Shard: A Scalable and Resize-optimized Hash Index on Disaggregated Memory

Hantian Zha
Renmin University of China
zhahantian1206@ruc.edu.cn

Teng Ma*
Alibaba Group
sima.mt@alibaba-inc.com

Baotong Lu
Microsoft Research
baotonglu@microsoft.com

Yuansen Wang
Renmin University of China
menphina@ruc.edu.cn

Dongbiao He
CNIC, CAS
dbhe@cnic.cn

Yuanhui Luo
Renmin University of China
losk@ruc.edu.cn

Dafang Zhang
Renmin University of China
dfzhang@ruc.edu.cn

Yunpeng Chai
Renmin University of China
ypchai@ruc.edu.cn

Yuxing Chen
Tencent Inc., China
axingguchen@tencent.com

Anqun Pan
Tencent Inc., China
aaronpan@tencent.com

ABSTRACT

Disaggregated memory (DM) separates memory and computing resources into distinct pools, improving resource utilization, scalability, and data sharing in data centers and cloud environments. These systems utilize RDMA-capable networks, which provide high throughput and low latency, making them well-suited for high-performance indexing in data storage systems. However, existing DM-optimized hash indexes face significant challenges in achieving the one RTT goal due to excessive remote read/write accesses, correctness issues in concurrent operations, high latency during resizing, and costly multi-node synchronization.

This paper addresses these challenges by introducing a novel architecture called Shard, designed to enhance the performance of hash indexes in disaggregated memory. We leverage the structure of Iceberg Hashing to ensure that each key is mapped to fewer buckets. We propose the Ordered-CAS technique to minimize read/write accesses and ensure correctness when handling duplicate keys. To address the trade-offs between resizing and synchronization, we adopt a lazy resizing strategy and propose the RDMA-combining and adaptive frequency synchronization (AFS) techniques. We implement Shard and conduct a comprehensive evaluation on DM. The results show that Shard outperforms state-of-the-art DM-optimized hash indexes by at most 6.7× (RACE), 3.6× (SepHash), and 1.8× (Outback) in YCSB workloads, respectively.

PVLDB Reference Format:
Hantian Zha, Teng Ma, Baotong Lu, Yuansen Wang, Dongbiao He, Yuanhui Luo, Dafang Zhang, Yunpeng Chai, Yuxing Chen, and Anqun Pan. Shard: A Scalable and Resize-optimized Hash Index on Disaggregated Memory. PVLDB, 19(4): 684 - 697, 2025.
doi:10.14778/3785297.3785309

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785309

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zhtshr/Shard.

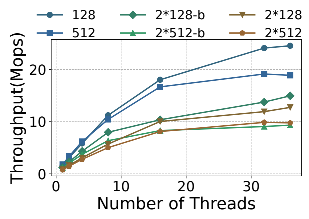
1 INTRODUCTION

Recent advancements in disaggregated memory (DM) decouple compute and memory nodes, organizing them into distinct compute and memory pools [20, 55, 63], and interconnecting these two resource pools via CXL [1] or RDMA [21]. This architectural shift is gaining popularity in modern data centers [17, 21, 60] due to its ability to improve resource utilization, achieve flexible hardware scalability, and facilitate efficient data sharing [34]. These systems typically leverage RDMA-capable networks for communication due to their notable advantages such as high throughput (100–400 Gbps), low latency (2–3 μs), and the ability to bypass remote CPUs and kernel [16, 61, 79].

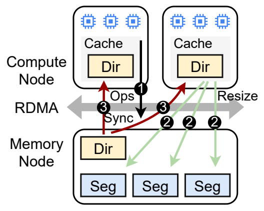
Within DM architecture, disaggregated hash indexes are widely employed in high-performance data storage systems, including databases [60, 73], key-value stores [16, 40, 59, 70, 84], and file systems [42]. In a DM architecture, network round-trip time (RTT) often becomes a performance bottleneck. Excessive RTTs and large data volumes can saturate the NIC's packet rate and bandwidth [26, 32, 70]. To study the effect of packet rate and data size on throughput, we test the read throughput of RDMA NIC with different configurations, as shown in Figure 1a. In existing works [43, 84], 128 bytes and 512 bytes are standard data transfer volumes per access because once the data block size per access exceeds 512 bytes, network bandwidth becomes the bottleneck [43]. We have found that less RTT always achieves higher throughput, even if the total data size read is greater. This phenomenon arises from each thread sending RDMA requests frequently, causing the NIC packet rate to

Table 1: The features of different hash indexes. (The index marked with an * indicates that it was originally a single-machine index and has been adapted for DM. For variable-length keys/values, a pointer is required, adding 1 RTT. Duplication indicates whether the index correctly handles duplicate inserted keys, missing indicates whether there will be data loss during data movement. The bucket group refers to a collection of buckets, whose combined size is smaller than a segment.)

Type	RTTs/latency			Correctness		Resizing			Sync
	search	insert	network	duplication	missing	granularity	strategy	latency	
FaRM	∼1/low	∼4/high	One-sided	✓	✓	hash table	blocking	high	low
CLevel*	4/high	5/high	One-sided	✓	✗	segment	non-blocking	high	high
RACE	2/medium	5-6/high	One-sided	✗	✗	segment	non-blocking	high	low
SepHash	2-3/medium	3-4/medium	One-sided	✓	✓	segment	blocking	medium	low
Outback	1/medium	1/medium	RPC	✓	✓	segment	blocking	medium	low
Iceberg*	∼3/medium	∼5/high	One-sided	✓	✓	bucket group	blocking	low	high
SHARD	∼1/low	∼2/low	One-sided	✓	✓	bucket	blocking	low	low



(a)



(b)

Figure 1: (a) RDMA-read throughput of an RDMA NIC with different RTTs and data sizes. (x refers to read x bytes, 2*x-b refers to 2 RTTs with doorbell batching, and 2*x refers to 2 RTTs.) (b) Hash architecture on DM. ❶ Basic operations. ❷ Resizing operations. ❸ Synchronization of the local cache.

fall behind the threads' sending frequency [28, 68]. Consequently, this discrepancy leads to the blocking of several threads.

Existing works [26, 32, 36, 41, 43, 65, 70, 84] propose various optimization solutions to reduce RTTs. As shown in Figure 1b, These solutions typically involve co-designing basic operations, resizing, and cache synchronization. Basic operations may require multiple RTTs for correctness, resizing involves modifying remote memory, and cache synchronization adds another RTT. Achieving a single RTT for basic operations on a hash index is challenging.

FaRM [16] uses hopscotch hashing [23] with lock-free reads, reducing read-path RTTs to 1.04, but writes require a lock, potentially triggering cascading data displacements. RACE [84] employs cuckoo hashing to resolve conflicts, requiring multiple reads per operation. Its re-read method for inserts and concurrent write scheme reduce blocking but may introduce errors. SepHash [43] uses a multi-segment structure to reduce resizing latency and improve insert performance, but increases RTTs due to more reads. Outback [36] achieves one-RTT communication using RDMA-RPC but offloads computation to memory nodes, creating a CPU bottleneck [22, 65].

To achieve the one RTT goal, there are the following challenges:

1) Correctness in concurrent operations. To improve system scalability, existing DM-optimized indexes [12, 84] employ a lock-free design for various index operations, even including the hash table resizing. However, due to the latency of network messages, we observe existing designs incur concurrency inconsistency [51, 65, 80, 84], causing duplication and missing errors. Some concurrency techniques, such as checksum, versioning, FaRM cache line versions, and dirty bit [10, 16, 17, 80], require non-negligible time or space overhead, which introduces more network roundtrips to basic operations.

2) High-latency and blocking resizing operation. In existing designs [12, 43, 84], resizing is processed at a coarse granularity. These designs typically employ a single thread to do the data movement. This process also causes other operations to be blocked, even if it is handed off to a background thread, it still introduces contention and leads to numerous retries. All of these factors contribute to high tail latency and a drop in throughput during resizing.

3) Dilemma between resizing and synchronization. After resizing, the compute nodes need to synchronize the metadata from remote memory to obtain the latest index structure. If a coarse-grained resizing strategy is employed, adjusting the local depth [43, 84] within the bucket can aid other threads in promptly verifying resizing. Alternatively, opting for a fine-grained resizing approach to minimize blocking during resizing necessitates an RDMA READ in each operation to validate potential metadata modifications, introducing an additional RTT. Therefore, there is a trade-off between resizing and synchronization.

