SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure

Xinhao Min
Kai Lu*
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
minxinhao@hust.edu.cn
kailu@hust.edu.cn

Pengyu Liu
Jiguang Wan
Changsheng Xie
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technolog
pyliu@hust.edu.cn
jgwan@hust.edu.cn
cs_xie@hust.edu.cn

Daohui Wang
Ting Yao
Huatao Wu
Huawei Cloud
wangdaogui@huawei.com
yaoting17@huawei.com
wuhuatao@huawei.com

ABSTRACT

Disaggregated memory separates compute and memory resources into independent pools connected by fast RDMA (Remote Direct Memory Access) networks, which can improve memory utilization, reduce cost, and enable elastic scaling of compute and memory resources. Hash indexes provide high-performance single-point operations and are widely used in distributed systems and databases. However, under disaggregated memory, existing hash indexes suffer from write performance degradation due to high resize overhead and concurrency control overhead. Traditional write-optimized hash indexes are not efficient for disaggregated memory and sacrifice read performance.

In this paper, we propose SepHash, a write-optimized hash index for disaggregated memory. First, SepHash proposes a two-level separate segment structure that significantly reduces the bandwidth consumption of resize operations. Second, SepHash employs a low-latency concurrency control strategy to eliminate unnecessary mutual exclusion and check overhead during insert operations. Finally, SepHash designs an efficient cache and filter to accelerate read operations. The evaluation results show that, compared to state-of-the-art distributed hash indexes, SepHash achieves a 3.3× higher write performance while maintaining comparable read performance.

PVLDB Reference Format:
Xinhao Min, Kai Lu, Pengyu Liu, Jiguang Wan, Changsheng Xie, Daohui Wang, Ting Yao, and Huatao Wu. SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure. PVLDB, 17(5): 1091 - 1104, 2024.
doi:10.14778/3641204.3641218

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

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641218

1 INTRODUCTION

Recently, disaggregated memory architecture has received widespread attention from both academia [20, 26, 36, 37, 45, 57, 58] and industry (e.g., Microsoft [21], Alibaba [5, 58], IBM [17]) due to its high resource utilization, scalability, and failure isolation advantages. Disaggregated memory decouples compute (CPU) and memory resources from traditional monolithic servers to form independent resource pools. The compute pool contains rich CPU resources but minimal memory resources, whereas the memory pool contains large amounts of memory but near-zero computation power. The compute pool accesses the memory pool through RDMA-capable networks such as InfiniBand, RoCE, and Omnipath [15, 38, 54], which offer salient features including remote CPU bypass, low latency, and high bandwidth [14, 40, 59].

Distributed hash indexes are widely used for high-performance data indexing services such as databases, key-value (KV) stores, memory pool systems, and file systems [1, 8, 24, 27, 46]. However, due to the near-zero computation power of memory pools, traditional solutions [14, 28, 46] based on two-sided RDMA operations cannot be applied efficiently to disaggregated memory architectures. Prior work, e.g., RACE [65], focused on designing extendible hash structures [19, 30] that are friendly to one-sided RDMA access. However, when faced with write-intensive workloads, RACE suffers from severe write performance degradation due to inefficient resize operations and high concurrency control overhead. One common way to improve the write performance of hash indexes is to introduce leveling index structures that optimize data movement, such as CLevel [9] and Plush [41]. However, these designs do not cater to disaggregated memory, leading to frequent RDMA communication induced by a large number of small reads and writes. Furthermore, the multi-level index structure significantly compromises the read performance. Direct transplanting of these solutions is inadequate to attain the desired performance. In summary, it is imperative to redesign high-performance hash indexes for disaggregated memory, yet it is confronted with the following challenges:

1) Significant resize overhead. Resize operations of hash indexes need to be transferred to clients due to the limited computation power of the memory nodes, resulting in significant bandwidth consumption. Existing methods concentrate on minimizing the aggregate data volume moved during each resize operation. Nonetheless, they utilize an entry-based transfer strategy that relocates data at the granularity of individual entries, causing substantial network overhead and becoming a performance bottleneck. Simultaneously, in the context of variable-length KV payloads, transferring each entry requires accessing the original KV, resulting in a significant increase in read amplification.

2) Concurrency control overhead. Concurrent access to the index requires robust mechanisms for ensuring data consistency. When multiple clients access an index concurrently, they may read or write the same key at the same time. To avoid issues such as insert-miss and duplicate key [9, 30, 65], current approaches generally adopt lock-based or reread-based concurrency control strategies. However, both methods incur additional round-trip time (RTT) and increase the latency of insert operations.

3) Sacrificing read performance. Achieving optimal write performance often requires compromising the orderliness of the data layout, which can negatively impact read performance. For instance, in a leveling hash structure, the read operation must search through multiple levels, causing severe read amplification. In addition, memory constraints on compute nodes and communication overhead make optimizing read performance challenging.

In this paper, we propose SepHash, a write-optimized hash index for disaggregated memory. To address the above challenges, we have introduced innovative techniques, including the separate segment structure for efficient resizing, an RTT-reduced concurrency control mechanism for reducing write latency, and efficient cache and filter structures for enhancing read performance. With these techniques, SepHash delivers high read and write performance.

Separate segment structure. To reduce the resize overhead, SepHash proposes a two-level separate segment structure that combines the benefits of extendible hash and leveling hash. The data flow between segments is completely batch-oriented, avoiding individual entry movements. By storing depth information (part of the hash) and state information in each entry, SepHash reduces access to original KVs and quickly empties old entries during resize.

RTT-reduced concurrency control. SepHash designs an RTT-reduced concurrency control strategy that uses append write, coroutine, and sliding windows. This policy allows highly concurrent access to the index without additional locking or rereading operations. All write operations can return immediately after inserting KV pointers, without waiting for the completion of subsequent meta-data updates, significantly reducing write latency.

Efficient cache and filter. To improve read performance, SepHash introduces a filter for the separate segment structure and maintains a client-side cache. These components can effectively reduce the access granularity of RDMA operations and reduce unnecessary access to indexes. SepHash designs a space-efficient cache structure and proposes an update-efficient filter structure that requires only one RDMA operation for each update.

We implement a prototype of SepHash and evaluate performance using Micro-Benchmarks and YCSB workloads [10, 50] . Our experiments demonstrate that, under write-intensive workloads, SepHash improves write throughput by 3.3× and reduces write latency by 30% compared to state-of-the-art hash indexes. Furthermore, under read-intensive workloads, SepHash achieves 7.1× higher read throughput compared to other leveling index structures. In summary, we have the following contributions:

- We perform an analysis of existing hash indexes on disaggregated memory and identify three factors contributing to poor write performance: entry-based resize strategies, read amplification caused by out-of-place KV, and additional RTTs caused by concurrent control strategies (§3).
- We design SepHash, a write-optimized hash index on disaggregated memory. SepHash employs a two-level structure, RTT-reduced concurrent control, and efficient cache and filter to achieve excellent performance (§4).
- We evaluate SepHash and compare it with state-of-the-art distributed hash indexes. Evaluations demonstrate that SepHash outperforms other indexes in terms of write performance while achieving balanced read performance (§5).

2 BACKGROUND

2.1 Disaggregated Memory

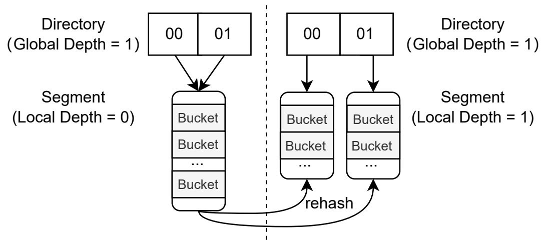
Disaggregated memory segregates compute and memory resources into separate pools [21, 22, 53], which are interconnected using high-performance RDMA networks. The compute pool has multiple CPUs (e.g., 100s) with a few GBs of DRAM. In contrast, the memory pool has a large number of memory resources (e.g., 100s–1000s of GBs) with weak computing capabilities (e.g., 1–2 wimpy cores) for handling communication and memory management. RDMA networks provide RDMA `READ`, `WRITE`, and `ATOMIC` interfaces, e.g., compare-and-swap (`CAS`) and fetch-and-add (`FAA`) operations. RDMA operations are based on the post-poll mechanism. Users initiate data transfer by posting work requests to the send queue. Requests in the same send queue are executed sequentially. By using doorbell batching [34, 43, 47], multiple RDMA operations can be combined into a single request. These requests are then read by the RDMA NIC, which asynchronously writes/reads data to/from remote memory. During data transfer, the completion queue is polled to check if the operation is complete. Waiting for network transmissions accounts for most of RDMA's latency overhead [7], making it suitable for optimization using coroutine techniques.

Coroutines are lightweight threads that are not managed by the operating system but by the program. Coroutines allow a program to pause execution at any location and resume execution later without interrupting the entire program. This makes coroutines well suited for asynchronous tasks like RDMA where I/O wait and compute can be executed under the same thread.

