Anthus: Index Shipping for LSM-based Key-Value Stores Utilizing Hybrid Key-Value Placement

Georgios Stylianakis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Supervisor: Prof. *Angelos Bilas*

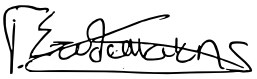
This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

University of Crete
Computer Science Department

Anthus: Index Shipping for LSM-based Key-Value Stores Utilizing Hybrid Key-Value Placement

Thesis submitted by
Georgios Stylianakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: 

Georgios Stylianakis

ANGELOS BILAS ANGELOS BILAS Aug 9 2023 12:15 PM

Committee approvals:

Angelos Bilas
Professor, Thesis Supervisor

KONSTANTINOS MAGOUTIS
09/08/2023 14:30

Kostas Magoutis
Associate Professor, Committee Member

Polyvios Pratikakis
Associate Professor, Committee Member

Departmental approval:

Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, July 2023

Anthus: Index Shipping for LSM-based Key-Value Stores Utilizing Hybrid Key-Value Placement

Abstract

Key-Value (KV) stores based on LSM tree have become a foundational layer in the storage stack of Data Centers and Cloud Services. In state-of-the-art distributed KV stores, the communication over the network is a severe performance bottleneck. Current designs reduce the network traffic by sending only user data across nodes to achieve system reliability and availability. As a result, they perform costly compaction operations to reorganize data in both primary and backup nodes. This approach increases device I/O traffic and CPU overhead and eventually hurts overall system performance.

Initially, *Tebis* introduced the notion of Index-Shipping. Index-Shipping is an efficient solution for two reasons: Firstly, *Tebis* uses RDMA thus reduces the network overhead compared to traditional practices (TCP/IP) and secondly it reduces resource consumption at the backup nodes by maintaining the replica index. However, *Tebis* utilizes KV separation. In case of small KV pairs, which dominate modern workloads, this method is impractical because it heightens garbage collection costs resulting to increased I/O amplification.

In this work we introduce *Anthus*; an efficient replicated LSM-based KV store which extends *Tebis*. In our implementation we utilize an Index-Shipping method for KV stores that relies on hybrid KV placement instead of KV separation. Hybrid KV placement is an emerging technique that reduces I/O amplification regardless of the KV pair sizes. Our results show that our design increases throughput by $1.06-2.90\times$, CPU efficiency by up to $1.21-2.78\times$ and decreases I/O amplification by $1.7-3.27\times$ over baseline implementations. Also, *Anthus* increases throughput by $1.06-1.95\times$, CPU efficiency by $1.14-1.8\times$ and minimizes I/O amplification by $1.5-1.87\times$ over *Tebis*.

Αποστολή ευρετηρίου για συστήματα Κλειδιού-Τιμής βασισμένα σε LSM που χρησιμοποιούν υβριδική τοποθέτηση Κλειδιού-Τιμής

Περίληψη

Τα συστήματά αποθήκευσης Κλειδιού-Τιμής που βασίζονται σε δέντρο LSM έχουν γίνει ένα βασικό κομμάτι στο λογισμικό αποθήκευσης δεδομένων στα Κέντρα Δεδομένων και των Υπηρεσιών Νέφους. Στα σύγχρονα κατανεμημένα συστήματά αποθήκευσης Κλειδιού-Τιμής, η επικοινωνία μέσω του δικτύου αποτελεί ένα σημαντικό περιοριστικό παράγοντα για την απόδοση τους. Κατά συνέπεια, οι τρέχοντες σχεδιασμοί μειώνουν την κίνηση στο δίκτυο, αποστέλλοντας μόνο δεδομένα χρηστών (user data) μεταξύ των κόμβων, για να επιτύχουν την αξιοπιστία και την διαθεσιμότητα του συστήματος. Ως αποτέλεσμα, πραγματοποιούν δαπανηρές λειτουργίες συμπύκνωσης (compaction operations) για την αναδιάταξη των δεδομένων στο σύνολο των κόμβων. Αυτή η προσέγγιση αυξάνει την κίνηση I/O, τον φόρτο του επεξεργαστή, και τελικά επηρεάζει αρνητικά την συνολική απόδοση του συστήματος.

Αρχικά, το Tebis εισήγαγε τη έννοια του Index-Shipping. Το Index-Shipping είναι μια αποτελεσματική λύση για δύο λόγους: Πρώτον, το Tebis χρησιμοποιεί RDMA για να μειώσει το κόστος του δικτύου σε σύγκριση με τις παραδοσιακές πρακτικές (TCP/IP) και δεύτερον, μειώνει την κατανάλωση πόρων στους αντίγραφους (backup) κόμβους αποστέλλοντας το ευρετήριο του πρωτεύοντος (primary) κόμβου. Ωστόσο, το Tebis χρησιμοποιεί την τεχνική διαχώρισης Κλειδιού-Τιμής (KV separation). Σε περίπτωσεις φόρτου εργασιών αποτελούμενων απο ζευγάρια Κλειδιού-Τιμής μικρού μεγέθους, οι οποίες κυριαρχούν στις μέρες μας, αυτή η μέθοδος είναι ανεφάρμοστη επειδή αυξάνει το κόστος της λειτουργίας ανάκτησης χώρου (garbage collection) οδηγώντας σε αυξημένο I/O.

Σε αυτήν την εργασία παρουσιάζουμε το Anthus, ένα αποδοτικό κατανεμημένο σύστημα αποθήκευσης Κλειδιού-Τιμής, βασισμένο στο δέντρο LSM, που επεκτείνει το Tebis. Στην υλοποίησή μας χρησιμοποιούμε μια μέθοδο Index-Shipping για συστήματα Κλειδιού-Τιμής που βασίζονται στην υβριδική τοποθέτηση Κλειδιού-Τιμής (Hybrid KV Separation) αντί για την μέθοδο του διαχωρισμού Κλειδιού-Τιμής (KV Separation). Η υβριδική τοποθέτηση Κλειδιού-Τιμής αποτελεί μια αποτελεσματική τεχνική καθώς μειώνει το επιπλέον I/O ανεξάρτητα από το μέγεθος των ζευγαριών Κλειδιού-Τιμής. Τα αποτελέσματα μας δείχνουν ότι ο σχεδιασμός μας αυξάνει την απόδοση κατά $1.06 - 2.90\times$, την αποτελεσματικότητα του επεξεργαστή κατά $1.21 - 2.78\times$ και μειώνει το επιπλεον I/O κατά $1.7 - 3.27\times$ σε σύγκριση με τις τρέχουσες τεχνικές. Επιπλέον, συγκριτικά με το Tebis, το Anthus αυξάνει την απόδοση κατά $1.06 - 1.95\times$, την αποτελεσματικότητα του επεξεργαστή κατά $1.14 - 1.8\times$ και μειώνει το επιπλεον I/O κατά $1.5 - 1.87\times$.

Acknowledgements

I am deeply grateful to all the individuals who surrounded me in the department and created a productive and enjoyable environment, to say at least. In particular, I want to thank my supervisor, Professor Angelos Bilas, for his support from my Bachelor studies until now. His teachings and mentorship all these years have been and will remain, invaluable to me. Furthermore, I want to express my appreciation to Prof. Polyvios Pratikakis and Prof. Kostas Magoutis for being a member of my M.Sc. Committee and their feedback.

I also want to express my heartfelt appreciation to all the members of the Computer Architecture and VLSI Systems Laboratory (CARV), especially the members of the Storage group, who each helped me in a distinct way. Among the members of the Storage group, I am grateful to Giorgos Saloustros, whose technical expertise and resources were instrumental in the success of this thesis.

I would like to extend my gratitude to my dear friends Sotiris, Vaggelis, Giannos, and Giorgos for their support and perfect collaboration during our MSc-PhD studies. In addition to these friends, I want to extend my gratitude to my dear friends Kostas S., Orfeas, Eva, and Xristina, who have been a constant source of support and encouragement throughout my academic journey. Last but not least, I also wish to thank my remote friend, only in terms of distance, Kostas D., and all the above, for the beautiful moments we shared in Crete.

Finally, I know by now that I owe more than I can grasp to my family. To my brothers Manolis and Iosif, who always supported me and made every visit back home a new experience, to say at least. To my parents, Konstantinos and Efstathia, whose goal was always to reach our goals. Without you, none of this would be possible.

Στους γονείς μου, Κωνσταντίνο και Ευσταθία

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Key-Value (KV) stores are the heart of modern datacenter storage stacks [27, 24, 14, 12, 2]. These systems typically use an LSM tree [28] 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 [15]. However, LSM-based KV stores suffer from high compaction costs for reorganizing the multi-level index [26, 29], including both I/O amplification and CPU overhead.

Furthermore, to provide reliability and availability, state-of-the-art KV stores [12, 24] replicate KV pairs in multiple, typically two or three [9], nodes. Current designs [34, 24, 12] perform costly compactions to reorganize 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 [33] 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 [36, 18]

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.

This work suggests *Anthus*, an efficient replicated LSM-based KV store which extends *Tebis* [39]. To reduce compaction overhead at the *backups*, *Anthus* similar to *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*. *Anthus* creates mappings between aligned *primary* and *backup* segments. Then, it

uses these mappings to rewrite device locations at the *backups* efficiently.

Anthus 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 [37]. In addition, to reduce CPU overhead for client-server communication, *Anthus* uses one-sided RDMA write operations. The protocol of *Anthus* supports variable-size messages using a single round trip to reduce the processing overhead at the server.

Each server in *Anthus* uses Parallax [41] LSM KV store to organize its data efficiently. 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 [17, 35].

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 [17].

We evaluate *Anthus* using a modified version of the Yahoo Cloud Service Benchmark (YCSB) [13] that supports variable KV sizes for all YCSB workloads, similar to Facebook's [10] production workloads. Our results show that our index shipping method requires $1.21 - 2.78\times$ fewer CPU cycles per operation compared to a baseline implementation that performs compactions at the *backups*. Furthermore, it achieves $1.06 - 2.90\times$ higher throughput and reduces I/O amplification by $1.7 - 3.27\times$. Overall, our technique of sending and rewriting a pre-built index trades CPU, memory, and read I/O amplification for increased network traffic.

Our work extends *Tebis* as follows. We redesign the *Tebis* storage engine to use hybrid KV placement [41, 25] instead of KV separation [29, 26, 30]. KV separation of small KV pairs is impractical for operational environments because it heightens garbage collection costs and thus increases I/O amplification, compared to LSM KV stores that always store KV pairs in-place (RocksDB [17]). 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. *Anthus* 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 $1.15\times$. 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 *Anthus* compared to RDMA.

The rest of this article is organized as follows: Section 2 provides background. Section 3 presents our design and implementation of *Anthus*. Section 4 presents our evaluation methodology and experimental results and Section 5 provides our conclusions.

1.1 Contributions

This thesis suggests Index Shipping for LSM-based KV stores that utilize hybrid KV placement. To achieve this, the author extends *Tebis* to support hybrid KV placement by redesigning its storage engine. This required a complete system re-implementation (referred to as *Anthus* throughout this work), as *Tebis* was closely tied to its original storage engine. Specifically, to support hybrid KV placement, *Anthus* introduces:

1. a new mechanism for replicating the value logs of its new storage engine.
2. an Index Shipping process to properly replicate and rewrite the index of the primary.
3. a new Build-Index process in order to properly measure its overheads.

