC	RocksDB	BlobDBGC
I/O amplification	2.0	17.4	27.4

Table 1. I/O amplification in BlobDB (with/without GC) and RocksDB for inserts of small (33 B) KV pairs.

data in the primary and backup nodes to ensure minimal network traffic by moving only user data across nodes. However, this approach significantly increases read I/O traffic, CPU utilization, and memory use at the backups. Since all nodes function simultaneously as primaries and backups, this approach eventually hurts overall system performance.

To increase the data processing capacity given the exponential data growth [40] and power restrictions, we need to design CPU-efficient KV stores. Toward this goal, in this work, we rely on two observations in the datacenter: 1) the wide use of flash devices (NVMe SSDs) and 2) the increased use of RDMA [20, 43].

Flash storage devices (NVMe SSDs) achieve hundreds of thousands of IOPs per device and operate at their maximum throughput even with I/O sizes in the order of tens of KB with adequate queue depths. Additionally, RDMA increases available throughput at low CPU utilization making it viable to trade network traffic for CPU and device I/O.

In this work, we design and implement *Tebis*, an efficient replicated LSM-based KV store. To reduce compaction overhead at the *backups*, *Tebis* ships a pre-built index from the *primary*. This approach reduces read I/O amplification, CPU overhead, and memory utilization in *backup* nodes. 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 KV pairs, using an efficient RDMA-based primary backup communication protocol that does not require the involvement of the *backup* CPUs in communication operations [44]. 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 using a single round trip to reduce the processing overhead at the server.

Our work in this article extends *Tebis* [47], as follows. We redesign the storage engine of *Tebis* to use hybrid KV placement [30, 50] instead of KV separation [31, 36, 37]. LSM KV stores that use KV separation append each KV pair in a value log and keep a pointer (<key, value_log_offset>) to the KV location in the LSM index. Due to update and delete operations, they perform garbage collection (GC) in the value log to reclaim space. GC process is costly for small KV pairs prominent in hyper-scalar workloads [11]. To illustrate the effect of GC in I/O amplification for small KV pairs, we show in Table 1 the effects of GC in I/O amplification using two state-of-the-art KV stores: RocksDB (no KV separation) and BlobDB (with KV separation) for small KV pairs. BlobDBnoGC is a configuration of BlobDB with GC disabled. Even though it is not a realistic setup, we use it to quantify the benefits of KV separation without GC. BlobDBnoGC has 8.7x less I/O amplification compared to RocksDB. When using GC, I/O amplification in BlobDB is 1.57x higher than RocksDB (27.4 vs. 17.4). These numbers include only the identification cost of valid values since there are no deletes or updates and no relocation occurs. Identifying valid keys consumes valuable read throughput from client get and scan operations. Delete and update operations increase I/O amplification because the GC mechanism needs to relocate valid KV pairs by re-appending them to the log and consuming valuable write throughput.

To overcome the shortcomings of Kreon [36], which uses KV separation, we alter the storage engine of *Tebis* to Parallax [50]. Parallax introduces a hybrid KV placement technique that reduces I/O amplification in the following manner. It uses different KV placement strategies for different KV pair sizes. Parallax stores small KV pairs in place within each LSM level. It uses a B+-tree index for each LSM level and stores small KV pairs in its index leaves while it performs transfers from level to level as in LSM-type approaches [19, 42].

For medium and large KV pairs, it purposefully introduces small and random I/Os to reduce I/O amplification. It always places large KV pairs in a log with a clear benefit in I/O amplification at low GC cost. For medium KV pairs, Parallax uses a new technique: It places medium KV pairs in a log up to the last level and then compacts the medium log to the last level, freeing the medium log. Given that it frees the medium log when KV pairs are replaced in the LSM structure, there is no GC overhead associated with the medium log. Therefore, medium KV pairs combine most of the I/O amplification benefits with almost no GC overhead. Using hybrid KV placement in multiple logs and in-place introduces challenges with ordering and recovery. Parallax uses log sequence numbers to maintain the ordering of keys within each region. In addition, Parallax offers crash consistency and can recover to a previous (but not necessarily the last) write, discarding all subsequent writes, as is typical in modern KV stores [19].

In contrast, hybrid KV placement is a state-of-the-art technique that reduces I/O amplification regardless of the KV pair sizes. Index shipping for hybrid KV placement poses the additional challenge of managing and keeping consistent multiple logs at the replicas. *Tebis* uses appropriate protocols between *primary* and *backups* to ensure correctness. In addition, we show that even though in hybrid KV placement, more data are sent through the network because of small and medium KV pairs, CPU gains compared to repeating the compaction process at the *backups* range between $1.21 - 2.78\times$.

Furthermore, we implement a protocol for last write recovery that can handle N-1 failures in a replica group, adding a small overhead in throughput up to 1.15×. Our protocol relies on the efficient, in terms of CPU and latency, message delivery of RDMA. Our protocol also incorporates the appropriate mechanisms to detect torn writes at the *backups*. Finally, we implement a client-server protocol based on TCP/IP and quantify the CPU gains for sending and receiving messages in *Tebis* compared to RDMA.

The rest of this article is organized as follows: Section 2 provides background on LSM KV stores and basic RDMA operations. Section 3 presents our design and implementation of *Tebis*, including the integration of Parallax. Section 5 presents our evaluation methodology and experimental results. Section 6 reviews related work, and Section 7 provides our conclusions.

2 BACKGROUND

In this section we briefly discuss LSM tree index structure and the basic RDMA operations.

LSM tree: LSM tree [34] 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 [19], for simplicity, we use the LSM terminology and refer to it as L_0. There are different ways to organize data in levels [24, 34]. 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 [34], 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 [16], 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 [12, 18, 27, 31, 36] use the ability of fast storage devices to operate at a high percentage (close to 80% [36]) of their maximum read throughput under small and random I/Os to reduce I/O amplification. The main techniques are KV separation [4, 12, 18, 27, 31, 36] and hybrid KV placement [30, 50]. 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 [30, 50] is a technique that extends KV separation and reduces garbage collection