We present the SHARD, a Scalable And Resize-optimized Hash index on Disaggregated memory, to achieve these goals. We leverage the structure of Iceberg Hashing [6, 48] to access fewer buckets. Iceberg Hashing is RDMA-friendly because it needs to access only one bucket on average during queries. It is also low-associativity, which means that each key is only mapped to a few positions. This enables us to look up one bucket with an extremely high probability (e.g., > 97%) during queries. To address the duplication error, ❶ we propose the **Ordered-CAS** technique. For the missing error, we employ a blocking resizing method. Simply using coarse-grained blocking would cause threads highly blocked, so ❷ we opt for a **lazy fine-grained blocking** approach and introduce the **RDMA-combining** technique to minimize resizing blocking time. The fine-grained approach renders the original synchronization method based on local depth invalid. To address this, ❸ we propose the

Adaptive Frequency Synchronization (AFS) technique to reduce synchronization overhead. We achieve the goal of one RTT while addressing all the previously mentioned challenges. Table 1 shows the summary of all the RDMA-based hash indexes. Iceberg hashing requires locking nodes to ensure correctness, which incurs significant overhead in DM. Previous work [2, 43] implemented CLevel on DM, so we include it in the comparison.

We implement Shard and evaluate its performance using the widely-used YCSB benchmark [15, 72]. Experiments show that Shard can achieve over 15.5 million read-only requests per second. This result shows that Shard outperforms state-of-the-art DM-optimized hash indexes by at most 6.7× (RACE), 3.6× (SepHash), and 1.8× (Outback) in YCSB workloads, respectively. Overall, Shard offers lower latency and better scalability than prior works.

The main contributions of this paper are as follows:

- We systematically identify three critical challenges for implementing correct and highly scalable hash indexing on DM. Subsequently, we conducted an in-depth analysis of the underlying causes of these challenges (§2).
- We propose four techniques to address these issues, including Ordered-CAS, lazy-resizing, RDMA-combining and AFS. Our Shard achieves the one RTT goal in basic operations, correctness, non-blocking resizing, and lightweight metadata synchronization (§3).
- We implemented Shard and evaluated its performance under YCSB and real-world benchmark (§4).

2 BACKGROUND AND MOTIVATION

In this section, we study the characteristics of DM in Section 2.1, then explore the existing disaggregated hash indexes and hash indexes that can be applied to DM in Section 2.2 and Section 2.3. Finally, we examine the existing hash index structures to motivate our study in Section 2.4.

2.1 Disaggregated Memory (DM)

DM architectures [8, 52, 75] separate computing and memory resources into distinct pools [30, 31, 74]. The computing pool contains multiple CPUs (e.g., 100) and limited DRAM (e.g., 1 GB), while the memory pool offers vast memory (e.g., 100-1000 GB) but fewer, weaker CPU cores (e.g., 1-2). These pools are connected via high-performance networks like InfiniBand and CXL [13]. RDMA, commonly used over InfiniBand, allows compute nodes to access remote memory with memory semantics. We focus on RDMA-enabled DM [29, 32, 39–41, 65, 76, 77], which provides one-sided operations—RDMA *READ*, *WRITE*, and *ATOMIC* (e.g., CAS, FAA). RDMA works on a post-polling mechanism where users post requests to a send queue, which are executed in order. Through doorbell batching [51, 71], multiple RDMA operations can be bundled into one request. The RDMA NIC asynchronously reads/writes data from remote memory, while the completion queue is polled to track operation status.

2.2 DM Based Hash Index

Disaggregated hash usually stores hash tables in each memory node, and the compute nodes have several working threads. High-speed RDMA networks interconnect the compute nodes and memory nodes. Typically, compute nodes will cache data that are less frequently modified to speed up the operation.

FaRM [16]. FaRM uses a hopscotch-hashing table [23]. As shown in Figure 2 (a), A lookup inspects at most H neighbors and verifies consistency by checking the version counter embedded in every cache line. During insertion, if no empty entry is found, entries are shifted to preserve the H-invariant. Every modified cache line's version is incremented. To optimize both space and lookup cost, FaRM keeps the hopscotch neighborhood small (H = 8) and introduces overflow buckets to host entries that cannot be placed within it. If neither an empty slot nor a successful shift can be found, the table is resized: the entire hash table is locked, all data is migrated, and every thread is blocked until the operation completes.

CLevel [12]. CLevel manages hash tables with a lock-free *level list*. As shown in Figure 2 (b), both insertion and search operations traverse the hash tables sequentially from lowest to highest level. The search returns data from the highest level, while insertion clears duplicates and adds the data to the topmost bucket. If no space is available, a larger hash table is allocated at the top and added to the global level list. A background thread continuously migrates data to consolidate hash tables until only two levels remain. In DM implementations [2, 43], the level list is cached on the compute node, and local memory access is modified to remote memory access.

RACE [84]. RACE uses an extendible hash scheme with a hash table made up of multiple segments and a directory. As shown in Figure 2 (c), the directory contains a *global depth* variable indicating the hash suffix length for indexing segments. Each segment has a *local depth* variable and contains several buckets, each holding 8 entries. Buckets share an *overflow bucket* to handle hash conflicts. After inserting a KV, RACE re-reads to check for duplicate keys. When a bucket is full, the entire segment is resized. RACE supports concurrent insertion, allowing data migration and insertion to happen in parallel. The directory is cached on the compute side to reduce remote access time for compute nodes.

SepHash [43]. SepHash uses extendible hashing with a separate segment structure to reduce resizing blocking. Figure 2 (d) shows the overall architecture of SepHash. Each directory entry has a small *CurSegment* and a large *MainSegment*. The CurSegment stores metadata (*CurSegMeta*) to filter non-existent fingerprints, while the MainSegment holds an *FPTable* for fingerprint range lookup. Searches involve both segments. SepHash uses append-only operations for insertions, updates, and deletions. When the CurSegment reaches capacity, it merges with the MainSegment, and when the MainSegment is full, it splits into two. Two fingerprints (fp and fp2) are used to improve key filtering. KV metadata (bitmap and fp2) is written asynchronously during insertion. To reduce RTTs, FPTables are cached locally.

Outback [36]. Outback uses the design of Ludo hashing [56]. Ludo hashing is a type of dynamic minimal perfect hashing (DMPH) that can achieve no hash collisions. A bucket locator distributes keys into different buckets. Ludo hashing uses brute force to find a hash seed, allowing keys to be mapped to different slots without collision. As shown in Figure 2 (e), Outback caches *bucket locator* and *bucket seeds* at compute nodes, enabling local bucket address computation. The search/insert/update/delete operations are sent via an RDMA-RPC request to achieve one RTT. Threads in memory nodes need to perform address read/write and resizing operations.

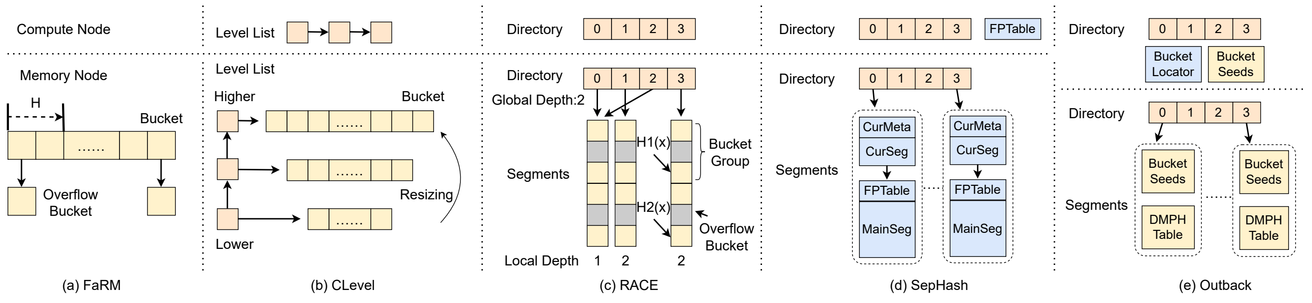


Figure 2: Hashing on DM.

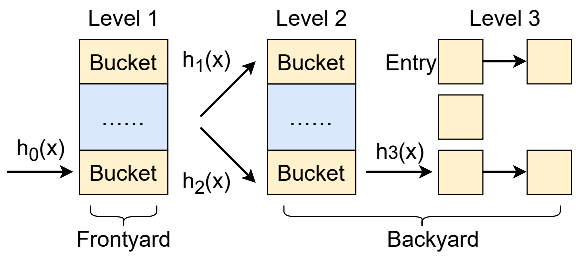


Figure 3: The iceberg hashing structure.

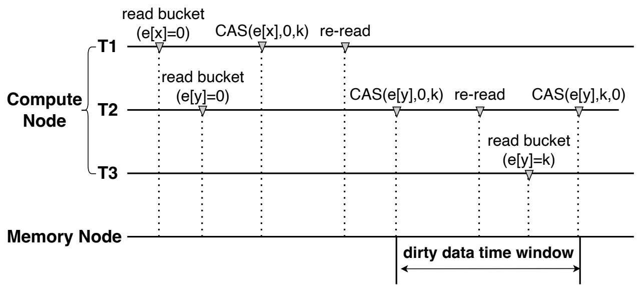


Figure 4: An error case of duplication.($e[x]$ and $e[y]$ are inserted duplicated key k.)

