Exploiting Hybrid Index Scheme for RDMA-based Key-Value Stores

Shukai Han
State Key Lab of Processors,
Research Center for Advanced Computer Systems,
Institute of Computing Technology, CAS,
University of Chinese Academy of Sciences
hanshukai@ict.ac.cn

Mi Zhang
State Key Lab of Processors,
Research Center for Advanced Computer Systems,
Institute of Computing Technology, CAS
zhangmi@ict.ac.cn

Dejun Jiang
State Key Lab of Processors,
Research Center for Advanced Computer Systems,
Institute of Computing Technology, CAS,
University of Chinese Academy of Sciences
jiangdejun@ict.ac.cn

Jin Xiong
State Key Lab of Processors,
Research Center for Advanced Computer Systems,
Institute of Computing Technology, CAS,
University of Chinese Academy of Sciences
xiongjin@ict.ac.cn

ABSTRACT

RDMA (Remote Direct Memory Access) is widely studied in building key-value stores to achieve ultra-low latency. In RDMA-based key-value stores, the indexing time takes a large fraction of the overall operation latency as RDMA enables fast data access. However, the single index structure used in existing RDMA-based key-value stores, either hash-based or sorted index, fails to support range queries efficiently while achieving high performance for single-point operations. In this paper, we explore the adoption of a hybrid index in the key-value stores based on RDMA, especially under the memory disaggregation architecture, to combine the benefits of a hash table and a sorted index. We propose HStore, an RDMA-based key-value store that uses a hash table for single-point lookups and leverages a skiplist for range queries to index the values stored in the memory pool. Guided by previous work on using RDMA for key-value services, HStore dedicatedly chooses different RDMA verbs to optimize the read and write performance. To efficiently keep the index structures within a hybrid index consistent, HStore asynchronously applies the updates to the sorted index by shipping the update log via two-sided verbs. Compared to state-of-the-art Sherman and Clover, HStore improves the throughput by up to 54.5% and 38.5% respectively under the YCSB benchmark.

CCS CONCEPTS

• **Computer systems organization → Architectures**.

KEYWORDS

RDMA, key-value store, hybrid index

SYSTOR '23, June 5–7, 2023, Haifa, Israel

ACM ISBN 978-1-4503-9962-3/23/06…$15.00
https://doi.org/10.1145/3579370.3594768

1 INTRODUCTION

Key-value store is a vital component in modern data centers for building various applications. Many existing systems, such as databases, social networks, online retail, and web services, use key-value stores as storage engines. The simple interfaces (e.g., PUT, GET, SCAN) and high performance of key-value stores enable users to efficiently store and access a large volume of data. Remote Direct Memory Access (RDMA) is widely explored to improve the performance of key-value stores in recent years, namely *RDMA-based* or *RDMA-enabled* key-value stores [9, 17, 23, 28, 39, 41]. RDMA communication provides two types of primitives, *one-sided* verbs allow direct access to the data in remote memory without involving the server CPU, and *two-sided* verbs enable fast message-based data transfer (like the conventional network protocols). The RDMA-based key-value stores utilize different RDMA verbs to provide key-value services, which can be classified into *server-centric*, *client-direct*, and *hybrid-access* designs. The server-centric stores only use two-sided verbs to support key-value operations, while the client-direct designs only leverage one-sided RDMA reads and writes. The hybrid-access stores exploit both one-sided and two-sided verbs for CPU efficiency and low latency, which combines the benefits of the server-centric and client-direct designs.

When handling key-value operations, indexing plays a critical role in the whole process, especially for RDMA-based key-value stores under the *memory disaggregation* architecture. Memory disaggregation separates the monolithic servers into independent components that are connected via high-speed RDMA networks, increasing resource utilization and hardware scalability [13, 30, 34]. With the support of memory disaggregation, the upper-layer applications can allocate a large amount of memory space and share the data in the memory pool with other applications efficiently. As RDMA significantly reduces the network communication overhead, the proportion of the software layer cost increases in the whole operation [26]. For single-point operations, the indexing performance largely determines the overall performance because retrieving a value from the memory pool through one-sided verbs can be very

fast. Thus, it is necessary to reduce the indexing latency when building low-latency key-value stores based on RDMA [34, 44].

Existing RDMA-based key-value stores typically leverage a single index for key lookups, either hash-based [6, 9, 17, 19, 23, 28, 35, 39, 44] or sorted (e.g., tree-backed, skiplist) [7, 10, 20, 29, 37, 43]. The hashing index locates a key-value pair based on the hash value of the key, which provides fast single-key lookups and can be easily completed using one-sided verbs. For example, RACE [44] hashing index executes all index requests using only one-sided RDMA verbs. However, it is hard to deal with range queries (i.e., SCAN) based on a hashing index. Thus, some RDMA-based key-value stores (e.g., Sherman [34]) choose a sorted index that maintains the order of key-value pairs to provide efficient range queries. Though the sorted index supports rich key-value operations, the sorted index incurs longer latency than the hashing index for single-key lookups. Searching along a sorted index requires the involvement of the server CPU to complete within one round trip; otherwise, it incurs multiple round trips if only using one-sided RDMA reads.

To provide rich key-value services with low latency, it is promising to combine a hash table and a sorted index for single-key lookups and range queries respectively in RDMA-based key-value stores. HiKV [40] realizes the idea of a hybrid index on a single server with hybrid memory and demonstrates the performance improvement of key-value operations. However, adopting a hybrid index in RDMA-based key-value stores under memory disaggregation poses new challenges. First, index management requires keeping two different index structures consistent with the key-value items in an efficient manner. Though the disaggregated memory provides high scalability for data storage, the memory server has little computing resource to handle the updates on the hybrid index. Thus, how to mitigate the management overhead of a hybrid index with strong consistency remains an open issue. Furthermore, as failures in distributed systems are commonplace, the key-value stores should consider how to support key-value services using the hybrid index and reconstruct the index under server failures.

We present HStore, an RDMA-based key-value store that combines a hash table and a skiplist to support rich key-value operations with high indexing performance. HStore leverages the monolithic servers as the index servers to efficiently manage the hybrid index with strong consistency, and stores the key-value pairs in the memory pool to benefit from the disaggregated memory. HStore places the hash table and the skiplist on two index servers separately, called hash table server (HTS) and skiplist server (SLS), to protect the hybrid index against server failures. To minimize the write latency, HStore batches the index updates in a log, ships the update log from the HTS to the SLS, and applies the updates to the skiplist asynchronously. It allows hybrid access using different RDMA verbs for CPU efficiency and low latency. That is, HStore uses two-sided verbs for write operations to guarantee strong consistency; for read requests, it uses one-sided verbs for GET operations to bypass the server CPU, and leverages two-sided verbs for range queries to simplify the design complexity of sorted index without degrading the SCAN performance. To the best of our knowledge, HStore is the first RDMA-based key-value store that leverages a hybrid index for key-value services under memory disaggregation.

We summarize our contributions as follows.

- We analyze the usage of a single index in RDMA-based key-value stores and motivate the adoption of a hybrid index by comparing the performance of different indexes.
- We propose HStore that combines a hashing index and a sorted index to support efficient single-key lookups and range queries via different RDMA verbs. HStore consistently updates the two index structures within the hybrid index and reduces the write latency by performing asynchronous updates on the sorted index.
- We implement HStore using eRPC [20] in two-sided communications and evaluate its performance. Our evaluation results show that HStore improves the throughput of Sherman (a B^+-tree index) and Clover (a hashing index) by up to 54.5% and 38.5% respectively under the YCSB workloads.

