HiStore: Rethinking Hybrid Index in RDMA-based Key-Value Store

Shukai Han, Mi Zhang, Dejun Jiang, Jin Xiong
SKL Computer Architecture, ICT, CAS; University of Chinese Academy of Sciences
{hanshukai, zhangmi, jiangdejun, xiongjin}@ict.ac.cn

Abstract

RDMA (Remote Direct Memory Access) is widely exploited in building key-value stores to achieve ultra low latency. In RDMA-based key-value stores, the indexing time takes a large fraction (up to 74%) of the overall operation latency as RDMA enables fast data accesses. 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 reconsider the adoption of hybrid index in the key-value stores based on RDMA, to combine the benefits of hash table and sorted index. We propose HiStore, an RDMA-based key-value store using hash table for single-point lookups and leveraging skiplist for range queries. To maintain strong consistency in a lightweight and efficient approach, HiStore introduces index groups where a skiplist corresponds to a hash table, and asynchronously applies updates to the skiplist within a group. Guided by previous work on using RDMA for key-value services, HiStore dedicatedly chooses different RDMA primitives to optimize the read and write performance. Furthermore, HiStore tolerates the failures of servers that maintain index structures with index replication for high availability. Our evaluation results demonstrate that HiStore improves the performance of both GET and SCAN operations (by up to 2.03x) with hybrid index.

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 the storage engines [3, 6, 8, 12, 25, 31]. 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 studied to improve the performance of key-value stores in recent years, namely RDMA-based or RDMA-enabled key-value stores [13, 20, 26, 29, 42, 44]. RDMA communication provides two types of primitives, one-sided verbs allow to directly access 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 systems. As RDMA enables fast data access, the indexing performance largely determines the overall performance of single-point operations. Our analysis demonstrates that the indexing latency accounts for 49-74% of the total operation time (for PUT and GET) in an RDMA-based key-value store. Therefore, it is important to reduce indexing latency when building low-latency key-value stores based on RDMA.

Existing RDMA-based key-value stores typically leverage single index for key lookups, either hash-based [7, 13, 20, 22, 26, 29, 38, 42, 46] or sorted index (e.g., tree-backed, skiplist) [9, 14, 23, 30, 40, 45]. 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 [46] 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 hashing index. Thus, some RDMA-based key-value stores choose sorted index which maintains the order of key-value pairs to provide efficient range queries. Though sorted index supports rich key-value operations, sorted index incurs longer latency than hashing index for single-key lookups. Searching along a sorted index requires involvement of the server CPU to complete within one round trip; otherwise, it incurs multiple
round trips if only using one-sided RDMA reads. Our study shows that when there is no specific optimization, the indexing latency of sorted index can be as high as three times of the latency using a hash table.

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 [43] realizes the idea of hybrid index on a single server with hybrid memory and demonstrates the performance improvement of key-value operations. However, adopting hybrid index in RDMA-based key-value stores poses new challenges. First, the index management requires to keep different index structures consistent with the key-value items in an efficient manner. Meanwhile, if the storage system replicates the index for high availability, the replicas should be updated consistently. Thus, how to mitigate the management overhead of hybrid index with strong consistency remains an open issue. Furthermore, as failures in distributed systems are commonplace, the key-value stores using hybrid index should consider how to support key-value services and reconstruct the index in the present of failures.

We present HiStore, an RDMA-based key-value store which combines a hash table and a sorted index (i.e., skiplist) to support rich key-value operations and achieve high indexing performance. HiStore combines different indexes in one index group (a unit of hybrid index) to efficiently manage them with strong consistency. It allows hybrid-access using different RDMA primitives for CPU efficiency and low latency. That is, HiStore 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, while leveraging two-sided verbs for SCAN operations to reduce the number of round trips. To minimize the write latency, HiStore batches the index updates and applies the updates to the skiplist asynchronously. Moreover, HiStore achieves high availability with index replication to protect index structures against failures. To the best of our knowledge, HiStore is the first RDMA-based key-value store that leverages hybrid index for key-value services.

We summarize our contributions as follows.

- We implement HiStore using eRPC [23] in two-sided communications and conduct extensive experiments to evaluate its read and write performance.

We will make the source code of HiStore publicly available after this paper is accepted.