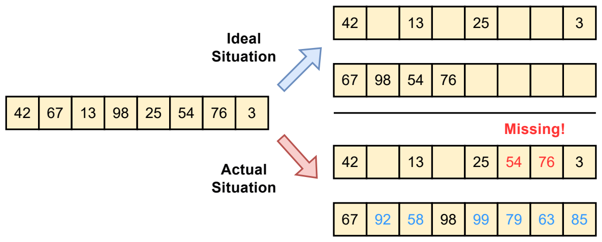


Figure 5: Missing error during resizing. (The blue keys are inserted by other threads and the red keys are unable to be moved to the new segment.)

2.3 Iceberg Hashing on DM

Iceberg hashing [6, 48] is a novel hash index aimed at optimizing time and space consumption simultaneously. As shown in Figure 3, Iceberg hashing adopts a three-level structure: the first level is the *frontyard*, which stores most of the elements (> 97%); the next two levels are the *backyard* used to store overflow elements. There is about $\frac{1}{log(n)}$ of the data in the backyard, hence most queries only need to access the first level. The operation process of Iceberg hashing is also very simple, prioritizing access to the first level, followed by the second and third levels. Iceberg hashing exhibits low associativity, which restricts each element to map to a small number of positions. This limitation reduces the number of memory locations to be examined when performing queries, thereby improving search efficiency.

The advantages of Iceberg hashing can also be applied to DM, i.e., read and write operations only access a bucket in most cases. However, it still faces some challenges in DM: 1) how to ensure correctness when data may be inserted into different layers; 2) the methodology for executing efficient resizing across distributed nodes; 3) a high-performance mechanism to synchronize metadata.

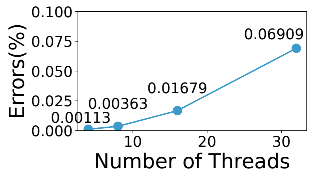
2.4 Challenges in DM Based Hash Index

Designing a correct and efficient hash index on DM is challenging due to the characteristics of DM and RDMA networks. An in-depth analysis of the three challenges associated with the design of hash indexes will be conducted, with a summary of disaggregated hash indexes presented in Table 1.

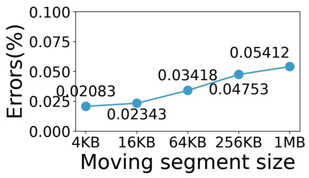
Insight 1: On DM where high throughput is pursued, read-write algorithms must be carefully designed to guarantee consistency. We observed that in previous works [12, 84], there were concurrency controls that could lead to inconsistency.

We found a **duplication** error in the design of RACE. The re-reading mechanism may lead to inconsistencies. Figure 4 shows the error case. When two threads concurrently write to the same key while another thread is reading, a dirty read may occur. We tested this scenario with the Zipfian YCSB workload (skewness = 0.99) in Figure 6a. The index has a non-negligible probability of encountering errors.

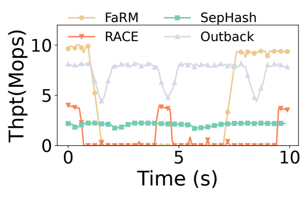
The **missing** error occurs in CLevel and RACE due to their non-blocking methods, as shown in Figure 5. In non-blocking mode, the insert operation thread competes with the resizing thread for empty entries, preventing the resizing thread from finding enough space



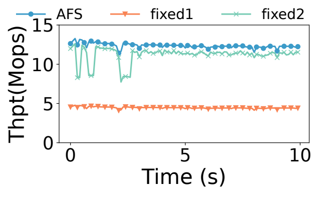
(a) Duplication error



(b) Missing error



(c) Realtime-indexes



(d) Realtime-Shard

Figure 6: Percentage of (a) duplication errors and (b) missing errors. A comparison of real-time throughput under write-only workloads: (a) different indexes and (b) different sync strategies of Shard (fixed1: freq=1, fixed2: freq=10000).

to move data. An experiment with 16 threads (Figure 6b) confirms this behavior, showing an error rate of 2.1%-5.4% across different segment sizes.

Insight 2: Fine-grained resizing can improve throughput. Since the non-blocking resizing strategy will cause the missing error, we consider the blocking approach. We fixed the bugs in RACE by employing a blocking method and evaluated the real-time insert performance of the four indexes. Figure 6c shows the result. Outback achieves the highest write throughput, FaRM's and RACE's blocking during resizing leads to lower throughput. SepHash proposes a two-segment structure and the minimal blocking granularity (64KB), achieving minimal performance degradation. Therefore, a finer-grained resizing is desired in designing a DM-based hash index.

Insight 3: An appropriate synchronization strategy is required to boost throughput and reduce resizing latency. Fine-grained resizing reduces blocking time and can be done concurrently by multiple threads. However, this requires all threads to proactively synchronize metadata in order to obtain the latest table information. We evaluated the performance of two synchronization strategies in Shard: synchronizing metadata every 1 operation and every 1,000 operations. As shown in figure 6d, the fixed-1 strategy remains more stable during resizing but yields lower overall throughput. In contrast, the fixed-1000 strategy delivers higher throughput; however, because fewer threads participate in synchronization and more KVs are inserted into the overflow level, performance drops more sharply during resizing. Our goal is to achieve higher throughput while maintaining stable performance.

3 DESIGN OF SHARD

We present Shard, a resize-optimized hash table on DM. The design of Shard is guided by three primary goals: (1) reducing RTTs during read and write operations with guaranteed correctness, (2) using

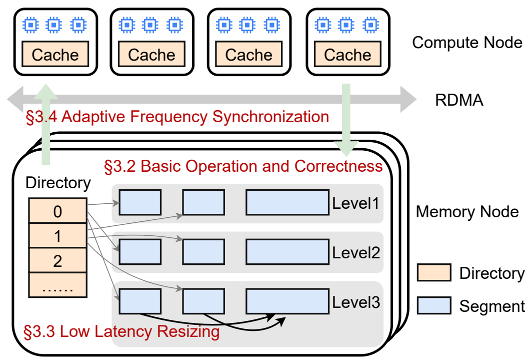

Cache
Cache
Cache
Cache
Compute Node
RDMA
§3.4 Adaptive Frequency Synchronization
§3.2 Basic Operation and Correctness
Directory
0
1
2
......
Level1
Level2
Level3
Memory Node
Directory
Segment
§3.3 Low Latency Resizing

Figure 7: The overall architecture of Shard.

fine-grained resizing to amortize the latency to all threads, and (3) efficiently synchronizing metadata. Specifically, the distinct features of Shard are described in the following.

- **Basic Operation and Correctness (§3.2) .** For the concurrent correctness of different threads and to reduce access overhead, we design the insertion/update/deletion operations as lock-free in-place modifications, where all read and write operations need to be executed in a fixed order (**Ordered-CAS**).
- **Low Latency Resizing (§3.3) .** In existing designs, the blocking time of resizing is not negligible. For this reason, we choose the **fine-grained and lazy resizing** approach. After that, we propose the embedded version number and **RDMA-combining** approach, which guarantees performance and correctness.
- **Adaptive Frequency Synchronization (§3.4) .** During resizing, an efficient synchronization is needed. Therefore, we designed the Adaptive Frequency Synchronization (**AFS**) algorithm, which enables each compute node to predict the time of resizing and execute synchronization promptly.

3.1 Architecture

Figure 7 shows the overall architecture of Shard on DM. Each memory node stores a separate hash table, which includes a directory and a three-level structure. We record the address of each segment in the directory. In the compute node, each thread maintains the directory cache to accelerate the retrieval process. Compute node use one-sided RDMA to perform read/write operations on the index.

We employ the design principles of iceberg hashing[6, 48] in Shard because this structure is RDMA friendly. To support resizing, each level consists of segments, and the segment number increases when resizing. Figure 8 shows the structure of segments in each level, a segment is composed of 2^b buckets (b is an adjustable parameter between 10-20). We don't need to record any metadata in segments and buckets, thus reducing space overhead. To minimize RTTs, the design allocates the maximum number of entries to buckets in level 1, which contain 64 entries. Subsequently, buckets in level 2 contain 8 entries, and those in level 3 are limited to a single entry. Consequently, the majority data is in level 1.