The source code of HStore is publicly available at: https://github.com/KinderRiven/HStore.

2 BACKGROUND AND MOTIVATION

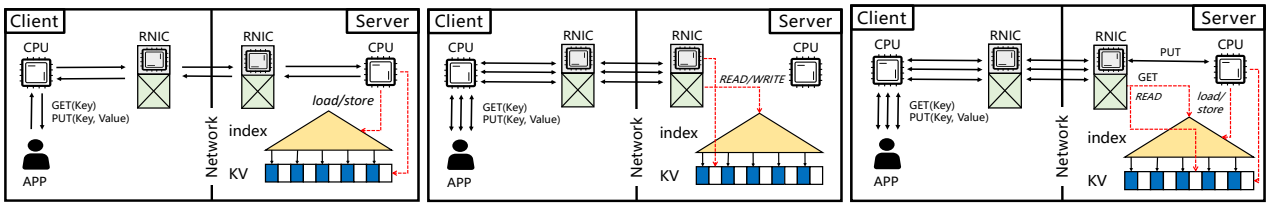
2.1 RDMA-based Key-Value Stores

RDMA basics. RDMA is an alternative to network protocols (e.g., TCP, UDP), which allows fast data transfers between the local and remote memory with kernel bypassing [18]. RDMA hosts establish communication using *queue pairs* (QPs) consisting of a send queue and a receive queue, and post operations to the queues via different *verbs*. RDMA communications provide two types of verbs, *one-sided verbs* (aka *memory verbs*) and *two-sided verbs* (aka *message verbs*). The one-sided verbs, including `READ`, `WRITE`, `CAS` (compare-and-swap), and `FAA` (fetch-and-add), enable direct access to a pre-allocated memory region on a remote server without involving the remote CPU. Two-sided verbs work like the conventional network protocols based on messages, where one process sends/receives a message using the `SEND`/`RECV` verb. Data transfer based on two-sided verbs incurs CPU cost on the remote server.

A large number of research works [6, 10, 17, 19, 20, 28, 29, 35, 37, 39, 43, 44] have studied how to leverage RDMA network protocols to optimize key-value stores in terms of their storage requirements and the characteristics of RDMA primitives. The architecture of RDMA-based key-value stores can be classified into, *server-centric*, *client-direct*, and *hybrid-access* designs.

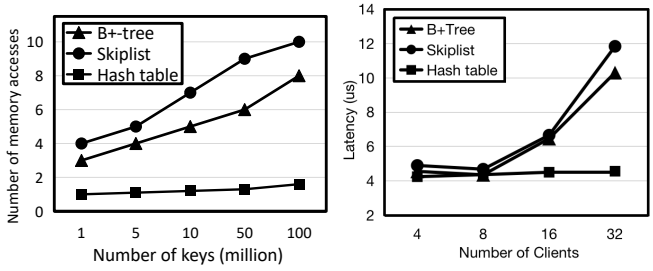
Server-centric design. Some works [17, 19, 20, 39, 43] adopt server-centric design by replacing the communication layer (e.g., RPC) in a key-value store with RDMA verbs. Figure 1a shows the process of GET and PUT operations, where the client sends a request to the server and the server returns the response after processing the request locally. Such a design only introduces one RTT, one half for sending the request and another half for receiving the response. Moreover, it simplifies the implementation because it only requires adding an RDMA-enabled communication module. However, the server-centric design still involves the remote CPU, which limits the scalability of the key-value stores and degrades the overall performance when the CPU becomes the bottleneck.

Client-centric design. To bypass the CPU of the remote server, some systems [6, 34, 44] choose client-direct architecture, enabling the clients to directly access a pre-allocated memory region on the server. As shown in Figure 1b, the client directly fetches/writes the data from/to the server using one-sided verbs. Though the client-direct approach reduces the server CPU cost, it requires multiple



(a) Server-centric design (b) Client-direct design (c) Hybrid-access design

Figure 1: The architecture of different key-value stores with RDMA.



(a) Single-key lookup in DRAM (b) Single-key lookup over RDMA

Figure 2: The performance of different index structures. Note that the number of memory accesses in (a) is equal to the number of round trips when we use one-sided verbs for the index without any optimization.

RTTs to complete one complex operation, e.g., traversing a tree-based index on the remote server. Hence, the client-direct design incurs long latency when handling some complex data structures.

Hybrid-access design. Many existing works [10, 28, 29, 35, 37] leverage hybrid-access design to combine the advantages of server-centric and client-direct architectures. Figure 1c illustrates how the hybrid-access design works where the usage of one-sided verbs is restricted to read-only requests (i.e., GET and SCAN). That is, the client can directly access the data from the remote server for read-only operations, while the server only needs to process the requests involving writes (i.e., PUT, DELETE, and UPDATE). It uses the client-direct approach for read-only requests and server-centric mode for other operations. Thus, the hybrid-access design not only harnesses the high performance of RDMA networks, but also relaxes the burden on the server CPUs.

2.2 RDMA-friendly Index Structures

As a key component of key-value stores, the indexes can determine the overall operation performance (i.e., the time of a key-value operation). Prior works have proposed many efficient index structures for RDMA-based key-value stores. Here we summarize the characteristics of different index structures, either hash-based or sorted, and their applications in key-value stores.

Hashing index. The hashing index has been widely adopted in RDMA-enabled key-value stores [6, 9, 17, 19, 28, 35, 39, 44], because the hashing index can provide fast lookup services. The simple data structure of the hash table enables the clients to directly fetch data from the remote server using one-sided RDMA primitives (i.e., client-direct or hybrid-access design), which mitigates the CPU overhead on the server. Thus, the key-value stores can achieve high performance for single-point operations (e.g., GET, PUT, UPDATE, DELETE) using the hashing index. However, the hashing index does not support range queries, i.e., SCAN operations, limiting its applications in the key-value stores.

Sorted index. To support efficient range queries, many RDMA-based key-value stores [7, 10, 20, 29, 37, 43] leverage a sorted index (e.g., B^+-tree, skiplist), which organizes the key-value pairs in an ordered manner. When locating a key-value pair, the sorted index requires multiple lookups. For example, the B^+-tree needs to search from the root node to the leaf nodes, while the skiplist requires multiple random accesses among its nodes. Note that the number of searching operations increases with the data amount as the tree or skiplist grows larger. Moreover, the complex structure of sorted indexes makes it complicated to update the index during writes. Hence, the RDMA-based key-value stores typically use two-sided verbs (i.e., server-centric or hybrid-access design) to deal with sorted index, which reduces the number of communication round trips but incurs server CPU cost.

2.3 Memory Disaggregation Architecture

Conventional data centers run key-value services on a cluster of *monolithic servers* that packs CPU and memory into the same physical server, suffering from low memory utilization and increased total cost of ownership [13, 30, 34]. To improve resource utilization, *memory disaggregation* decouples the computing and memory resources into independent components (i.e., compute and memory resource pools), which are connected through RDMA networks to reduce the access latencies between the *compute servers* and *memory servers*. Recent works explore building high-performance key-value stores under the memory disaggregation architecture, by placing the key-value pairs in the memory pool and designing efficient index structures [32, 34, 43, 44]. RACE [44] proposes a hashing index based on one-sided verbs, providing lock-free remote concurrency control and efficient remote resizing; FG [43] and Sherman [34] build a B^+tree index for disaggregated memory. Our work benefits from the disaggregated memory by storing the values on the memory servers and focuses on designing an efficient hybrid index for both single-point lookups and range queries.