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 local and remote memory with kernel bypassing [21, 34]. 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 API, 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 to directly access 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. Note that data transfer based on two-sided verbs incurs CPU cost on the remote server.

Different architectures of RDMA-based key-value stores. A large number of research works [7, 14, 20, 22, 23, 29, 30, 38, 40, 42, 45, 46] have studied how to leverage RDMA 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, based on the usage of different RDMA verbs.

- Some works [20, 22, 23, 42, 45] adopt server-centric design by replacing the communication layer (e.g., RPC) in a key-value store with RDMA primitives. Figure 1a shows the workflow of processing a request, where the client sends a request to the server via RDMA network and the server returns the response after processing the request locally. Such design depends only two RDMA operations, one for sending request and one for receiving response, which simplifies the implementation because it only requires adding an RDMA-enabled communication module. However, 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.

- To bypass the CPU of the remote server, some systems [7, 37, 46] 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 network round trips 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 dealing with some complex data structures.

- Many existing works [14, 29, 30, 38, 40] 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). Thus, the hybrid-access designs not only harness the high performance of RDMA networks, but also relax 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 overall time of a key-value operation). Prior works have proposed many efficient index structures for key-value stores with RDMA. Here we summarize the characteristics of different index structures, either hash-based or sorted (e.g., tree-backed, skiplist), and the applications of the indexes in RDMA-based key-value stores.

**Hashing index.** Hashing index has been widely adopted in the RDMA-enabled key-values stores [7, 13, 20, 22, 29, 38, 42, 46], because the hashing index can provide fast lookup services. Figure 2a shows the data structure of a basic hash table, where a value is indexed by the hashing value of a key. 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 hashing index. However, the hashing index does not support range queries, i.e., SCAN operations, limiting its applications in building key-value stores.

**Sorted index.** To support efficient range queries, many RDMA-based key-value stores [9, 14, 23, 30, 40, 45] leverage sorted index (e.g., B+-tree, skiplist), which organizes the key-value pairs in an ordered manner. Figure 2b and 2c illustrate the data structure of B+-tree and skiplist respectively. 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. Also, the complex structure of the sorted index 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 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 Intel Xeon Gold 5215 CPU (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 previous study [6].

**Observation #1:** The RDMA-based systems using sorted index should leverage two-sided verbs if possible, to minimize the number of round trips without any specific optimization. Although some recent works [37, 45] explore to support 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 [37], a state-of-the-art distributed B+-tree optimized for writes. For read operations, both FG [45] and Sherman require to cache index locally on the client to avoid traversing the tree nodes, but there still remain some index lookup operations with 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

---

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

(a) Server-centric design  
(b) Client-direct design  
(c) Hybrid-access design
server, which equals to the amount 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\textsuperscript{+}-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 3a shows the number of memory accesses of sorted index and hashing index under different data amounts (1 to 100 millions key-value pairs). With larger number of key-value pairs, the number of memory accesses of B\textsuperscript{+}-tree and skiplist, increases from 3 to 10, because B\textsuperscript{+}-tree expands itself by adding more nodes while skiplist splits itself into more lists. For hash table, the number of memory accesses remains relatively stable (around 1) with 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. Therefore, if there is no constraint on the choices of RDMA commands (e.g., in case of disaggregated memory), we should employ two-sided verbs to support sorted index efficiently without modifying the data structures and adding other optimizations.

**Observation #2:** The index lookup performance of sorted index drops down a lot with more clients, as the server CPU becomes the bottleneck. We evaluate the indexing latency (time of index lookup over RDMA) of different indexes under different number of client threads. Based on observation #1, we implement B\textsuperscript{+}-tree and skiplist using eRPC [23] (an RPC library based on two-sided verbs), and access hash table via one-sided verbs. Here we load 100 millions keys to each index structure, and start different number of client threads issuing GET requests over all keys uniformly. Figure 3b illustrates the indexing latency of different index structures under various number of clients. Here the indexing latency includes the RDMA transfer time and the key lookup time. For the sorted index (i.e., B\textsuperscript{+}-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 number of clients because there is no CPU cost through one-sided verbs. The indexing latency of sorted index is almost three times of that of hash table when there are 32 client threads issuing requests to the server. Therefore, hashing index provides higher performance than sorted index, which motivates our idea of combining hashing index and sorted index in RDMA-based key-value stores.