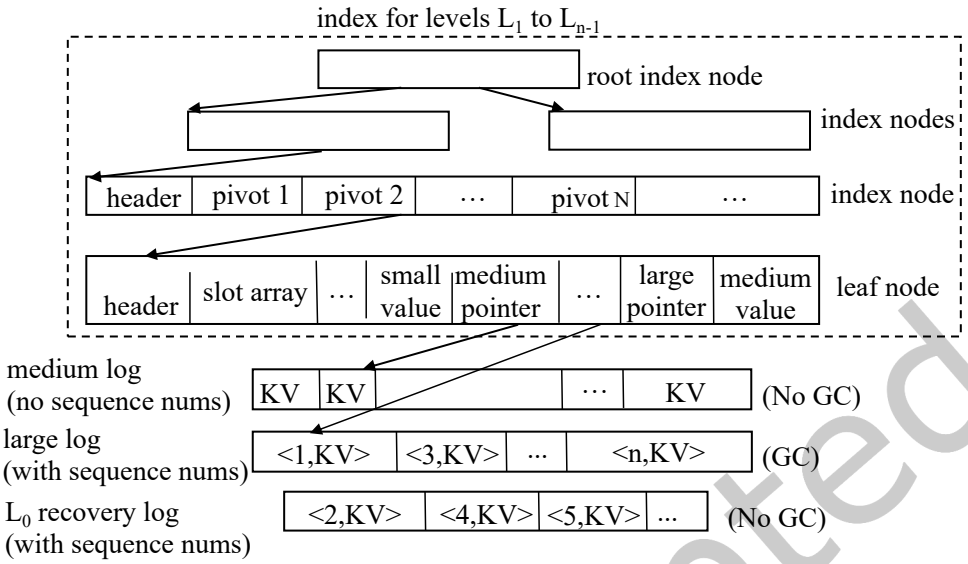


Fig. 1. Overview of index and log design for levels L_1 to L_{n-1} in Parallax. Levels L_0 and L_n always store medium values in place.

overhead, especially for medium (≥ 100 B) and large (≥ 1000 B) KV pairs [11]. 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.

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

3 DESIGN

In this section, we first describe the architecture of Parallax LSM KV store, which *Tebis* uses to organize its data in its local devices. Then, we present the design of *Tebis*.

3.1 Parallax Key-Value Storage Engine

Parallax [50] is a leveled, LSM KV store that offers a dictionary API (insert, delete, update, get, scan) for variable size KV pairs. KV pairs are organized in non-overlapping ranges, called *regions*. Each level in a region uses a concurrent B+-tree index [6, 7, 36, 42] and the region as a whole uses three logs: L_0 recovery log, medium log, and large log. Figure 1 shows an overview of Parallax. All KV logical structures consist of fixed size segments on the device (Figure 2). Currently, Parallax uses 2 MB segments.

The B+-tree index for each level consists of two types of nodes: *index* and *leaf* nodes. Index nodes store pivots (full keys), whereas leaf nodes store either (a) the pair <key, pointer> to the KV location or (b) the actual <key, value> pair. Index and leaf nodes have a configurable size. We set their size to 8 KB which leads to better performance as we have seen experimentally.

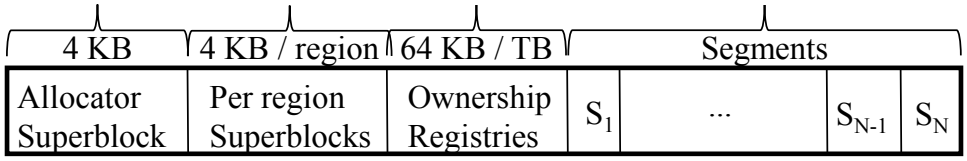


Fig. 2. Device layout used by Parallax and its allocator. A segment may belong to the large, medium, L_0 recovery, or region allocation log, or to a level index.

Get operations examine hierarchically all levels from L_0 to L_n and return the first occurrence. Similar to other KV stores, Parallax uses bloom filters per level to reduce I/Os during lookup operations. *Scan* operations create one scanner per level and use the index to fetch keys in sorted order. They combine the results of each level to provide a global sorted view of the keys.

Insert operations write each KV pair in L_0, similar to all LSM KV stores. *Delete* operations mark keys with a tombstone and defer the delete operation similar to RocksDB, freeing up space at the next compaction. *Update* operations insert the new version of the KV pair in L_0. *Tebis* detects older versions of the same KV pair in higher levels of the LSM tree and discards them during compaction.

3.2 Handling Small KV Pairs

Parallax stores small KV pairs in place in the B+-tree leaves and moves them from level to level with regular compactions. Parallax compacts level L_i to L_{i+1} in a bottom-up fashion: It merges the sorted leaves of levels L_i and L_{i+1} and builds the B+-tree index for L'_{i+1}. In this manner, leaves are always full and compactions require fewer CPU cycles because they avoid index traversals.

3.3 Handling Large KV Pairs via GC

For large KV pairs, Parallax always performs KV separation [12, 18, 31, 36]: It places large KV pairs in the large log, maintains and compacts pointers in the index from level to level, and uses GC to reclaim space, as follows.

First, Parallax avoids full scans in the large log. For this purpose, it stores in a recoverable map per region, named *GC map*, information about the space used by invalid KV pairs in each segment as a pair <segment, invalid-byte-count>. We represent segments with their 8-byte device offset and use an 8-byte invalid-bytes counter per segment. Each compaction thread updates a private GC map when it discovers a deleted or updated large KV pair. At the end of the compaction, it atomically updates the GC map of the region with the contents of its private GC map. Parallax logs and commits in a batch these updates in a per-region operation log to be recoverable after failures. Although this operation incurs overhead, it is only required for large KV pairs and eliminates the need for full log scans. In addition, reads to the GC map are always served in memory because the GC region is small and fits in memory, e.g., 8 MB for a 1 TB device with 2 MB segments.

Then, Parallax reclaims segments with invalid KV pairs. It places segments that exceed a preconfigured threshold (10%) in an eligibility list. If there are no such segments and there is capacity pressure, it considers all segments in the GC region eligible. Parallax uses a dedicated GC thread to scan eligible segments. It iterates over all KV pairs for each segment and uses lookup operations to identify valid KV pairs and append them to the large log. After all valid KV pairs are appended, it reclaims the segment.

3.4 Handling Medium KV Pairs

For medium KV pairs, Parallax uses a hybrid technique: It performs KV separation for all levels except the last one or two levels by placing medium KV pairs in the medium log. As a result, it defers compaction of medium

values up to the last few levels. At the last level(s), e.g. L_n, it moves medium KV pairs in place, similar to small KV pairs. Once medium values are moved in place, Parallax can reclaim the medium log segments without needing expensive GC. Placing medium KV pairs in a log and deferring their compaction raises two questions: (a) What is the size of the medium log and the associated space amplification? (b) How can we efficiently merge the medium log back into the LSM structure?