Moreover, the author designs and implements a protocol for last write recovery that can handle N-1 failures in a replica group. The protocol relies on the efficient, in terms of CPU and latency, message delivery of RDMA. Also, this work includes the implementation of a client-server protocol based on TCP/IP, to quantify the CPU gains for send/receive operations in *Anthus* compared to RDMA. Furthermore, the author modifies the I/O path of Parallax to maximize the I/O utilization by exploiting the performance capabilities of both direct and memory-mapped I/O.

Finally, *Anthus* is evaluated through multiple configurations using a modified version of the Yahoo Cloud Service Benchmark (YCSB). This modified benchmark supports variable KV sizes for all YCSB workloads, emulating production workloads similar to Facebook [10].

Our results show that in all setups where Send-Index has the same L0 size with Build-Index, Send-Index method of *Anthus* increases throughput by $1.06 - 2.90\times$, CPU efficiency by up to $1.21-2.78\times$ and decreases I/O amplification by $1.7-3.27\times$.

Our approach increases network traffic by $1.32-3.76\times$, creating a trade-off between network utilization and backup region servers resource use.

Compared to Tebis, in the same setups, *Anthus* increases throughput by $1.06-1.95\times$, CPU efficiency by up to $1.14-1.8\times$, decreases I/O amplification by $1.50-1.87\times$, while it increases network utilization by $1.21-2.06\times$.

Chapter 2

Background

2.1 LSM tree

The LSM tree is a write-optimized data structure commonly used in modern database systems for storing and retrieving data. It organizes data into multiple levels, with each level's size growing by a constant growth factor, denoted as 'f'. The first level, known as L_0 or the memtable, resides in-memory and has a size in the order of hundreds of megabytes. The subsequent levels are stored on the underlying storage device.

In LSM tree implementations, different strategies exist for organizing data across levels. 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.

To minimize I/O amplification, which refers to the increase in I/O operations compared to the number of writes the application expects, a growth factor of f = 4 results in the minimum I/O amplification. However, production KV stores often utilize larger growth factors, typically ranging from 8 to 12. While larger growth factors increase overall I/O amplification, they reduce the number of levels required. This level reduction leads to decreased space usage on the storage device, particularly in scenarios with high update ratios, assuming intermediate levels primarily contain update and delete operations.

One critical aspect of LSM trees is the compaction process, which manages the merging and removal of redundant data across different levels. Compaction is necessary to ensure efficient read performance and disk space utilization. The compaction process is typically triggered when certain conditions are met, such as when data size in a level exceeds a predefined threshold or when the system is under low disk utilization.

During compaction, the LSM tree merges multiple sorted files from different levels into a new file, eliminating redundant key-value pairs and creating a more compact representation. The merged file is then written to the next level, replacing the existing files. By compacting the data, the LSM tree reduces the number of disk reads required during query processing.

The frequency and strategy of compaction operations can vary across LSM tree implementations and depend on factors such as system workload, storage constraints, and performance requirements. Some systems employ background compaction processes that run continuously or periodically in the background, while others may use more aggressive strategies, to optimize performance and space utilization.

2.1.1 The Notion of KV Separation

Current KV store designs [23, 26, 29, 11, 16] exploit the ability of fast storage devices to operate at a high percentage (close to 80% [29]) of their maximum read throughput under small and random I/Os to reduce I/O amplification. The main techniques are KV separation [23, 26, 29, 11, 16, 4] and hybrid KV placement [41, 25]. 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 [25, 41] 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 [10]. Hybrid KV placement also places large KV pairs in a separate log, small KV pairs in-place within each LSM tree level, and medium KV pairs in a separate value log until the last, one or two, level(s) where it reclaims the medium value log.

2.2 Remote Direct Memory Access

RDMA (Remote Direct Memory Access) is a networking technology that enables efficient data transfer between remote systems in a distributed computing environment. RDMA allows one computer to directly access the memory of a remote computer without involving the operating system at any host. This enables zero-copy transfers, reducing latency and CPU overhead.

2.2.1 One-Sided RDMA Operations

One-sided RDMA operations, also known as remote memory operations, allow a process on one system to directly access the memory of another system without involving the remote host's CPU. These operations enable efficient data transfer by bypassing the need for traditional message passing or remote procedure calls.

In one-sided RDMA, a process initiates a memory operation by specifying the remote memory address and the local data buffer. The data is transferred directly from the local buffer to the remote memory, without any involvement of the remote host's CPU. This approach minimizes the overhead of data movement and provides high throughput and low latency.

2.2.2 Two-Sided RDMA Operations

Two-sided RDMA operations involve communication between two processes, where both processes actively participate in the data transfer. Unlike one-sided RDMA operations, two-sided operations require coordination between the sender and receiver processes.

In a two-sided RDMA operation, the sender process initiates the operation by posting a work request that includes the remote memory address and the data buffer. The receiver process, upon receiving the request, validates the request and performs the necessary operations on its side, such as reading or writing data. The sender process is then notified about the completion of the operation.

Two-sided RDMA operations are commonly used in message passing paradigms, where explicit coordination between sender and receiver processes is required. These operations allow for more flexibility in terms of control and synchronization compared to one-sided operations.

2.2.3 Reliable Connections

Reliable Connections(RC) is one of the transport protocols provided by RDMA networks. Reliable Connections offer several important features that ensure reliable and ordered data transfer between endpoints. One of these features is the FIFO ordering of messages. When multiple messages are sent over a Reliable Connection, they are guaranteed to be delivered in the same order they were sent. In addition to message ordering, using Reliable Connections also guarantees that RDMA writes are applied in increasing address order (e.g., if the last byte of a message is written in-memory, then it is guaranteed that the whole message is written).

These features of Reliable Connections provide strong guarantees for data integrity and ordering, making them particularly valuable for applications that require precise control over data consistency and reliability.

2.3 Tebis

2.3.1 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 2.1):

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.

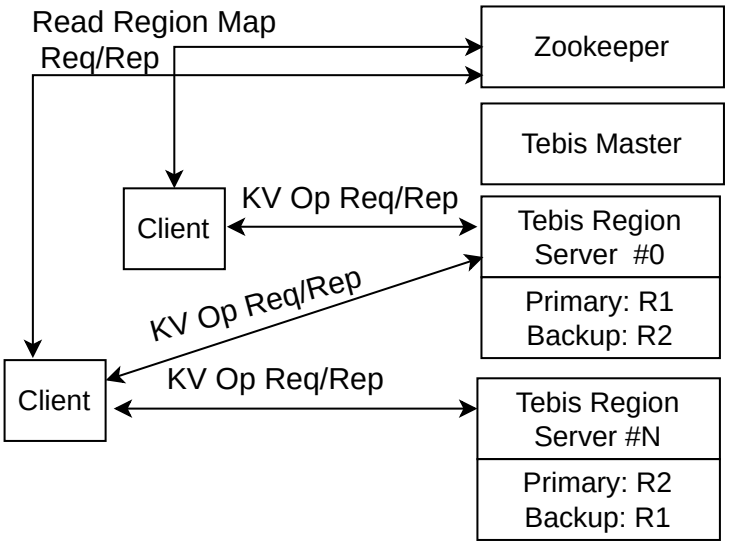


Figure 2.1: *Tebis* overview.

3. Zookeeper [20] 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.

2.3.2 Value Log Replication

When Tebis receives updates and inserts from clients, the primary server replicates each operation to its set of backup servers in three steps (Figure 2.2). First, it inserts the key-value pair in Kreon, which returns the offset of the pair in the value log tail segment. Then, it appends(via an RDMA-write operation) the key-value pair to the RDMA buffer of each backup server at the corresponding offset. Tebis waits for acknowledgment that all the RDMA write operations have been replicated in the memory of the backup servers. Tebis uses the work completion events of reliable queue pairs [36].

Persisting the tail segment involves the CPU of both the primary and backup servers. When the tail segment of the log in the primary server becomes full, it flushes the segment to persistent storage and sends a flush tail message to each backup server to persist their RDMA buffer. Upon receiving the flush request, the backup servers write their corresponding RDMA buffer to persistent storage and send an acknowledgment to the primary server.

Each backup region maintains a log map that specifies the location of each segment on the storage device in both the primary and backup servers. The log

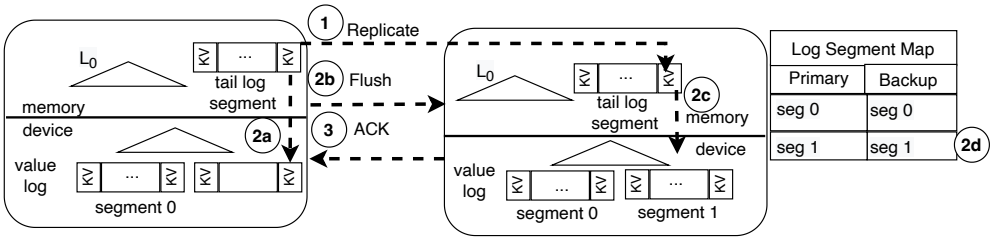


Figure 2.2: *Tebis* log replication.

map has a small memory footprint and is updated when a primary server changes due to a failure or load balance operation.

2.3.3 Index Shipping in *Tebis*

In this Send-Index method, when L_i becomes full, the *primary* region executes the compaction process of L_i and L_{i+1} and sends the resulting index L'_{i+1} to the *backup* regions. *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.

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 2.3) consists of leaf and index nodes. For each KV pair, leaf nodes contain a key prefix, which reduces I/O operations to the value log [29], 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 2.3).

Backups keep track of two mappings for segments: the log map and the index map. 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).

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.

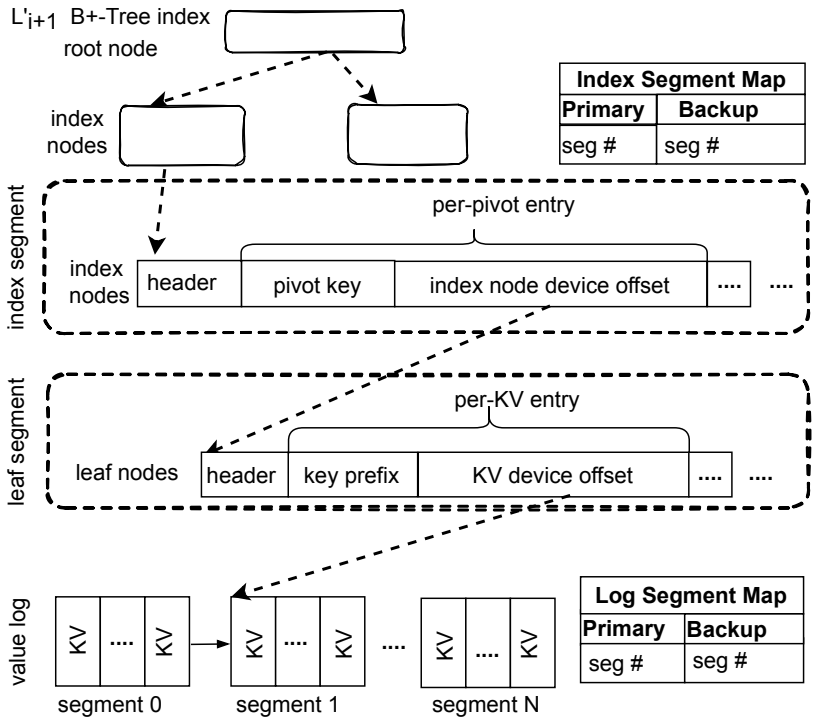


Figure 2.3: *Tebis* index shipping.