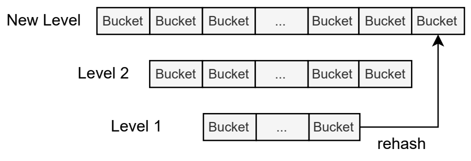
2.2 Write Optimized Hash

Current write-optimized hash indexes include extendible hash [25, 30, 62] and leveling hash [9, 41, 55, 63, 64], as shown in Figure 1.

Extendible hash uses a three-level structure comprising a directory, segments, and buckets. The directory contains a *global depth* variable, which represents the length of the hash suffix used to index segments in the directory. Each segment holds a *local depth* variable, which represents the hash suffix length shared by all KVs in that segment. A segment includes multiple buckets stored continuously, and the corresponding bucket is selected using another part of the hash value to insert KVs. Each bucket has multiple entries to hold KVs with the same hash value. When storing variable-length KVs, each entry holds a pointer to the KV data. When a bucket is full, a resize operation is triggered for the entire segment. During a

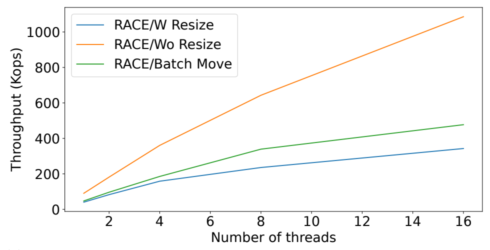


(a) Extendible hash

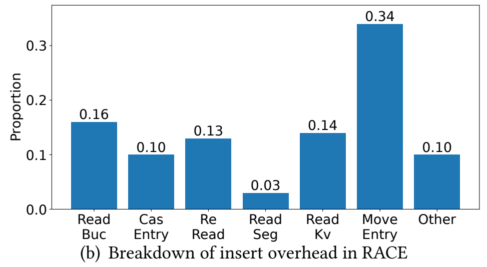


(b) Leveling hash

Figure 1: Write optimized hash indexes.



(a) Insert throughput of RACE under write-intensive workloads



(b) Breakdown of insert overhead in RACE

Figure 2: Performance analysis of RACE.

resize operation, two new segments are allocated. All KVs in the old segment are scanned to determine which new segment they should write to based on the *local depth*+1 bit of their hash value. RACE [65] further optimizes the extendible hash for disaggregated memory. With RACE, each KV corresponds to two buckets within a segment, and adjacent buckets share an overflow bucket, providing more write space for hash conflicts. The directory cache is kept on the client side to shorten the client's remote access path.

Leveling hash maintains a multi-level index structure of increasing size. CLevel [9] manages all hash tables with a lock-free level list. Both insert and search operations iterate through all hash tables level by level from bottom to top. The insert operation clears all duplicate keys encountered and inserts the data into the bucket closest to the top, while the search operation retains the data at the highest level as the final result. If a key cannot find an insertion position in all levels, a larger new hash table is allocated at the top and inserted at the end of the global level list. The background rehash thread continuously migrates data from the lowest level to the top-level hash table until only two levels of hash tables remain. Plush [41] introduces the idea of LSM-Tree [32, 51, 52] for leveling hash. Each level of Plush is an extendible hash with a fixed global depth, and all data is first inserted into the first level. When the corresponding bucket at the first level is full, a flush operation is triggered to transfer KVs to the next level. The global depth is increased by 4 per level, and the bucket is expanded into a group consisting of multiple buckets starting from the second level. The flush operation is triggered recursively when the group space is also full. The search operation starts from the first level and returns immediately after encountering the target key.

3 MOTIVATION

3.1 Low Insert Performance of RACE

RACE's search operation exhibits excellent performance. However, we observe that RACE's insert operation suffers from severe resize overhead and concurrency collision detection overhead. By manipulating the initial size of the RACE hash table, we test the performance of RACE inserting 100 million keys from scratch with and without resizing operations. As shown in Figure 2(a), RACE's insert performance is reduced by 50% in the presence of resize.

Furthermore, we perform a time decomposition of RACE's insertion. The insert overhead of RACE comes mainly in two parts: the normal write operations and the resize operations. The write operation involves three steps: (1) reading the bucket from remote memory (ReadBuc); (2) using the RDMA CAS instruction to write the KV pointer to the empty entry in the bucket (CasEntry); (3) rereading the bucket to check for duplicate keys (ReRead). The resize operation also includes three steps: (4) reading the segment (ReadSeg); (5) reading the KV data corresponding to each entry (ReadKV); (6) moving the entry to the corresponding new segment (MoveEntry). As shown in Figure 2(b), the overhead introduced by splitting accounts for half of the total time (51%), of which the overhead of moving entries accounts for the largest part. Reading the corresponding KV for each entry accounts for 27% in the resize operation. The ReRead overhead introduced by the concurrent control policy takes up one-third of the normal write operation.

3.2 Limitations of Leveling Hashes

The leveling hash provides a more cost-effective way to resize operations. However, the current leveling hashes are not optimized for disaggregated memory and do not consider RDMA access characteristics. When applied to disaggregated memory, leveling hashes face three limitations: 1) Multi-level structure suffers from high communication overhead. The leveling structure increases the access path of the search operation, which significantly lowers search performance. 2) Small-grained access does not fully utilize RDMA bandwidth. Both CLevel and Plush are designed for persistent memory, and they issue many small-grained reads and writes to index metadata due to the low access latency of NVM [42, 49]. However, by evaluating RDMA bandwidth and latency changes with access

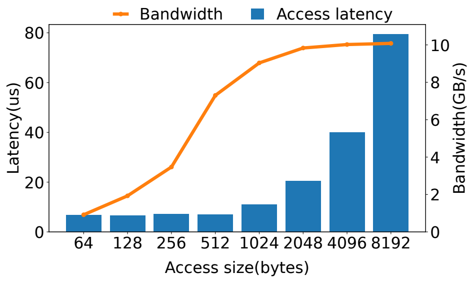


Figure 3: RDMA access patterns.

granularity (Figure 3), we observe that smaller access granularity leads to lower bandwidth, while larger access granularity leads to increased latency. Therefore, to design an efficient index structure for disaggregated memory, it is essential to select appropriate access granularity. 3) There is still room for write optimization. Plush introduces a batch-move strategy to eliminate write amplification caused by moving entries individually. Applying this strategy to RACE can improve insert throughput (the green line in Figure 2(a)), but there is still a large gap compared to the ideal performance.

4 SEPHASH DESIGN

4.1 Overview

This paper proposes SepHash, a write-optimized hash index on disaggregated memory with three core design principles: 1) *low resize overhead*, 2) *low write latency*, and 3) *balanced read performance*. Figure 4 shows its key design components:

- **Separate segment structure** (§4.2). SepHash introduces a separate segment structure that consists of a small unordered segment (called CurSegment) and a large ordered segment (called MainSegment). Each CurSegment has a corresponding MainSegment, indexed by a directory according to extendible hash schemes. Whenever the data accumulation in the CurSegment or MainSegment reaches the size limits, a resize operation (called segment merge or split) is triggered to batch-transfer entries to a new MainSegment.
- **RTT-reduced concurrency control** (§4.3). All concurrent write operations are atomically performed in CurSegment in an append-only manner. Subsequent write operations directly overwrite old data without rereading or locking the segment. The coroutine-optimized RDMA interface and sliding-window scheme are adopted to reduce write latency.
- **Efficient cache and filter** (§4.4). Each CurSegment is equipped with an update-efficient filter to reduce unnecessary reads. For each MainSegment, a metadata table (called FPTable) recording entry distribution is cached in the clients and is compacted using a record point based scheme.

As shown in Figure 4, the main index structure of SepHash is stored in the memory pool and consists of two-level hash tables. Each level is organized according to the extendible hash scheme and indexed using a common global directory. In the compute pool, the client maintains a directory cache to accelerate index access and uses pure one-sided RDMA verbs to perform read/write operations on the index structure in the memory pool.

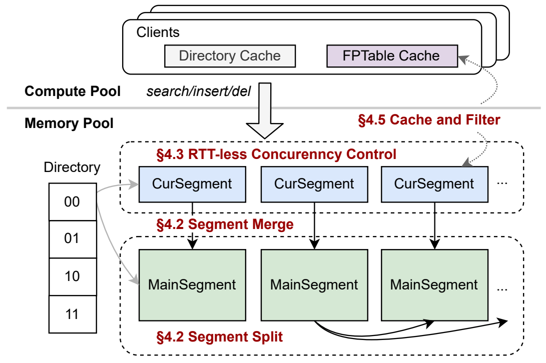


Figure 4: The overall architecture of SepHash.