Efficient merging of the medium log: The compaction process in LSM tree [34] requires inserting keys from L_i to L_{i+1} in sorted order to amortize I/O costs. Otherwise, this process will result in excessive data transfers. When Parallax merges medium KV pairs from the log in place, e.g. in level N, the index of level N-1 already contains sorted pointers to the KV pairs of the medium log. However, a full scan of an unsorted medium log will cause a significant penalty in traffic: Medium KV pairs in the order of hundreds of bytes will result in 4 KB I/O operations, i.e. up to 40x traffic for 100 B KV pairs.

To overcome this cost, Parallax maintains sorted segments in the medium log. To achieve this, Parallax uses L_0 and its recovery log as follows. Initially, it inserts medium KV pairs *in place* in L_0 (in memory), and it also appends them to the L_0 recovery log, along with small KVs for recovery purposes. During compaction from L_0 to L_1, it uses the L_0 index to store medium KV pairs in the medium log as a sorted run of segments, and it stores pointers in the L_1 index. Merging the medium log in place requires at most one segment from each sorted run of segments in the medium log.

One concern for merging the medium KV pair log in place is the amount of buffering required in memory for the sorted runs, during the merge operation. Assuming only inserts of distinct keys, medium KV pairs, and a device capacity of about 10 TB, the medium log at L_{n-1} is about 1 TB with a growth factor of 10. If L_0 is 64 MB, there are about 16 K sorted runs for medium KV pairs in a 1 TB L_{n-1} medium log. Given that fast storage devices, such as SSDs, operate at peak throughput with I/Os of tens or hundreds of KB, we need only to fetch in memory, e.g. a 64 KB chunk for each sorted run of the medium log. As a result, we need at most 1 GB memory for buffering purposes, when merging at L_{n-1}. If medium KV pairs are merged at L_{n-2}, then we only require about 100 MB of buffering.

In the presence of updates, the maximum size for the medium log can exceed the above calculation. In case updates occur to different keys, e.g. each key is updated once, then the medium log size will remain manageable. In case updates occur to a small set of keys, then the size of the medium log can become excessive. In this case, when the medium log exceeds a threshold, Parallax merges medium KV pairs in place earlier and reclaims the medium log. Essentially, the percentage and type of updates affect how late or early the medium log will be merged in place.

Finally, during the compaction that merges medium KV pairs in place, e.g. at L_n, the size of L_{n-1} and L_n needs to be calculated based on their number of bytes rather than the number of keys to satisfy the growth factor and maintain the properties of the LSM tree.

3.5 *Tebis* 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 hybrid KV placement. *Tebis* consists of three main entities, as shown in Figure 3:

(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 primary *region server* of each region in the *Tebis* cluster.
(3) Zookeeper [23] 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

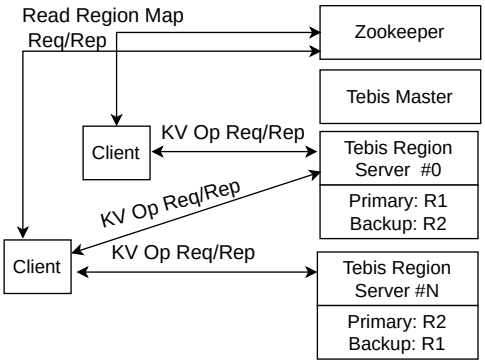


Fig. 3. *Tebis* overview.

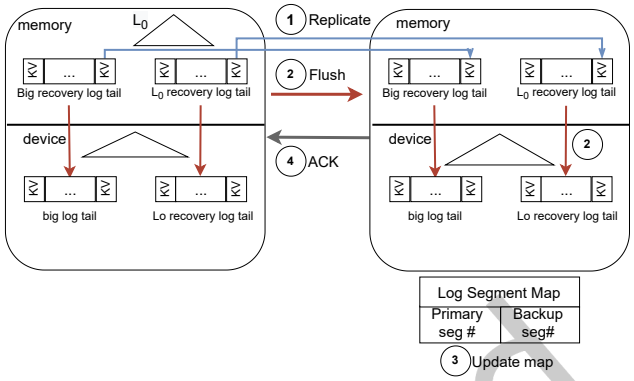


Fig. 4. Value log replication in *Tebis*.

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, 38, 48] that provide strongly consistent metadata replication and notification services.

During initialization, clients read and cache the region map. 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 [10, 44]. *Tebis* serves all write and read operations only from the primary server of each region. Furthermore, different regions hosted in the same region server (with either primary or backup role) are independent. Next, we discuss how *Tebis* replicates its log (Section 3.6) and index (Section 3.7).

3.6 Primary-Backup Value Log Replication

Tebis replicates its log without involving the CPU of *backups*. When it receives updates, inserts, and deletes from clients, the *primary* replicates each operation to its set of *backup* servers in four steps (Figure 4). It inserts the KV pair in Parallax which categorizes the KV pair according to its KV size: small, medium, or large. It then assigns a log sequence number (LSN) to the KV pair and appends it to L_0 recovery or big log for recovery purposes. Then, it appends (via an RDMA write operation) the KV pair to the corresponding (big or L_0 recovery) RDMA buffer of each *backup* at the corresponding offset (step 1 in Figure 4). It is important to notice that replication requests occur with increasing order according to the LSNs.

On the other hand, persisting the tail segment involves the CPU of both the *primary* and *backups*. When the tail segment of either big or L_0 recovery log in the *primary* becomes full, the *primary* flushes the segment to persistent storage and sends a *flush tail* message to each *backup* to persist both RDMA buffers (step 2 in Figure 4). Upon receiving a flush request, *Backups* write their RDMA buffers to persistent storage. Finally, they send an acknowledgment to the *primary* (step 4 in Figure 4).