2.4 Motivation and Challenges

We conduct some experiments with RDMA communications to compare the indexing performance (e.g., index lookup, index insert) of different index structures. We use two machines equipped with two ten-core Intel Xeon Gold 5215 CPUs (2.5 GHz), 64 GB memory,

and one 100 Gbps Mellanox ConnectX-5 Infiniband NIC, to be a key-value store server and a client respectively. The server and client are connected by a 100 Gbps switch. We set the key/value size to be 16 B/32 B respectively according to the previous study [5].

Observation #1: Using one-sided verbs to implement a sorted index can bypass the server CPU, but introduces more RTTs to traverse the index, thereby degrading the performance of single-point lookups. Although some recent works [34, 43] explore supporting a sorted index efficiently with pure one-sided verbs (to deploy on disaggregated memory), the optimized read and write operations still need multiple round trips. For example, 94.1% of write operations need at least three round trips in Sherman [34], a state-of-the-art distributed B$^+$-tree optimized for writes. For read operations, both FG [43] and Sherman require caching index locally on the client to avoid traversing the tree nodes, but some index lookup operations incur read retries (even experience nine times).

We compare the performance of different index structures by measuring the *number of memory accesses* on a key-value server, which equals the number of round trips when we adopt one-sided verbs for the index without any optimization (e.g., combining dependent RDMA commands). We test three common index structures (i.e., B$^+$-tree [2], chained hash table, and skiplist [1]) respectively, by accessing each key one by one using a single lookup thread. All indexes reside in the memory. Figure 2a shows the number of memory accesses of the sorted index and hashing index under different data amounts (1 to 100 million key-value pairs). With a larger number of key-value pairs, the number of memory accesses of B$^+$-tree and skiplist, increases from 3 to 10, because B$^+$-tree expands itself by adding more nodes while skiplist splits itself into more lists. For the hash table, the number of memory accesses remains relatively stable (around one) with a slight increase as more buckets are accessed in case of hash conflicts. Our experiment results confirm the previous analysis that using one-sided verbs for sorted index requires several round trips for one single-key search. Thus, the sorted index based on one-sided verbs increases the latencies of single-point lookups though it avoids involving the server CPU.

Observation #2: The single-key lookup performance of sorted index based on two-sided verbs drops down a lot with more clients, as the server CPU can become the bottleneck. We evaluate the *indexing latency* (time of index lookup over RDMA) of different indexes under different numbers of client threads. Based on observation #1, we implement B$^+$-tree and skiplist using eRPC [20] (an RPC library based on two-sided verbs), and access the hash table via one-sided verbs. Here we load 100 million keys to each index structure, and start a different number of client threads issuing GET requests over all keys uniformly. Each client thread issues 20 million GET requests to the server. We use four RPC threads and four processing threads on the server to process the client requests, by enabling hyper-threading on the four CPU cores (i.e., each physical core runs an RPC thread and a processing thread). Figure 2b illustrates the indexing latency of different index structures under various numbers of clients. Here, the indexing latency includes the RDMA transfer time and the key lookup time. For the sorted index (i.e., B$^+$-tree, skiplist), the indexing latency increases with the number of clients as the server CPU fails to handle all the client requests in time. For the hashing index, the indexing latency remains relatively stable across different numbers of clients because there is no CPU cost through one-sided verbs. The indexing latency of the sorted index is almost three times that of the hash table when there are 32 client threads issuing requests to the server. Therefore, the hashing index provides higher performance than the sorted index, which motivates our idea of combining a hashing index and a sorted index in RDMA-based key-value stores.

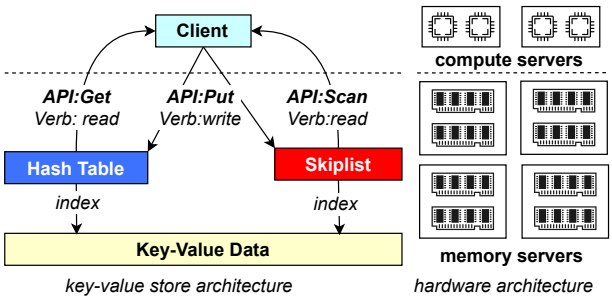


Figure 3: An intuitive approach to build key-value service with hybrid index under the memory disaggregation architecture.

However, it is non-trivial to realize the idea of a hybrid index efficiently in RDMA-based key-value stores, especially under the memory disaggregation architecture. The first challenge is to preserve strong consistency between the hashing index, the sorted index, and the key-value items. As the memory server in the memory pool does not have much computing power, it needs to reduce the synchronization and serialization overhead of different index structures within a hybrid index. Second, though the read operations can benefit from the hybrid index, the write performance drops down because the system needs to maintain two index structures and update both of them for write operations. How to mitigate the overhead of keeping and updating the hybrid index remains an open problem. Last but not least, when a failure occurs (e.g., one index server crashes), it is challenging to support key-value services efficiently and achieve fast index recovery based on the remaining index structures.

3 DESIGN OF HSTORE

In this section, we first introduce an intuitive approach of a hybrid index which is hard to efficiently maintain strong consistency between different index structures. We then present our hybrid index scheme under the memory disaggregation architecture and build HStore, an RDMA-enabled key-value store based on a hybrid index.

3.1 An Intuitive Approach

One natural approach to leverage a hybrid index in RDMA-based key-value stores under the memory disaggregation architecture is using one-sided verbs to access both hashing index and sorted index as the memory servers have limited computing resources. Figure 3 shows the intuitive design of building a key-value store based on a hybrid index which consists of a hash table and a skiplist. In such a design, the hybrid index and the key-value pairs are located on memory servers where the compute servers access them via the one-sided verbs. To achieve high read performance, the client on the compute server can access the hash table or the skiplist on the memory servers based on the type of read operations via

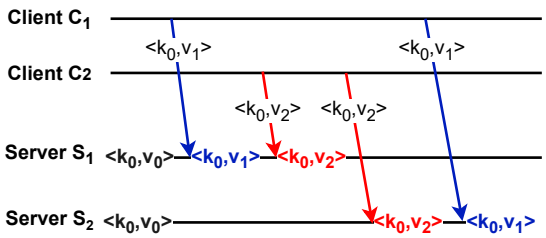


Figure 4: An example that makes the two index structures within a hybrid index inconsistent. Here the server S_1 and S_2 hold the hash table and skiplist for the key k_0 respectively.

READ. That is, the client can query the hash table directly for single-point lookups while leveraging the skiplist for range queries. For write operations, the client needs to update both the hash table and skiplist using the WRITE verb.

Accessing the index on the memory servers via one-sided verbs seems to maximize the system performance, but poses unique challenges to maintain consistency between the two index structures within a hybrid index. The key problem is how to update the hashing index and sorted index in an atomic and efficient approach. As network interruptions and server crashes are commonplace in a distributed environment, it is possible that one index is updated successfully while another fails to update. The two index structures within a hybrid index become inconsistent, such that a key-value pair is indexed by only one index structure or none of them.