4.2 The Index Structure of SepHash

4.2.1 Separate segment structure. To reduce resize overhead and optimize both read and write performance, we design a sophisticated separate segment structure, as shown in Figure 5, which includes four major designs:

1) Dual level structure. To balance insert and search performance, we divide the index into two levels of segments: top-level CurSegments and bottom-level MainSegments. The insert operation is completed after writing the KV pointer to CurSegment. Once the CurSegment size reaches its limit, a merge operation is triggered, combining the CurSegment with the corresponding MainSegment using a single RDMA write. When the MainSegment size reaches its limit, a split operation is triggered, dividing the MainSegment into two smaller new segments. Dual-level index structure allows data to be moved in batches while avoiding degradation in read performance due to excessive levels.

2) Shared CurSegment. To reduce the number of resize operations, we design a CurSegment structure that eliminates the intermediate bucket hierarchy. CurSegments consist of contiguously stored entries with a segment size suitable for RDMA access (e.g., 512 bytes, 1024 bytes). All entries in CurSegment share the same hash suffix. Each entry is 8 bytes, and all inserts compete for these entries in order using RDMA CAS. Extending the hash collision space to the entire segment avoids splits triggered by uneven data distribution among buckets. Reduce the frequency of resize operations while improving index space utilization. We store shared metadata (called CurSegMeta) in the CurSegment header, which helps avoid storing metadata for each bucket. We also store a filter in CurSegMeta to shorten the search path.

3) Resize-optimized entry. To mitigate read amplification during resize, we pre-store a portion of the hash value (depth information) in each entry. Specifically, we select 4 bits from the hash value as depth information during each insert based on CurSegment's local depth. In each subsequent segment split, a bit of depth information is used to determine the moving destination of the entry. This reduces access to the original KV during split operations. Moreover, it reduces space consumption by 50% compared to storing the entire hash value of each entry while providing a similar acceleration effect. To reduce the massive overhead of emptying old entries in the resize operation, we add a sign bit for each entry and CurSegMeta. Entries with the same sign bit as CurSegMeta

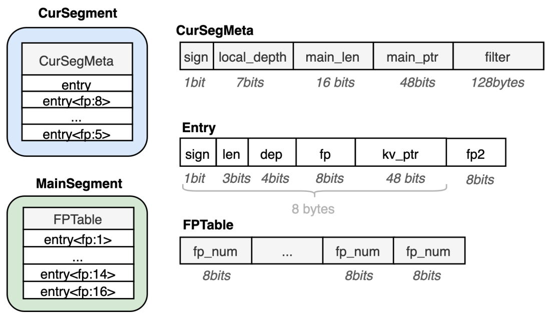


Figure 5: Separate segment structure in SepHash.

are in an empty state. The merge operation can quickly empty all entries in CurSegment by flipping the sign bit in CurSegMeta.

4) Sorted MainSegment. To speed up the search process, we store fingerprints (a small sequence of hash-generated bits) of keys in each entry. Only entries that match the fingerprint of the target key need to read the original KV for comparison. However, MainSegment is much larger than CurSegment due to the merge mechanism. Reading the entire MainSegment leads to high latency and reduced throughput, as shown in Figure 3. To address this issue, we sort entries based on their fingerprints and design an FPTable that stores the number of entries for each fingerprint. Each read operation only reads the entry array corresponding to a certain fingerprint, which can be quickly located through the FPTable. An RDMA READ can read this array, and the size is only a tiny part of the MainSegment (e.g., 1/256 if using 8-bit fingerprints). However, due to the enormous size of MainSegment, an array corresponding to 8-bit fingerprints may contain up to a dozen entries. For each entry, the corresponding KV needs to be read for comparison, which will introduce multiple communications. To store a longer fingerprint while maintaining an entry size suitable for RDMA CAS, we maintain two 8-bit fingerprint fields fp and fp2 in each entry. Fp is directly embedded in the entry, fp2 is stored at the end of the entry, and all entries in the MainSegment are sorted by fp.

Combining all the design elements, the details of the separate segment structure are shown in Figure 5. CurSegment and MainSegment both consist of a contiguous array of entries. All entries in a CurSegment share the same hash suffix. Entries in a CurSegment are unordered, while MainSegment entries are sorted by fp. CurSegMeta includes a sign bit for finding empty entries, a 7-bit local_depth for extendible split, a pointer main_ptr to the corresponding MainSegment and its length, and a filter for read access. Each entry includes a 1-bit sign indicating its free status, a 3-bit len indicating the length of the KV after alignment to 64 bytes, a 4-bit dep field for depth information, a 48-bit KV pointer, and two 8-bit fingerprints fp and fp2. FPTable holds the number of entries for each fp. Based on the separate segment structure, the flow of insert and search operations is as follows:

Insert: To insert a KV, the client first calculates the hash value and indexes the corresponding CurSegment address in the local directory cache. After reading CurSegment from remote memory, the client looks for an empty entry with the same sign bit as CurSegMeta and initializes it according to CurSegment's local depth. As

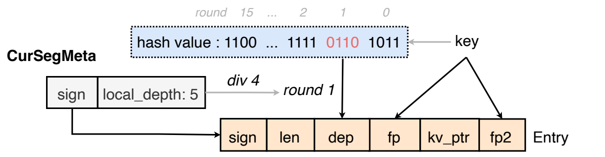


Figure 6: Initialization of entry.

shown in Figure 6, we organize 4 split operations as a round and divide the hash value into several 4 bits accordingly. The current round is determined by dividing the local depth by 4. Initialize the dep field by taking the 4-bit hash value corresponding to the current round. Two 8-bit hash values are then calculated as fp and fp2. The client writes the initialized entry to the remote CurSegment using RDMA CAS. Finally, the client uses two RDMA WRITEs to update the filter in CurSegMeta and fp2 at the end of the entry. Update or delete is converted into insert using the new or blank value, according to the RTT-reduced Concurrency Control in §4.3.

Search: To search for a target key, the client first obtains CurSegment and MainSegment pointers from the local directory cache. Then, the client retrieves the CurSegment filter and the MainSegment FPTable using RDMA READs. Based on the filter, the client determines whether to read the entry array in CurSegment. Using the FPTable, the client determines the address and length of the entry array to be read in MainSegment. After reading the corresponding entry array, the client reads KVs for entries that match both the fp and fp2 fields and finally matches the key locally.

4.2.2 Segment merge and split. When the data accumulation in CurSegment or MainSegment reaches the size limits, a segment merge or split is triggered to transfer data to a new MainSegment. Through depth information cached in the entries and batch clearing the old entries, the main flow of both segment merge and split can be completed within 5 RTTs, as shown in Figure 7.

Segment merge: ❶ The CurSegment is read into the client's memory, and the entries are sorted according to their fp. For entries with the same fp, if they also share the same fp2, we read the KV to verify if they are duplicate keys. According to the RTT-reduced concurrency control in §4.3, we only keep the entries closest to the bottom for duplicate keys. ❷ The corresponding MainSegment is read and merged with CurSegment. Replace the MainSegment entry with the CurSegment entry that points to the same key. The merged entry array becomes the new MainSegment if it does not exceed the size limit. ❸ The new MainSegment is written to the remote memory, ❹ followed by updating the main_ptr in the CurSegment. ❺ The entire CurSegment is emptied by flipping the sign bits.

Figure 7 illustrates an example of segment merge. For simplicity, we only use fp in this example and do not involve fp2. When sorting the CurSegment, two entries with fp=8 are identified as duplicates, with only the latter entry being retained. The merge operation with MainSegment adds the entry with fp=4 directly, as it has no matched entries. The entry with fp=5 in the CurSegment replaces the old entry in the MainSegment. The entry with fp=8 in the CurSegment is appended to the MainSegment since it does not correspond to the same KV in the MainSegment. In practice, since both fp and fp2 are used for matching, it is rarely necessary to read the KVs.

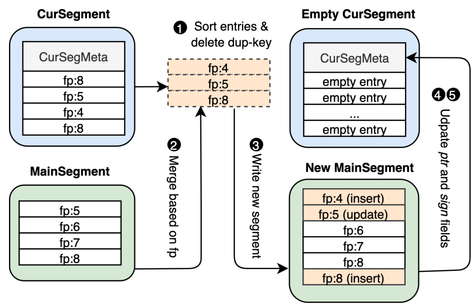


Figure 7: Segment merge.

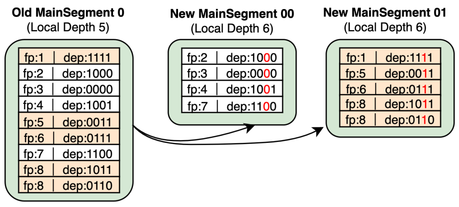


Figure 8: Segment split.