Tebis supports either a) last write recovery from the memory of the *backups* or b) recover to a previous consistent point, which may not include the last acknowledged write(s). In the first case, the *primary* responds to the clients

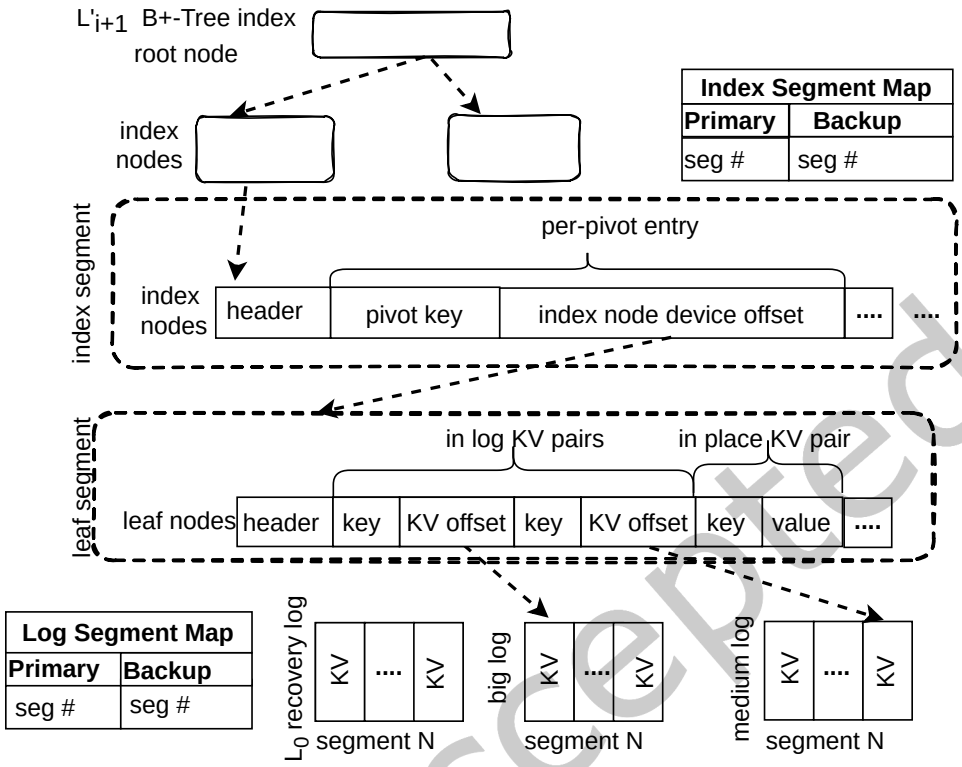


Fig. 5. *Tebis* B+ tree index and value log organization on the storage devices.

after it has received completion from the remote NICs, which ensures that the KV pair is at the memories of the backups [3]. In the second case, *primary* responds to the client requests after issuing the asynchronous RDMA write operation to all of its *backups*. Therefore, *Tebis* can recover to a previous consisent point, which may not include the last acknowledged write(s). Most KV stores (and *Tebis*) can be configured to acknowledge writes after they are written to the device for all *backups* or to perform more frequent flush operations, but this is not commonly used as it increases acknowledgment delay or I/O overhead.

Each *backup* region maintains a log map with entries <primary segment number, backup segment number>, specifying the location of each big and medium log 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 3.7. 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 numbers in its storage device. *Backup* servers, after persisting their value log tail segments, use this information to create the corresponding entries in their log map (step 3 in Figure 4). 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*.

3.7 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 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, 8, 22].

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 reduces the memory budget for L_0 by 2× for one replica per region or 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 throughput. 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 logs (L_0 recovery, medium, big) and the B+-tree indexes of the levels. *Tebis* stores its logs and the B+-tree indexes as a list of fixed segments. Each segment is 2 MB in size 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 5) consists of leaf and index nodes. Leaf nodes (bottom in Figure 5) contain in-log and in-place KV pairs. In-log KV pairs belong to the big or medium category (which *Tebis* has not transferred in-place). In contrast, in-place KV pairs are either small or medium that *Tebis* has transferred in-place.

In-place KV pairs consist of the key and value, whereas in-log KV pairs consist of the key and a device offset pointing to the location of the KV pair in either the big or medium 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 5).

Backups keep track of two mappings for segments: The log map and the index map. The log map is updated during the flush operation of the log (Section 3.6). The log map contains mappings for the big and medium logs. 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}. The *primary* builds its index bottom-up and left to right during compaction. As a result, the *primary* can send the new index incrementally as it is being built, 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 in-log KV pairs (leaf nodes). For each source device offset, it replaces the high order bits with the local segment from the segment map. In *Tebis*, the primary server of each region serves all read and write operations. As a result, during the index rewrite operation, all servers can serve read and write operations for the regions in which they have the primary role.

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.

Space reclamation in the *backup* regions include segments on the device for (a) index and (b) logs (L_0 recovery, medium, and big). *Backup* regions reclaim space of their index segments and trim their L_0 recovery log at the end of each compaction operation. On the other hand, trim operations in the medium log take place during

L_{n-1} to L_n compactions when *primary* moves medium KV pairs in place. To keep the log segment map at the *backups* synchronized, the *primary* instructs at the end of an L_{n-1} to L_n compaction the *backups* to trim their corresponding log segments and update their log segment map. In addition, *primary* performs GC in its big log. GC includes two steps: 1) reinsertion of valid KV pairs in L_0 and 2) reclaim the big log segment. The *primary* defers the reclaim operation until the completion of an L_0 to L_1 compaction, where it informs its *backups* for reclaiming big log segments and updating their log segment map.

It is important to note that the index shipping and rewriting technique can be applied to KV stores that perform full compactions such as RocksDB or use KV separation [18, 30, 31]. 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*.

3.8 Last-Write Recovery from the Memory of the *Backups*

State-of-the-art persistent KV stores [19] offer crash consistency with last-flush recovery (group commit) semantics. In last-flush recovery, the system recovers to a previous (but not necessarily the last) write, discarding all subsequent writes. KV stores implement last-flush recovery semantics mainly for performance reasons.

In *Tebis*, RDMA enables the design of a last-write recovery protocol with a small overhead of up to 4% in throughput compared to last-flush recovery, as we experimentally show in (Section 5.3). *Tebis* can support either last-flush or last-write recovery. In *Tebis*, last-write recovery can tolerate up to N-1 failures, where N is the replication group size. In particular, *Tebis* recovers writes from the memory of *backups* in case of a *primary* failure. However, in the scenario where all servers in a replica group fail, the system recovers to a previous state, discarding all subsequent writes. Future work should examine placing the replication buffers in Non-Volatile Memory (NVM) to address this scenario.

In last-write recovery, *primary* acknowledges a client write request after receiving acknowledgments from the NICs of all backups, ensuring that the KV pair is in their memory [3].

In last-write recovery, the main challenge is to handle torn writes. Torn writes can occur when the *primary* crashes during a KV pair replication operation. In this case, the contents of the KV pair may be partially applied in the memories of the *backups*, resulting in corrupted KV pairs. To address torn writes, *backups* introduce a mechanism to verify the integrity of KV pairs stored in their RDMA buffers to recover them successfully.