2.3.4 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 host-name 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

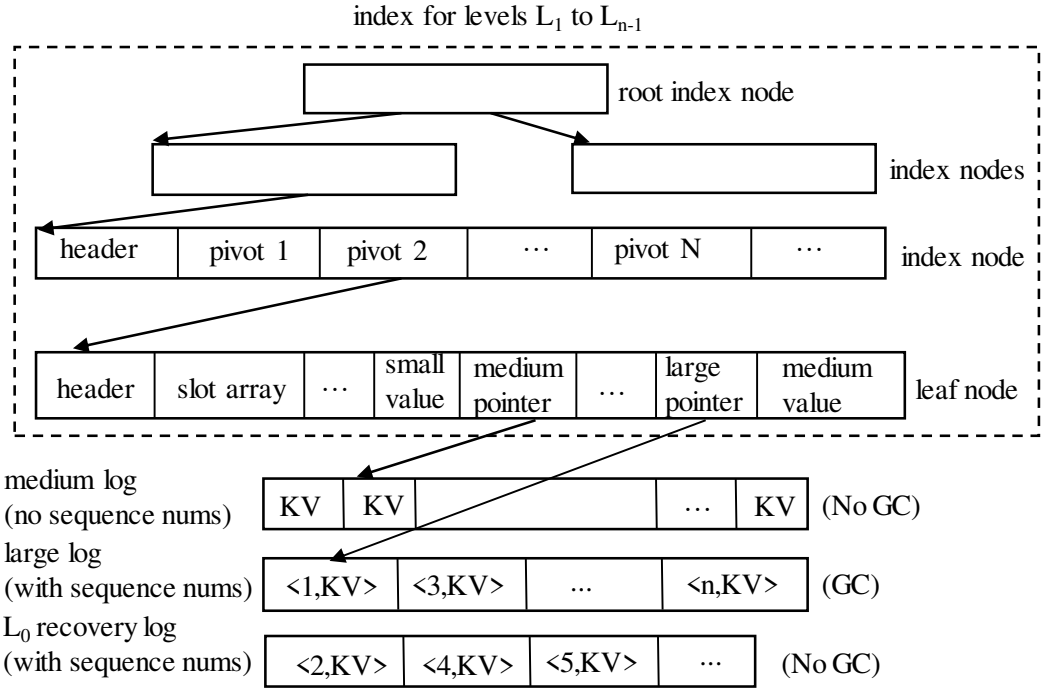


Figure 2.4: Overview of index and log design for levels L_1 to L_{n-1} in Parallax. Levels L_0 and L_n always store medium values in-place.

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.

2.4 Parallax

Parallax [41] 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 [7, 6, 35, 29] and the region as a whole uses three logs: L_0 recovery log, medium log, and large log. Figure 2.4 shows an overview of Parallax. All KV logical structures consist of fixed size segments on the device (Figure 2.5). Currently, Parallax uses 2 MB segments.

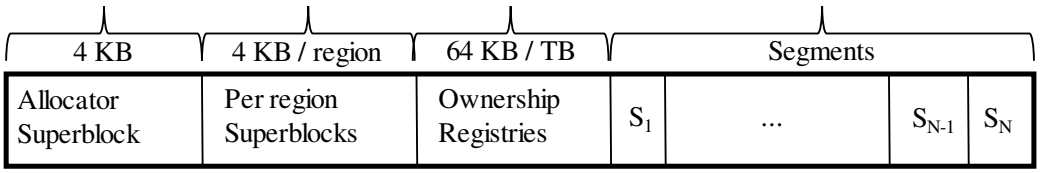


Figure 2.5: Device layout used by Parallax and its allocator. A segment may belong to the large, medium, L_0 recovery, or region allocation log, or to a level index.

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.

When a KV pair is placed in a log, Parallax stores in the index leaf the key and a pointer to the KV pair. To store a variable number of KV pairs with variable size in leaves, we use two dynamically growing segments, the slot array and the data segment [11, 24]. The slot array is a small array with 4 B cells that grows from left to right. We reserve the three high-order bits of each cell to store the KV category. The rest of the bits contain an offset inside the leaf with the location of the actual data. The data segment is an append only buffer that contains either pointers to the log or the in-place KV pairs themselves, and grows from right to left. When the slot array and the data segment meet, the leaf becomes full. Update operations append the new value to the data segment and update the slot array.

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. *Delete* operations mark keys with a tombstone and defer the delete operation similar to RocksDB [17], freeing up space at the next compaction. *Update* operations are similar to a combined *insert* and *delete*. Finally, *insert* operations write each KV pair in L_0, similar to all LSM KV stores. At each insert operation, the KV pair is categorized as small, medium, or large derived from its KV size. Based on the category, Parallax uses its hybrid strategy for placement, as we discuss next.

2.4.1 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.

2.4.2 Handling Large KV Pairs via GC

For large KV pairs, Parallax always performs KV separation [16, 26, 29, 11]: 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 pre-configured 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.

2.4.3 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 [28] 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 KV pairs 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.

2.4.4 Recovery

Recovery in Parallax relies on value logs. In addition to the medium and large logs, for recovery purposes, Parallax uses the L_0 recovery log as a temporary log for small KV pairs (Figure 2.4). Conceptually, Parallax places each KV pair in the respective log. After a failure, Parallax can recover all KV pairs by replaying the logs.

Parallax always places large KV pairs directly in the large log upon insert, where they remain for their lifetime. We place small KV pairs in the L_0 recovery log and write them in-place in L_0. When Parallax compacts L_0 in L_1, it can fully reclaim the L_0 recovery log. Parallax could place medium KV pairs directly in the medium log. However, this would require more I/O traffic to sort medium long runs. For this reason, it initially inserts medium KV pairs in-place in L_0. When

compacting L_0, it also sorts medium KV pairs and places them in the medium log as a sorted run. To ensure medium KV pairs are recoverable while they live in L_0, Parallax also writes them to the L_0 recovery log. Medium KV pairs do not affect reclaiming the L_0 recovery log, which still occurs after L_0 compactions.

Parallax needs to maintain-order across the large and L_0 recovery logs (but not the medium log) and handle cases with KV pairs that change the category after an update. For this reason, it uses region 8-byte Log Sequence Numbers (LSNs). Each appended KV pair has an LSN, which we increment atomically before appending. During region recovery, Parallax replays each log entry of the two logs with the correct order, as indicated by their LSN number.

Finally, like other KV stores [21], Parallax acknowledge writes as soon as they are written in-memory. KV pairs are recoverable after a group commit operation that flushes the log tail asynchronously to the device, currently in chunks of 256 KB for I/O purposes. Therefore, Parallax can recover to a previous consistent point, which may not include the last (acknowledged) write(s). Most KV stores (and Parallax) can be configured to acknowledge writes after they are written to the device or to perform more frequent flush operations, but these are not commonly used as they increase acknowledgment delay or I/O overhead.

2.4.5 Direct and Memory-Mapped I/O

Parallax carefully utilizes two I/O paths, direct I/O via system calls and memory-mapped I/O. Parallax performs device I/O on the following occasions:

1. During compactions (Section 2.1), Parallax reads the source and destination level and writes the new level by issuing large, full-segment (2MB).

2. Large reads of full log segments (2 MB) during garbage collection.

3. Large writes of log chunks (256 KB) at log flushes.

4. Small (B-KB) reads and writes for region metadata.

5. Small, random reads of the index and leaf nodes (B KB) and log pages (B-KB) during get, scan operations.

Parallax uses direct I/O for all cases except 1) the GC operation, 2) the compaction of the medium-log to the last level, and 3) the get and scan operations. Parallax uses direct I/O because (a) it is more efficient for large I/Os, (b) it allows explicit control of the allocator state in the device for recovery purposes, and (c) it avoids pollution of the read cache with data that need not reside in-memory. Although memory-mapped I/O is suitable for get and scan operations as it eliminates system calls and data copies, its usage in the GC and the compaction of the medium-log operations can result in significant system overheads.

2.5 ZooKeeper

ZooKeeper is a distributed coordination service that plays a crucial role in building reliable and highly available distributed systems. At its core, ZooKeeper is designed to solve the challenges associated with coordinating distributed systems. In a distributed environment, it is essential to ensure that multiple nodes or processes work together in a consistent and coordinated manner. This coordination includes tasks such as leader election, configuration management, distributed locking, and synchronization.

One of the key features of ZooKeeper is its simplicity. It offers a file system-like hierarchical namespace, where each node in the hierarchy is called a "znode." These znodes can store small amounts of data, typically a few kilobytes, and are accessed using unique paths. ZooKeeper provides APIs that allow applications to create, read, update, and delete znodes, as well as watch for changes on specific znodes.

Except from data storage, ZooKeeper provides several powerful primitives to facilitate coordination among distributed processes. It offers sequential and ephemeral znodes, enabling ordered access and temporary presence, respectively. Additionally, ZooKeeper supports watches, allowing applications to receive notifications when the state of a znode changes. This event-driven model enables efficient and reactive coordination between distributed components.

ZooKeeper's architecture follows a server-client model. A set of ZooKeeper servers form a quorum to provide high availability and fault tolerance. Clients connect to any server in the ensemble, which then handles requests and coordinates with other servers to ensure data consistency. ZooKeeper employs a consensus protocol to maintain consistency across servers, ensuring that all updates are applied in the same order and clients observe a linear view of the system's state.