Segment split: In ❷ of the segment merge, if the size of the newly generated MainSegment exceeds the size limit, a segment split is triggered. Each split creates two new MainSegments and entries in the old MainSegment will be assigned to one of the two new MainSegments based on their hash suffix. Because we pre-store the first 4 bits of the suffix for each split round in the dep field, we can complete the entry assignment without reading the original KV. Assuming the current local depth is *l*, the bit at *l mod 4* of the dep field is the first bit of the suffix used by the current split. Entries are divided into two new MainSegments according to the *l mod 4* bit of the dep. After splitting the MainSegment in client-side memory, they are bulk written to remote memory using an RDMA `WRITE`. When *l mod 4* equals 3, the last split in the current split round is in progress and all information in the dep field will be consumed. Read the original KV to obtain a new dep.

An example is shown in Figure 8, the local depth of the old MainSegment is 5, so the 1st bit (5 mod 4) of the dep field is used. All entries in the old MainSegment with the 1st depth bit of 0 are split to form a new MainSegment 00, while all entries with the 1st depth bit of 1 are split to form a new MainSegment 01. In the next splitting of segments 00 and 01, the second bit of dep field is used.

Merge or split operations update the main_ptr or local_depth in CurSegMeta. When the insert operation reads the CurSegment, it checks the main_ptr and local_depth to detect merges and splits occurring in remote memory and updates the local directory cache.

4.3 RTT-reduced Concurrency Control

If two concurrent clients insert different values of the same key into the same bucket, the search operation cannot determine the latest value. Traditional indexes use lock-based [41] or reread-based [65] schemes to avoid duplicate keys, both incur extra RTTs. As shown in Figure 9, lock-based schemes obtain the bucket lock before insert operations to prevent the writing of the same key. Reread-based solutions reread the bucket after insert operations and remove all duplicate keys. We propose a concurrency control strategy with fewer RTTs using the following three techniques:

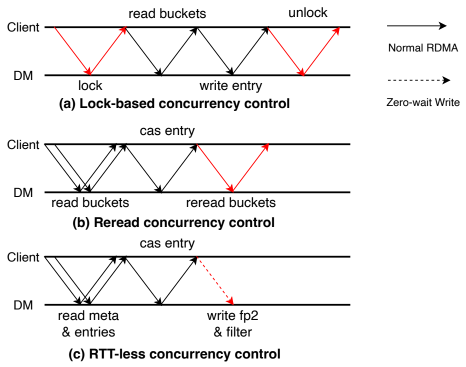


Figure 9: RTT-reduced concurrency control. (*The solid lines mean normal RDMA requests, and the dotted line means zero-wait write. The strategies's overhead is marked in red.*)

Append write: All writes to the CurSegment occur sequentially. Initially, all CurSegment entries are empty, and inserts can only occur on the first entry. All clients use RDMA `CAS` to compete for the first entry. When the first entry is written, all clients move to the second entry and repeat the process. All entries are written atomically using RDMA `CAS`, ensuring that only one client can successfully write the current first empty entry without locking the segment. In addition, the append write ensures that the latest version of a key will appear in the entry closest to the end. Search operations can obtain consistent results for duplicate keys, so there is no need to reread the bucket. Update and delete operations are converted to inserts, competing for CurSegment entries in the same order. It ensures that no writes will occur to CurSegment when the merge operation is in progress, avoiding data loss.

Sliding window: Reading the entire CurSegment to find the first empty entry causes high write latency. We divide the entry array in CurSegment into fixed-length entry windows. Each client maintains a *offset* variable for each CurSegment that indicates the position of the last first empty entry. Each insert starts from the locally recorded *offset* and reads the entry window one by one to find the first empty entry. Each successful write updates the local *offset*. Whenever a merge on the remote CurSegment is detected, *offset* is reset to 0. This prevents any insertion into the window behind the first empty entry. Due to the uniformity of the hash function, the first empty entry is usually found on the first attempt.

Zero-wait write: The separate segment structure introduces additional updates to fp2 and CurSegMeta. However, unlike `READ` and `CAS` operations, RDMA `WRITE` don't require a return result. Based on this, we design a zero-wait write interface. We use a coroutine framework to schedule and overlap all RDMA requests from different clients, with each coroutine corresponding to a client. We share a back-end RDMA connection (*back-conn*) for clients that belong to the same thread. For RDMA `WRITE`s that don't need to wait for completion, we return after submitting it to the post

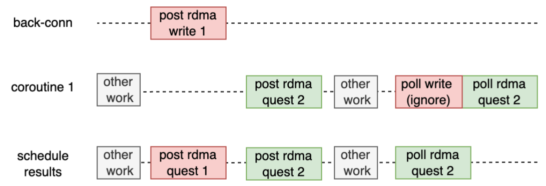


Figure 10: Zero-wait write.

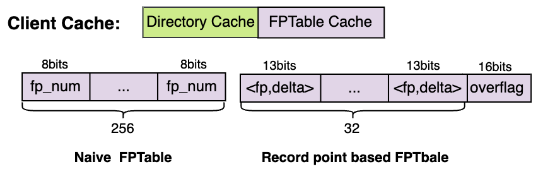


Figure 11: Client cache structure.

queue in back-conn. For instance, in Figure 10, coroutine 1 needs to send an RDMA WRITE without waiting and another normal RDMA request. It returns after submitting the WRITE request into back-conn, followed by sending the normal RDMA request to the client's connection. During the polling process, the write operation posted to back-conn is polled together through the coroutine framework. The final schedule result is shown at the bottom of Figure 10, with the latency of polling write results completely hidden.

The flow of RTT-reduced concurrency control is illustrated in Figure 9(c). In the first RTT, depending on the local offset, the client uses doorbell batching to simultaneously read the CurSegMeta and CurSegment entry window. In the second RTT, the client selects the first empty entry based on the sign bit in CurSegMeta and occupies it using an RDMA CAS operation. If the CAS operation succeeds, the client uses zero-wait write to post changes to fp2 in the tail of the entry and filter in CurSegMeta. Compared to lock-based and reread-based schemes, the RTT-reduced concurrency control significantly reduces RTTs and has lower latency.

4.4 Filter and Cache

The leveling structure inevitably reduces read performance. To speed up search operations, we design a filter for CurSegment and a client cache for MainSegment.

4.4.1 FP filter for CurSegment. We adopt a bitmap as a filter for CurSegment. Traditional filters, such as Bloom filter [2, 35], Quotient filter [18, 33], Cuckoo filter [4, 13], need to modify/read multiple discrete bits for a single insert/query. This brings multiple communication overheads in disaggregated memory. The bitmap is equivalent to a bloom filter using a single hash function, and inserts and queries on it require only one RDMA Read. The bitmap records all fps that appear in CurSegment, and the i-th bit equal to 1 means that the entry with fp=i exists in CurSegment. For insert operations, after occupying the empty entry, we write a 1 to the bit at fp of bitmap. For search operations, clients use doorbell batching to read the MainSegment entries and the CurSegment bitmap simultaneously. Clients read the CurSegment entry array only if the bit corresponding to the given key in the bitmap is 1.

4.4.2 FPTable cache for MainSegment. On the client side, in addition to the directory cache, we design an FPTable cache to hold MainSegment metadata. FPTable needs to store fp distribution in MainSegment. As shown in Figure 11, a naive way of caching is to record the number of entries per fp. However, this method will result in vast space overhead and is not suitable for the compute side with limited memory resources. Due to the uniformity of the hash function, the number of entries corresponding to each fp in MainSegment is very close (in our tests, the average data skew is less than 18%). Therefore, we propose a compact FPTable cache scheme using record points. We use a set of linear functions with the same slope to fit the fp distribution in MainSegment. Suppose the number of entries in MainSegment is E and the total number of fp is N. For fp=i, we use the following formula (1) to predict the starting address (pre_i) of the corresponding entry array.

$$avr = \frac{E}{N}, \quad pre_i = avr \times i + \delta \tag{1}$$

where avr is the average length of the entry arrays corresponding to fps, and δ is used to correct prediction errors. The δ is initially 0 and is updated according to formula 2 when the FPTable is created.

$$|pre_i - start_i| < 0.5 \times avr \tag{2}$$

where $start_i$ is the actual starting address of the entry array corresponding to fp=i. When the prediction error exceeds 0.5 times avr, a new record point is created in the FPTable, as shown in Figure 11, recording the fp and the new delta value (a 5-bit sign integer). FPTable reserves 32 record points, which is sufficient in most cases. An overflow flag (overflag) is set at the end to indicate the number of overflow record points, stored sequentially after it. The record point based solution can save space compared to the naive solution. For the search operation, the latest record point that does not exceed the fp of the given key is found in the FPTable. Then, the delta value is used to predict the start address of the entry array. To ensure reading the complete entry array, several more entries are read before and after the predicted entry array. If the entry at the beginning and end of the array still has the given fp, then the array has not been read completely, and an entry of size $0.5avr$ is read further. In most cases, only one reading is required.

5 EVALUATION

In this section, we evaluate SepHash's performance using different workloads. The experiment setup is introduced first (§5.1). Next, we compare SepHash with three other hash indexes using Micro-Benchmark and YCSB benchmark (§5.2 and §5.3). Then, we compare the space overhead of these indexes (§5.4). Finally, we evaluate the impact of SepHash's technologies and parameters (§5.5 and §5.6).