The last-write recovery protocol relies on two observations: 1) *Tebis* issues RDMA write operations in each RDMA buffer sequentially, in increasing addresses and 2) *Tebis* uses reliable queue pairs which apply memory writes in FIFO order. Based on these two observations, *Tebis* introduces two markers in the in-memory KV pair layout to enable backups to handle torn writes in case of a *primary* failure.

The in-memory KV pair layout consists of a header that includes the size of the key and the value. The rest contains the variable size KV data. *Tebis* adds two marker bytes, one after the header and one after the variable size KV data. The first marker indicates that the *backup* has received the header correctly, while the second marker shows that *backup* has received the KV payload successfully. In case of a *primary* failure, *backup* iterates its RDMA buffers until it finds a corrupted KV pair or reaches the end of the buffer. Finally, to ensure the correctness of the markers, *backup* zeroes the contents of its RDMA buffers after a flush buffer request from the *primary*.

3.9 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. Prior to any region reconfiguration due to a failure or load

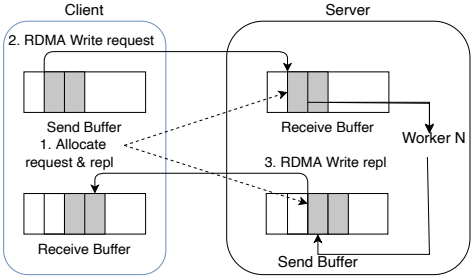


Fig. 6. Allocation and request-reply flow of *Tebis* RDMA Write-based communication protocol.

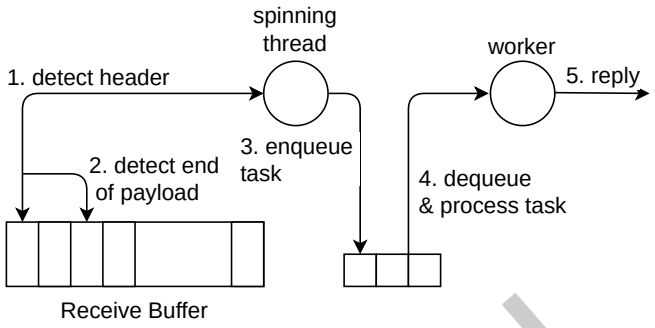


Fig. 7. Message detection and task processing pipeline in *Tebis*. For simplicity, we only draw one circular buffer and a single worker.

balancing operation, *master* first log its intention in a redo log named *region log* which keeps in Zookeeper for persistence and high availability purposes.

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 *primary* server to coordinate the replication group to transfer in parallel their region data to the new *backup*. During the synchronization of the new *backup* process, the region experiencing the failure remains unavailable for the replica group to reach a consistent state. We leave as future work protocols that enable the operation of the region in parallel with the synchronization of the new *backup*.

In case of a *primary* failure, the *master* first promotes the next in line of the list of *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 serving 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*. The new *master* replays the region log to build a consistent state about the regions currently in the system. 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 becomes unavailable until a new *master* is elected and it handles the primary or backup failure as before.

3.10 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.

3.10.1 RDMA Buffer Management. *Tebis* performs client-server communication via one-sided RDMA write operations [26] to avoid network interrupts and reduce the CPU overhead in the server [25, 26]. 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 6). 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.

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 [45] 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. The detection of new messages in one-sided RDMA write operations includes two possible options: 1) Polling rendezvous points, which *Tebis* does and is the fastest option in terms of latency, or 2) instruct the host network adapter (HCA) of the target to generate a completion event with infomation about the arrival of the new message. 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.

3.10.2 Variable Size Messages and Task Scheduling. *Tebis* uses one-sided RDMA write operations for all protocol messages. There are three possible ways to detect the arrival of a new message when using RDMA one-sided write operations: 1) targeted polling (busy wait), 2) blind polling, and 3) interrupts. We do not use interrupts in *Tebis* to avoid the associated CPU costs.

In targeted polling, the sender instructs the receiver's network card to generate a completion event on the arrival of a new message. The completion event entry (CQE) contains info regarding the queue pair that received the message and the number of bytes written. The receiver actively polls its completion queue (CQ), shared among its queue pairs, for new CQE entries.

In blind polling, the receiver checks the memory area of the communication buffer of each queue pair where the new message should be received. Notably, blind polling does not include completion events at the receiver. In *Tebis*, we employ blind polling, as it omits the requirement of CQE events atop PCI from the receiver end.

The main design challenge with variable size messages is how to detect their arrival at the *region server* given the absence of network interrupts and completion events generated at the receiver.

All messages in *Tebis* consist of a *message header* of size 32 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 7. 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 one byte 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

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
Run E	95% scans, 5% inserts

(a) Operation mix for YCSB.

	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	13.7	1.7
L	0-0-100	100M	114.3	14.2
SD	60-20-20	100M	27.4	3.4
MD	20-60-20	100M	31.7	3.9
LD	20-20-60	100M	71.9	8.9

(b) KV size distributions.

Table 2. YCSB workloads(Table 2a) and KV size distributions (Table 2b). Workloads use Zipfian distribution and Run D uses the latest distribution. For (b) small KV pairs are 33 B, medium KV pairs are 148 B, and large KV pairs are 1228 B. We report the record count, dataset size, and cache size per server used with each KV size distribution.

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 7). 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.

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 3.10.0. 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 2b.

In our experiments, we run the YCSB benchmark [14] and its workloads Load A and Run A to Run D. Table 2a 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 two threads per process. To generate enough outstanding requests for each server, each client process uses one queue pair per server which is 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.

In our evaluation, we also vary the KV pair sizes according to the KV sizes proposed by Facebook [11], as shown in Table 2b. 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 [39] and we modify it to produce different values according to the KV pair size distribution we study.

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 [11], as well as a medium dominated (*MD*) and a large dominated (*LD*) workload. We summarize these KV distributions in Table 2b.

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 128 MB per server. 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 hyper-threads.

I/O amplification measures the ratio of device traffic performed in excess of reading and writing user data; compaction I/O represents a significant portion of all excess 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.

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, acknowledgments), therefore, network traffic is always higher than application data. In addition, our RDMA client-server protocol uses a minimum payload of 64 B to reduce CPU usage 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?