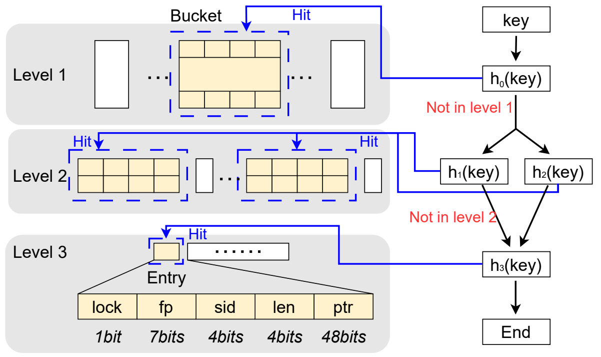


Figure 8: The segment structure and the layout of an entry (left), and the index operation workflow (right).

The layout of an entry is shown in Figure 8, the first 16 bits are used to record metadata which is used to further reduce operation latency. Each entry contains a 1-bit *lock* to lock a bucket during resizing, a 7-bit fp (fingerprint) to filter out different keys, a 4-bit *sid* to pre-know the segment id after resizing for acceleration like SepHash, and a 4-bit *len* to present the length of the KV block in 64B units. If the KV length exceeds this limit (1024B), we can store the next block pointer at the end of the block. Therefore, we can write metadata into an empty entry using only a CAS.

3.2 Basic Operation and Correctness

3.2.1 Basic Operation. To ensure correctness and minimize retrieval RTTs, we propose the **Ordered-CAS** technique. Each operation follows a predefined access sequence. The workflows of insert/search/update/delete operations are shown in Figure 8.

Insertion: Algorithm 1 shows the process of the insert operation. To insert a KV, the thread will prioritize insert in level 1. ❶ The thread first writes the KV block in the memory pool (line 6), ❷ then calculates $hash(key)$, and obtains the segment address from the local cache to get bucket address (line 7). ❸ After that, the thread read the bucket with an RDMA READ (line 8). ❹ The thread finds an empty entry from front to back of the bucket and writes the entry with an RDMA CAS (lines 12-14). If a CAS operation fails, it is necessary to check whether the already inserted key is identical. If so, the operation should switch to an update; otherwise, the insertion must continue searching for an empty entry. If there is no empty entry, the thread keeps inserting in level 2 and level 3 (lines 18-28). When the insertion operation still cannot find an empty position at the third level, we will use the open addressing method which has 8 entries in each chain node to tolerate hash collisions. The strategy of only searching for the first empty entry may leave redundant and invalid KV pairs in the index [10]. Fortunately, this scenario occurs infrequently and does not compromise correctness (we will substantiate it in section 3.2.2). Thus, we can eliminate these entries during the resizing process to reduce latency in basic operations. We will introduce the synchronization and moving during resizing in Section 3.3.

Search: The search process is similar to that of insertion, as shown in algorithm 2. ❶ The thread first calculates $hash(key)$ and

Algorithm 1 Insert(key,value)

```
1:  $thread.insert\_num+ = 1$
2:  $check\_sync()$                                   ▷ sync from remote
3:  if $load\_factor() \geq THRESHOLD$ then
4:      $resize()$                                    ▷ trigger resizing
5:  end if
6:  $new\_entry = write\_key\_value(key, value)$
7:  $segment, offset = hash(key)$
8:  $bucket = read\_lv1\_bucket(segment, offset)$
                      ▷ read the level 1 bucket using remote address
9:  if $!bucket.lock$ and $is\_dirty(bucket)$ then
10:     $move\_lv1\_bucket(bucket)$                    ▷ move in the first visit
11: end if
12: if $insert\_bucket(bucket, new\_entry) == true$ then
13:     return                                        ▷ insert into lv1 buckets
14: end if
15: if $!bucket.overflowbit$ then
16:     $set\_overflowbit(bucket)$                     ▷ insert oveflow
17: end if
18: for $i \leftarrow 1$ to 2 do
19:     $segment, offset = hash_i(key)$
20:     $bucket = read\_lv2\_bucket(segment, offset)$
21:     if $!bucket.lock$ and $is\_dirty(bucket)$ then
22:         $move\_lv2\_bucket(bucket)$
23:     end if
24:     if $insert\_bucket(bucket, new\_entry) == true$ then
25:         return                                    ▷ insert into lv2 buckets
26:     end if
27: end for
28: $insert\_lv3\_bucket(new\_entry)$                  ▷ insert into lv3 buckets
```

reads the bucket like `insert` operation (lines 2-3). ❷ Then the thread retrieves KV from front to back (lines 7-10). The thread will read the KV block if the fingerprint (fp) matches. If the KV is not found, the thread continues retrieving level 2 and level 3 (lines 11-22). The search operation returns the first matched key, but it will travel throughout all three levels if a key doesn't exist. To avoid unnecessary RTTs, we leverage an overflow bit in each bucket to indicate insertion overflow to the next level (line 8 and lines 15-17 in algorithm 1).

Update: To update a KV, the thread first writes the KV block, and then uses `search` operation to find the first matched key. Once the target key exists, the entry is modified using an RDMA CAS to refer to the new KV block.

Deletion: For correctness, the deletion thread should identify all the matched KV from level 3 to level 1, in reverse sequential order, then set the entries to zero with an RDMA CAS. When the last matched KV is deleted, the key is invisible, hence the deletion operation is atomic.

Under skewed workloads, write operations are subject to heavy contention, resulting in many RDMA CAS failures. To solve this problem, we employ the read delegation and write combining techniques in SMART [41], which uses a local lock table to resolve hash conflicts in the same thread.

Algorithm 2 Search(key)

```
1:  $check\_sync()$                                   ▷ sync from remote
2:  $segment, offset = hash(key)$
3:  $bucket = read\_lv1\_bucket(segment, offset)$
                             ▷ read the level 1 bucket using remote address
4:  if !$bucket.lock$ and $is\_dirty(bucket)$ then
5:      $move\_lv1\_bucket(bucket)$                   ▷ move in the first visit
6:  end if
7:  $value = search\_bucket(bucket, new\_entry)$
                                                      ▷ search lv1 buckets
8:  if $value \neq NULL$ || !$bucket.overflowbit$ then
9:      return $value$
10: end if
11: for $i \leftarrow 1$ to 2 do
12:     $segment, offset = hash_i(key)$
13:     $bucket = read\_lv2\_bucket(segment, offset)$
14:     if !$bucket.lock$ and $is\_dirty(bucket)$ then
15:         $move\_lv2\_bucket(bucket)$
16:     end if
17:     $value = search\_bucket(bucket, new\_entry)$
18:     if $value \neq NULL$ then
19:         return $value$
20:     end if
21: end for
22: return $search\_lv3\_bucket(new\_entry)$
```

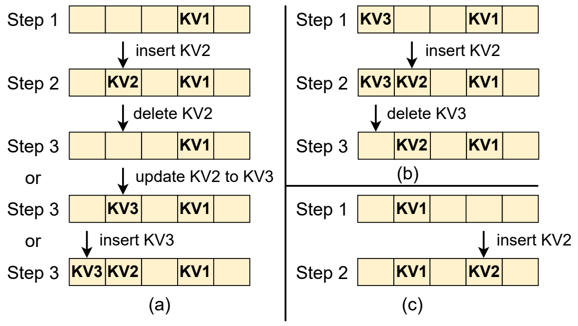


Figure 9: All possible scenarios. (KV1, KV2, and KV3 are in the same bucket and have the same key.)

3.2.2 Correctness. We will demonstrate that Shard is linearizable in the presence of mixed workloads involving search, insertion, update, and deletion, attributed to the protocol features of RDMA NIC [51]. We prove the linearizability by demonstrating the following two invariants are correct [24, 57].

I1: Multiple readers read the same value of a key. Readers can always get the same value simultaneously since they retrieve it in the same order.

I2: After any reader returns a new value, all subsequent readers must also return the new value. We prove the property of Shard by contradiction. Supposing that KV1 and KV2 have the same key but different values. The visible KV transitions from KV1 to KV2, and then back to KV1. There are three cases altogether.

Case 1: As shown in Figure 9 (a), the position of KV2 is in front of KV1. KV2 is visible when KV2 is inserted. KV2 becomes invisible when KV2 is deleted, or KV2 is updated, or the same key (KV3) is inserted in front of KV2. In the deleted case, KV1 was inserted first, the thread that deletes KV2 will also read KV1 because RDMA READ and RDMA CAS are not out-of-order[1] [51]. In this case, the deletion thread will delete KV1 first (delete in reverse order). Consequently, KV1 will not be visible again. In the other two cases, this conclusion also holds.

Case 2: As shown in Figure 9 (b), KV2 is in front of KV1, and KV3 is in front of KV2. KV2 is visible when KV3 is deleted. In this condition, KV1 is also invisible because readers will find out KV2.