5.1 Experimental Setup

Hardware Platform. We run all experiments on 8 machines, each with two 26-core Intel Xeon Gold 5218R CPUs, 384 GiB DRAM, and one 100Gbps Mellanox ConnectX-5 IB NIC. Each NIC is connected to a 100Gbps Mellanox IB switch. We use a machine to simulate a memory pool and limit its CPU resources to a single CPU core, which is only used for memory registration during the index initialization stage [23, 56, 61, 65]. Memory is registered using huge pages to reduce NIC page translation cache misses [14]. The remaining

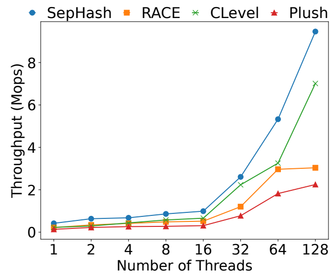


Figure 12: Insert throughput.

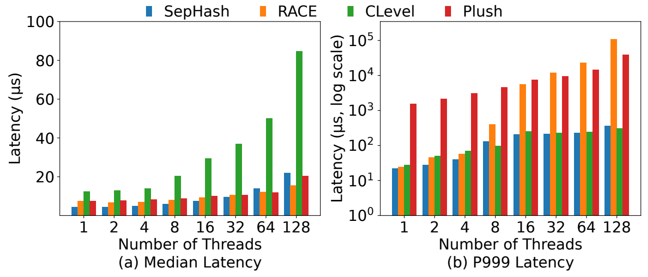


Figure 13: Insert latency.

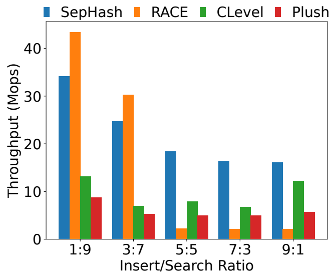


Figure 14: Hybrid workloads.

machines constitute compute pools. Since current RDMA NICs do not support remote memory allocation[23, 65], we reserve memory space at the memory pool for each client.

Comparisons. We compare SepHash with three distributed hash indexes. 1) RACE [65] is the only distributed hash designed for disaggregated memory, and because RACE is not open-source, we implement it from scratch. 2) CLevel [9] and 3) Plush [41] are leveling hash indexes designed for a single machine, and we port them to disaggregated memory by replacing local memory read/write with RDMA `READ/WRITE`. To ensure fairness, we optimize CLevel and Plush for disaggregated memory, such as removing persistence logs and using bucket sizes that are appropriate for RDMA. Additionally, we maintain client-side caches of similar size to SepHash for Plush and CLevel. For Plush, we employed Monkey's [12] policy to set and cache the bloom filter. For CLevel, we maintain a KV-cache due to its frequently updated structure. For SepHash, CurSegment is set to 1024 bytes, and MainSegment is 64 times CurSegment's size by default. Based on previous work, we use a configuration of 8 entries per bucket for RACE. For all comparisons, we ensure that each hash table initially has a similar number of entries to make the conditions for the first resize operation comparable. In addition, CLevel uses a client as a background thread for resizing the index. We use 16-byte keys and 32-byte values that are representative of KV stores in real-world workloads [3, 6, 23, 26, 43, 65].

5.2 Micro-Benchmark

We test the basic performance of each index by inserting and reading 100 million KVs from scratch. Then, we adjust the number of concurrent threads to observe their concurrent scalability. When the number of threads is less than 16, they run on the same compute node. For thread numbers 32, 64, and 128, they run on 2, 4, and 8 compute nodes respectively [23, 65]. Each thread runs with 1-4 coroutines, choosing the number of coroutines that provide the highest performance as the final configuration.

5.2.1 Insert performance.

Insert throughput. As shown in Figure 12, SepHash achieves the highest insert throughput, increasing by 1.4×–4.2× compared to other indexes. SepHash also shows good scalability, and insert throughput increases linearly with the number of concurrent threads. RACE's insert throughput gradually reaches a bottleneck at 64 threads. CLevel also outperforms RACE, showing excellent insert throughput and scalability. Plush shows the lowest write performance of all concurrent configurations.

RACE's low insert throughput mainly comes from high resize overhead. As shown in Table 1, RACE's resize operations occupy half of the insert time. The entry-based move strategy makes it impossible to empty the entries in the resized segment in time. When concurrent clients access the resized segment, they attempt to initiate a new resize operation but are blocked. Under 128 threads, RACE experiences 80× more resize operations and takes 21× more resize time than a single thread. Thanks to higher entry utilization and the larger MainSegment maintained by the merge mechanism, SepHash splits only 1/8 as much as RACE. SepHash's efficient resize strategy makes its resize time only 1/30 of RACE, and therefore, less client-side blocking due to resize. Under 128 threads, SepHash's resize time has only increased by 4.2×. Plush's resize operation also takes a lot of time. Under a single client, the resize operation time accounts for 45%. This is because Plush needs to read the KV corresponding to each entry to determine which group they belong to in subsequent levels. Moreover, RACE allows KV to be inserted into empty buckets of the segment being resized, while Plush completely prevents insertion into other free buckets on the insertion path. As a result, Plush's insert time increases more with the number of concurrent threads than RACE, increasing by 30× under 128 threads. CLevel's resize operation only needs to insert a pointer into the remote level list, so the resize overhead is negligible.

Insert latency. Figure 13 shows the median and tail latency for different index insert operations. The tail latency is presented using logarithmic coordinates. SepHash's median latency under a single client is 4.3μs, only 58% of RACE (7.4μs). CLevel's median latency is significantly higher while growing rapidly with increasing concurrent threads. Plush's median write latency is close to RACE. SepHash and CLevel have stable tail latency, while RACE's tail latency increases rapidly, eventually increasing by three orders of magnitude compared to SepHash. Plush has the highest tail latency.

When the number of concurrent threads is low, SepHash's low insert latency benefits from the RTT-reduced concurrency control strategy that avoids extra RTTs. While SepHash's merge and split operations are more efficient, its local cache fails more frequently than RACE. Consequently, SepHash's insert process experiences more retries than RACE, as shown in Table 1, which causes slightly higher insert latency under high concurrent threads (as shown in Figure 13 a). This is acceptable compared to the huge throughput boost gained in SepHash. Most Plush inserts only need to access the first level of the index and contain only 6 RDMA operations, so the median latency is similar to RACE. CLevel has a higher median insert latency because it checks all levels for conflicts before the

Table 1: Index Overhead

1-client /128-clients	resize_cnt (10^4)		resize_time (10^7 μs)		insert_time (10^8 μs)	resize_ratio	retry/level_cnt	ReadBuc
SepHash	merge_cnt	split_cnt	merge_time	split_time	5.14/81.5	3.5%/1.01%	1.00/6.05	2.05/2.07
	86.1/86.1	1.64/1.64	0.39/1.10	1.82/8.22				
RACE	12.9/1050		68/1470		13.4/290	50.79%/50.48%	1.00/3.82	2.0/2.0
CLevel	0.0012/0.0018		0.049/0.0008		14.9/113	0.00%/0.00%	2.00/8.28	4/16.6
Plush	59.6/59.6		64.7/329		14.7/444	45.35%/7.39%	1.00/1.00	34/34

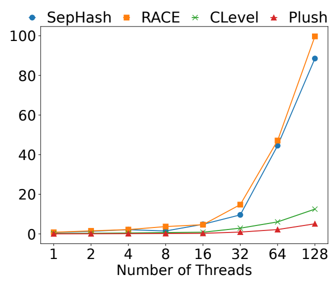


Figure 15: Search throughput.

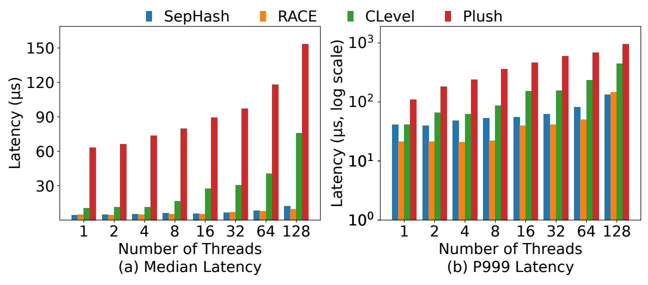


Figure 16: Search latency.