**Observation #3:** The indexing latency accounts for a large proportion of the overall operation time. We further quantify the percentage of indexing latency in the overall operation time (the total time of a complete PUT/GET operation), which includes index insert/lookup and value insert/fetch over RDMA. We start 32 client threads to write/read 100 millions key-value pairs uniformly. Figure 3c and 3d depict the percentage of indexing latency in the overall latency of PUT and GET operations respectively, where the latency of writing/reading KV means the time to write/read a key-value pair using one-sided verbs once the index identifies the value location. For the sorted index, the indexing latency includes the time of
index insert/lookup and network transfer through RPC. As we access the hash table using one-sided verbs, the indexing latency equals to the time of index insert/lookup shown in the figure (without RPC). The indexing latency of $B^+$-tree and skiplist accounts for 70-74% of the whole process, while the indexing latency with hash table is about half of the total time. Hence, reducing the indexing latency with hybrid index can significantly improve the overall performance of a key-value store based on RDMA.

However, it is non-trivial to realize the idea of hybrid index efficiently in RDMA-based key-value stores. The first challenge is to preserve strong consistency between the hashing index and the sorted index, the index structures and the key-value items. Also, if the system replicates the index and data for fault tolerance (e.g., using three-way replication), the replicas should be consistent with each other. Thus, we should carefully perform the synchronization between different index structures, as RDMA networks can aggravate this problem (e.g., when updating the hybrid index, only one index is updated successfully while the other fails to be updated due to network interruption). Second, though the read-only operations 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 hybrid index remains an open problem. Last but not least, when failure occurs (e.g., one index server crashes), it is challenging to support all key-value services efficiently and achieve fast index recovery based on the remaining index structures.

3 Design of HiStore

In this section, we first introduce an intuitive approach of hybrid index which is hard to efficiently maintain strong consistency between different index structures. We then present our hybrid index scheme and build a RDMA-enabled key-value store called HiStore using hybrid index.

3.1 An Intuitive Approach

One natural approach to leverage hybrid index in RDMA-based key-value stores is using one-sided verbs to access the hashing index while employing two-sided verbs to access the sorted index, driven by our previous observations (in Section 2.3). Figure 4 shows the intuitive design of building a key-value store based on hybrid index, e.g., a hash table and a skiplist. To distinguish the server maintaining index from that storing data, we call the server managing index as index server, and the server storing values as data server. There can be multiple index servers and many data servers in a key-value store. Here we store the hash table in some index servers and place the skiplist in the other index servers. The client chooses to access different index structures based on the type of read operations. That is, the client directly accesses the hash table for GET operations while leveraging the skiplist for range queries. During writes, the client needs to update both the hash table and the skiplist using one-sided and two-sided verbs respectively.

Such approach seems to maximize the system performance, but poses unique challenges to maintain consistency between the two indexes. One one hand, this design bypasses the server CPU as much as possible (i.e., read/write the hash table using one-sided verbs) and minimizes the number of round trips (i.e., access the skiplist via two-sided verbs). One the other hand, as the two indexes are totally separate, partitioning the data according to the hash values of the keys helps address the issue of skewed access while the ranges of the key-value pairs can be maintained by the sorted index. However, it is hard to implement a lightweight mechanism to keep the hashing index consistent with the sorted index in such a design. The root problem is how to atomically and efficiently update the two different indexes. It can be prevalent in distributed environment that one index is updated successfully but the other one is not, caused by the network interruptions or server crashes. Thus, the value can be indexed by one of the indexes or none of them. One solution to the inconsistency problem is to employ distributed transactions [5], which requires to access the transaction table during writes. The system can judge the validity of an update based on the transaction table, and then complete/roll back the whole operation. This complicated method introduces additional overhead to the write process, degrading the write performance.