Chapter 3

Implementation

3.1 *Anthus*

3.1.1 Overview

Anthus, partitions the key-value space into non-overlapping key ranges, named *regions*. *Anthus* 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. *Anthus* consists of three main entities:

1. *Anthus 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 [20] stores information about the metadata of each region. Zookeeper is not in the common path of client operations in *Anthus* since changes in regions are triggered either by membership changes due to failures or load balancing operations. Furthermore, the *master* of *Anthus* uses the membership service of Zookeeper to detect changes in server status (join or fail) and trigger appropriate action. *Anthus* can make use of similar systems [1, 31, 40] that provide strongly consistent metadata replication and notifications 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

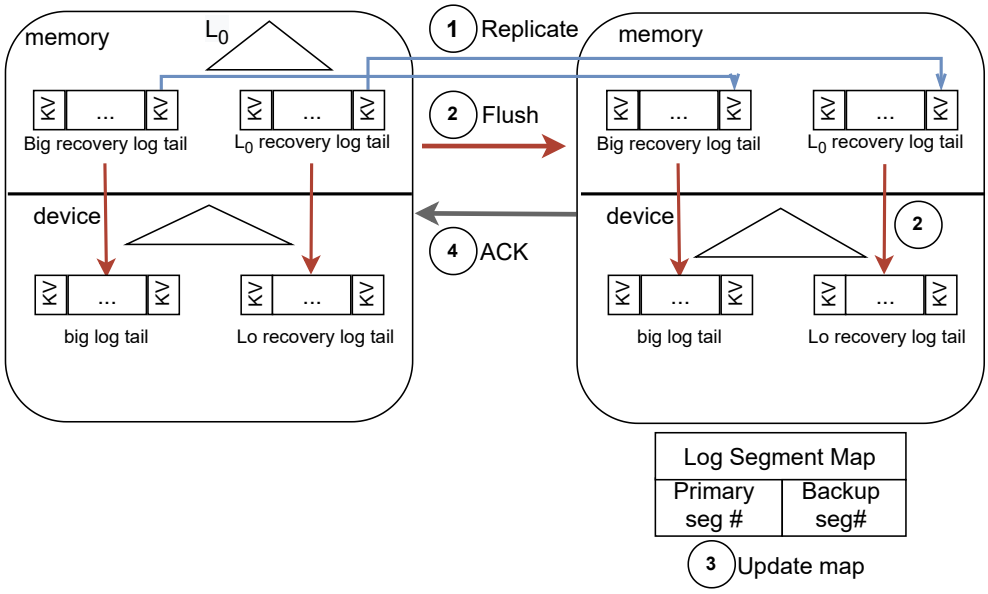


Figure 3.1: The three scenarios that can happen when a primary crashes while writing a KV pair to its backup nodes.

a system reconfiguration, the *region server* instructs the client to update its region map.

3.1.2 Primary-Backup Value Log Replication

3.1.2.1 Replication Mechanism in *Anthus* ①

In *Anthus*, similar to *Tebis*, the log replication occurs without involving the CPU of the backup servers, thanks to the exclusive use of RDMA-write operations for communication. Whenever the primary node receives updates, inserts, or deletes from clients, it replicates each operation to its set of backup servers. Both primary and backup nodes in *Anthus* maintain in-memory buffers, which temporarily store the KV pairs that require replication.

Upon applying an operation at the primary, *Tebis* performs out-of-order replication. Each worker thread appends (via an RDMA-write operation) the corresponding KV pair to the remote memory of backups concurrently, and atomically alters the metadata related to buffer fullness. However, we believe out-of-order replication should be replaced with in-order replication based on two observations. Firstly, as shown in Section 4.3.7, the performance benefits of out-of-order replication are minimal. Secondly, out-of-order replication cannot provide last-write (group commit) guarantees in the event of a primary crash. When relying on out-of-order replication, *Tebis* cannot determine which KV pairs are live in the remote memory buffers of its backup servers, making it unable to provide reliable last-write guarantees by design.

This work introduces in-order replication in *Anthus*. To achieve this, when a primary node detects a new operation (update, insert or delete), it inserts the KV pair in Parallax, categorizing 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 the KV pair to the corresponding (big or L_0 recovery) RDMA buffer of each backup at the corresponding offset, ensuring that the replication requests occur with increasing order to the LSNs.

3.1.2.2 Replication Acknowledgement ②

When using *Anthus*, it is important to determine when to acknowledge client operations. In this work, we focus on two possible scenarios. The primary can choose to (1) respond to client requests after initiating asynchronous RDMA write operations to all of its backups. Alternatively, it can (2) respond to clients once it receives completion signals from the remote NICs, indicating that the KV pair has been successfully stored in the backups' memory.

Each solution carries specific implications, particularly in recovery scenarios. In the first case, if a primary crash occurs, the system can recover to a previous consistent point, but it may not include the last acknowledged write(s). On the other hand, in the second case, the system can recover from the last acknowledged write, as the primary awaits acknowledgments from remote NICs before responding to clients.

In our work, we have designed *Anthus* to support both (last-flush recovery) and (b) last-write recovery from the memory of the backups. In the first case, the primary responds to client requests only after initiating asynchronous RDMA write operations to all its backups, thus enabling Tebis to recover from the last persistent flush request. In the second case, we have developed a protocol for last-write recovery capable of handling up to N-1 failures in a replica group. Section 3.2 provides a detailed description of this protocol.

3.1.2.3 A Working Paradigm: Value Log Replication

Here, we combine the mechanisms above and describe their operation and interaction. For this purpose, we illustrate an operation paradigm in Figure 3.1. When *Anthus* receives updates, inserts, and deletes from clients, the primary replicates each operation to its *backup* servers in four steps. As discussed in ①, the log replication occurs without involving the CPU of the backup servers. *Anthus* inserts the KV pair in Parallax, categorizing the KV pair according to its KV size. Then, it appends in an increasing LSN order, the KV pair to the corresponding (big or L_0 recovery) RDMA buffer of each *backup* at the corresponding offset(step 1 in Figure 3.1).

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 3.1). 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 3.1).

As discussed in ②, *Anthus* can support either a) last-write recovery or b) last-flush recovery. In the first case, the primary responds to clients after it has received completion from the remote NICs, while in the second case, the primary responds after issuing the RDMA-write operation to all of its backups.

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 3.3.1. 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 the 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 3.1). 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, *Anthus* 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*.

3.1.2.4 Medium Value Log Replication

Parallax uses a hybrid KV placement strategy that involves three logs: the L_0 recovery log, the big log, and the medium log. Initially, medium KV pairs are inserted in-place into L_0. During the compaction of L_0, medium KV pairs are placed in the medium log as a sorted run (Section 2.4). This poses the problem that the value log replication of *Anthus* alone is insufficient, as it does not replicate the medium log. *Anthus* achieves to replicate its medium log as follows:

During an L_0 to L_1 compaction, the primary instructs the backup nodes to create and register a dedicated memory region explicitly for receiving the context of the medium log segments. Once a compaction operation fills a medium log segment, the primary performs the following steps before flushing the segment to the device:

1. The primary writes the context of the segment to the remote memory region of its backups.

2. It sends a flush-segment message to all its backups, including the segment's number as a piggybacked parameter.

Subsequently, the primary node flushes the segment into the medium log and waits for the backup nodes to complete their tasks.

Upon receiving the flush-segment message, a backup node a) persists the contents of its memory region into its local medium log and b) updates its log map mappings, similar to the flush tail procedure. Then, it acknowledges the operation to the primary. It is important to note that RDMA's Reliable connections deliver messages in a FIFO order. This FIFO guarantee ensures that when a backup flushes its memory region to the medium log, all the related data are present, preventing any potential data inconsistencies or corruption.

3.2 Last-Write Recovery from the Memory of the Backups

State-of-the-art persistent KV stores [17] 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 mainly adopt last-flush recovery for performance reasons. This choice aligns with the conventional wisdom of distributed system design that the network is a severe performance bottleneck. Replication protocols that incorporate last-flush semantics aim to optimize network utilization by generating larger messages. As we show in Section 4.3.3 aggressive network access, especially for transmitting small KV pairs which dominate modern workloads, can result in substantial performance overhead on the CPU due to the message processing overhead imposed by the operating system.

In *Anthus*, RDMA enables the design of a last-write recovery protocol with small overhead up to 4% in throughput compared to last-flush recovery, as we experimentally show in (Section 4.3.3). *Anthus* can support either last-flush or last-write recovery. In *Anthus*, last-write recovery can tolerate up to N-1 failures, where N denotes the number of replicas in the replica group. In particular, *Anthus* recover writes from the memory of backups in case of a primary failure. 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].

The main challenge in providing last-write recovery semantics 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 we need to enable backups to verify the integrity of KV pairs stored in their RDMA buffers and recover them successfully.

Our last-write recovery protocol relies on two observations. Because *Anthus*

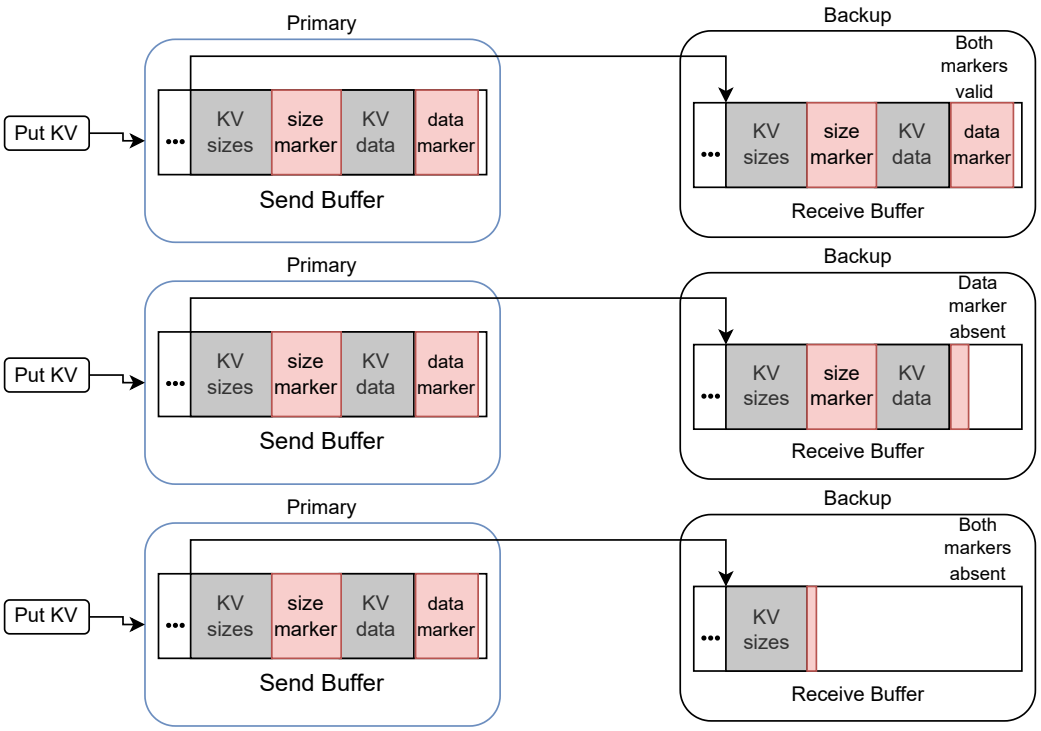


Figure 3.2: The three possible outcomes of the replication process of a KV pair.

uses RDMA's reliable queue pair connections, *Anthus* 1) issues RDMA write operations in each RDMA buffer sequentially, in increasing address order and 2) apply remote memory writes in FIFO order. Based on these two observations, *Anthus* 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 and the actual variable size KV data. *Anthus* 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. Eventually, the in-memory KV pairs will be flushed or inserted into the storage engine of *Anthus*. In order for the storage engine to work properly, we enabled Parallax to use the new KV pair layout for all its internal operations.

Figure 3.2 illustrates how backup nodes handle corrupted data in the event of a primary failure. In the first scenario, a valid KV pair transmission is depicted with successfully replicated markers, allowing backup nodes to recover all data up to that KV pair safely.