Case 3: As shown in Figure 9 (c), KV2 is behind KV1. In this case, KV2 is invisible. Hence the visible KV would not transform to KV2.

This conclusion remains applicable to multi-level structures, ensuring accuracy without introducing extra costs.

3.3 Low Latency Resizing

Algorithm 3 Move_bucket(bucket)

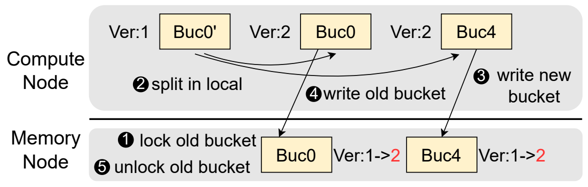
```
1:  if !$RDMA\_CAS(bucket.lock, 0, 1)$ then
2:      return                                        ▷ failed to get lock
3:  end if
4:  $new\_bucket = new\_bucket()$          ▷ allocate new bucket in local
5:  for $i \leftarrow 0$ to $bucket.len - 1$ do
6:      if $need\_move(bucket.entry[i].sid)$ then
7:          $new\_bucket.entry[i] = bucket.entry[i]$
8:          $bucket.entry[i] = 0$                     ▷ move to new bucket
9:      end if
10: end for
11: $RDMA\_WRITE(new\_bucket)$
12: $bucket.lock = 0$
13: $RDMA\_WRITE(bucket)$
```

When the number of KV entries reaches the threshold of load factor (0.8 of the total size), a resize is triggered. The first thread identifying this condition takes responsibility for resizing. The thread ❶ initially acquires a lock on the hash table, ❷ then allocates a new segment in every level of the remote memory pool, with the size of the new segment being the total of all preceding segment sizes. After that, ❸ updates the metadata in the remote memory pool, and ❹ subsequently releases the hash table lock. Afterward, half of the entries will be moved to new entries. All the threads should update metadata first in each operation (line 1 in algorithm 1 and algorithm 2). In case of metadata changing, they cannot continue with the operation until all threads finish the synchronization, otherwise the stale threads would cause consistency issues.

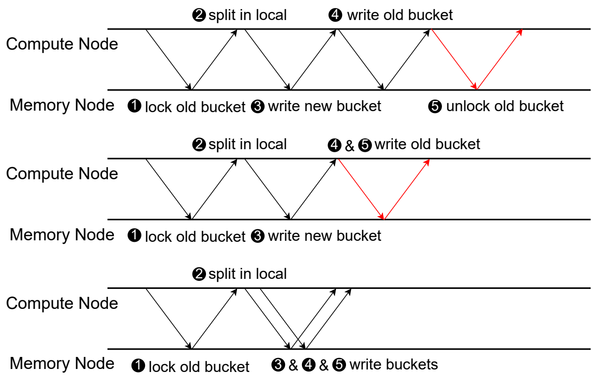
To reduce resizing latency and blocking time, we adopt the **lazy resizing** strategy, which amortizes data movement to all threads. The granularity of the data movement is similar to that of the search

[1]The invariant order is: insert KV1 using CAS, insert KV2 using CAS, and the delete operation must read both KV1 and KV2.

operation, which is a bucket. This task will be accomplished by the first thread that accesses it. Therefore, we can add moving buckets to the base operation (lines 8-10, 17-19 in algorithm 1 and lines 4-6, 14-16 in algorithm 2). During the moving process, we can detect redundant and invalid KV pairs and purge them without introducing additional overhead. Since the move operation is always executed before all insertion operations, we can avoid missing errors (mentioned in 2.4). We can distinguish which buckets should be moved by version numbers. Version numbers are related to global resizing times, so the needed bits are $O(log(log(N)))$. Consequently, we can embed the version number in lock bits for every 8 entries (the size is a cacheline).



(a) The workflow of moving data in a bucket.



(b) RDMA-combining. (red lines: the reduced RTTs)

Figure 10: The workflow of move and the RDMA-combining technique.

For instance, Shard has been resized twice, resulting in a global version of 2. The version number of $bucket_0$ is 1, which is behind the global version. Figure 10a and algorithm 3 shows the five steps of moving data from $bucket_0$ to $bucket_4$: ❶ lock the old bucket by an RDMA CAS (lines 1-3); ❷ distinguish which entries to move via segment ID (lines 5-10); ❸ write new data and version numbers to the new bucket using an RDMA write, each cacheline should be written new version numbers (line 11); ❹ write new data and version numbers to the old bucket; ❺ unlock the old bucket (lines 12-13). When a thread is moving, it reads the bucket into the compute node. Hence it doesn't need an additional RTT to read the bucket during basic operations.

We proposed **RDMA-combining** technology based on the moving process. Observation reveals that we could discriminate inconsistent entries with the version number in each cacheline, hence we

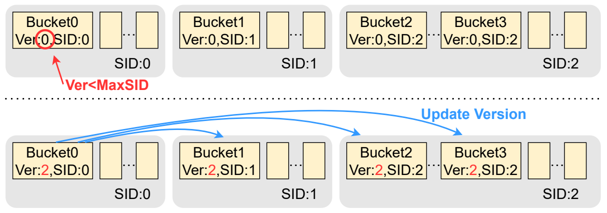


Figure 11: The corner case of splitting.

do not need to unlock at the end of the moving operation. Above all, the operation of writing and unlocking the old bucket can be combined into a single RDMA write. Furthermore, due to the order-preserving feature of RDMA write [16, 51], we can even write buckets at a time by using doorbell batching. Consequently, we need only two extra RTTs to move a bucket, as shown in Figure 10b, bringing little blocking time.

The current solution still has a corner case: when the version number is behind k ($k > 1$), there are 2^k possible locations for the key of the currently split bucket, as shown in Figure 11. So we need to extend the move strategy: since there are only $O(log(n))$ possible buckets for keys, and the version number of these buckets is monotonically decreasing. We can bisect the version number to find the stale bucket ($Version < MaxSID$) with the smallest segment ID. The smallest stale bucket is $bucket_{0-0}$ in Figure 11. Then we move entries to the correct locations separately and modify the version number in order. These buckets are visited only once, thus the overhead of amortization is low.

3.4 Adaptive Frequency Synchronization

Algorithm 4 Check_sync()

```
 1: $sync\_ferq = \alpha * table.size * (resize\_thr - load\_factor())$
 2: if $++thread.cnt >= sync\_freq$ then
 3:     $RDMA\_FAA(table.num, thread.num)$
                                        ▷ add inserted num to global
 4:     $RDMA\_READ(table.meta)$
 5:     if $changed(table.meta)$ then
 6:         $RDMA\_FAA(sync\_threads, 1)$
 7:         for $sync\_threads < total\_threads$ do
                              ▷ wait until all threads have synchronized
 8:             $RDMA\_READ(sync\_threads)$
 9:         end for
10:     end if
11:     $thread.cnt -= sync\_freq$        ▷ sync every sync_freq times
12: end if
```

To facilitate the detection of resizing and speedup operations, each compute node caches **metadata** consisting of segment count, bucket number, starting address of each segment and the number of inserted entries. This approach is akin to previous works [43, 84]. Metadata changes are only affected by resizing, except for the number of inserted entries. Therefore, threads don't need to synchronize in each operation. We can decrease the synchronizing frequency such as synchronizing in every f operation. Delaying

Table 2: YCSB workloads.

Workload	A	B	C	D	F
Type	R : U	R : U	R	R : I	R:RMW
Ratio	50 : 50	90 : 10	100	95 :5	50:50

synchronization will not affect correctness, because at this time, the new threads are waiting for synchronization to complete, only the old threads that have the same metadata are executing concurrently.

As shown in algorithm 4, each thread first adds its locally inserted entries to the global table's metadata and then reads the remote-side metadata (lines 3-4). If a resize occurs, the current thread must wait until all threads have synchronized (lines 5-10). The concrete implementation is that each thread atomically increments a fixed remote address by one and spins until the counter reaches the total number of threads.

How to choose the synchronization frequency is a dilemma. To reduce RTTs and improve overall throughput, a low synchronization frequency should be selected. However, an excessively low synchronization frequency forces threads that hold the latest metadata to block, waiting for stale threads to catch up to maintain correctness. This prevents the operation from proceeding. The move postponement will cause extensive insertion overflow to level 2 and even level 3, seriously hurting overall throughput.