3.2 Hybrid Index Scheme

3.2.1 Index Groups

To efficiently manage the hybrid index with strong consistency, we propose index group which combines a part of hash table and a range of sorted index (e.g., a tree-backed index or a skiplist). An index group is a unit of hybrid index. The key idea here is to associate one index with another by in-
Index groups. As shown in the intuitive approach, it is hard to update the two different index structures in an efficient way, if there is no any relation between the hash-based index and the sorted index. Hence, we map the sorted index to the hash table in the same index group, such that the indexes in one group can get updated consistently. This idea is similar to primary-backup replication in the hybrid transaction/analytical processing (HTAP) systems [35], where the OLTP and OLAP workloads run on primary and backup replicas separately. Following the naming of primary-backup replication, we call the index server managing hash table as primary server and the index server keeping skiplist as backup server, because GET operations which will be served by the hash table dominate in real-world key-value workloads [3].

Figure 5 depicts how the index groups combine two different indexes. Here we show three index groups consisting of a hash table and a skiplist. As each index group is responsible to manage a set of hash values, the index groups are independent with each other. When inserting one key, it is first allocated to an index group based on its hash value, and then inserted to the skiplist with two-sided verbs to keep the order of the keys. For example, a key “brand” belonging to the second index group is inserted to the hash table of index group 2 and the corresponding skiplist.

3.2.2 Index Updating

For write requests (PUT, DELETE, UPDATE operations), HiStore updates both the hash table and the sorted index belonging to an index group to keep them consistent. Compared to updating a hash table, 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, HiStore performs asynchronous updates to the sorted index like HiKV [43] to keep the write latency low, while synchronously updates the hash table for single-key lookups.

In case of index server failures, HiStore 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 to the local index structure. When receiving a write request from a client, the primary server first records the update in its local log and sends the update to the backup server; the backup server stores the update in its log, replies to the primary server, and asynchronously applies the update to the skiplist; after receiving successful response from the backup server, the primary server updates its hash table and returns to the client. As the sorted index is updated asynchronously, the backup server updates the index based on its log before answering SCAN requests for strong consistency. In other words, HiStore supports serializability that the written items can always be accessed during single-point reads and range queries.

3.2.3 Consistency Guarantee

HiStore maintains strong consistency from two perspectives: (i) consistent index structures in each index group, 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 consistent with the hash table in the same group. A complete index update means that the update has been recorded in the logs on both the primary server and the backup server, and the hash table gets updated. As the log on the primary server and the backup server is 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, HiStore 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 a data server and gets the value address from the data server; then chooses an index group based on the hash value of the key, and connects to the primary server for index update. If any step fails during the write process, the write operation fails and the client can restart a write operation.

3.2.4 Choices of RDMA Verbs

HiStore uses different RDMA verbs for different index structures to achieve low latency. For single-key lookups, HiStore allows the client to directly access the hash table on the primary server using one-sided verbs, which bypasses the primary server CPU. For range queries, the client sends the request to the backup servers maintaining sorted index via two-sided verbs, which can be completed in one round trip.
3.3 Fault Tolerance

Our basic approach of hybrid index cannot tolerate index server failures as an index group consists of only one hash table and one sorted index. Though the sorted index can be considered as a backup of the hash table, neither the hash table nor the sorted index has an exact copy. That is, only a hash table or a skiplist remains when one index server in a group fails. If the primary server fails, the backup server keeping the skiplist requires to provide single-key lookups, which increases the latency of GET operations. In case of the backup server failure, the remained hash table cannot support SCAN operations until the completion of rebuilding the sorted index. Thus, we should address the issue of fault tolerance in hybrid index for high availability.

We consider adding one backup index structure to a group which basically consists of a hash table and a sorted index, as distributed systems typically use three-way replication to achieve high availability in the presence of failures [16, 32]. The added index can be a hash table or a sorted index. Figure 6 shows the two possible backup strategies for hybrid index. If we replicate the hash table in one additional backup server, there are two same hash tables and one skiplist in an index group. As shown in Figure 6a, the primary server and one backup server manage two hash tables, and the other backup server keeps the skiplist. Thus, when the primary server fails, the backup server with the hash table can directly act as the primary server. Moreover, the backup server containing the hash table can also provide single-point lookups, which relieves the request burden (for GET operations) on the primary server in some degree. However, this approach cannot tolerate the failure of the index server with skiplist. Another alternative is to replicate the skiplist on one additional backup server, which tolerates the failure of one index server with skiplist but fails to tolerate the primary server failure. When the primary server fails, one backup server with skiplist needs to support single-point operations until the primary server is repaired. Figure 6b depicts such backup strategy where the primary server maintains a hash table while two backup servers keep two same skiplists. In this case, both the two backup servers with skiplist can support range queries.