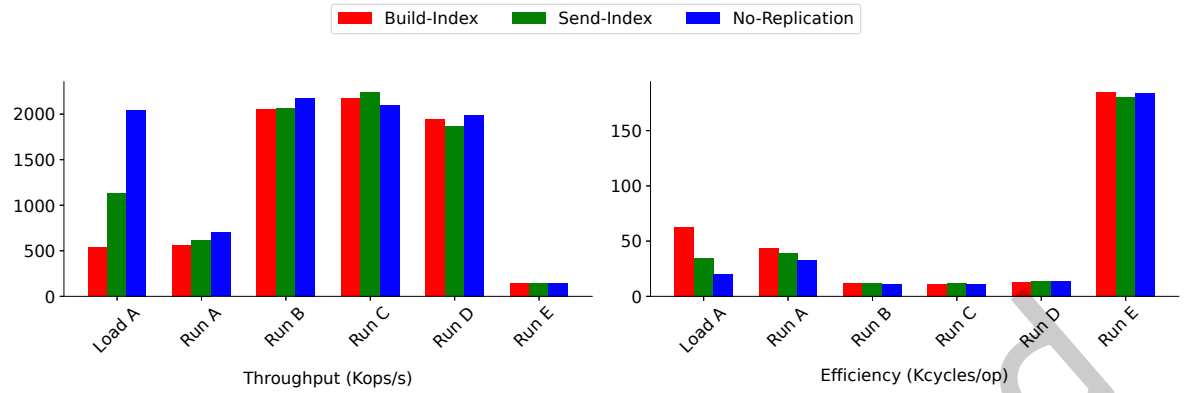
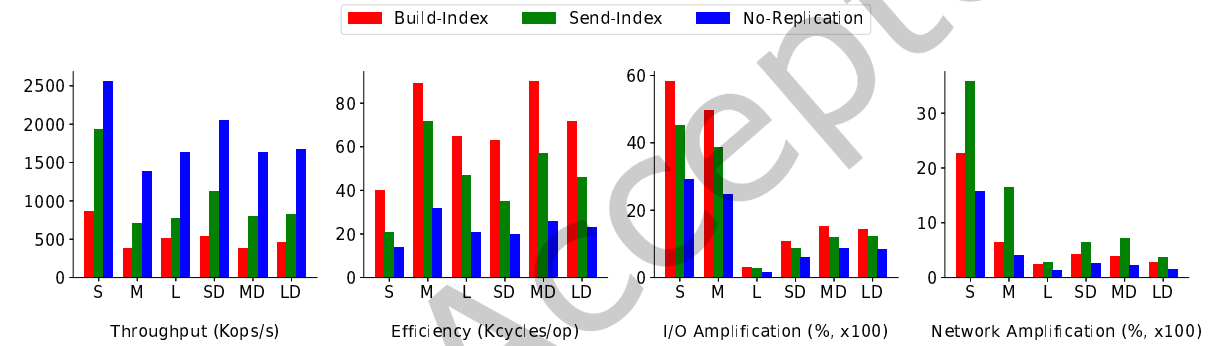
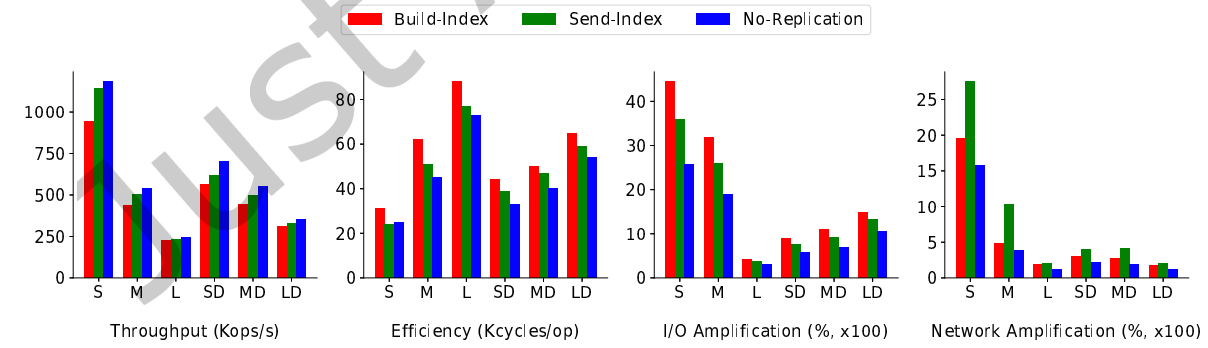


Fig. 8. Performance and efficiency of *Tebis* for workloads Load A, Run A – Run E with the SD KV size distribution.

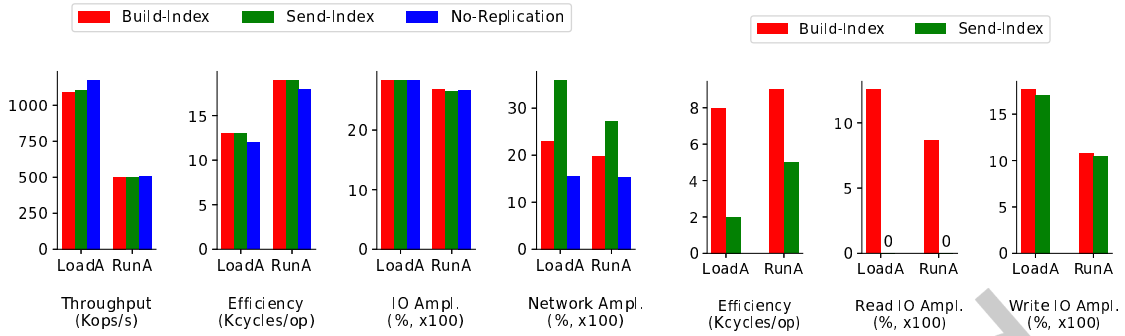


(a) Load A



(b) Run A

Fig. 9. 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.



(a) Primary (b) Backup

Fig. 10. Overhead breakdown for the primary and backup role.

(4) What are the gains in throughput, efficiency, and I/O amplification for three-way replication?
(5) Does using a smaller L_0 in Build-Index, to counterbalance the L_0 memory budget compared to Send-Index, have an impact on performance, efficiency, and I/O amplification?
(6) What are the CPU efficiency gains of RDMA vs TCP/IP?