Figure 4 shows an example that makes the two index structures within a hybrid index inconsistent. Assume that the hash table and the skiplist of a key-value pair $<k_0, v_0>$ are located on server S_1 and S_2 respectively, and there are two clients C_1 and C_2 updating the same key-value pair $<k_0, v_0>$ where C_1 writes $<k_0, v_1>$ and C_2 writes $<k_0, v_2>$. When a client needs to update the value of k_0, it requires updating the index entries of k_0 on the server S_1 and S_2 by issuing two WRITE requests. On the server S_1, the index entry is updated to $<k_0, v_2>$ as the request from the client C_2 comes after the client C_1's. However, for the index on the server S_2, it is updated to $<k_0, v_1>$ because the C_2's request arrives earlier than that of the client C_1. Therefore, the index structures on the server S_1 and S_2 become inconsistent due to the incorrect concurrent updates.

The main reason leading to the inconsistent hybrid index is that the client performs update operations on two servers independently via one-sided verbs, making atomic update and serialization of the hybrid index hard to realize. One solution to this inconsistency problem is to employ distributed transactions [4]. It requires checking the transaction table during writes, such that the client can judge the validity of an update operation based on the transaction table, and then complete or roll back the whole operation. This complicated method inevitably introduces additional overhead to the write process, thereby degrading the write performance [42]. Therefore, we need to adopt a lightweight mechanism to efficiently manage the hybrid index in an RDMA-based key-value store.

3.2 Hybrid Index Scheme

Overview. Figure 5 plots the architecture of HStore where the hybrid index resides on two monolithic servers and the values are stored in the memory pool consisting of multiple memory servers.

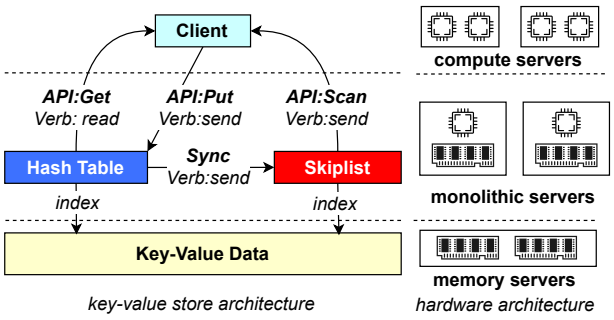


Figure 5: The hybrid index scheme deployed on two monolithic servers in HStore.

As updating the hybrid index on the memory servers increases the complexity of index synchronization, HStore builds a hybrid index atop the monolithic servers with both computing and memory resources. We deploy the hashing index and sorted index on two servers separately in order to tolerate single-server failures and reduce the recovery time. When one index server fails, the remained server can continue to provide partial key-value services until the failed index gets reconstructed. We refer to the server holding the hashing index as Hash Table Server (HTS), and the server containing the skiplist as SkipList Server (SLS). To achieve low latency for read operations, HStore performs single-key lookups by accessing the hash table on the HTS, and answers range queries based on the skiplist on the SLS. Compared to updating the hybrid index with only one-sided verbs, HStore simplifies the synchronization between the HTS and SLS through two-sided verbs. If there are more index servers (more than two servers) to hold a bigger data structure, we can use sharding to spread the keys across multiple hash tables where each hash table has a corresponding sorted index. That is, the current hybrid index in HStore is a unit to preserve strong consistency between the hashing index and sorted index.

Choices of RDMA verbs. HStore uses different RDMA verbs for different read operations to achieve low latency. For single-key lookups, HStore allows the client to directly access the hash table on the HTS using one-sided verbs, which bypasses the CPU. For range queries, the client sends the request to the SLS maintaining a sorted index via two-sided verbs, which can be completed in one round trip. Upon receiving SCAN requests, the SLS searches its local skiplist and returns the results to the client. Using two-sided verbs to handle SCAN requests reduces the number of RTTs, but limits the scalability as it involves the server CPU (as shown in Figure 2). We argue that the advantages of this design outweigh using one-sided verbs to access the sorted index, because the latter requires redesigning the index structure algorithm and handling the conflict of different clients' requests [34]. Moreover, as the overhead of SCAN operations mainly depends on accessing the values rather than the index, the performance gain of using one-sided verbs for sorted index may not be much higher than that of using two-sided verbs (based on the evaluation results in Section 5.2). HStore leverages two-sided verbs to deal with write requests, because the client requires getting a response from the HTS and the HTS needs to receive the response from the SLS. In summary, HStore uses one-sided verbs for GET and two-sided verbs for other operations.

Index updating. For write requests (PUT, DELETE, UPDATE operations), HStore updates both the hashing index and the sorted index to keep them consistent. Compared to updating a hashing index, updating a sorted index is a much more costly operation as it involves additional operations to maintain the order of the keys. For example, inserting one key into a B$^+$tree index may trigger node splitting and merging while updating a skiplist may require list splitting. Thus, HStore performs asynchronous updates to the sorted index like HiKV [40] to keep the write latency low, while synchronously updating the hash table for single-key lookups.

In case of server failures, HStore stores the updates in an append-only log before applying the updates to the index structures. Each log entry consists of a key, a value address, and a mark named "isApplied" to indicate whether this key has been inserted into the local index structure. When receiving a write request from a client, the HTS first records the update in its local log and sends the update to the SLS; the SLS stores the update in its log, replies to the HTS, and asynchronously applies the update to the skiplist; after receiving a successful response from the SLS, the HTS updates its hash table and returns to the client. Note that the update requests of the same key are processed by the same thread, such that we do not need to serialize the concurrent updates. We elaborate on the implementation of index updating in Section 4.2. As the sorted index is updated asynchronously, the SLS updates the index based on its log before answering SCAN for strong consistency. In other words, HStore supports serializability in that the written items can always be accessed during single-point reads and range queries.

Consistency guarantee. HStore maintains strong consistency from two perspectives: (i) consistent index structures between HTS and SLS, and (ii) consistent index with the key-value data. For the first consistency issue, our approach to update the index based on the log can keep the sorted index on SLS consistent with the hash table on HTS. A complete index updating means that the update has been recorded in the logs on both the HTS and the SLS, and the hash table gets updated. As the logs on the HTS and the SLS are the same, the sorted index is consistent with the hash table when the sorted index finishes applying the updates asynchronously. For the second consistency point, it means that the data can be indexed during reads after the data is successfully written to the key-value store. That is, if a write fails, the invalid key-value pair should not be indexed during the reads. To ensure the index structures consistent with the data stored, HStore uses sequential writes to store the value and update the index. When performing a write request, the client first stores the key-value pair to the memory pool and gets the value address, then connects to the HTS for index updating. If any step fails during the write process, the write operation fails and the client can restart the write operation. Note that when updating an existing key-value pair, HStore returns the old value address to the client, such that the client can release the old value space asynchronously. If some errors occur during the process of space release, HStore can subsequently scan out the orphaned key-value data to reclaim the memory space.

Failure handling. HStore starts to reconstruct the failed index when detecting some index server failures. We assume that there is a centralized manager like ZooKeeper to monitor the health status of the HTS and SLS [16]. During index recovery, HStore blocks the write operations and serves read requests using the remained index. When the HTS fails, the hash table can be recovered from SLS by scanning the skiplist; it reduces the recovery time as the whole process does not need to retrieve the key-value pairs in the memory pool. If the SLS fails, the only method to recover the skiplist is to scan all key-value pairs across the memory servers, because the HTS only stores the hash value of the key field instead of the complete key (details are described in Section 4.1). Specifically, the recovery of the skiplist involves retrieving all key-value pairs' addresses from the HTS and accessing the complete key at each address. Thus, it is time-consuming to reconstruct a skiplist as the recovery needs to scan the entire key space in the memory pool.