HiStore replicates the skiplist in an index group by default, i.e., the primary server keeps a hash table while two backup servers maintain skiplists. First, the hybrid index can provide rich key-value operations in case of any index server failures in a group. The failure of one backup server does not affect the functioning of the whole system, i.e., the hash table on the primary server and the remained skiplist are sufficient to answer all requests. If the primary server fails, the backup server can process single-point requests though the GET performance reduces without the hash table. Second, HiStore can quickly rebuild a hash table to recover the primary server, restarting to support GET requests efficiently. As the recovery time of a hash table is much shorter than that of a skiplist, we prefer to rebuild a hash table to reduce the impact of server failures on the whole key-value store. Third, the SCAN performance can benefit from one additional server because the management of skiplist relies on the server CPU. If replicating hash table on one additional server, the performance improvement of GET requests is limited, as the operations based on one-sided verbs do not involve the server CPU. Therefore, HiStore configures one hash table and two skiplists in one index group to achieve high availability and low latency.

**Putting it all together.** Figure 7 shows the architecture of HiStore, which consists of multiple index groups and many data servers. The index groups manage the hybrid index while the data servers are responsible for storing the values. In each index group, a primary server maintains a hash table for single-key lookups based on one-sided verbs while two backup servers keep the skiplists for range queries using two-
sided verbs. During writes, the client first stores the values on the data servers, and then asks the primary server to update the index using two-sided verbs. HiStore asynchronously applies the update to the skiplists on the backup servers to improve the write performance.

4 Implementation

We implement HiStore in C++, which realizes two-sided communications based on eRPC [23]. In this section, we first introduce the threading model and data structures of hybrid index in HiStore. We then explain how HiStore provides key-value services and recovers hybrid index when failure occurs.

4.1 Hybrid Index

Threading Model. HiStore 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 process RPC requests, and worker threads, which perform index updates and key lookups. Figure 8 depicts the threading model of hybrid index. During writes, the worker threads on the primary server update the hash table while the worker threads on the backup servers apply updates to the skiplist. For range queries, the worker threads on the backup servers look up the keys among the sorted lists. We explain how the RPC threads and worker threads support each key-value operation in Section 4.2.

Data Structures. HiStore currently adopts a chained hash table and a basic skiplist to constitute an index group. 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 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 next pointer is not empty, the next bucket is queried based on the next pointer. When a bucket is full during writes (e.g., hash collision occurs), the client issues an RPC request for adding a bucket to the server, which links a new bucket after the last bucket using the next pointer; after receiving the successful response from the server, the client can rewrite the key-value pair via one-sided verbs. To avoid resizing (which introduces one more RDMA round trip), we allocate more buckets than required by consuming a little more memory space. For the skiplist, HiStore divides the whole list to 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.

4.2 Key-value Services

Write operations. HiStore uses two-sided verbs to perform index updating and data storing. For PUT and UPDATE operations, the client first stores values on the data servers 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 primary server. The RPC threads on the primary server append the updates to different logs based on the hash values of the keys. The worker threads then send the updates to the backup servers using two-sided verbs, and wait for the responses from the backup servers. To improve the write throughput, the worker threads perform log synchronization between the primary server and the backup servers in a batch. On the backup server, the RPC threads append the updates to the log and send successful responses to the primary server, 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 backup servers, the worker threads on the primary server apply the updates to the hash table and return success to the client. To handle concurrent writes and reads, the primary server updates the hash table by compare and swap operation with CPU, which makes each update an atomic operation. If any step fails during writes, the incomplete write operation is considered as a write failure; the client can restart the write process if it does not receive successful response from the primary server after a period of time.

GET. HiStore handles GET requests using one-sided verbs, totally bypassing the server CPU. It leverages the hash table on the primary server 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 data server according to the value address. The whole process avoids incurring CPU cost on both the index server and the data server.

SCAN. HiStore provides efficient range queries based on the skiplists. The client submits SCAN requests via eRPC to the backup servers, 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 none index updates remains 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.