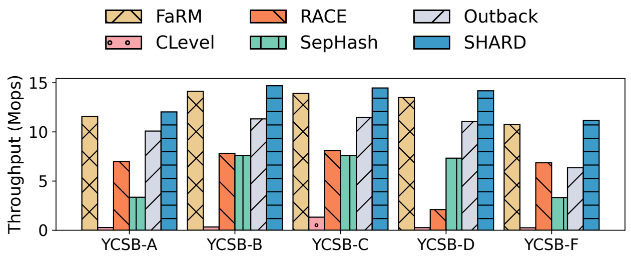
To address this problem, we propose the **AFS** technique by inspecting the relationship between resizing and synchronization. If the load factor of the hash table is low, the probability of resizing is also low, we do not need synchronization, and vice versa. We can adjust the synchronization frequency according to the probability by using a function. To simplify the designs, we use a linear function to fit it. Formally, for total size T, resize threshold R and the current load factor L, the synchronization frequency F is calculated as $F = \alpha T(R - L)$. Where α is a tunable parameter (0.1 by default).

3.5 Discussion

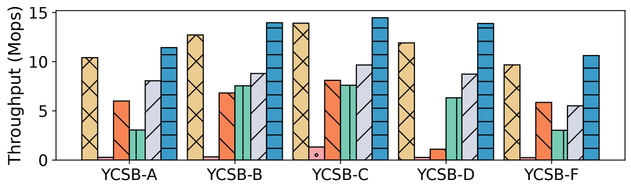
Fault tolerance. Our resizing strategy requires all threads to synchronize, but delays can occur due to factors like OS scheduling or hardware errors. To address this, we set a timeout for each waiting thread. For short delays, threads can wait briefly, but if the timeout is exceeded due to hardware issues, all threads reduce the metadata of total threads and continue processing requests.

Index placement across memory nodes. Our current strategy to solve index placement is to map hash values to different memory nodes. But if there is a demand for increasing or decreasing memory nodes, we should travel throughout all the memory nodes and choose a new hash function to rebuild the index. To tackle it, we can leverage the consistent hashing algorithm [27, 44] to reduce the data volume we should move.

Combining with other works. In our current implementation, searching variable-length keys still requires pointer chasing, which incurs an additional RTT. There exists an orthogonal approach (PRISM [7]) that enables single-RTT pointer dereference. We plan to incorporate this into our system in future work.



(a) Uniform (108 threads)



(b) Zipfian (108 threads, 0.99 skewness)

Figure 12: Comparison of throughput for various indexes using YCSB.

4 EVALUATIONS

In this section, we evaluate Shard to answer the following questions:

- How does Shard perform under different workloads, including real-world workloads, and how scalable is Shard when varying compute threads (4.2, 4.3, and 4.4)?
- How does Shard perform in realtime throughput (4.5)?
- How do the different techniques employed in Shard contribute to overall performance and throughput (4.6)?
- How does the parameter α affect performance (4.7)?
- What is the space consumption of Shard (4.8)?

4.1 Experiment Setup

Testbed. We run all experiments on three nodes. Each node has two 20-core Intel Xeon Silver 4114R CPUs clocked at 2.2GHz (each with 48.5MB of L3 Cache), 288GB DRAM across two sockets, and a 100Gbps Mellanox ConnectX-5 InfiniBand NIC connected to a 100Gbps InfiniBand switch.

We configure each machine to act as one compute node and one memory node. We allocate 36 cores for the compute node and use the remaining 4 cores for the memory node on each machine, similar to the settings used in previous works [39, 78]. Our experiments use all 108 (36×3) compute-side threads and 12 (4×3) memory-side threads in the cluster. Each thread is pinned to a physical core.

Workloads. Most experiments are based on standard workloads from the YCSB [72]. We mainly focus on YCSB because it has a variety of workloads [14]: update heavy (A), read mostly (B), read only (C), read latest (D), and read-modify-write (F), as shown in Table 2. When evaluating each index, we first insert one trillion data entries and then perform three trillion operations. Each KV pair consists of an 8-byte key and an 8-byte value.

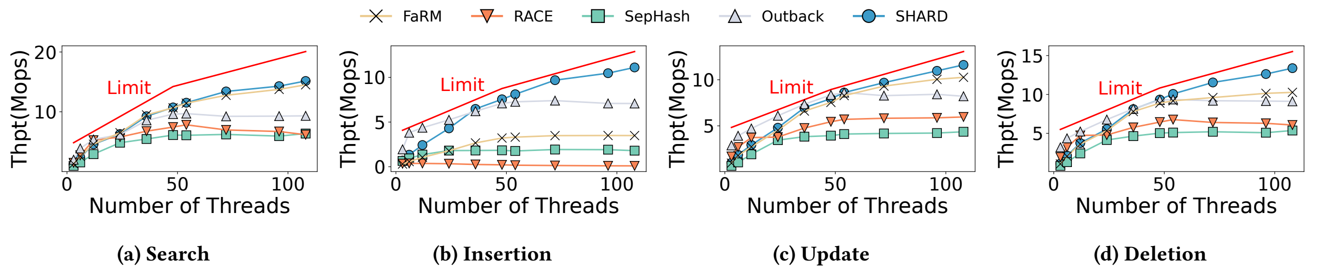


Figure 13: Comparison of scalability for various indexes under single operations. (The red line indicates the throughput limit.)

Comparisons. We developed a Shard prototype with over 4000 lines of C++ code and compared it to five distributed hash indexes. CLevel, a leveling hash index designed for persistent memory, was tested with its RDMA version. FaRM, RACE, SepHash, and Outback are distributed hash indexes for DM. Except for FaRM, these indexes were implemented from open-source projects [2, 3]. CLevel and RACE's concurrent insertion during resizing can cause missing errors, which we address with a blocking batch write. For Shard, α is set to 0.1 by default. Each test runs with 1-4 coroutines per thread, and the configuration with the highest performance is selected.

4.2 YCSB Benchmark

In this experiment, we use the YCSB benchmarks [15] to test the performance of these indexes. We run the benchmark with an uniform and a Zipfian (skewness = 0.99) access distribution. The results are shown in Figure 12.

For uniform distribution, Shard can achieve a throughput of 14.5 M/s. Shard outperforms FaRM, CLevel, RACE, SepHash, and Outback by 1.1×, 10.9×, 1.6×, 1.9×, and 1.2×, respectively. CLevel exhibits poor performance because it needs to access buckets in each level, requiring 4 RTTs. For update-intensive workloads (YCSB A and F), SepHash exhibits low throughput because its out-of-place update strategy triggers multiple resizing. RACE shows poor performance under YCSB D for its insert operations require multiple round-trip times (RTTs) to ensure correctness.

In Zipfian distribution, CLevel achieves lower throughput due to its need to access four buckets and its blocking resizing strategy. RACE's insert operations involve reading many duplicate keys during the re-read phase, and its in-place update policy leads to CAS failures, wasting RTTs. As a result, RACE performs poorly in workload D. However, Shard scales well in Zipfian distribution by using a local lock table to reduce RDMA CAS conflicts. In read-intensive workloads, Shard outperforms FaRM, CLevel, RACE, SepHash, and Outback by 1.2×, 12.9×, 1.8×, 1.9×, and 1.5×, respectively.

We excluded CLevel from subsequent tests due to its poor performance in RDMA architectures.

4.3 Performance of Single Operations

We evaluate the performance of basic operations for each index. The data distribution is a uniform access pattern. To quantify the gap between Shard and an ideal hash table, we test the system's throughput when reading 64-byte buckets and equally-sized KV blocks, simulating the theoretical upper bound of search operations.

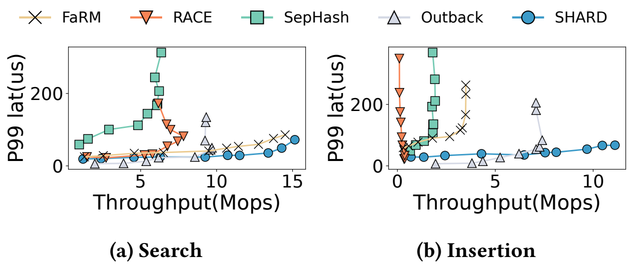


Figure 14: Comparison of tail latency for various indexes under single operations.

Table 3: Average round-trip times (RTTs) per operation. (The network round-trip of Outback is always 1 RPC.)

Index	Search	Insert	Update	Delete
RACE	2.03	12.79	3.03	3.03
SepHash	2.20	11.56	3.20	3.20
Shard	**1.00**	**2.59**	**2.00**	**2.00**

The same approach was applied to all other operations. Figure 13 and Figure 14 show the scalability and tail latency respectively.

Search. As shown in Figure 13a, Shard achieves the highest search throughput and scalability, improving 1.1×-2.5× over FaRM, RACE, SepHash, and Outback. This is because Shard requires only one RDMA READ, while SepHash and RACE require two (Table 3). When 16 threads per node are reached, the RDMA NIC packet rate saturates, limiting further throughput gains. RACE outperforms SepHash due to fewer RTTs (0.17 on average) and smaller data reads (128B vs. SepHash's 512B). Although Outback also uses one RTT, its reliance on RPC creates a bottleneck at the memory node's CPU, limiting scalability. Figure 14a shows that Shard has the lowest tail latency, while SepHash and RACE's latency increases with higher RDMA NIC contention.