In the second scenario, the backup received the KV pair but not the data marker. In this case, the integrity check fails. The backup can recover all the previous KV pairs but ignores the latest. Notably, the Put procedure of *Anthus*

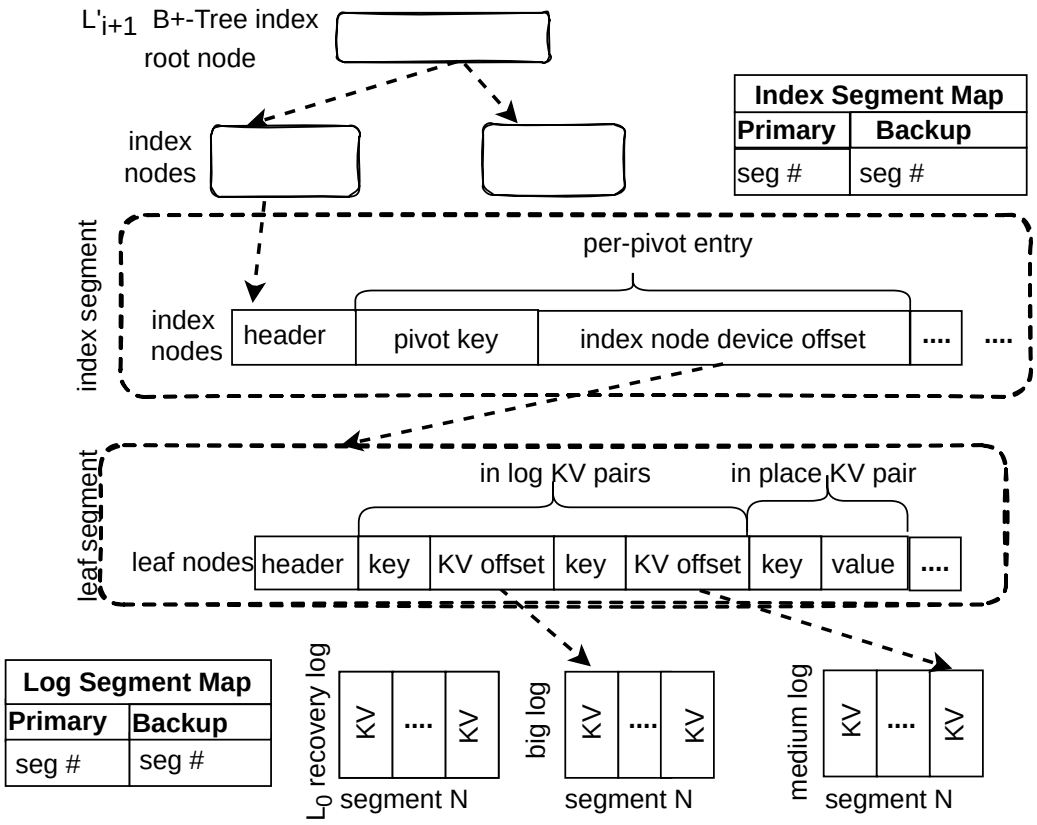


Figure 3.3: Index Shipping in *Anthus*.

has yet to acknowledge the insertion of this specific KV pair, so ignoring it is safe.

The third scenario is similar to the second, where backups ignore the currently processed KV pair. Although the marker for the sizes of the key-value pair may or may not be successfully transmitted, there is no guarantee regarding the transmission of the actual KV pair data. Finally, to ensure the correctness of the markers, backup zeroes the contents of its RDMA buffers after a flush buffer request from the primary.

3.3 Index Shipping

3.3.1 Index Shipping and Rewrite at the Backup

Anthus 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 *Anthus* is to rewrite the index at the *backups* to contain valid device addresses since servers do not share a global storage name space [2, 19, 8].

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 $2\times$ for one replica per region or $3\times$ for two replicas. This is important because a server may host hundreds of regions, especially with increasing device capacities, for concurrency and load balancing purposes. As a results, assuming and L_0 size of 64 MB, the reduction of the memory budget for the Send-Index method is in the order of tens of GB.

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 the 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 *Anthus*, the main device structures are the logs (L_0 recovery, medium, big) and the B+-tree indexes of the levels. *Anthus* 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, *Anthus* replaces the high-order bits of the *primary* segment with the new segment number in the *backup* device.

The index of a region (Figure 3.3) consists of leaf and index nodes. Leaf nodes (bottom in Figure 3.3) contain in-log and in-place KV pairs. in-log KV pairs belong to the big or medium category (which *Anthus* has not transferred in-place). In contrast, in-place KV pairs are either small or medium that *Anthus* 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 3.3).

Backups keep track of two mappings for segments: The log map and the index map. The log map is updated during the flush operation (Section 3.1.2.3) and 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). Each source device offset replaces the high-order bits with the local segment from the segment map. During this procedure, the primary proceeds with its compaction process. The primary stalls only when producing a new index segment, but its backups have not finished rewriting the previous one. This collision is not in the common path since the index rewriting is an in-memory operation. When the index rewriting of a segment finishes, *backup* flushes the segment to the persistent storage of Parallax.

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 full compactions, such as RocksDB or use KV separation [26, 16, 25]. In these systems, SSTs may contain device offsets of the *primary* to its value log or an internal SST index that needs rewriting similar to *Anthus*.

3.3.2 Send-Index Interface

The Send-Index functionality of *Anthus* revolves around a Send-Index API, which enables seamless integration with other storage engines. Our primary objective was to ensure that engineers can easily utilize *Anthus* by designing it agnostic to the underlying storage engines. To achieve this, we provide an asynchronous interface describing the send index procedure through callback functions. These callbacks specifically handle a storage engine's compaction process and log management.

This approach allows third-party engineers to effortlessly notify *Anthus* and manage events within their storage engine. As a result, the required modifications to the storage engine's code are minimal, while the core functionality can be implemented within *Anthus*. Notably, the Send-Index API can be extended to accommodate the specific needs of a storage engine. The following listings present pseudo code about Parallax's implementation of the Send-Index API.

```
struct send_index_callback_funcs {
    void (*compaction_started_cb)(void *context, ...);
    void (*compaction_ended_cb)(void *context, ...);
    // In the event of a compaction with an empty destination level,
    // Parallax swap the levels by updating in-memory metadata
    void (*swap_levels_cb)(void *context, ...);
    // A new index segment is produced, send it to the backups
    void (*comp_write_cursor_flush_segment_cb)(void *context, ...);
    // Checks if backups have rewrited the previous index segment
    // and replied
    void (*comp_write_cursor_got_flush_replies_cb)(void *context, ...);
    };
```

3.3.3 Build-Index Procedure in *Anthus*

In *Anthus*, the Build-Index method serves as the baseline metric for measuring the performance gains of Index Shipping (Section 3.3.1). The primary objective of the Build-Index method is to minimize network traffic by replicating only the user data while leaving the *backup* nodes to handle the resource-intensive compactions necessary for reorganizing their data. We implement the Build-Index method as follows:

Each backup node maintains one RDMA buffer in its memory. When *Anthus* receives an insert, update, or delete operation, the primary first applies the operation locally. After, it replicates the corresponding KV pair to the RDMA buffer of each backup node and updates the metadata associated with the RDMA buffer's fullness. Once the RDMA buffer reaches its capacity, the primary sends a blocking flush tail message to all the backup servers.

Each backup node has a consumer thread responsible for inserting new KV pairs into its storage engine, similar to the primary, and a producer thread that posts new KV pairs for insertion. When a new flush tail message is detected, the producer thread copies the contents of the RDMA buffer to a private buffer in a buffer pool owned by the consumer thread. Then, the producer thread acknowledges the operation to the primary. The consumer thread scans its buffer pool, detects new KV pairs, and inserts them into the backup's storage engine.

This approach allows us to accurately measure the performance of the Build-Index method since the primary does not wait for the backup nodes to apply the changes in their RDMA buffers. Waiting for backup nodes to apply changes could result in significant stalls lasting tens to hundreds of seconds due to compactions. Therefore, our approach ensures a more efficient evaluation of the Build-Index method, unaffected by potential compaction delays.

3.4 Communication Protocol

3.4.1 RDMA Buffer Management

Anthus performs a client-server communication approach using one-sided RDMA write operations [22] to minimize network interrupts and reduce CPU overhead on the server [22, 21]. Once the connection is established, the server and client allocate a pair of buffers with configurable sizes (currently set to 8 MB) and zero their contents. These buffers are freed by the server when a client disconnects or encounters a failure. To ensure the validity of inactive queue pairs, a dedicated thread monitors them through a heartbeat procedure. This thread sends an empty RDMA write message to the server and waits for its completion. If the RDMA write is successfully completed, RDMA's Reliable Connection guarantee the server's online status. Currently, the thread continuously polls the RDMA buffer, but an alternative sleep-wake up approach could also be implemented.

To avoid synchronization among workers on the server, clients manage both

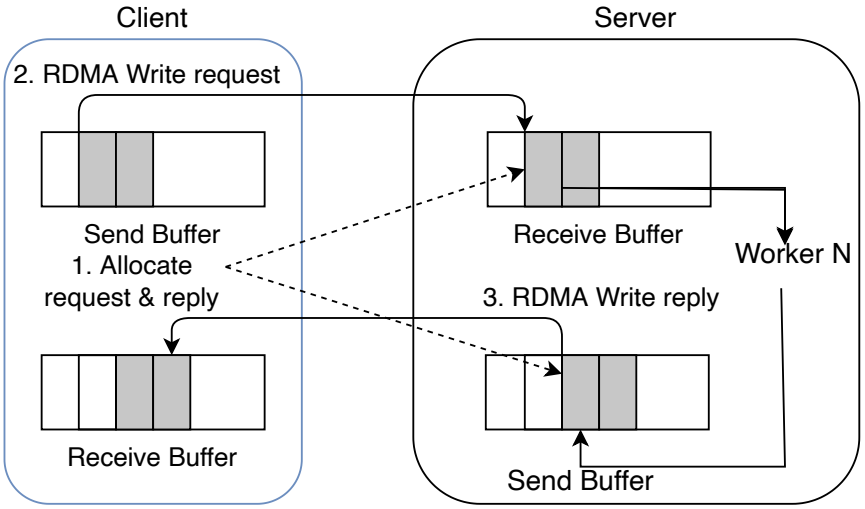


Figure 3.4: Allocation and request-reply flow of Tebis RDMA Write-based communication protocol.

request and reply buffers. Before each KV operation, clients consult their local copy of the region map to determine the primary region server to which they should send their request. Then, the clients allocate a pair of messages for each KV operation: one for their request and one for the server's reply (refer to step 1 in Figure 3.4). The request header includes the buffer offset where the server can write its reply.

Workers on the server asynchronously complete requests and respond to clients out-of-order. To track the arrival of server responses, clients use a dedicated thread called the *reply checker*. This thread iterates through the communication buffers where the server posts its replies and acknowledges whether an operation was successful or not to the the clients. Furthermore, after the acknowledgement of a server reply, the *reply checker* thread is responsible for freeing the memory allocated for both the request and the reply.

For put requests, the reply size is fixed, allowing the client to allocate the required memory before performing the operation. Conversely, forget and scan requests have variable and unknown reply sizes. If the value size exceeds the buffer size of the reply, the server sends a partial reply and informs the client to increase its allocation size for reply buffers to avoid similar scenarios in subsequent requests. The client can then retrieve the remaining portion of the value from the offset provided by the server. Consequently, the penalty in this case is a round trip with a minimal impact on the overall latency.