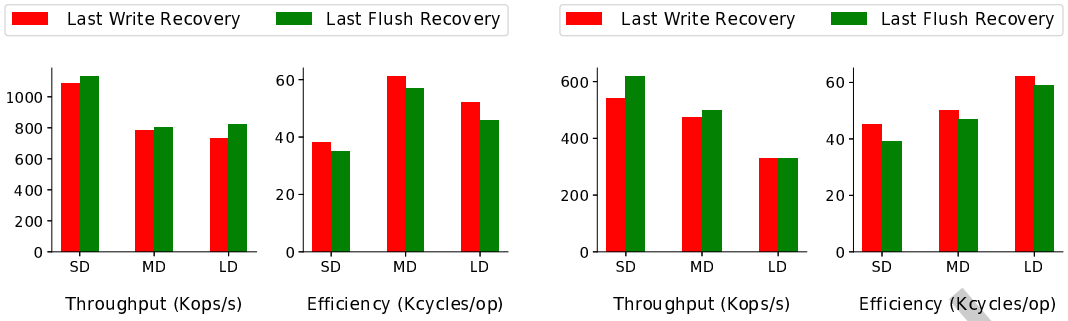
5.1 *Tebis* Performance and Efficiency

In Figure 8, we evaluate *Tebis* for two-way replication using YCSB workloads Load A and Run A to Run E for the SD KV distribution [11]. Since replication does not have impact on read-dominated workloads, the performance in workloads Run B to Run E 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 8. Figure 9 shows that compared to Build-Index and for all KV size distributions, Send-Index increases throughput by 1.04 – 2.24×, CPU efficiency by 1.06 – 1.9×, and reduces I/O amplification by 1.08 – 1.28×. 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.

5.2 Overhead Breakdown

To assess the overheads of each server role of *Tebis*, we run a Load A and Run A for the S KV distribution experiment with two-way replication. One server assumed the primary role, while the other served as the backup for all 32 regions, enabling accurate correlation of server overheads with their respective roles. As shown in Figure 10, I/O amplification is constant for the primary role as neither Send-Index nor Build-Index approaches impose excessive device traffic. The advantages of using the Send-Index approach are evident in the backup role, where it eliminates Read I/O Amplification compared to Build-Index, and CPU usage is reduced by 1.8 – 4×. This is because Send-Index backups only need to perform index rewriting operations for translating the primary index, which is significantly less taxing on the CPU than Build-Index's merge sort operations for compactions. Additionally, Read I/O Amplification elimination is the result of the index-shipping, where backup nodes only need to persist the pro-constructed primary index.



(a) Load A

(b) Run A

Fig. 11. Throughput (Kops/s) and efficiency (Kcycles/op) for last flush and last write recovery protocols.

5.3 Replication Protocols Trade-offs

This experiment examines the performance and efficiency of *Tebis*'s replication protocols: the last flush and last write recovery protocols. To evaluate these protocols, we use a growth factor of 8 and maintain one replica per region (two-way replication). We run the Load A and Run A workloads for all YCSB's mixed KV distributions. As shown in Figure 11, the last write recovery protocol exhibits a minimal reduction in throughput by $1 - 1.14\times$ and a slight decrease in CPU efficiency by $1.05 - 1.15\times$ compared to the last flush recovery protocol. For both protocols, the I/O and network amplification remain the same.

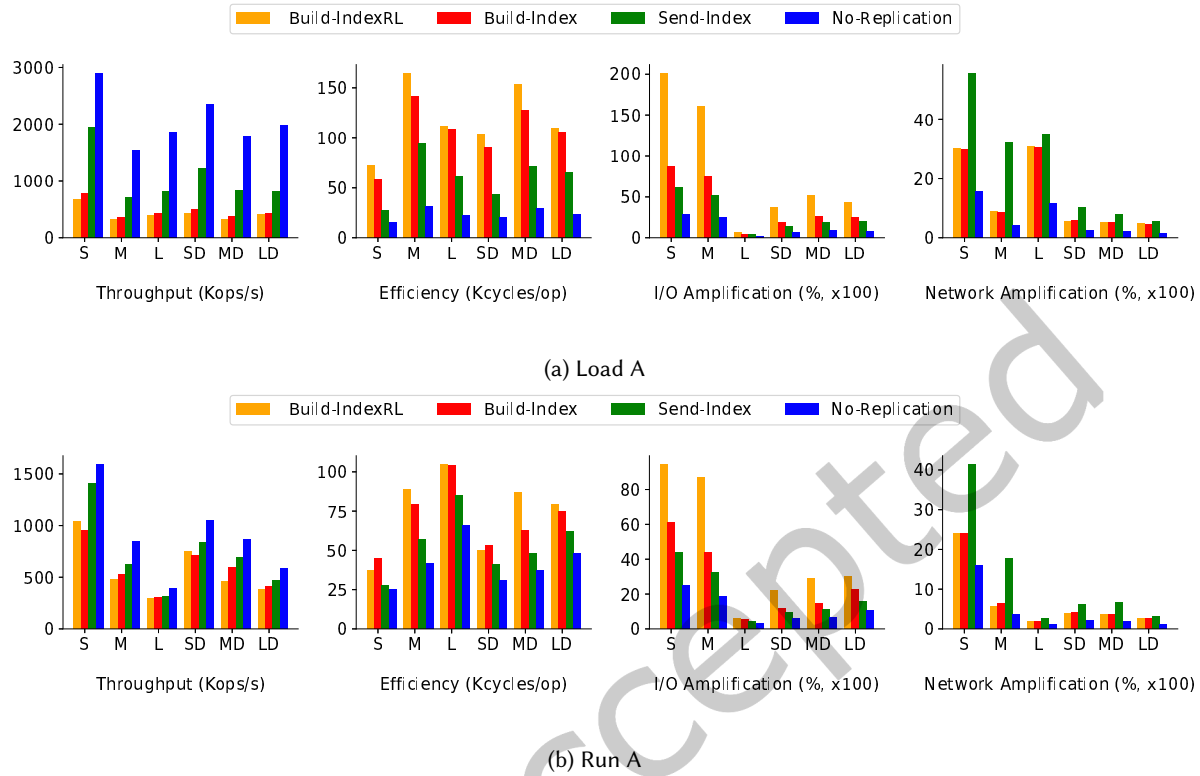
5.4 Three-way Replication

We run Load A and Run A for all six KV distributions with a growth factor of 8. In this experiment, we keep two replicas per region, in addition to the *primary* copy. We set the L_0 size to 128 MB per server for the No-Replication, Build-Index, and Send-Index configurations.