Insert. Figure 13b shows that Shard outperforms FaRM, RACE, SepHash, and Outback by 1.3~29.6× in insertion throughput. Table 3 indicates that Shard requires only 2.59 RTTs for insertion. Despite SepHash using background threads for metadata writing, its NIC packet rate becomes a bottleneck. RACE requires 12.79 RTTs, with blocking during resizing causing significant performance degradation. FaRM's blocking resize strategy limits its scalability. Outback

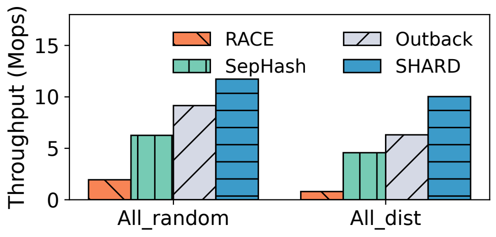


Figure 15: Real world workload.

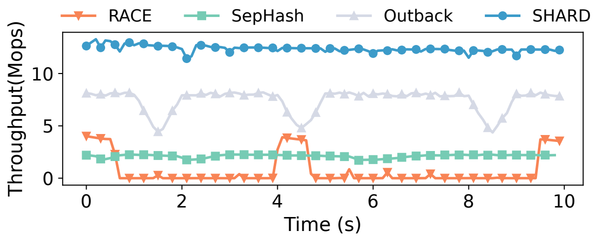


Figure 16: Real-time throughput.

uses bucket locking for inserts, leading to increased contention with more than 50 threads. Figure 14b shows that FaRM, RACE, and SepHash have much higher tail latencies than SHARD, with FaRM and RACE experiencing more blocking during resizing, and SepHash suffering from excessive RTTs.

Update. Figure 13c shows the update throughput of different indexes. When the number of threads exceeds 24, SHARD still has the highest throughput. Table 3 shows the RTTs of update operation, SHARD requiring 2 RTTs to update a KV entry, SepHash and RACE require 3.20 RTTs and 3.03 RTTs because the retrieval process requires one more RDMA READ. SHARD outperforms FaRM, RACE, SepHash and Outback by 1.2×-2.7×.

Delete. Figure 13d shows the deletion throughput of different indexes. SHARD achieves 1.4~2.5× higher throughput than FaRM, RACE, SepHash and Outback. The required number of RTTs is the same as for update operations in Table 3.

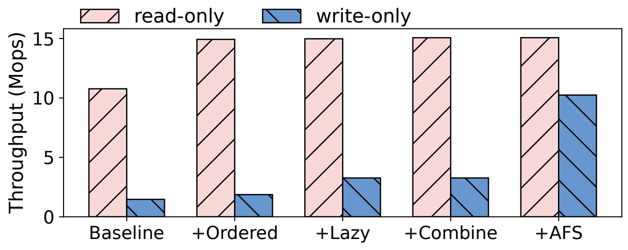
4.4 Real-World Workload

To test the performance of indexes on real-world datasets, we used the benchmark tools in RocksDB [9] and evaluated two workloads from ZippyDB, which follow the configuration described in the paper. Figure 15 shows the peak throughput of SHARD and other indexes on two real-world workloads. Performance is also mainly bottlenecked by RNIC RTTs. In datasets with hotspot workload (all_dist), the performance of all four indexing techniques decreases, but SHARD still maintains a significant advantage. In each workload, SHARD outperforms RACE, SepHash and Outback by 6.0×/12.6×, 1.9×/2.2×, and 1.3×/1.6× respectively.

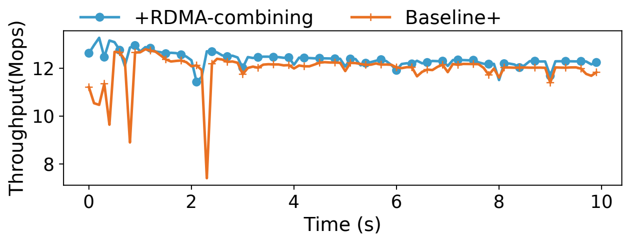
4.5 Realtime Analysis

To analyze the impact of resizing, we test real-time insertion throughput across different hash indexes, using the same workload as in Section 4.3. The results in Figure 16 show that SHARD outperforms all other schemes. It achieves at least 2.8×, 4.8×, and 1.4× higher throughput than RACE, SepHash, and Outback, respectively. The performance degradation during resizing is only 0.3s at worst, while SepHash takes 0.8s and RACE takes 4.8s to recover. SHARD avoids resizing blocking with fine-grained movement and mitigates throughput loss using RDMA-combining and AFS. SepHash reduces blocking time with a two-segment structure, while RACE faces significant throughput drops due to frequent CAS retries. Outback suffers a performance drop due to its coarse-grained blocking approach.

4.6 Step-by-Step Analysis



(a) Contributions of all techniques. (Baseline: naive implementation of Iceberg hashing, +Ordered: Ordered-CAS technique, +Lazy: lazy-resizing technique, +Combining: RDMA-combining technique, +AFS: AFS technique.)



(b) Contribution of RDMA-combining.

Figure 17: Contributions of techniques (uniform workloads).

To analyze SHARD 's performance, we break down the performance gap between a baseline and SHARD by applying each technique step by step. The **baseline** is a naive implementation of Iceberg hashing on DM. Figure 17a shows the results under uniform workloads. **+Ordered** stands for the Ordered-CAS strategy and optimization for minimizing the RTT. **+Lazy** is the lazy-resizing and the fine-grained moving techniques. **+Combining** represents RDMA-combining technique. **+AFS** indicates the AFS technique.

From Figure 17a, we make the following observations: In read-only workloads, Ordered-CAS improves throughput by 1.4×. Without Ordered-CAS, SHARD requires locking and optimistic reads to ensure consistency, adding 1-2 additional RTTs. In write-only workloads, SHARD's lazy-resizing strategy boosts throughput by 1.5×, eliminating resizing blocking and reducing remote reads. AFS increases throughput by 4.1× in write-only workloads. In highly

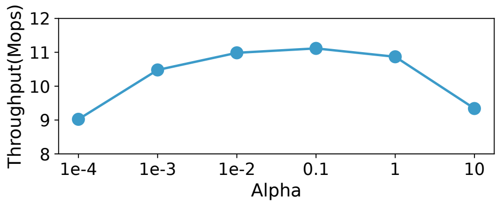


Figure 18: The sensitivity analysis for performance.

Table 4: Space overhead of different hash indexes. (Utilization refers to the ratio of the space occupied by the entries to the total used space.)

Index	#Entry	Utilization	Total(GB)	Cache(MB)
RACE	6.44 Ti	88.9%	314	108.00
SepHash	**7.82 Ti**	88.3%	319	326.05
Outback	4.29 Ti	97.0%	**290**	59.61
SHARD	4.83 Ti	**100%**	295	**2.36**

concurrent scenarios, an additional RTT per operation can significantly impact performance. With AFS, SHARD avoids extra RTTs for metadata synchronization during insertions, improving throughput and reducing RDMA NIC bottlenecks.

To evaluate RDMA-combining's contribution, we test real-time throughput in a write-only workload (Figure 17b). **Baseline+** represents SHARD without RDMA-combining. RDMA-combining improves worst-case throughput by 1.5× by reducing RTTs during resizing. While the total throughput improvement is modest, it helps stabilize performance by mitigating degradation during resizing.

4.7 Sensitivity Analysis

To analyze how the parameter α of SHARD affects its performance, we evaluate the peak insertion throughput under different α configurations, as shown in Figure 18. As α grows from 1e-4 to 10, the throughput continuously increases until reaching its maximum at 0.1, then steadily declines. When α is too small, the high synchronization frequency of basic operations causes throughput to decrease. When α is too large, delayed resizing triggers cause data overflow into the second and third layers, resulting in increased latency. Therefore, $\alpha = 0.1$ is the optimal configuration.

4.8 Space Consumption

In this experiment, we compare the load factor and space utilization between SHARD, SepHash and RACE.

Table 4 shows the space utilization of the indexes, as well as the number of entries and space overhead of each index after inserting 4 trillion KVs. SHARD does not record additional metadata in the bucket, so the space utilization is 100%. SepHash records a 1B fp2 for each entry, the space utilization is $\frac{113*8}{1024} = 88.3\%$. RACE records a 4B local depth and a 4B suffix in each bucket, The space utilization of RACE is $\frac{64}{72} = 88.9\%$. Outback stores a seed for every bucket (4 entries), gaining the space utilization of $\frac{32}{33} = 97.0\%$.