Scaling the RDMA protocol of *Anthus* 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, *Anthus* could divide this memory elastically between more and less active clients. Also, other approaches

such as LITE [38] 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 *Anthus* and distinguishing hot from cold clients to reduce the polling frequency. We leave these as extensions for future work.

3.4.2 An Overview on Segmented vs. Non-Segmented Communication

After discussing the RDMA buffer management for client-server communication, it is crucial to consider the representation of messages inside these RDMA communication buffers and the trade-offs associated with different approaches. In this work, we focus on two approaches: segmented communication and non-segmented communication. Possible implementations of these in the context of *Anthus* could be the following:

Segmented communication involves dividing messages into equal-sized chunks. The first chunk, typically containing the message header with relevant metadata, is followed by a variable number of chunks containing the actual KV data and additional control information. Each chunk includes a metadata field (e.g., a receive field) indicating whether it has been received. When a message is allocated and filled with data, the client initializes the receive fields of the first and last chunks to a magic number. The server waits for the arrival of the receive fields of the first and last chunks to detect new messages. To free a message, the server simply zeros the receive fields of all the chunks associated with that message. In this way, there are no conflicts between freed and future message chunks.

On the other hand, non-segmented communication allocates space for the entire message, consisting of a message header with metadata followed by the actual data. The client fills the receive field of the message header and includes a receiving field at the end of the variable-sized payload. The server detects new messages by waiting for the arrival of the message header and spinning on the receive field of the payload. To prevent conflicts with future messages, the server must zero all the memory associated with that message.

3.4.2.1 Trade-Offs Between Segmented and Non-Segmented Communication

Choosing between segmented and non-segmented communication involves a trade-off between network traffic and CPU usage. Segmented communication allocates more memory than necessary for storing the KV pairs, which increases network traffic. On the other hand, non-segmented communication requires the complete freeing of message contents on the server side to properly detect new messages, increasing the CPU usage. In the case of *Anthus*, we have chosen the segmented communication approach due to its lower CPU usage on the backups.

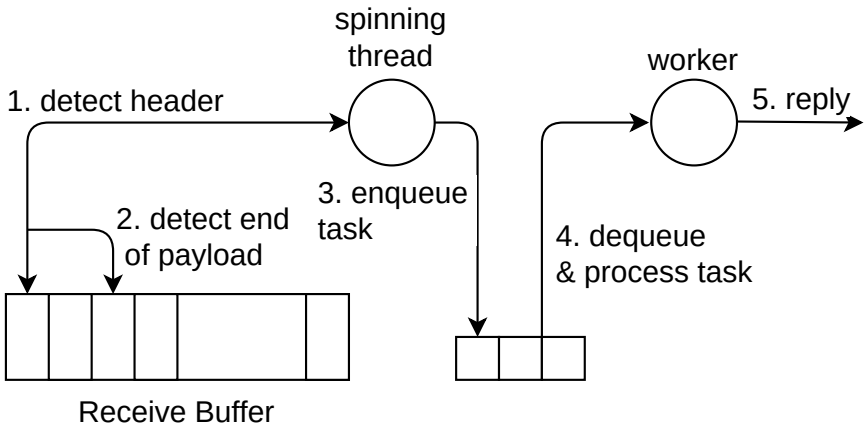


Figure 3.5: Message detection and task processing pipeline in *Anthus*. For simplicity, we only draw one circular buffer and a single worker.

3.4.3 Segmented Communication in *Anthus*

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 3.5. 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 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 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 NO-OP 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 NO-OP reply. When the clients detect the NO-OP reply, it proceeds as in case (a).

3.4.4 Task Scheduling

To limit the max number of threads, *Anthus* use a configurable size of workers. Each worker has a private *task queue* to avoid the CPU overhead associated with shared accesses contention. In this queue, the spinning thread places new tasks, as shown in (Figure 3.5). Workers poll their queue to retrieve a new request and sleep if the spinning thread does not assign them a new task within a period (currently 100 μs). The primary goal of *Anthus*'s task scheduling policy is to limit the number of wake-up operations, which include user/kernel crossings. 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 all running workers exceed the task queue limit, it wakes a sleeping worker and en-queues this task to their task queue.

3.4.5 RDMA-Write Detection in *Anthus*

Anthus 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: targeted polling (busy wait), blind polling, and interrupts. However, to minimize CPU costs, *Anthus* does not utilize interrupts.

Targeted polling involves the sender instructing the receiver's network card to generate a completion event when a new message arrives. This completion event entry (CQE) contains relevant information such as the queue pair that received the message and the number of bytes written. To check for new CQE entries, the receiver actively polls its completion queue (CQ), which is shared among its queue pairs. By continuously monitoring the CQ, the receiver can detect the arrival of new messages.

In contrast, blind polling does not rely on completion events at the receiver's end. Instead, the receiver independently checks the memory area of the communication buffer associated with each queue pair to determine if a new message has arrived. This method eliminates the need for CQE events on top of the PCI at the receiver's end, thereby reducing CPU costs.

Choosing between blind polling and targeted polling is a trade-off between message cost and CPU. *Anthus* employs blind polling to detect incoming RDMA messages because it provides the fastest messages.

3.5 Failure Detection and Recovery

Anthus, similar to *Tebis*, utilizes the ephemeral nodes mechanism of Zookeeper for efficient failure detection. By leveraging this feature, Zookeeper automatically

removes an ephemeral node when a node stops responding to heartbeats. The failure detection process is critical in ensuring system reliability and availability.

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*.

When a *primary* failure occurs, the *master* promotes the next *backup* in line with the *primary* role and updates the region map accordingly. To achieve recovery, *Anthus* initiates the closure of all DBs associated with the failed region, ensuring any pending operations are discarded or flushed to the storage engine. Subsequently, the new *primary* clears its state and communicates with its backups to clear its states. Finally, the new *primary* establishes new connections with its backups. It is important to note that the new *primary* possesses a complete KV log and index for levels L_i, where $i \geq 1$. By replaying the last few segments of its value log, the new *primary* constructs L_0 in memory before becoming operational to serve client requests which incur some downtime 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.

3.6 Changes in the I/O Path of Parallax

As discussed in Section 2.4 Parallax carefully utilizes two I/O paths, direct I/O via system calls and memory-mapped I/O. It uses direct I/O for all cases except a) the GC operation, b) the compaction of the medium-log to the last level, and c) the get and scan operations. Although memory-mapped I/O is suitable for get and scan operations as it eliminates system calls and data copies, its usage in the GC, and the compaction of the medium-log operations can result in significant system overheads.

During the compaction of the medium log to the last level(s), Parallax moves medium in-log KV pairs to in-place, similar to small KV pairs. This process involves fetching the actual KV data from the medium log, performing merge sort, and writing them in-place to the new level. When using memory-mapped I/O, fetching data from the medium log can trigger a page fault, leading to small random I/O if there is no valid memory mapping. When memory pressure is high, the system's

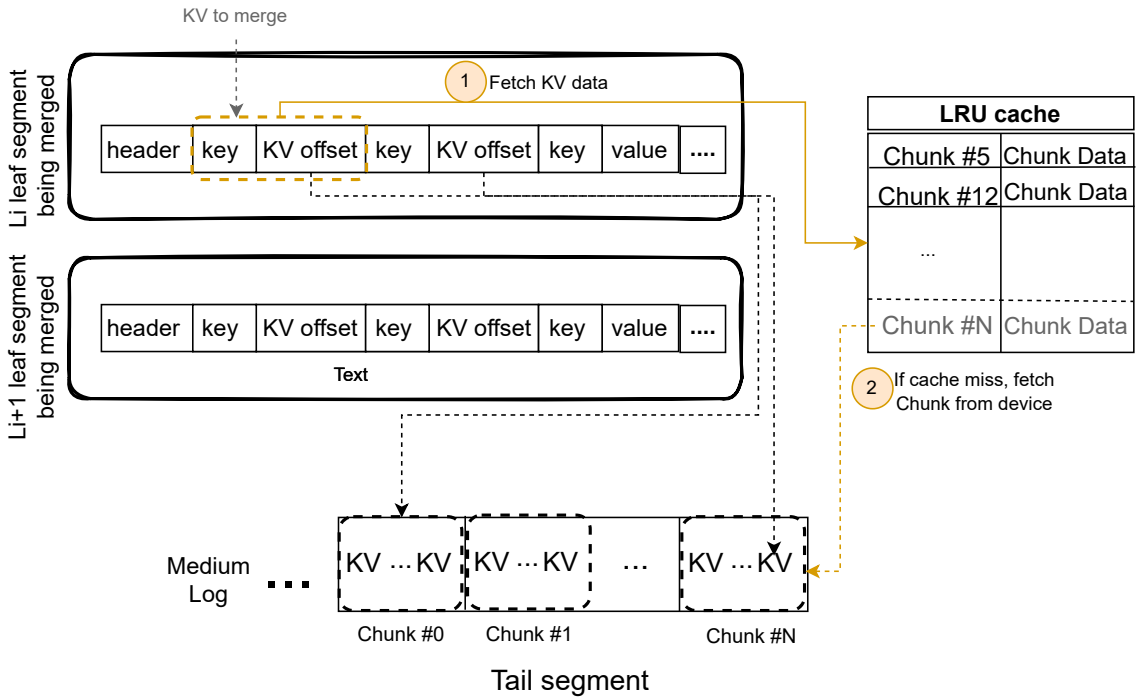


Figure 3.6: Parallax's LRU cache

page cache fills up, forcing the operating system to evict pages from memory, resulting in significant system overheads. Similarly, the garbage collector parses a set of segments to determine the liveliness of KV pairs. Since the segments are parsed sequentially, this introduces a page fault for every 4K segment chunk.

Furthermore, these operations may overlap with the system serving client-read requests. In such cases, page faults pollute the read cache, significantly impacting the performance of read operations. Due to these considerations, direct I/O should be preferred over memory-mapped I/O for these operations. In this work, we tackle this problem by changing the I/O path of the GC and the compaction of the medium log to the last level from memory-mapped to direct I/O as follows:

During the compaction process, which involves transferring in-log medium KV pairs to in-place, we introduce an in-memory LRU cache as shown in Figure 3.6. This cache is designed to store recently accessed segment chunks and is initialized at the start of the compaction process. The LRU cache consists of a HashTable and a linked list. The linked list is sorted based on node accesses, with the most recently accessed node at the tail and the oldest accessed node at the head. HashTable entries are represented as pairs, denoted as <chunk_offt, node pointer>. The chunk_offt value serves as a unique identifier, indicating the starting offset of the chunk on the disk. The node pointer field refers to the location of the actual chunk data within the linked list.

During compaction, before accessing the device to retrieve the contents of an in-log medium KV pair, we initiate a search in the LRU cache using the chunk offset of the KV. If there is a cache hit, we retrieve the contents directly from the cache. In the case of a cache miss, we fetch the segment chunk from the device where the KV pair is located. As the cache operates in an LRU manner, the fetched chunk is inserted into the cache, replacing the oldest entry in the case of insufficient space.