4.3 Failure Handling

When an index server fails, HiStore still provides key-value services and starts a recovery process to rebuild the failed index on the new index server. In case of the primary server
failure, one backup server acts as the primary server temporarily to process all writes while the other backup server handles the range queries; the GET requests is distributed to any backup server randomly. To rebuild a hash table, the new primary server retrieves the skiplist from the temporary primary server via eRPC. When the hash table is rebuilt completely, the new primary server can handle key-value operations and the index group can work normally. When a backup server fails, another backup server processes all SCAN requests. The new backup server fetches the hash table from the primary server via eRPC to generate a skiplist, and works as a normal backup server when its skiplist is up-to-date.

5 Evaluation

5.1 Experiment Setup

We evaluate the performance of HiStore on a local cluster. Our local cluster consists of five servers, each of which runs CentOS Linux release 7.6.1810 with 4.18.8 kernel and is equipped with two Intel Xeon Gold 5215 CPU (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 data storage, one to send requests as a client, and deploy an index group consisting of a primary server and two backup servers on the remaining machines. We use db_bench [15, 17] and YCSB [10] benchmark to evaluate the performance of HiStore. We run five times for each experiment and plot the average result.

We implement another two systems called all-hashtable and all-skiplist for comparison, as none existing system realizes hybrid index based on RDMA. The all-hashtable keeps three hash tables on three servers, while the all-skiplist maintains a skiplist on each server. For PUT operations, both of the systems work as HiStore where the primary server sends the index updates to the backup servers for asynchronous updates, and updates the local index on the primary server. The all-hashtable allows the client to directly access the hash tables via one-sided verbs, but does not support range queries. In the all-skiplist, the client sends GET and SCAN operations to a randomly-chosen server using two-sides verbs. Moreover, we compare HiStore with a single server running a hash table (i.e., single-hashtable) or a skiplist (i.e., single-skiplist). For the hash table used, we allocate more buckets to avoid resizing, i.e., the amount of keys that the hash table can store is higher than the number of keys to store.

5.2 Performance of Basic Operations

To evaluate the performance of basic operations in HiStore, we first load 100 millions (100 M) key-value pairs with key size of 16 B and 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 performance of basic operations under different number of client threads ranging from 4 to 64. We plot the throughput and latency of PUT, GET, and SCAN operations in Figure 9 and 10.

For the PUT operations, HiStore achieves similar performance to all-hashtable, and higher performance than all-skiplist. Compared to single index, the performance of HiStore is lower than that of single-hashtable, and higher than that of single-skiplist when there are more than 48 client threads. Figure 9a and 10a show the throughput and latency of PUT operations. The workflow of processing a PUT operation in HiStore is almost the same as that in all-hashtable and all-skiplist. The only difference is that HiStore and all-hashtable update the hash table, while all-skiplist update the skiplist on the primary server. Thus, the PUT performance of all-skiplist is lower than that of HiStore and all-hashtable when there are more than four client threads, because updating a skiplist takes longer time than updating a hash table. The PUT throughput of HiStore is 1.45 times of that of all-skiplist when there are 64 clients. The single-hashtable achieves the lowest latency among all schemes. Compared with the single-hashtable, HiStore increases the latency by 50-124% because HiStore requires to send the index updates to other backup servers and wait for their responses. When the number of clients is small, HiStore achieves lower performance than single-skiplist as single-skiplist only needs to update one skiplist. The latency
of single-skiplist increases with the number of clients because the server CPU becomes the bottleneck. HiStore reduces the latency of single-skiplist by 15-20% when the number of client threads exceeds 48.

HiStore achieves the best read performance among all schemes by leveraging its hybrid index for GET and SCAN operations. Figure 9b and 10b depict the throughput and latency of GET operations. The GET performance of HiStore is similar to that of single- hashtable and all- hashtable, because the client can directly access the hash table using one-sided verbs in all three schemes. HiStore achieves higher performance than both all-skiplist and single-skiplist. The latency of GET operations in all-skiplist and single-skiplist increases with the number of clients, as these two schemes incur CPU cost during key lookups. Note that the latency of single-skiplist increases significantly while that of all-skiplist increases slowly, meaning that three skiplists running on three servers relax the CPU burden on one server. The GET throughput of HiStore is 2.03 times that of all-skiplist under 64 clients. Figure 9c and 10c show the throughput and latency of SCAN operations. HiStore achieves similar SCAN performance to single-skiplist and all-skiplist. The number of keys covered by each scan operation in our evaluation is 100. Although all three schemes use different number of skiplists for range queries, 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).