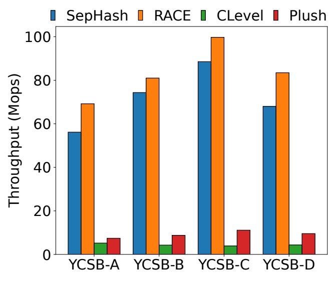


Figure 17: YCSB benchmarks.

final insert. CLevel's median insert latency increases rapidly with higher concurrency due to the background rehash thread being unable to move data in time. This results in insert operations having to traverse more levels. For instance, Table 1 shows that under 128 threads, the median number of levels traversed by CLevel increases to 8.28. SepHash and CLevel have the lowest tail latency, which reflects the overhead of different resize schemes. For SepHash, moving entries in bulk and caching depth-info can minimize the number of RDMA requests per resize operation. CLevel's background rehash scheme makes the delay of resize operations not manifest in foreground operations. In contrast, Plush and RACE have extremely high tail latency due to their expensive resize scheme.

5.2.2 Search performance. Figure 15 shows the search throughput of different indexes. RACE and SepHash outperform Plush and CLevel by 7.8×-22×. Figure 16 illustrates that SepHash and RACE have the lowest search latency, while Plush has 2×-10× higher read latency than other indexes. The last column of Table 1 shows the median number of read buckets (ReadBuc) in each index's search operation. RACE only needs to access two buckets per search operation. With the two-level index structure and the fp filter, SepHash also only needs 2 RDMA READs to complete the search operation in most cases. In contrast, each group at the bottom levels of Plush corresponds to 16 buckets, which must be read sequentially during the search process. CLevel also only needs to query two buckets per index level but needs to access multiple levels. Table 1 shows that Plush needs to read 34 buckets per search operation, and ReadBuc of CLevel has grown 4× from a single thread to 128 threads, resulting in high search latency for both.

5.2.3 Hybrid workload. Figure 14 shows the performance of different indexes under mixed workloads with different insert/search operation ratios. We pre-insert 10 million KVs for each index, then run a mixed workload containing 10 million operations. Each index records maximum throughput under 128 threads. Under a low insert ratio, RACE performs well because there are few insert operations and no resize operation is triggered. When the insert ratio reaches more than 50%, resize operations are triggered and RACE's throughput drops rapidly. Due to the low-overhead resize strategy, SepHash maintains stable performance in the face of increased insert operations. Finally, SepHash provides 8× the overall throughput of RACE in a write-intensive scenario. The overall performance of CLevel and Plush is stable but low.

5.3 YCSB Benchmark

We use the standard YCSB benchmarks [10] to test the performance of these indexes. We pre-insert 100 million KVs and perform 10 million operations under four workloads including (A) update heavy (50% updates), (B) read mostly (95% read), (C) read only, (D) read latest (5% insert). As shown in Figure 17, RACE and SepHash exhibit higher performance among all YCSB workloads. In standard YCSB benchmarks, the amount of loaded data is 10 times that of hybrid workloads in the Micro-Benchmark, and all indexes are fully expanded. In addition, there are more read operations in YCSB benchmarks. As a result, the resize operations are not triggered even under the YCSB A workload. The updates of RACE only involve updating the KV pointer in each entry after a search. This makes RACE perform quite well. SepHash maintains performance close to RACE through its read optimization strategy, while Plush and CLevel perform poorly due to search performance limitations.

5.4 Space Overhead Analysis

Space overhead is an important metric for hash indexes [9, 25, 30, 64]. To compare the space cost of different hash indexes, we calculate the space utilization of different indexes and the space cost of each component.

5.4.1 Space utilization. To compare the space efficiency of different hash indexes, we introduce two metrics: entry utilization and space

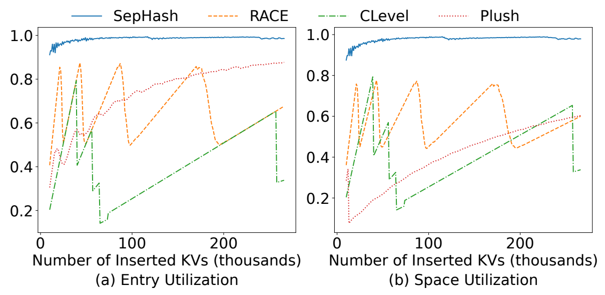


Figure 18: Entry utilization and space utilization.

utilization. **Entry utilization** is the ratio of the number of entries holding valid data to the total number of entries. **Space utilization** refers to the proportion of the space overhead of valid entries to the total space overhead of the index. We first insert 10,000 KV, which is close to when the index triggers the first resize operation, and record entry/space utilization for every 1,000 KV inserted. For CLevel, we record the utilization after waiting for the background resize thread to complete. As shown in Figure 18, experimental results show that SepHash achieves more than 90% entry utilization and space utilization. Moreover, MainSegment maintains 100% entry utilization. As more data is inserted, a larger MainSegment results in higher entry utilization and space utilization. RACE and CLevel have a maximum entry utilization of around 90%, but resize operations significantly decrease their entry utilization. CLevel's entry utilization can even fall below 20% due to full table resize. RACE has low space utilization due to the retention of metadata in each bucket. Plush has excellent entry utilization but low space utilization due to the large filters in each bucket.

5.4.2 Space overhead. We calculate the space overhead of each component after inserting 100 million KVs, as shown in Table 2. BucMeta refers to metadata stored in the bucket, such as local_depth and suffix stored in RACE's bucket, and fp bitmap stored in SepHash's CurSegment. DirMeta refers to the segment and bucket information in the directory, such as segment pointer in RACE, and FPTable in SepHash. The experimental results show the following: 1) For metadata overhead (BucMeta and DirMeta), although SepHash maintains additional fp filter and FPTable, BucMeta is very small because the entire CurSegment shares metadata. In addition, Sephash maintains a low global depth via the merge mechanism and large MainSegments, resulting in a small DirMeta. RACE and Plush have high metadata overhead. RACE needs to store local depth and suffixes in each bucket, while Plush needs to maintain a bloom filter of more than 128 bytes for each bucket after the second level. CLevel only includes a list of all levels, and its metadata overhead is negligible. 2) For total overhead, the index size is mainly determined by the number of its entries. SepHash has lower space overhead because SepHash can be expanded more finely while keeping the number of entries low. Table 2 shows that the total number of entries in SepHash is only 101 million after inserting 100 million KVs. RACE with full-segment resize and CLevel with full-table resize result in more space usage. After inserting 100 million KVs, the total number of entries in the two indexes is 197 million and 800 million, which is a huge space overhead. Plush uses a resize

Table 2: Space overhead

	Depth /Level	**BucMeta** (MB)	**DirMeta** (MB)	**Total** (GB)	**#Entry** (billion)
SepHash	14	2.37	2.96	0.85	1.01
RACE	18	188.56	6.00	1.66	1.97
CLevel	16	0	0	6.00	8.05
Plush	4	0	213.79	1.03	1.11

Table 3: The space overhead of FPTable cache

MainSeg/CurSeg	64	32	16	8
Naive	9 MB	18 MB	36 MB	73 MB
Record Point Based	2.68 MB	5.36 MB	10.8 MB	22 MB

strategy similar to SepHash, resulting in a lower total number of entries and lower space overhead.

Table 3 shows the FPTable cache space of different MainSegment sizes. Results show that the record point based scheme can save up to 67% space compared to caching the FPTable directly. Even using a small MainSegment will not cause excessive cache consumption.

5.5 In-Depth Analysis

To analyze the performance of SepHash and verify its optimization design, we decompose the performance gap between RACE and SepHash by applying SepHash's key techniques one by one. Figure 19 describes the evaluation results for insert and search operations.

5.5.1 Write optimization. In Figure 19(a), **Base** represents the basic version of SepHash, including the use of a dual-level structure, append-write policies, and offset-based sliding windows. **+resize-op entry** represents adopting the resize-optimized entry structure with depth information and empty status on top of the Base scheme. **+zero-wait write** indicates that the coroutine-optimized RDMA `WRITE` interface is further applied. The Base scheme uses segment merge and split to transfer the entries in the resize operation as a batch, better matching the high bandwidth access granularity of RDMA. As a result, SepHash allows more simultaneous writes under high concurrency conditions. With 128 client sides, SepHash gains a 1.6× throughput improvement over RACE. Resize-optimized entry structure alleviates the huge read/write amplification caused by reading KV and emptying entries during resize, further improving performance by 2× compared to the Base solution. The coroutine-optimized RDMA `WRITE` interface conceals the additional update operations caused by the separate segment structure and ultimately obtains a 3.3× performance improvement over RACE.

5.5.2 Read optimization. In Figure 19(b), **Base** refers to the scheme of directly scanning the entire CurSegment and MainSegment. **+FPTable** means enabling FPTable for sorted MainSegment and maintaining the corresponding cache on the client side. **+BitMap** means adding fp bitmap as a filter for CurSegment. The Base scheme performs poorly due to the large size of CurSegment and MainSegment. Reading the entire CurSegment and MainSegment consumes much bandwidth and cannot scale with the number of threads. FPTable reduces access granularity to the MainSegment to an entry

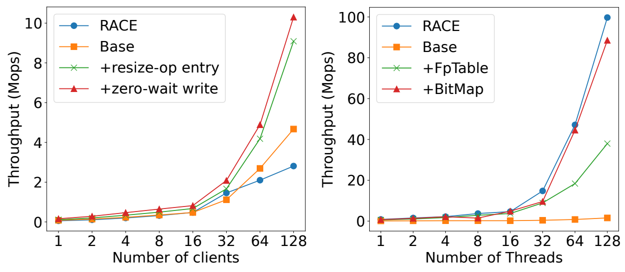


Figure 19: In-depth analysis.