Regarding garbage collection, when fetching a segment for GC, we utilize direct I/O to retrieve the entire segment. This shift eliminates the use of memory-mapped I/O in garbage collection.

Chapter 4

Evaluation

4.1 Evaluation Methodology

Our experimental setup consists of three identical region 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 region 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 4.2.

In our experiments, we run the YCSB benchmark [13] and its workloads Load A and Run A to Run D. Table 4.1 summarizes the operations run during each workload. We run Tebis with 32 regions equally distributed across all region servers. Furthermore, each server has two spinning threads and eight worker threads in all experiments. region servers use the remaining cores to perform compactions.

In all experiments, we use two separate region 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 [10], as shown in Table 4.2. 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 [32] 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

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 4.1: 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

Table 4.2: KV size distributions.

KV pairs. We use a small-dominated (SD) KV size distribution as proposed by Facebook [10], as well as a medium dominated (*MD*) and a large dominated (*LD*) workload. We summarize these KV distributions in Table 4.2.

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 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:

$$efficiency = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}} \; cycles/op, \tag{4.1}$$

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 excess device traffic generated due to compactions (for primary and backup regions) by Tebis, and we define it as:

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

where *device_traffic* is the total number of bytes read from or written to the storage device and *dataset_size* is the total size of all key-value requests issued during the experiment.

We measure network amplification as traffic to all region 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 64 B to reduce CPU usage for detecting variable size messages in the region 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.

4.2 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 *Anthus* 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?

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?

6. What are the CPU efficiency gains of RDMA vs TCP/IP?

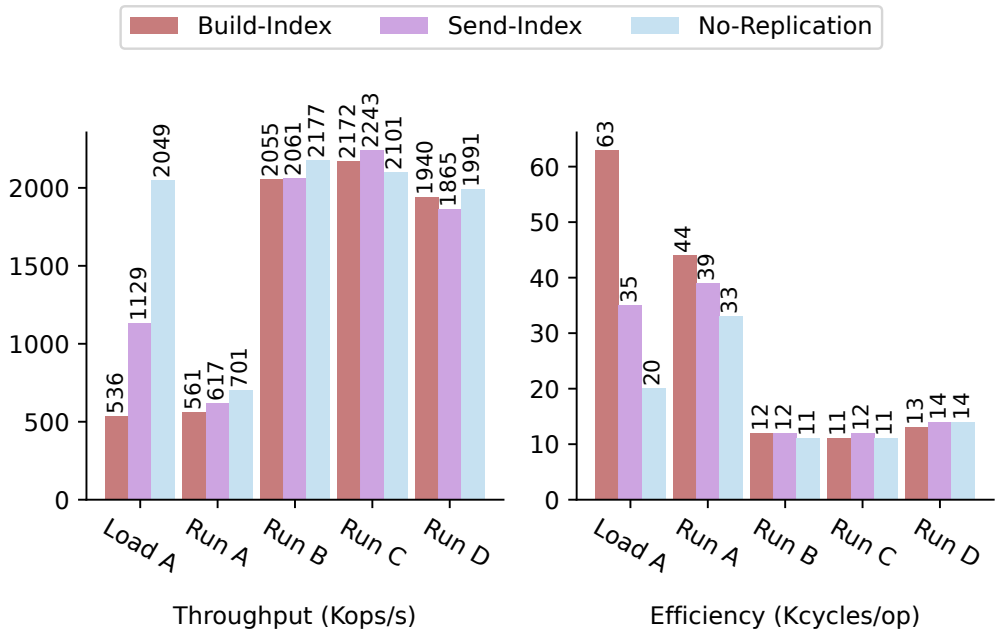


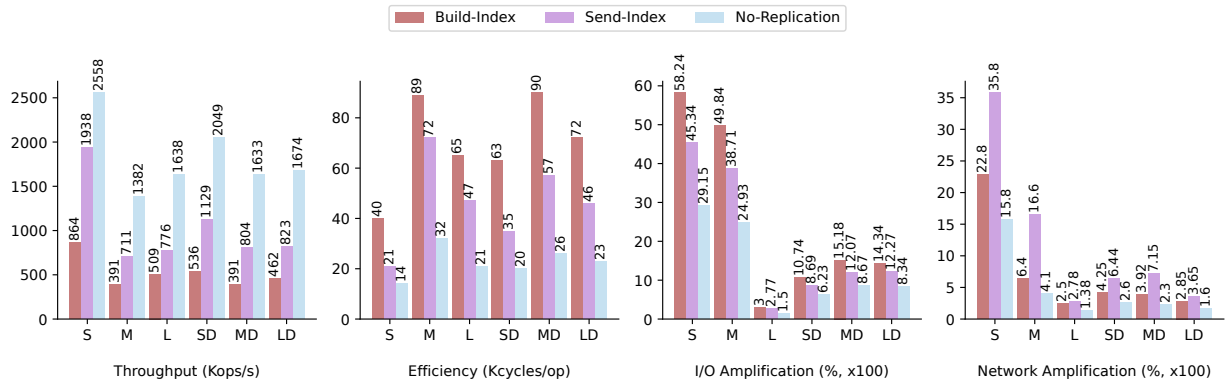
Figure 4.1: Performance and efficiency of *Anthus* for workloads Load A, Run A – Run D with the SD KV size distribution.

4.3 Experimental evaluation

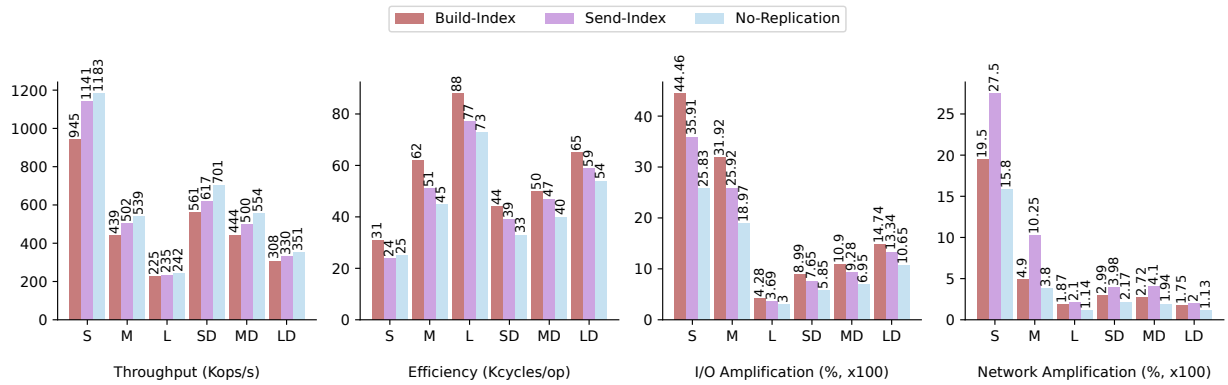
4.3.1 *Anthus* Performance and Efficiency

In Figure 4.1, we evaluate *Anthus* for two-way replication using YCSB workloads Load A and Run A to Run D for the SD KV distribution [10]. 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 8. Figure 4.2 shows that compared to Build-Index and for all KV size distributions, Send-Index increases throughput by $1.04 - 2.24\times$, CPU efficiency by $1.06 - 1.9\times$, and reduces I/O amplification by $1.08 - 1.28\times$. 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 *Anthus* since it uses available network throughput to reduce device I/O traffic and CPU usage.

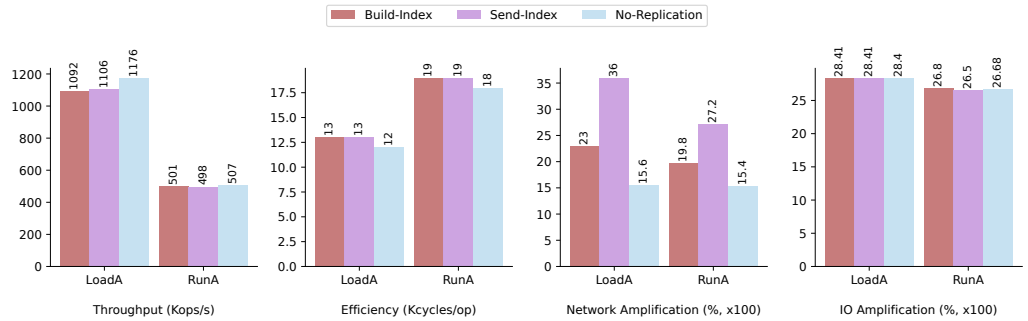


(a) Load A

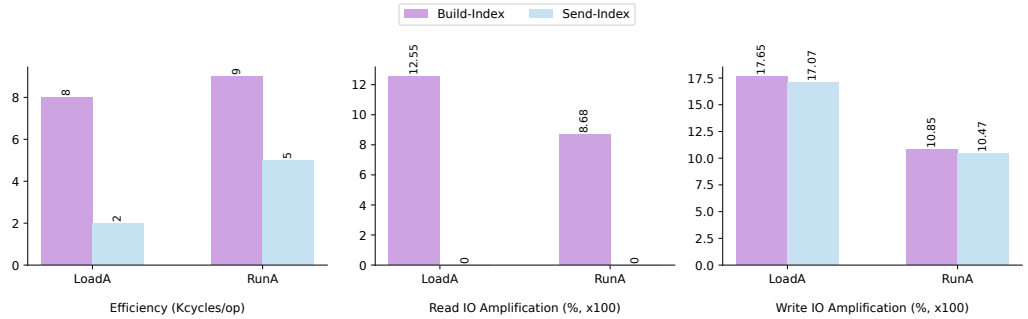


(b) Run A

Figure 4.2: 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

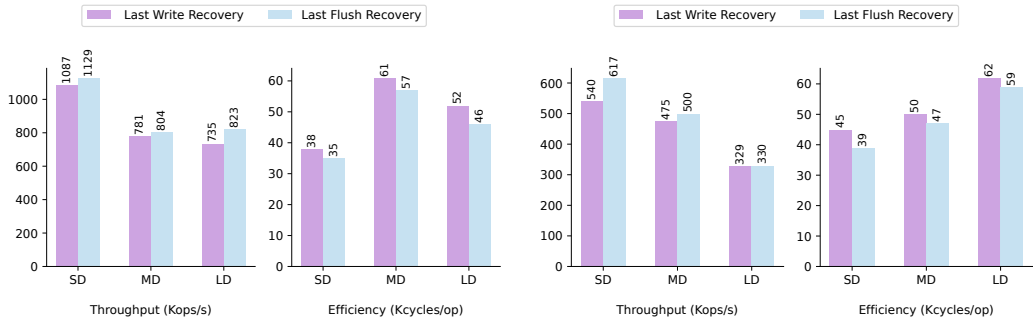


(b) Backup

Figure 4.3: Overheads breakdown for the primary and backup role

4.3.2 Overhead Breakdown

To assess the overheads of each server role of *Anthus*, 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 4.3, 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 $1.8-4\times$. 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 compaction's merge sort operations. 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.



Load A Run A

Figure 4.4: Throughput (Kops/s) and efficiency (Kcycles/op) for last flust and last write recovery protocols.

4.3.3 Replication Protocols Trade-offs

This experiment examines the performance and efficiency of *Anthus*'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 4.4, 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.