![Graphs showing throughput, latency, and proportion of HiStore, All-HashTable, All-Skiplist, Single-HashTable, and Single-Skiplist for basic operations](image)

**Figure 9:** The throughput of basic operations.

**Figure 10:** The latency of basic operations.

**Figure 11:** Performance breakdown of basic operations.

5.3 Microbenchmarks

We conduct microbenchmarks for HiStore, all-hashtable, and all-skiplist using db_bench. We measure the time of the following phases when handling a request: (i) index rpc, the communication time between a client and an index server based on eRPC; (ii) queue wait, waiting in a queue to process; (iii) index access, performing update or search on an index; (iv) log sync, log synchronization between the primary server and backup servers during writes; and (vi) data access, storing values on data servers or retrieving data based on the value address. Figure 11 shows the performance breakdown of ba-
5.4 Performance under YCSB Workloads

We evaluate the performance of HiStore under different YCSB workloads. We use the default setting with key size of around 20 B, value size of 32 B, and a Zipfian request distribution with a Zipfian constant of 0.9. The number of keys requested by each scan operation is 100. We first load 100 M key-value pairs, and execute each workload by issuing 20 M requests.

Figure 12: Performance under YCSB workloads. This figure shows the throughput and latency normalized to the all-skiplist under YCSB workload A (50% reads and 50% updates), B (95% reads and 5% updates), C (100% reads), D (95% reads and 5% inserts), E (95% scan and 5% inserts), and F (50% reads and 50% read-modify-writes).

For PUT operations, the time of log sync takes the largest fraction (43-49%) of the whole latency in all three schemes. The latency of index access (i.e., inserting to a hash table) in HiStore and all-hashtable is quite short, which can be ignored during writes, while the time of updating a skiplist takes 4% of the total write time. HiStore has the same performance breakdown as all-hashtable for GET operations based on one-sided verbs, where the index access and data access takes 48% and 52% of the read time respectively. For all-skiplist, the index rpc and queue wait take 70% of the read time in total, while data access only takes 21%. When handling a SCAN operation, HiStore and all-skiplist spend about 96% of the time in data access while the remaining time is used for index queries.

5.5 Recovery and Degraded Performance

We measure the recovery time in case of the primary server and backup server failures, i.e., the latency of rebuilding a hash table or a skiplist. Figure 13 shows the recovery time of the primary server and a backup server. When the number of keys increases from 10 M to 100 M, it takes 14-155 s and 15-176 s to recover the primary server and a backup server respectively. Compared to rebuilding the hash table, the reconstruction of the skiplist is longer; it incurs more CPU cost.

We also evaluate the degraded performance of HiStore when the primary server or one backup server fails. Figure 14 illustrates the degraded throughput and latency normalized to the normal performance of HiStore. In case of primary server failure, the PUT and GET performance reduces, because HiStore needs to update a skiplist instead of a hash table and provides single-key lookups based on the skiplist. When a

HiStore under six YCSB workloads respectively, which are normalized to that of all-skiplist. The performance of HiStore is close to that of all-hashtable. Compared to all-skiplist, HiStore increases the throughput by 139-196% under the workloads without range queries, because HiStore supports GET operations using the hash table. For workload E that involves 95% range queries and 5% insert, HiStore achieves similar performance to all-skiplist by leveraging the skiplist in the hybrid index.
backup server fails, the PUT performance increases as the primary server only requires to update one backup server, while the GET performance is not affected. The performance of range queries does not change in the presence of single-server failures, as there is at least one index server with skiplist.

### 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 structure, either hash-based or sorted index. These works are orthogonal to HiStore, i.e., HiStore can utilize two different types of indexes (i.e., one hashing index and one sorted index) to combine their benefits.