Figure 12 shows that for Load A, compared to Build-Index, Send-Index improves throughput by $1.05 - 2.46\times$, increases CPU efficiency by $1.2 - 2.1\times$, and decreases I/O amplification by $1.11 - 1.42\times$. Compared to two-way replication we see that the gains increase for throughput from $1.04 - 2.24\times$ to $1.05 - 2.46\times$, for efficiency from $1.06 - 1.9\times$ to $1.2 - 2.1\times$, and for I/O amplification from $1.08 - 1.28\times$ to $1.11 - 1.42\times$. 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.

5.5 L_0 Memory Usage

It is important to note that compared to Send-Index, Build-Index uses $2\times$ more memory for L_0 when keeping two replicas and $3\times$ 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 128 MB for all *primary* and *backup* regions.



(a) Load A

(b) Run A

Fig. 12. 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.

cycles/op in Load A, SD, 100 M KV pairs

	Recv	Send	Parallax	Other	Total
TCP/IP	22223	14815	15344	531	52913
RDMA	2895	2481	15096	208	20680
Reduction	87%	83.3%	1.7%	61%	61%

Table 3. Breakdown of the cycles spent by all server threads in each component of *Tebis* for TCP/IP and RDMA.

Compared to Build-IndexRL, Send-Index improves throughput by 1.06 – 2.90×, increases CPU efficiency by 1.21 – 2.78×, and decreases I/O amplification by 1.7 – 3.27×. 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.

5.6 RDMA gains over TCP/IP

In this experiment, we quantify the CPU efficiency gains of RDMA in addition to a version of *Tebis* that uses TCP/IP for communication with the clients. We use a standalone *Tebis* server (No-replication) with 32 regions and run the Load A phase of YCSB benchmark for the SD, MD, and LD KV size distributions. As we observe from

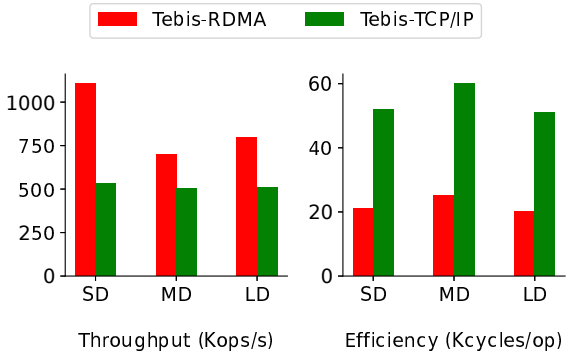


Fig. 13. Throughput (Kops/s) and CPU efficiency (Kcycles/op) for TCP/IP and RDMA, using Load A workload with the SD, MD, and LD KV size distributions.

Figure 13, compared to TCP/IP, RDMA increases throughput from 1.39× up to 2.07× and CPU efficiency from 2.4× up to 2.55×.

To further investigate the benefits of RDMA, we run Load A using the SD KV size distribution with 100 M KV pairs. We divide each operation into four stages and show for each operation the CPU cycles spent per stage (Table 3). The four stages are:

(1) Recv: Data path from the network card up to the copy of data in the *Tebis* server application buffer.
(2) Send: Data path for sending data from an application buffer of *Tebis*
(3) Parallax: includes cycles spent per operation in KV storage engine and other time.

As we observe from Table 3, the receive and send path are 87% and 83% more CPU efficient. The CPU efficiency gains in the RDMA come from the removal of interrupts and memory copies.

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: RubbleDB [29] is a replicated LSM KV store that ships the index to the backups to avoid repeating compactions. It stores all of its KV pairs in the index, uses chain replication [46] to replicate data, and uses NVMe-OF to write data the NVMe devices of the backups. On the contrary, *Tebis* implements the appropriate mechanisms to support in-place and in-log KV pairs (hybrid KV placement) and last write recovery. Furthermore, in *Tebis*, backup servers are the control plane of the index shipping mechanism of *Tebis*, allowing the system to add extra features such as compression and encryption easily.

Acazoo [21] 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 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 [51] 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 [41] 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 [35] 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 [44] 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.* [26] 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 [25], 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 [17, 33, 49] 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.

In all setups where Send-Index has the same L_0 size with Build-Index, our evaluation shows that Send-Index increases throughput by $1.06 - 2.90\times$, CPU efficiency by up to $1.21 - 2.78\times$ and decreases I/O amplification by $1.7 - 3.27\times$ for Load A and Run A. Our approach increases network traffic by $1.32 - 3.76\times$, creating a tradeoff between network utilization and backup servers resource use.

ACKNOWLEDGMENTS

We thankfully acknowledge the support of the European High-Performance Computing Joint Undertaking (EuroHPC JU) under project EUPEX (grant agreement No 101033975). The EuroHPC JU receives support from the

European Union's Horizon 2020 research and innovation programme and France, Germany, Italy, Greece, United Kingdom, Czech Republic, Croatia. We thankfully acknowledge the support of the European Commission under the European High-Performance Computing Joint Undertaking (JU), through project EUPEX (grant agreement No 101033975).

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] Rudolf Bayer and Edward McCreight. 2002. *Organization and maintenance of large ordered indexes.* Springer.
[7] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-trees. *Acta Inf.* 9, 1 (March 1977), 1–21.
[8] 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
[9] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop apache project* 53, 1-13 (2008), 2.
[10] 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
[11] 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.
[12] 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
[13] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide* (second ed.). O'Reilly Media. http://amazon.com/o/ASIN/1449344682/
[14] 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
[15] 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.
[16] 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
[17] 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.
[18] Facebook. 2018. BlobDB. http://rocksdb.org/. Accessed: April 16, 2024.
[19] Facebook. 2018. RocksDB. http://rocksdb.org/.
[20] 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
[21] 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
[22] 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
[23] 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

[24] 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.

[25] 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

[26] 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.

[27] 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

[28] 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

[29] Haoyu Li, Sheng Jiang, Chen Chen, Ashwini Raina, Xingyu Zhu, Changxu Luo, and Asaf Cidon. 2023. RubbleDB: CPU-Efficient Replication with NVMe-oF. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 689–703. https://www.usenix.org/conference/atc23/presentation/li-haoyu

[30] 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.

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-F érez, 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

[37] Anastasios Papagiannis, Giorgos Saloustros, Pilar Gonzá lez 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

[38] 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

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

[40] Seagate. [n. d.]. Data Age 2025. https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf. Accessed: April 16, 2024.

[41] 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

[42] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[43] 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

[44] 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

[45] 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

[46] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 7.

[47] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2022. Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3492321.3519572

[48] 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

[49] 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.

[50] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid Key-Value Placement in LSM-Based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 305–318. https://doi.org/10.1145/3472883.3487012

[51] 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