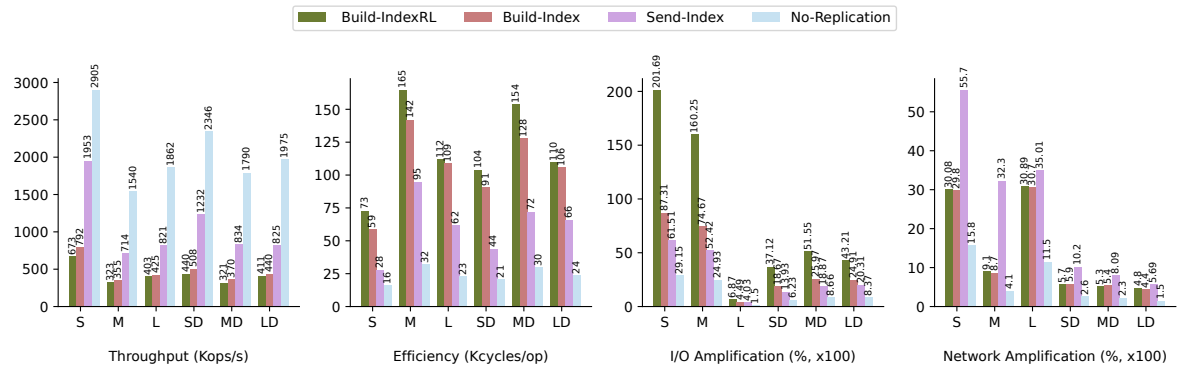
4.3.4 Three-way Replication

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.

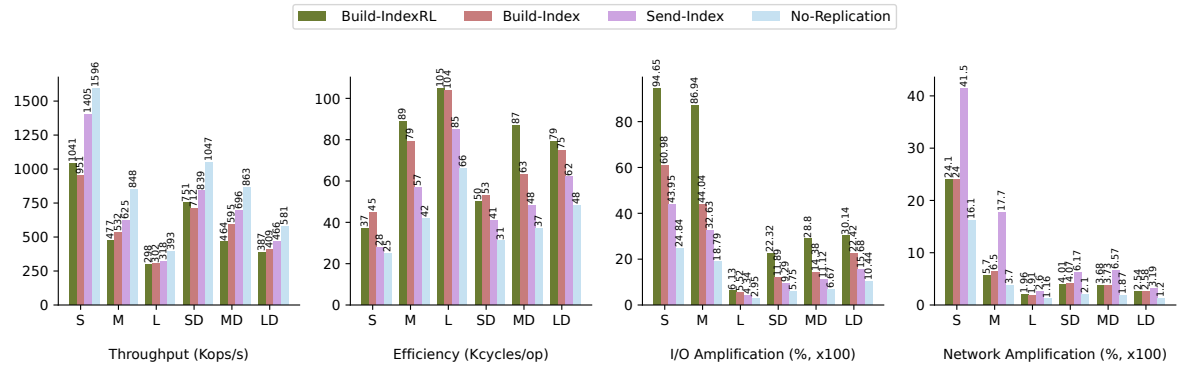
Figure 4.5 shows that for Load A, compared to Build-Index, Send-Index improves throughput by $1.05 - 2.46\times$, increases CPU efficiency by $1.2 - 2.1\times$, and decreases I/O amplification by $1.11 - 1.42\times$. Compared to two-way replication we see that the gains increase for throughput from $1.04 - 2.24\times$ 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.

4.3.5 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) Load A



(b) Run A

Figure 4.5: 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.

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.

Compared to Build-IndexRL, Send-Index improves throughput by $1.06-2.90\times$, increases CPU efficiency by $1.21-2.78\times$, and decreases I/O amplification by $1.7-3.27\times$. Compared to Build-Index, we observe that the $3\times$ 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.

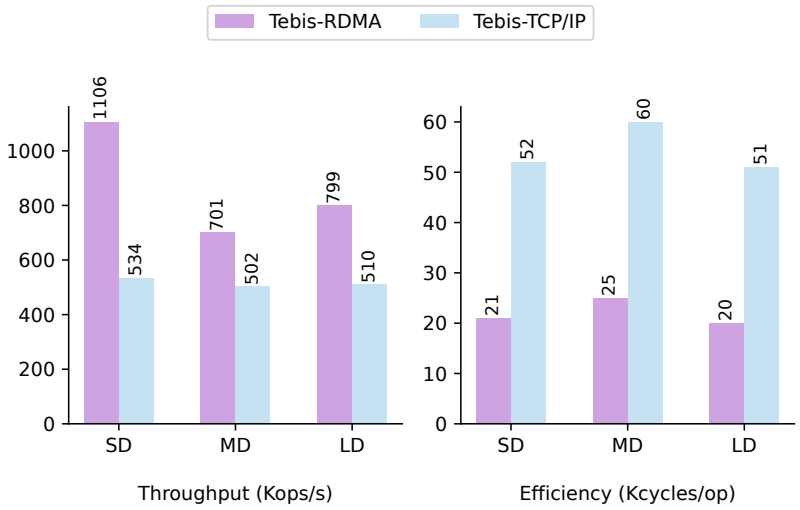


Figure 4.6: *Anthus* throughput (Kops/s) and CPU efficiency (Kcycles/op) for Load A workload using the SD, MD, and LD KV size distributions.

4.3.6 RDMA gains over TCP/IP

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 4.3: Breakdown of the cycles spent by all server threads in each component of *Anthus* for TCP/IP and RDMA.

In this experiment, we quantify the CPU efficiency gains of RDMA in addition to a version of *Anthus* that uses TCP/IP for communication with the clients. We use a standalone *Anthus* 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 Figure 4.6, 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 4.3). The four stages are:

1. Recv: Data path from the network card up to the copy of data in the *Anthus* server application buffer.

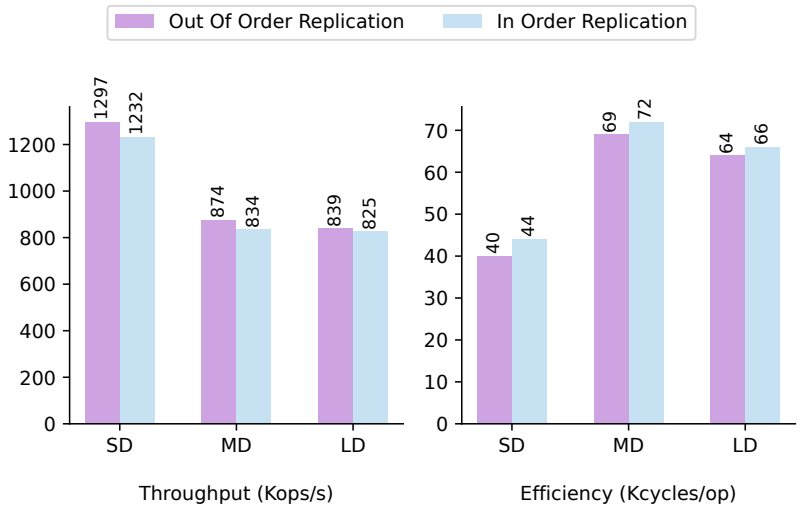


Figure 4.7: *Anthus* throughput (Kops/s) and CPU efficiency (Kcycles/op) for Load A workload using the SD, MD, and LD KV size distributions.

2. Send: Data path for sending data from an application buffer of *Anthus*
3. Parallax: includes cycles spent per operation in KV storage engine and other time.

As we observe from Table 4.3, 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.

4.3.7 Out-of-Order Replication vs In-Order Replication

In this experiment, we quantify the throughput and CPU efficiency gains of *Anthus* when replicating out-of-order compared to in-order. We run Load A for the SD,MD and LD 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.

Figure 4.7 shows that for Load A, compared to in-order replication, the performance gains of out-of-order replication on both throughput and CPU efficiency are negligible. Specifically, throughput increases by $1.01 - 1.05\times$ and for CPU efficiency by $1.03 - 1.1\times$.

Chapter 5

Conclusions

In this thesis, we propose *Anthus*, a replicated persistent LSM KV store that targets fast storage devices and fast RDMA-based networks. This work suggests a Send-Index method for KV stores utilizing hybrid KV placement, to keep efficiently an up-to-date index at the backups. Instead of performing compactions at the backups region servers, the primary in *Anthus* 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 trade-off between network utilization and backup region servers resource use. Compared with *Tebis*, in the same setups, *Anthus* increases throughput by $1.06 - 1.95\times$, CPU efficiency by up to $1.14 - 1.8\times$, decreases I/O amplification by $1.50 - 1.87\times$ while it increases network utilization by $1.21 - 2.06\times$.

Additionally, we propose a last-write replication protocol capable of handling N-1 failures in a replica group. Leveraging the efficiency of RDMA message delivery in terms of CPU and latency, we demonstrate that the last-write recovery protocols impose minimal overhead on throughput, ranging from $1 - 1.15\times$.

Bibliography

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

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

[3] INFINIBAND TRADE ASSOCIATION. Ib specification vol 1, 03,2015. release-1.3. 2015.

[4] Aurelius. Titandb, June 2012.

[5] Nikos Batsaras, Giorgos Saloustros, Anastasios Papagiannis, Panagiota Fatourou, and Angelos Bilas. Vat: Asymptotic cost analysis for multi-level key-value stores, 2020.

[6] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Inf.*, 9(1):1–21, March 1977.

[7] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[8] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated compute and storage for distributed lsm-based databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.

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

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies*, FAST '20, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[11] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in kv storage via hashing. In *Proceedings of the*

2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, pages 1007–1019, Berkeley, CA, USA, 2018. USENIX Association.

[12] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, second edition, 5 2013.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

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

[15] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[16] Facebook. Blobdb. `http://rocksdb.org/`, 2018. Accessed: August 8, 2023.

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

[18] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021.

[19] Haoyu Huang and Shahram Ghandeharizadeh. *Nova-LSM: A Distributed, Component-Based LSM-Tree Key-Value Store*, page 749–763. Association for Computing Machinery, New York, NY, USA, 2021.

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

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

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

[23] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *MSST*, pages 1–14. IEEE Computer Society, 2015.

[24] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[25] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated key-value storage management for balanced i/o performance. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)*, pages 673–687. USENIX Association, July 2021.

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[27] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow.*, 13(12):3217–3230, August 2020.

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[29] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-F érez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 490–502, New York, NY, USA, 2018. ACM.

[30] Anastasios Papagiannis, Giorgos Saloustros, Pilar Gonzá lez Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[31] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Jinglei Ren. Ycsb-c. `https://github.com/basicthinker/YCSB-C`, 2016.

[33] Seagate. Data age 2025. `https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf`. Accessed: August 8, 2023.

[34] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: Compressed, log-structured replication. *Proc. VLDB Endow.*, 1(1):526–537, August 2008.

[35] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[36] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. Cliquemap: Productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic rdma-based replication. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 851–863, Berkeley, CA, USA, 2018. USENIX Association.

[38] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tebis: Index shipping for efficient replication in lsm key-value stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 85–98, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 94–107, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. Parallax: Hybrid key-value placement in lsm-based key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 305–318, New York, NY, USA, 2021. Association for Computing Machinery.