4 IMPLEMENTATION

We implement HStore in C++, which realizes two-sided communications based on the eRPC library [20]. We first introduce the threading model and data structures of the hybrid index used in HStore. We then explain how HStore provides key-value services (i.e., GET, PUT, and SCAN operations) with the hybrid index.

4.1 Hybrid Index

Threading model. HStore employs multi-threading model for writes and range queries which involve the server CPU. Each index server starts several *RPC threads*, which are responsible to receive and handle RPC requests, and *worker threads*, which perform index updates and key lookups. Figure 6 depicts the threading model of hybrid index scheme. During writes, the worker threads on the HTS update the hash table while the worker threads on the SLS apply updates to the skiplist. For range queries, the worker threads on the SLS look up the keys among the sorted lists. We explain the details of processing the key-value operations by the RPC threads and worker threads in Section 4.2.

Data structures. HStore currently adopts a chained hash table and a basic skiplist to constitute a hybrid index. Note that the hash table and the skiplist can be replaced with other optimized hash table and sorted index (e.g., tree-backed) respectively. Each chain consists of multiple buckets, each of which is of 64 B. A bucket contains seven hash slots, and a pointer of 8 B to link the next bucket. Each hash slot records the information on a key-value pair, consisting of the hash value of the key (1 B), the length of the key-value item (1 B), and the value address (6 B). For single-key lookups, the client first computes the bucket address based on the key locally, reads a bucket using the `READ` verb, and then searches for the key within the bucket. The client needs to check each slot by comparing the signature (i.e., the hash value of the key) and the exact key. If the comparison succeeds, the value is returned. If no valid slot is matched and the next pointer is not empty, the next bucket is queried based on the next pointer. When a bucket is full during writes (e.g., a hash collision occurs), the server links a new bucket after the last bucket using the next pointer. To avoid resizing, we allocate more buckets than required by consuming a little more memory space. For the skiplist, HStore divides the whole list into several partitions based on the hash values of the keys, such that each partition can be searched concurrently by multiple threads to reduce the latency of range queries.

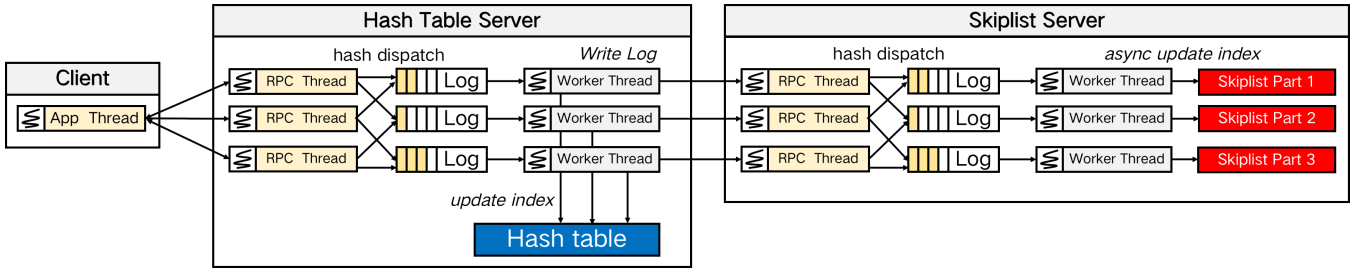


Figure 6: The threading model of hybrid index in HStore.

4.2 Key-value Services

Write operations. HStore uses two-sided verbs to perform index updating and data storing. For PUT and UPDATE operations, the client first stores values in the memory pool to get the value addresses before updating the hybrid index. Then the client sends requests for updating the index to the RPC threads on the HTS. The RPC threads on the HTS append the updates to different logs based on the hash values of the keys. The worker threads then send the updates to the SLS using two-sided verbs, and wait for the responses from the SLS. To improve the write throughput, the worker threads perform log synchronization between the HTS and the SLS in a batch. On the SLS, the RPC threads append the updates to the log and send successful responses to the HTS, while the worker threads asynchronously update the skiplist. As the skiplist is divided into several partitions, different partitions can be updated by multiple worker threads concurrently. Upon receiving the successful responses from the SLS, the worker threads on the HTS apply the updates to the hash table and return success to the client. The updates of the same key are processed by the same thread due to the above processing approach: 1) the RPC thread writes a request to a log based on the hash value of the key; 2) the write requests with the same key are placed in the same log, which is processed by the same worker thread. To handle concurrent writes and reads, the HTS updates the hash table by a compare-and-swap operation with the CPU, which deals with each update as an atomic operation. If any step fails during a write, the incomplete write operation is considered as a write failure; the client can restart the write process if it does not receive a successful response from the HTS after a period of time.

GET. HStore handles GET requests using one-sided verbs, totally bypassing the server CPU. It leverages the hash table on the HTS for single-key lookups. To read a key-value pair, the client directly accesses the hash table using one-sided verbs to obtain the value address. Then, the client retrieves the value from the memory server according to the value address. The whole process avoids incurring CPU cost on both the index server and the memory server.

SCAN. HStore provides efficient range queries based on the skiplist. The client submits SCAN requests via eRPC to the SLS, where the worker threads search among the skiplist partitions concurrently to return the results to the client. Note that the worker threads make sure that no index updates remain before processing the query. That is, if there are index updates left, the worker threads will first apply the updates to the skiplist and then answer the SCAN request.

5 EVALUATION

5.1 Experiment Setup

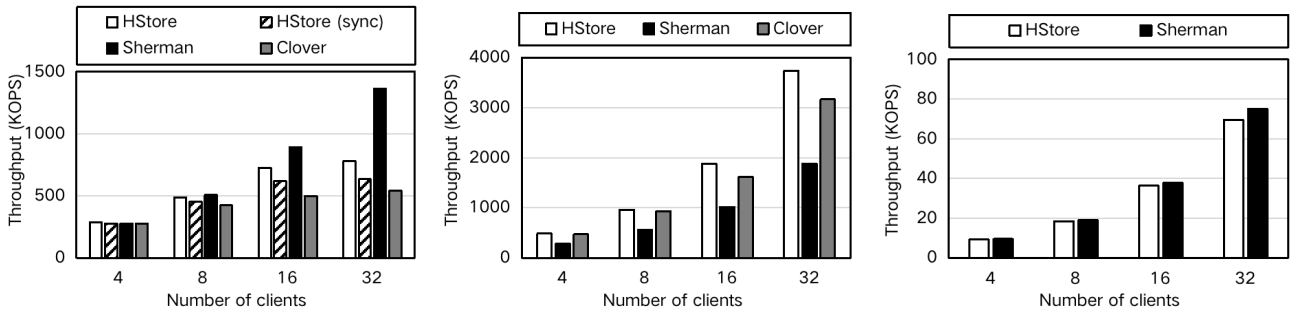
We evaluate the performance of HStore on a local cluster. Our local cluster consists of four servers, each of which runs CentOS Linux release 7.6.1810 with 4.18.8 kernel and is equipped with two ten-core Intel Xeon Gold 5215 CPUs (2.5 GHz), 64 GB memory and one 100 Gbps Mellanox ConnectX-5 Infiniband NIC. All machines are connected via a 100 Gbps switch. We use one machine for value storage as a memory server, deploy the hybrid index on two machines, and start multiple client threads on one machine to send requests. Each index server runs four RPC threads and four worker threads on the four CPU cores by enabling hyper-threading, where each physical core runs an RPC thread and a worker thread. Each client thread establishes a QP pair with the HTS. We use LevelDB's db_bench [11] and YCSB [8] benchmarks to evaluate the performance of HStore. The clients use closed-loop tests here. We run five times for each experiment and plot the average results.