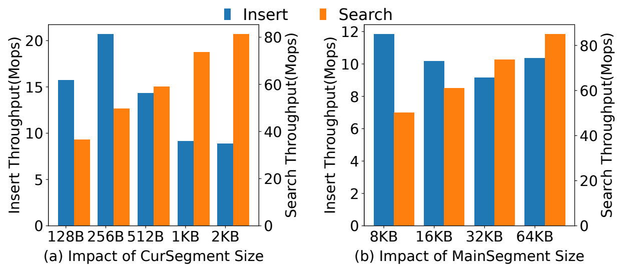


Figure 20: Impact of segment size.

array of only 1/256 of the entire segment, resulting in a 22× increase in read throughput over the Base solution. Fp bitmap avoids accessing a KB-sized CurSegment for each read. In most cases, CurSegment does not contain the target key, and unnecessary access can be avoided by reading 1 bit in the bitmap. Using fp bitmap further improves the read performance of SepHash by 2×, which is close to the read performance of RACE.

5.6 Sensitivity Analysis

We test the impact of critical parameters on SepHash, including segment size, the number of memory nodes, and variable KV size.

5.6.1 Segment size. The size of CurSegment and MainSegment determines the frequency of segment merge and split and the number of entries accessed in each operation. We test the impact of different CurSegment and MainSegment sizes on read/write performance.

CurSegment size. Figure 20(a) shows how SepHash performance varies with CurSegment size from 128 bytes to 2 KB. In general, smaller CurSegments improve insert performance. Because MainSegments decrease synchronously with CurSegments, the overhead of reading CurSegment and MainSegments during merging and splitting is also reduced. In addition, each insert operation accesses smaller CurSegments, reducing read amplification. In reducing CurSegments from 2 KB to 256 bytes, SepHash's insert throughput achieves a 2× performance improvement. Too small CurSegment, however, can lead to frequent merge and split operations, declining the concurrent scalability of indexes. When using a CurSegment of 128 bytes, SepHash's insert performance decreases compared to 256 bytes. SepHash's search performance decreases as CurSegment decreases. Because smaller CurSegment reduces the MainSegment proportionally, thus increasing the overall depth of the SepHash. As the number of CurSegments increases, more read operations will hit CurSegment, and additional RDMA access will significantly degrade search performance.

MainSegment size. We fix CurSegment to 1 KB and MainSegment size varies from 8 KB to 64 KB. As shown in Figure 20(b), insert performance gradually improves as the MainSegment size decreases from 32 KB to 8 KB. Because the smaller MainSegment reduces the read amplification caused by accessing the MainSegment during merge and split. On the other hand, a 64 KB MainSegment achieves better write performance because it reduces the frequency of split. Increasing the MainSegment size can improve search performance. Because larger MainSegments will hit more search accesses, shortening access paths compared to search CurSegments.

5.6.2 Number of memory node. We test the impact of the number of memory nodes on index performance. The rest of the index is evenly distributed across all memory nodes except the directory. CLevel is incompatible with multi-memory nodes because each level requires large contiguous memory. We test the throughput of inserting and searching 100 million KVs using different numbers of memory nodes. As shown in Figure 21, test results show that the insert performance of SepHash scales well with memory nodes. For RACE and Plush, adding a memory node can improve their insertion throughput while adding more memory nodes leads to a gradual decline in performance. The read performance of RACE and SepHash also increases first and then decreases with memory nodes. Plush's search performance continues to increase slightly.

Analysis. For indexes limited by the processing power of the memory NIC, adding memory nodes can increase performance to some extent. When the NIC processing power is sufficient, increasing the memory node can lead to a decrease in index performance due to the increase in RDMA connections [14, 29, 46, 60]. We introduce an Array comparison to illustrate this problem. Array maintains the array length through a lock and inserts data by modifying the array length exclusively and inserting KV pointers. For insert, Array clients have a lot of competition for global locking. Due to the low bandwidth of RDMA CAS operations, the concurrent locking of 128 clients quickly consumes NIC's processing power. Therefore, adding a memory node can improve Array's insert performance. When the lock competition is satisfied, the indexing performance begins to decline. For search operations, since the Array only reads the 8-byte KV pointer and the 48-byte KV, NIC processing power is sufficient, resulting in search performance decreasing with the memory node. Similarly, SepHash allows more concurrent operations through efficient resize operations and zero-wait writes, so its insert performance scales well with memory nodes. The resize operations of RACE and Plush block the insert of concurrent clients and therefore scale limited with memory nodes. The search operations of SepHash and RACE are more complex than Array and require more memory nodes to process requests.

5.6.3 Variable KV sizes. Figure 22 shows the SepHash performance with different KV lengths. The insert performance is almost unaffected by KV sizes, while search performance degrades when the KV size exceeds 256 bytes, which is the PCIE packet size [31]. As shown in Figure 3, the latency of a single RDMA operation rises sharply after the access granularity exceeds 256 bytes, leading to degraded read performance. The bottleneck in insert performance is the RDMA CAS operation, which has a lower upper limit than

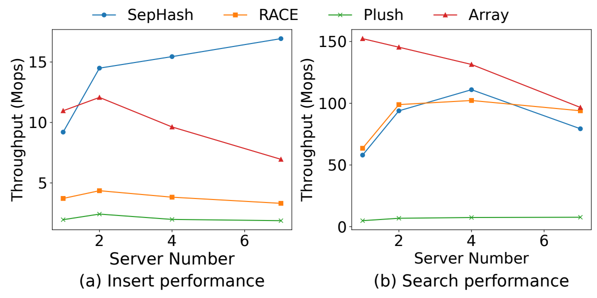


Figure 21: Impact of the number of memory nodes.

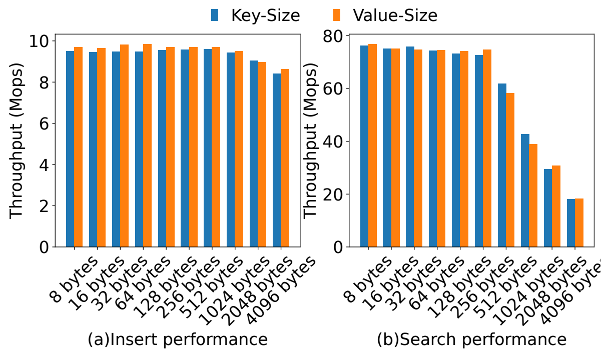


Figure 22: SepHash performance with different KV sizes.

RDMA WRITE/READ operations, so KV size has no significant impact on performance. Therefore, reducing the use of CAS operations in insert operations is important for designing write-optimized indexes. We will explore further optimization in future work.

6 DISCUSSION

Apply to other platforms. SepHash aims to optimize hash resize and concurrency control with trade-offs between RDMA bandwidth and latency. We believe that SepHash can provide write performance advantages on devices that prefer block read/write, making it suitable for various scenarios from SSD-based persistent storage to disaggregated memory based on CXL (Compute express link) [11, 21]. Although CXL has lower latency, it uses the same PCIE interface as RDMA, and we believe it still has a bandwidth and latency trade-off similar to RDMA.

Other write-optimized indexes. dLSM [44], Sherman [43] are also optimized for write-intensive workloads. However, there are notable differences compared to SepHash. First, hash indexes provide high single-point query performance, while tree-based indexes have poor search performance because they must consider range query operations. Our test results show that SepHash achieves over 20× higher read performance compared to dLSM. Therefore, even under write-intensive conditions, the overall performance of tree-based indexes is much lower than SepHash. Second, tree-based indexes generally introduce huge client-side caches or buffers, ranging from a few hundred MB to GB, significantly larger than SepHash's cache overhead of 3 to 20 MB. This can be unacceptable for clients with tight memory resources on disaggregated memory.

Garbage collection. As in previous work [23, 65], we assume that the memory pool will provide the interface for allocating and reclaiming memory. SepHash's memory space is mainly large chunks of segments, which can be easily managed using techniques such as epoch [16, 48] and reference counting [39]. Old segments that are no longer accessible are reclaimed after merging and splitting.