Many RDMA-enabled key-value stores leverage hashing index to achieve fast lookup services, and further optimize the usage of hash tables on RDMA [7, 13, 20, 22, 26, 29, 38, 42, 46]. Pilaf [29] uses a n-way cuckoo hashing algorithm [33] to compute n different hash buckets for every key by n orthogonal hash functions, where n = 3 achieves the best memory efficiency. FaRM [13] proposes a chained associative hopscotch hashing [19] to achieve high space efficiency and a small number of RDMA reads for lookups. HydraDB [38] proposes a cache-friendly compact hash table based on the consistent hashing algorithm [24]. DrTM [42] presents cluster hashing which is similar to a chained hashing with associativity. RACE [46] 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 index with RDMA to support range queries efficiently [9, 14, 23, 30, 40, 45]. The updated version of FaRM [14] that supports distributed transactions leverages B-Tree for range queries. Cell [30] proposes a hierarchical B-tree where a global tree consists of local trees. DrTM-R [9] provides an ordered store in the form of a B+-tree in DBX [39]. Masstree+eRPC [23] extends MashTree [28], an in-memory ordered key-value store, with eRPC (a RDMA-based RPC library). Ziegler et al. [45] study different design alternatives for tree-based index structure on RDMA and a B+-tree using local cache, local and global lock tables based on one-sided verbs, to reduce the number of round trips during writes. Tebis [36] explores to ship 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 [4, 43] mainly consider the usage on one single machine, while HiStore targets on distributed index based on RDMA. HiKV [43] proposes a hybrid index consisting of a hashing index [27] and B+-tree to enable fast index searching and range queries in DRAM and NVM. FloDB [4] presents a hierarchical memory design which is indexed by a small high-performance concurrent hash table [11] and a larger concurrent skiplist [18]. NAM-DB [44] 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 efficient update of the indexes.

### 7 Conclusion

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

### References

[1] Skiplist. https://github.com/begeekmyfriend/skiplist.

[2] STX B+ Tree. https://github.com/bingmann/stx-btree.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of A Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[4] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proc. of ACM EuroSys*, 2017.

[5] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distribute Systems. In *Proc. of USENIX OSDI*, 2006.

[6] 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.

[7] 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.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of USENIX OSDI*, 2006.
[9] 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.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In Proc. of ACM SoCC, 2010.

[11] T. David, R. Guerraoui, and V. Trigonakis. Asynchronous Concurrency: The Secret to Scaling Concurrent Search Data Structures. In Proc. of ACM ASPLOS, 2015.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon’s Highly Available Key-value Store. In Proc. of ACM SOSP, 2007.

[13] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In Proc. of USENIX NSDI, 2014.

[14] 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.

[15] Facebook. RocksDB. http://rocksdb.org/.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In Proc. of ACM SOSP, 2003.

[17] Google. LevelDB. https://github.com/google/leveldb.

[18] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers Inc., 2012.

[19] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In Proc. of ACM DISC, 2008.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In Proc. of ACM SIGCOMM, 2014.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In Proc. of USENIX ATC, 2016.

[22] 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.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter RPCs can be General and Fast. In Proc. of USENIX NSDI, 2019.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Hashing Protocols for Relieving Hot Spots on the World Wide Web. In Proc. of ACM STOC, 1997.

[25] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu’s Key-Value Storage System for Cloud Data. In Proc. of IEEE MSST, 2015.

[26] 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.

[27] 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.

[28] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In Proc. of ACM EuroSys, 2012.

[29] 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.

[30] 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.

[31] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In Proc. of USENIX NSDI, 2013.

[32] D. Ongaro and J. K. Ousterhout. In Search of an Understandable Consensus Algorithm. In Proc. of USENIX ATC, 2014.

[33] R. Pagh and F. F. Rodler. Cuckoo Hashing. Journal of Algorithms, 51(2):122–144, 2004.

[34] L. RDMA. RDMA Core Userspace Libraries and Daemons. https://github.com/linux-rdma/rdma-core.

[35] S. Shen, R. Chen, H. Chen, and B. Zang. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In Proc. of USENIX OSDI, 2021.

[36] 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.
[37] Q. Wang, Y. Lu, and J. Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In Proc. of ACM SIGMOD, 2022.

[38] 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.

[39] 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.

[40] X. Wei, R. Chen, and H. Chen. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In Proc. of USENIX OSDI, 2020.

[41] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In Proc. of USENIX OSDI, 2018.

[42] 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.

[43] 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.

[44] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The End of a Myth: Distributed Transactions can Scale. In Proc. of VLDB, 2017.

[45] 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.

[46] 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.