We compare HStore to *Sherman* [34], a key-value store based on the B$^+$tree index, and *Clover* [32], a key-value store using the hashing index. Both of them use one-sided verbs for index lookups under the memory disaggregation architecture. As Sherman and Clover use a single index, we use one server to store their index as an index server. For other configurations, we use the same settings as HStore for Sherman and Clover. We also evaluate the write performance of HStore (sync). HStore makes sure that the logs on the HTS and SLS are synchronized before the write operation is completed and asynchronously updates the skiplist on the SLS, while HStore (sync) synchronously updates both the hash table on the HTS and the skiplist on the SLS during writes.

5.2 Performance of Basic Operations

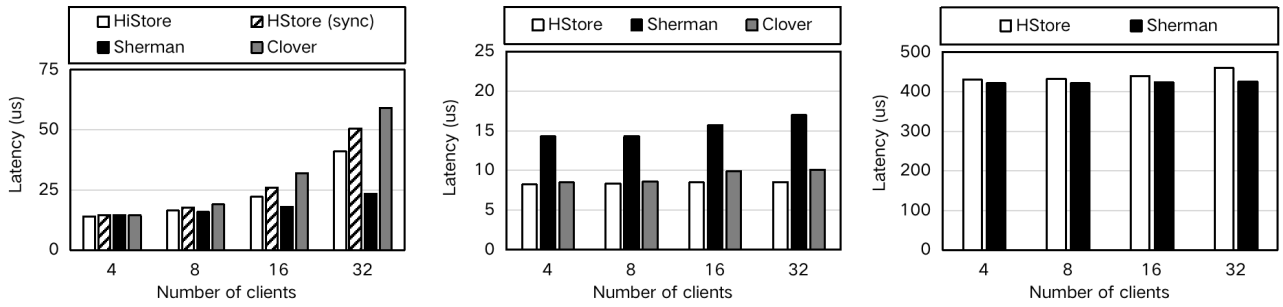
To evaluate the performance of basic operations (i.e., PUT, GET, SCAN) in HStore, we first load 100 million (100 M) key-value pairs with a key size of 16 B and a value size of 32 B. We then issue 20 M PUT requests, 20 M GET requests, and 1 M SCAN requests using db_bench. We start four RPC threads and four worker threads on each index server. We collect the throughput and latency of HStore, HStore (sync), Sherman, and Clover under different numbers of client threads ranging from 4 to 32. Figures 7 and 8 plot the throughput and latency of PUT, GET, and SCAN operations.

For write performance, HStore achieves similar performance to Sherman and Clover under eight or fewer client threads. When the number of client threads increases to 16 or higher, the write



(a) PUT operation **(b) GET operation** **(c) SCAN operation**

Figure 7: The throughput of basic operations.



(a) PUT operation **(b) GET operation** **(c) SCAN operation**

Figure 8: The latency of basic operations.

throughput of HStore is about 1.4 times Clover's because Clover needs to handle write conflicts at the remote server with more RTTs. Compared to Sherman, HStore reduces the throughput by 18.7% and 42.9% under 16 and 32 client threads respectively. As Sherman implements multiple specific optimizations for the write operations (e.g., accelerating lock operations), it is reasonable that Sherman achieves higher write performance than HStore which requires updating two index structures. HStore achieves 4.0-22.9% higher write throughput than HStore (sync) under different numbers of client threads, because asynchronous skiplist updates on the SLS reduce the waiting time during writes. The evaluation results demonstrate the effectiveness of asynchronous index updates in HStore.

For GET operations, HStore achieves the best performance among Sherman and Clover. When there are eight or fewer client threads, HStore achieves similar throughput and latency to Clover because both of HStore and Clover leverage the hash table for single-key lookups. HStore achieves higher performance for GET operations than Clover when there are more client threads, increasing the throughput by 16.6% and 18.1% under 16 and 32 clients respectively. The reason is that HStore handles hash conflicts by allocating multiple slots such that it can retrieve one bucket within one RTT, while Clover may need multiple RTTs to access a target chained list. Compared to Sherman, HStore increases the throughput by 71.3-98.6% and reduces the latency by 41.6-49.6% under different numbers of client threads. Sherman needs to recursively search from the root node to the leaf nodes when performing read queries. When the depth of the tree is deeper, it needs more RTTs to complete a single-point query. Our evaluation results show that it requires at least two RTTs to complete a single-point query. In contrast, HStore only needs one RTT to obtain the requested index entry from a bucket in the hashing index. The results show that HStore benefits from the hashing index for single-key lookups. Note that when the number of client threads increases from 4 to 32, the GET latencies of all three systems do not significantly increase, indicating that the above throughputs are not the maximum. As all three systems use one-sided verbs for index queries, they can achieve good system scalability when the server computing resources are insufficient. In other words, if the number of clients continues to increase, the total throughput of each system will increase. The maximum throughput is related to many factors, such as the number of QPs established between the clients and servers, network bandwidth, and the processing capacity of the RNIC [20]. The primary goal of our work is to leverage a hybrid index to reduce the RTT overhead during index access, thereby achieving a low indexing latency. Thus, we do not explore the maximum throughput of the systems here.

As Clover does not support range queries, we only plot the SCAN performance of HStore and Sherman. The number of keys covered by each scan operation in our evaluation is 100. HStore achieves a similar SCAN performance to Sherman. There is nearly no performance difference, because the SCAN latency depends on the time of data access rather than the indexing time (shown in Section 5.3).

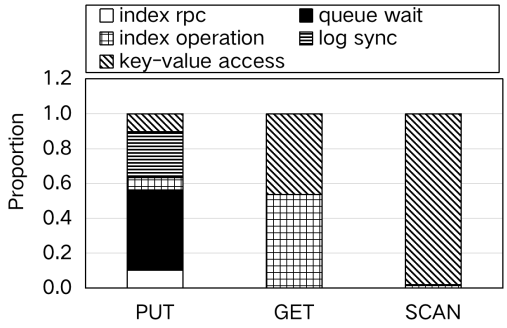


Figure 9: Performance breakdown of basic operations.

5.3 Microbenchmarks

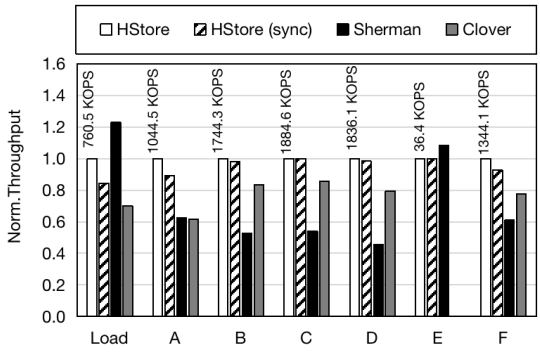
We conduct microbenchmarks for HStore using db_bench. We measure the time of the following phases during a request: (i) *index rpc*, the communication time between a client and an index server based on eRPC; (ii) *queue wait*, waiting in a log queue to be processed by a worker thread (we do not count the waiting time in the RPC queue as it is quite short due to the low concurrency); (iii) *index operation*, performing update or search on an index; (iv) *log sync*, log synchronization between the HTS and SLS during writes; and (vi) *key-value access*, writing/reading values to/from data servers.