7 RELATED WORK

Indexes on disaggregated memory. RACE [65] is currently the only hash index designed for disaggregated memory. RACE achieves good read performance through shared overflow buckets and a lock-free concurrency strategy. However, its entry-based resize scheme and reread-based concurrency strategy seriously increase resize overhead and write latency, making it inefficient for write-intensive workloads. Sherman [43] and FG-Tree [61] implement distributed B+Tree indexes based on disaggregated memory. FG-Tree distributes B + Tree nodes in different ways and tests their performance. Sherman uses hierarchical locking stored on RDMA NIC memory to optimize concurrent access. dLSM [44] implements LSM-Tree based on disaggregated memory, achieving high write performance compared to B+Tree. ROLEX [23] accelerates access to remote data by caching learned indexes on the client side. SMART [26] finds that the radix tree is a better ordered index for disaggregated memory, providing smaller read and write amplification. By using fine-grained locks and merging accesses to remote memory, SMART improves significantly over B+Tree-based indexes.

Write optimized hash index. Level Hash[64] contains two levels of hash tables: a top level and a bottom level. The top level is twice the size of the bottom level. Each resize operation migrates the data at the bottom level to a new hash table four times its size, thus reducing the amount of data that needs to be transferred for each resize. CLevel[9] further allows multiple hash table levels to coexist and uses a background resize thread to continuously transfer data from the bottom to the top level. Plush [41] combines extendible hash with LSM-Tree to divide resize operations into small data flows between levels. Dual-Stage index [55] divides the index into dynamic and static stages. When data accumulated in the dynamic stage reaches the upper limit, it is merged into the static phase.

8 CONCLUSION

In this study, we propose SepHash, a write-optimized hash index for disaggregated memory. By designing a dual-level separate segment structure and RTT-reduced concurrency control strategy, SepHash eliminates the huge bandwidth consumption in the index resize operation and achieves extremely low write latency. In addition, SepHash designs an FPTable cache and fp filter to improve read performance. Evaluations indicate that SepHash outperforms existing solutions in write-intensive workloads with lower space overhead.

ACKNOWLEDGMENTS

This work was sponsored by the National Natural Science Foundation of China under Grant No.62072196, the National Key Research and Development Program of China under Grant No.2023YFB45027-01, the Key Research and Development Program of Guangdong Province under Grant No. 2021B0101400003, the Creative Research Group Project of NSFC No.61821003.

REFERENCES

[1] 2023. Memcached-a distributed memory object caching system. https://memcached.org/.

[2] Paulo Sérgio Almeida. 2023. A Case for Partitioned Bloom Filters. *IEEE Trans. Computers* 72, 6 (2023), 1681–1691. https://doi.org/10.1109/TC.2022.3218995

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale (Eds.). ACM, 53–64. https://doi.org/10.1145/2254756.2254766

[4] Alex D. Breslow and Nuwan Jayasena. 2020. Morton filters: fast, compressed sparse cuckoo filters. *VLDB J.* 29, 2-3 (2020), 731–754. https://doi.org/10.1007/s00778-019-00561-0

[5] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2477–2489. https://doi.org/10.1145/3448016.3457560

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[7] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz (Eds.). ACM, 1060–1073. https://doi.org/10.1145/3603269.3604833

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1077–1091. https://doi.org/10.1145/3373376.3378515

[9] 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 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[10] 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, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[11] SM CXL Consortium et al. 2022. Compute express link: The breakthrough CPU-to-device interconnect. *Retrieved February* 2 (2022), 2023.

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 79–94. https://doi.org/10.1145/3035918.3064054

[13] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 365–378. https://doi.org/10.1145/3448016.3457273

[14] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[15] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João 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 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[16] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.

[17] IBM. 2018. Advancing Cloud with Memory Disaggregation. https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/.

[18] Young Bae Jun, Sun Shin Ahn, and Hee Sik Kim. 2001. Quotient structures of some implicative algebras via fuzzy implicative filters. *Fuzzy Sets Syst.* 121, 2 (2001), 325–332. https://doi.org/10.1016/S0165-0114(00)00008-7

[19] Per-Åke Larson. 1988. Dynamic Hash Tables. *Commun. ACM* 31, 4 (1988), 446–457. https://doi.org/10.1145/42404.42410

[20] Se Kwon 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. *Proc. VLDB Endow.* 15, 13 (2022), 4023–4037. https://doi.org/10.14778/3565838.3565854

[21] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2022. First-generation Memory Disaggregation for Cloud Platforms. *CoRR* abs/2203.00241 (2022). https://doi.org/10.48550/arXiv.2203.00241 arXiv:2203.00241

[22] 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 *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 1168–1181. https://doi.org/10.1109/HPCA56546.2023.10070986

[23] 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 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[24] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 21–35. https://doi.org/10.1145/3035918.3064015

[25] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *CoRR* abs/2003.07302 (2020). arXiv:2003.07302 https://arxiv.org/abs/2003.07302

[26] 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 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 553–571. https://www.usenix.org/conference/osdi23/presentation/luo

[27] 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 2022, Santa Clara, CA, USA, February 22-24, 2022*, Dean Hildebrand and Donald E. Porter (Eds.). USENIX Association, 313–328. https://www.usenix.org/conference/fast22/presentation/lv

[28] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[29] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 212–227. https://doi.org/10.1145/3477132.3483576

[30] 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 2019, Boston, MA, February 25-28, 2019*, Arif Merchant and Hakim Weatherspoon (Eds.). USENIX Association, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[31] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, Sergey Gorinsky and János Tapolcai (Eds.). ACM, 327–341. https://doi.org/10.1145/3230543.3230560

[32] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[33] Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1386–1399. https://doi.org/10.1145/3448016.3452841

[34] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. 2021. RDMA is Turing complete, we just did not know it yet! *CoRR* abs/2103.13351 (2021).

arXiv:2103.13351 https://arxiv.org/abs/2103.13351

[35] Pedro Reviriego, Alfonso Sánchez-Macián, Stefan Walzer, Elena Merino Gómez, Shanshan Liu, and Fabrizio Lombardi. 2023. On the Privacy of Counting Bloom Filters. *IEEE Trans. Dependable Secur. Comput.* 20, 2 (2023), 1488–1499. https://doi.org/10.1109/TDSC.2022.3158469

[36] 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 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[37] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[38] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 255–270. https://www.usenix.org/conference/nsdi19/presentation/shrivastav

[39] Håkan Sundell. 2005. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[40] Jérôme 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 *IEEE 20th Annual Symposium on High-Performance Interconnects, HOTI 2012, Santa Clara, CA, USA, August 22-24, 2012*. IEEE Computer Society, 48–55. https://doi.org/10.1109/HOTI.2012.19

[41] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (2022), 2895–2907. https://www.vldb.org/pvldb/vol15/p2895-vogel.pdf

[42] Daniel G. Waddington, Clem Dickey, Luna Xu, Travis Janssen, Jantz Tran, and Kshitij A. Doshi. 2020. Evaluating Intel 3D-Xpoint NVDIMM Persistent Memory in the Context of a Key-Value Store. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, 202–211. https://doi.org/10.1109/ISPASS48437.2020.00035

[43] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1033–1048. https://doi.org/10.1145/3514221.3517824

[44] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2835–2849. https://doi.org/10.1109/ICDE55515.2023.00217

[45] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. 2020. Efficient usage of one-sided rdma for linear probing. In *Eleventh International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (AMDS'20)*.

[46] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 87–104. https://doi.org/10.1145/2815400.2815419

[47] 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 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 523–536. https://www.usenix.org/conference/atc21/presentation/wei

[48] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. 2018. Interval-based memory reclamation. *ACM SIGPLAN Notices* 53, 1 (2018), 1–13.

[49] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227. https://doi.org/10.1109/JPROC.2010.2070050

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

[51] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. 2019. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, Arif Merchant and Hakim Weatherspoon (Eds.). USENIX Association, 159–171. https://www.usenix.org/conference/fast19/presentation/yao

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[53] 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, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 266–282. https://doi.org/10.1145/3552326.3567488

[54] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The End of a Myth: Distributed Transactions Can Scale. *CoRR* abs/1607.00655 (2016). arXiv:1607.00655 http://arxiv.org/abs/1607.00655

[55] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*. 1567–1581.

[56] 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 2022, Santa Clara, CA, USA, February 22-24, 2022*, Dean Hildebrand and Donald E. Porter (Eds.). USENIX Association, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[57] 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 *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1345–1359. https://doi.org/10.1145/3514221.3517856

[58] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912. https://doi.org/10.14778/3467861.3467877

[59] 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. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye (Eds.). ACM, 523–536. https://doi.org/10.1145/2785956.2787484

[60] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2 (2023), 131:1–131:26. https://doi.org/10.1145/3589276

[61] 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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 741–758. https://doi.org/10.1145/3299869.3300081

[62] Xiaomin Zou, Fang Wang, Dan Feng, Janxi Chen, Chaojie Liu, Fan Li, and Nan Su. 2020. HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory. *Mass Storage Systems and Technologies* (2020).

[63] Pengfei Zuo and Yu Hua. 2018. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Trans. Parallel Distributed Syst.* 29, 5 (2018), 985–998. https://doi.org/10.1109/TPDS.2017.2782251

[64] 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 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

[65] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory.. In *USENIX Annual Technical Conference*. 15–29.