We also count the cache overhead of the indexes. SHARD consumes only 2.36MB per node, while RACE, SepHash and Outback require 108MB, 326.05MB and 59.61MB of cache, respectively, significantly higher than SHARD. This is because SHARD's directory structure occupies merely $O(log(s))$ space, s is the number of segments, whereas RACE, SepHash and Outback demand linear space with complexity $O(s)$.

5 RELATED WORK

Resource Disaggregation. Resource disaggregation has gained attention for improving utilization and scalability. Previous work has explored memory disaggregation in areas like operating systems [55], hardware architectures [34, 35], memory management [4, 53, 59, 60, 63, 69], networking [18, 47, 49, 58, 79], and emerging needs [5, 20]. SHARD focuses on index structure design in DM, complementing these works. With the rise of CXL, some studies have aimed to reconstruct RDMA-based disaggregated applications [50, 67]. SHARD is not limited to RDMA and can be applied to CXL-based disaggregated systems.

Disaggregated Data Structure. There is growing interest in indexing techniques for disaggregated memory systems, including hash and tree-based indexes. Sherman [65] and FG-Tree [81] adapt B+Tree structures for disaggregated memory. dLSM [66] uses LSM-Tree for better write performance, while Deft [64] reduces remote access granularity with unordered node structures. SMART [41] utilizes a radix tree to reduce read and write amplification. Learned indexes like XStore [70] and Rolex [32] use linear models for faster retrieval, with HTM and locks ensuring correctness. However, learned indexes on DM still have $O(log(n))$ time complexity. SHARD addresses amplification issues with novel designs, requiring only about one RDMA RTT per operation.

Hash indexes in different architectures. For DRAM-based hash indexes, MemC3 [19] uses a global lock for multi-reader, single-writer access in cuckoo hashing, while Libcukoo [33] applies fine-grained locking for multi-reader, multi-writer access. Several other proposals [11, 12, 25, 37, 38, 45, 46, 48, 54, 62, 82, 83] focus on concurrent hashing for persistent memory, primarily addressing local memory access. In contrast, SHARD tackles the challenge of concurrent access in disaggregated hash indexes.

6 CONCLUSION

This paper presents our observations on the three major challenges for RDMA-based disaggregated hash tables. Then we address these challenges and propose SHARD, which employs a high load factor structure to reduce remote memory access. Besides, it uses a lazy resizing method to alleviate the resize overhead and designs a metadata synchronization algorithm. The evaluations show that SHARD significantly improves operation throughput and latency compared with state-of-the-art works such as RACE, SepHash and Outback.

ACKNOWLEDGMENTS

We appreciate reviewers' constructive comments for improving the quality of this paper. This work is supported by the National Key Research and Development Program of China (No. 2023YFB4503600), National Natural Science Foundation of China (No. U23A20299).

REFERENCES

[1] 2022. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org.

[2] 2024. SepHash open-source code. https://github.com/minxinhao/SepHash.

[3] 2025. Outback open-source code. https://github.com/yliu634/outback.

[4] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 775–787.

[5] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 120–126.

[6] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg hashing: Optimizing many hash-table criteria at once. *J. ACM* 70, 6 (2023), 1–51.

[7] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan RK Ports. 2021. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 228–242.

[8] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 79–92.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[10] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[11] Qi Chen, Hao Hu, Cai Deng, Dingbang Liu, Shiyi Li, Bo Tang, Ting Yao, and Wen Xia. 2023. Eeph: An efficient extendible perfect hashing for hybrid pmem-dram. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1366–1378.

[12] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 799–812.

[13] Compute Express Link Consortium et al. 2020. Compute Express Link Specification.

[14] B. F. COOPER. [n.d.]. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*. 54–70.

[18] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. {NICA}: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 345–362.

[19] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 371–384.

[20] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 249–264.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 202–215.

[22] Shukai Han, Mi Zhang, Dejun Jiang, and Jin Xiong. 2023. Exploiting Hybrid Index Scheme for RDMA-based Key-Value Stores. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 49–59.

[23] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *International Symposium on Distributed Computing*. Springer, 350–364.

[24] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[25] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A hybrid PMem-DRAM persistent hash index with fast recovery. In *Proceedings of the 2022 International Conference on Management of Data*. 1049–1063.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[27] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[28] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding {RDMA} microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

[29] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory (Extended Version). *arXiv preprint arXiv:2209.08743* (2022).

[30] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[31] Haifeng Li, Ke Liu, Ting Liang, Zuojun Li, Tianyue Lu, Hui Yuan, Yinben Xia, Yungang Bao, Mingyu Chen, and Yizhou Shan. 2023. Hopp: Hardware-software co-designed page prefetching for disaggregated memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1168–1181.

[32] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. {ROLEX}: A Scalable {RDMA-oriented} Learned {Key-Value} Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[33] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[34] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news* 37, 3 (2009), 267–278.

[35] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.

[36] Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. 2025. Outback: Fast and Communication-efficient Index for Key-Value Store on Disaggregated Memory. *arXiv preprint arXiv:2502.08982* (2025).

[37] Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A performant extendible adaptive hashing index. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[39] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2603–2616.

[40] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 110–126.

[41] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. {SMART}: A {High-Performance} Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 553–571.

[42] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. 2022. {InfiniFS}: An efficient metadata service for {Large-Scale} distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 313–328.

[43] Xinhao Min, Kai Lu, Pengyu Liu, Jiguang Wan, Changsheng Xie, Daohui Wang, Ting Yao, and Huatao Wu. 2024. SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1091–1104.

[44] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 587–604.

[45] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. {Write-Optimized} dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[46] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[47] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. 2019.

Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 97–108.

[48] Prashant Pandey, Michael A Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[49] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous NIC offloads. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 18–35.

[50] Amit Puri, Kartheek Bellamkonda, Kailash Narreddy, John Jose, Venkatesh Tamarapalli, and Vijaykrishnan Narayanan. 2024. DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems. In *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 3–14.

[51] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. {RDMA} is Turing complete, we just did not know it yet!. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 71–85.

[52] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. 2024. Scaling Up Memory Disaggregated Applications with Smart. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 351–367.

[53] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[54] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. 1–8.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[56] Shouqian Shi and Chen Qian. 2020. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (2020), 1–32.

[57] Daniel Sorin, Mark Hill, and David Wood. 2011. *A primer on memory consistency and cache coherence.* Morgan & Claypool Publishers.

[58] Yupeng Tang, Seung-seob Lee, Abhishek Bhattacharjee, and Anurag Khandelwal. 2025. PULSE: Accelerating Distributed Pointer-Traversals on Disaggregated Memory. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 858–875.

[59] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated {Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.

[60] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 306–324.

[61] Jerome Vienne, Jitong Chen, Md Wasi-Ur-Rahman, Nusrat S Islam, Hari Subramoni, and Dhabaleswar K Panda. 2012. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. IEEE, 48–55.

[62] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2023. {SEPH}: Scalable, Efficient, and Predictable Hashing on Persistent Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 479–495.

[63] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.

[64] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2025. Deft: A scalable tree index for disaggregated memory. In *Proceedings of the Twentieth European Conference on Computer Systems*. 886–901.

[65] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 international conference on management of data*. 1033–1048.

[66] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-based index for memory disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2835–2849.

[67] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. 2024. Rcmp: Reconstructing RDMA-Based Memory Disaggregation via CXL. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–26.

[68] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. 2023. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1–14.

[69] Zixuan Wang, Xingda Wei, Jinyu Gu, Hongrui Xie, Rong Chen, and Haibo Chen. 2025. {ODRP}:{On-Demand} Remote Paging with Programmable {RDMA}. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 1101–1115.

[70] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–32.

[71] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and optimizing remote persistent memory with {RDMA} and {NVM}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 523–536.

[72] Yahoo. 2015. YCSB-C. https://github.com/basicthinker/YCSB-C.

[73] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data*. 295–308.

[74] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 266–282.

[75] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. {FORD}: Fast one-sided {RDMA-based} distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 51–68.

[76] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking data management systems for disaggregated data centers. In *Conference on Innovative Data Systems Research*.

[77] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the effect of data center resource disaggregation on production DBMSs. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[78] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing data-intensive systems in disaggregated data centers with teleport. In *Proceedings of the 2022 International Conference on Management of Data*. 1345–1359.

[79] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

[80] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[81] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 international conference on management of data*. 741–758.

[82] Pengfei Zuo and Yu Hua. 2017. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2017), 985–998.

[83] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. {Write-Optimized} and {High-Performance} hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.

[84] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided {RDMA-Conscious} extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 15–29.