Figure 9 shows the performance breakdown of basic operations with 32 clients. For PUT operation, queue wait accounts for the highest proportion, about 50-66% of the whole latency. As HStore handles all write requests on the index server, the server CPU becomes a bottleneck when there are too many requests and a number of write requests wait in the queue to process. The second highest time-consuming operation is log sync, which takes 24% of the total write latency. For the GET operation, HStore completes the index lookup through one RTT and the key-value access through one RTT. Therefore, the proportion of index operation and key-value access is basically the same. When handling a SCAN operation, HStore spends about 96% of the time in key-value access while the remaining time is used for index lookups. The proportion of time depends on the number of elements covered by the range query. For a SCAN operation, HStore uses two-sided verbs to obtain the value addresses of all required keys within an RTT. As HStore stores the key-value pairs in the remote memory pool and needs to access them through one-sided verbs, fetching each value takes one RTT. For example, for a SCAN operation with the range of 100, accessing the index requires one RTT, and fetching the values needs 100 RTTs.

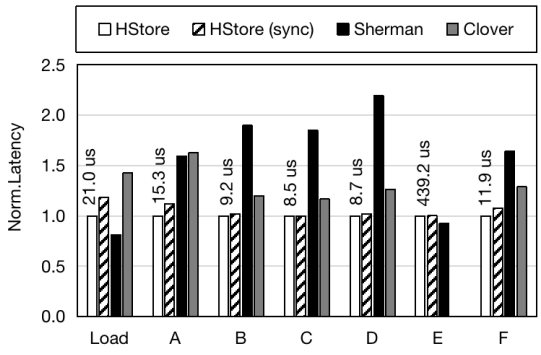
5.4 Performance under YCSB Workloads

We evaluate HStore, Sherman, and Clover under YCSB workloads A-E [8]. We first load 100 M key-value pairs with the default key-value size where the key size is about 20 B and the value size is around 200 B. We then run each workload with 20 M requests, which are issued by 16 clients. HStore performs the update operation through the PUT method, and performs the read-modify-write operation by the combination of GET and PUT operations.

Figure 10 depicts the performance of HStore, HStore (sync), Sherman, and Clover under the YCSB workloads. Compared to Sherman and Clover, HStore improves the throughput by 37.3-54.5%



(a) Throughput



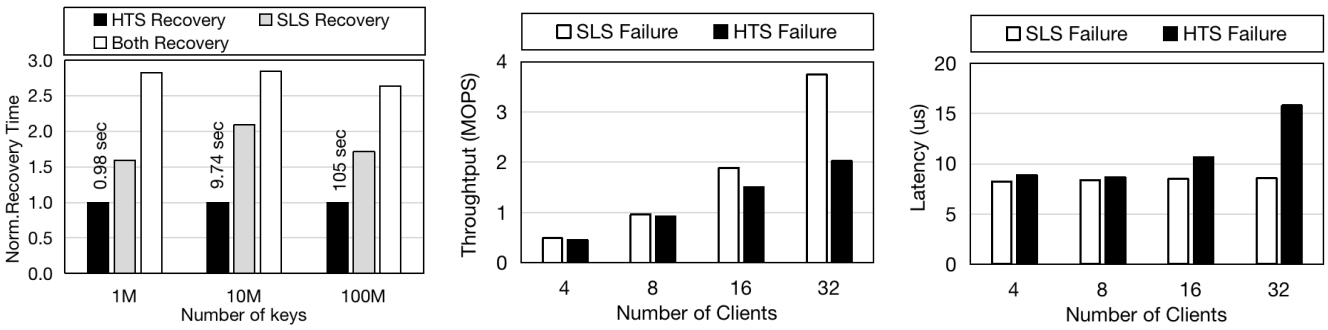
(b) Latency

Figure 10: Performance under the YCSB workloads: A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% reads for latest keys, 5% inserts), E (95% range queries, 5% inserts), and F (50% reads, 50% read-modify-writes). We use Zipfian distribution with the default skewness 0.99.

and 14.2-38.5% respectively under workloads A-D and F. HStore achieves better performance than Sherman and Clover, because it incurs fewer RTTs by using the hybrid index. For workload E with 95% range queries and 5% inserts, the throughput of HStore is a little lower than that of Sherman as HStore using two-sided verbs has worse scalability. Compared to HStore (sync), HStore achieves higher performance under write-intensive workloads (i.e., load phase, workloads A and F) and similar performance under other workloads. The results show that the asynchronous skiplist updates improve the write performance without degrading the scan performance. In summary, compared with the key-value stores based on a single index, HStore achieves better performance under workloads containing single-key lookups and similar performance to the sorted index under workloads involving range queries.

5.5 Recovery and Degraded Performance

We measure the recovery time of HStore in case of HTS and SLS failures, i.e., the latency of rebuilding a hash table or a skiplist. Figure 11a shows the recovery time of the HTS, SLS, and both of them. When the number of keys increases from 1 M to 100 M,



(a) Recovery time **(b) GET throughput** **(c) GET latency**

Figure 11: Recovery performance under index server failures.

it takes 0.98-105 s, 1.56-180 s, and 2.77-276 s to recover the HTS, the SLS, and both of them respectively. The repair time of the SLS is much longer, almost doubling the HTS' recovery time. The reason includes: 1) reconstructing a hash table takes less CPU cost than recovering a skiplist; 2) HStore can rebuild the hash table by scanning the skiplist, but needs to scan the key-value pairs in the memory pool to reconstruct the skiplist. Thus, the recovery of the HTS is faster than that of the SLS. HStore prioritizes the failure recovery process to avoid blocking the key-value services (e.g., write requests and range queries) too long.

We also evaluate the degraded performance of HStore when the HTS or SLS fails. HStore can serve single-point requests and range queries under the HTS failure, but can only process the GET operations under the SLS failure. When the HTS fails, the processing of a SCAN request is the same as in the normal case. Thus, we only show the performance of GET operations under different index server failures. Figures 11b and 11c show the throughput and latency of GET operation under the HTS and SLS failure, respectively. The SLS failure does not have any impact on the read performance, while the HTS failure degrades the read performance a lot because HStore needs to access the skiplist on the SLS. When the number of client threads varies from 4 to 32, the GET latency under the HTS failure increases that under the SLS failure by 7.90-85.03%, while the GET throughput under the HTS failure reduces that under the SLS failure by 8.16-45.99%.

6 RELATED WORK

Single index on RDMA. A large body of research on RDMA-based key-value stores in the literature has explored single index structures, either hash-based or sorted index. These works are orthogonal to HStore, i.e., HStore can utilize two different types of indexes (i.e., one hashing index and one sorted index) to combine their benefits in various key-value operations.

Many RDMA-enabled key-value stores leverage hashing index to achieve fast lookup services, and further optimize the usage of hash tables on RDMA [6, 9, 17, 19, 23, 28, 35, 39, 44]. Pilaf [28] uses a n-way cuckoo hashing algorithm to compute n different hash buckets for every key by n orthogonal hash functions, where $n = 3$ achieves the best memory efficiency. FaRM [9] proposes chained associative hopscotch hashing to achieve high space efficiency and a small number of RDMA reads for lookups. HydraDB [35] proposes a cache-friendly compact hash table based on a consistent hashing algorithm [21]. DrTM [39] presents cluster hashing which is similar to chained hashing with associativity. RACE [44] is a one-sided RDMA-conscious extendible hashing index that supports lock-free remote concurrency control and efficient remote resizing.

Some key-value stores adopt sorted indexes with RDMA to support range queries efficiently [7, 10, 20, 29, 37, 43]. The updated version of FaRM [10] that supports distributed transactions leverages B-Tree for range queries. Cell [29] proposes a hierarchical B-tree where a global tree consists of local trees. DrTM-R [7] provides an ordered store in the form of a B^+-tree in DBX [36]. Masstree+eRPC [20] extends Masstree [27], an in-memory ordered key-value store, with eRPC (an RDMA-based RPC library). Ziegler et al. [43] study different design alternatives for tree-based index structures on RDMA. XStore [37] maintains a B^+-tree index at the server, which is implemented by extending the B^+-tree index [36] with DrTM+H [38] (a hybrid RDMA framework). Sherman [34] is a write-optimized distributed B^+-tree using a local cache, and local and global lock tables based on one-sided verbs, to reduce the number of round trips during writes. Tebis [33] ships the B^+-tree index on the primary server to the backup servers over RDMA in LSM-based key-value stores.

Hybrid index in key-value stores. Existing works on hybrid index [3, 40] mainly consider the usage on one single machine, while HStore targets on distributed index based on RDMA. HiKV [40] proposes a hybrid index consisting of a hashing index [25] and a B^+-tree to enable fast index searching and range queries in DRAM and NVM. FloDB [3] presents a hierarchical memory design that is indexed by a small high-performance concurrent hash table and a larger concurrent skiplist [15]. NAM-DB [41] maps a value of the secondary attribute to a primary key using a hashing index and a B^+-tree, but it does not consider the consistency between the two different indexes and the efficient update of the indexes.

Memory disaggregation. The architecture of memory disaggregation separates computing resources from memory resources physically to allocate different resources on demand, thereby solving the problem of low memory utilization in data centers [13, 30]. Recently, many works [12, 14, 22, 24, 31, 32] study the deployment

of memory disaggregation in practice from the perspectives of resource management, performance optimization, data reliability, and hardware usage. Clio [14] adopts a combination of software and hardware to improve the performance of disaggregated memory. pDPM [32] explores the design of disaggregated persistent memory system and builds Clover, a key-value store based on a hashing index. Hydra [22] and FUSEE [31] revisit the issue of data reliability in disaggregated memory. DirectCXL [12] and Pond [24] utilize CXL to implement a high-performance memory pool. HStore stores KV data in the memory pool under the memory disaggregation architecture, while placing the hybrid index on the monolithic nodes to improve the indexing performance.

7 CONCLUSION

This paper presents HStore which leverages a hybrid index consisting of a hash table and a sorted index in an RDMA-based key-value store under the memory disaggregation architecture. HStore combines the benefit of hashing index and sorted index and stores the key-value pairs in the memory pool, to achieve efficient single-key lookups and range queries. To minimize the index lookup and update overhead, we dedicatedly use different RDMA primitives for read and write operations, and apply asynchronous updates to the sorted index while maintaining strong consistency among different index structures. Our evaluation results show that HStore efficiently manages the hybrid index with strong consistency and provides rich key-value services with low latency.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Ming Liu, and the anonymous reviewers for their comments. This work is supported by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92270202) and the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB44030200). The corresponding author is Mi Zhang.

REFERENCES

[1] Skiplist. https://github.com/begeekmyfriend/skiplist.
[2] STX B+ Tree. https://github.com/bingmann/stx-btree.
[3] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proc. of ACM EuroSys*, 2017.
[4] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distribute Systems. In *Proc. of USENIX OSDI*, 2006.
[5] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proc. of USENIX FAST*, 2020.
[6] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, and X. Liu. Nessie: A Decoupled, Client-Driven Key-Value Store Using RDMA. *IEEE Trans. on Parallel and Distributed Systems*, 28(12):3537–3552, 2017.
[7] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and General Distributed Transactions using RDMA and HTM. In *Proc. of ACM EuroSys*, 2016.
[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.
[9] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proc. of USENIX NSDI*, 2014.
[10] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. of ACM SOSP*, 2015.
[11] Google. LevelDB. https://github.com/google/leveldb.
[12] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proc. of USENIX ATC*, 2022.
[13] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who Limits the Resource Efficiency of My Datacenter: AnAnalysis of Alibaba Datacenter Tracess. In *Proc. of ACM IWQoS*, 2019.
[14] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang. Clio: A Hardware-software Co-designed Disaggregated Memory System. In *Proc. of ACM ASPLOS*, 2022.
[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* 2012.
[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX ATC*, 2010.
[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proc. of ACM SIGCOMM*, 2014.
[18] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proc. of USENIX ATC*, 2016.
[19] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proc. of USENIX OSDI*, 2016.
[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter RPCs can be General and Fast. In *Proc. of USENIX NSDI*, 2019.
[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM STOC*, 1997.
[22] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin. Hydra : Resilient and Highly Available Remote Memory. In *Proc. of USENIX FAST*, 2022.
[23] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proc. of ACM SOSP*, 2017.
[24] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proc. of ACM ASPLOS*, 2023.
[25] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of USENIX NSDI*, 2014.
[26] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *Proc. of USENIX ATC*, 2017.
[27] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proc. of ACM EuroSys*, 2012.
[28] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proc. of USENIX ATC*, 2013.
[29] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *Proc. of USENIX ATC*, 2016.
[30] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. of USENIX ATC*, 2019.
[31] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store (Extended Version). *CoRR*, abs/2301.09839, 2023.
[32] S. Tsai, Y. Shan, and Y. Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proc. of USENIX ATC*, 2020.
[33] M. Vardoulakis, G. Saloustros, P. González-Férez, and A. Bilas. Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. In *Proc. of ACM EuroSys*, 2022.
[34] Q. Wang, Y. Lu, and J. Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proc. of ACM SIGMOD*, 2022.
[35] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. HydraDB: A Resilient RDMA-Driven Key-Value Middleware for In-Memory Cluster Computing. In *Proc. of ACM SC*, 2015.
[36] Z. Wang, H. Qian, J. Li, and H. Chen. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proc. of ACM EuroSys*, 2014.
[37] X. Wei, R. Chen, and H. Chen. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proc. of USENIX OSDI*, 2020.
[38] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proc. of USENIX OSDI*, 2018.
[39] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast In-memory Transaction Processing using RDMA and HTM. In *Proc. of ACM SOSP*, 2015.
[40] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proc. of USENIX ATC*, 2017.
[41] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The End of a Myth: Distributed Transactions can Scale. In *Proc. of VLDB*, 2017.
[42] M. Zhang, Y. Hua, P. Zuo, and L. Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *Proc. of USENIX FAST*, 2022.
[43] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proc. of ACM SIGMOD*, 2019.
[44] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *Proc. of USENIX ATC*, 2021.