CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory

Xuchuan Luo[♯♩], Jiacheng Shen[♩], Pengfei Zuo[♈], Xin Wang[♯♉], Michael R. Lyu[♈], Yangfan Zhou[♯♩]

[♯]*School of Computer Science, Fudan University*
[♩]*National Key Laboratory of Parallel and Distributed Computing, China*
[♩]*Duke Kunshan University* [♈]*Huawei Cloud* [♈]*The Chinese University of Hong Kong*
[♉]*Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*

Abstract

Disaggregated memory (DM) is a widely discussed datacenter architecture in academia and industry. It decouples computing and memory resources from monolithic servers into two network-connected resource pools. Range indexes are widely adopted by storage systems on DM to efficiently locate and query remote data. However, existing range indexes on DM suffer from either high computing-side cache consumption or high memory-side read amplifications. In this paper, we propose **CHIME**, a hybrid index combining B+ trees with hopscotch hashing, to achieve low cache consumption and low read amplifications simultaneously. There are three challenges in constructing CHIME on DM, *i.e.*, the complicated optimistic synchronization, the extra metadata access, and the read amplifications introduced by hopscotch hashing. CHIME leverages 1) a *three-level optimistic synchronization* scheme to synchronize read and write operations with various granularities, 2) an *access-aggregated metadata management* technique to eliminate extra metadata accesses by piggybacking and replicating metadata, and 3) an effective *hotness-aware speculative read* mechanism to mitigate the read amplifications of hopscotch hashing. Experimental results show that CHIME outperforms the state-of-the-art range indexes on DM by up to 5.1× with the same cache size and achieves similar performance with up to 8.7× lower cache consumption.

CCS Concepts: • **Information systems** → **Distributed storage**; **Data structures**.

Keywords: Disaggregated Memory, RDMA, Hybrid Index, B+ Tree, Hopscotch Hashing

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695959

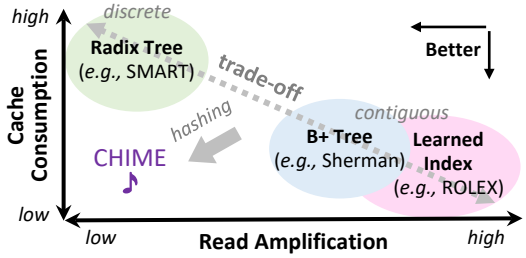


Figure 1. The trade-off between cache consumption and read amplifications for existing DM range indexes [32, 36, 56].

1 Introduction

The disaggregated memory (DM) architecture is being widely discussed due to its potential to improve datacenter resource utilization [2, 21, 28, 43, 48, 50]. It decouples computing and memory resources into independent computing and memory pools and interconnects them with fast networks, *e.g.*, InfiniBand [6] and compute express link (CXL) [11].

Range indexes [32, 36, 56] are cornerstones for storage systems on DM, *e.g.*, databases and key-value (KV) stores [9, 30, 51–53]. They are capable of conducting both point and range queries. For a practical range index on DM, computing-side cache consumption [32, 58] and memory-side read and write amplifications [36, 56] are two critical aspects. First, read and write amplifications waste network bandwidth between computing and memory pools. As network bandwidth is limited, range indexes should minimize amplifications to achieve high throughput [24, 36]. Second, existing approaches cache part of the index structure and addresses of KV items in the computing pool to reduce the overhead of remote index traversals [5, 32, 36, 56]. Range indexes should also reduce cache consumption due to the limited computing-side memory.

Unfortunately, existing approaches cannot simultaneously achieve both low computing-side cache consumption and low memory-side read amplifications because there exists a trade-off between these two aspects, as shown in Figure 1. Range indexes on DM can be classified into two categories, *i.e.*, those that store KV items contiguously [32, 56] and discretely [36]. Discretely storing KV items, *e.g.*, SMART [36], reduces read amplifications since each KV item is mapped to a unique memory address and accessed individually. However, they suffer from high computing-side cache consumption since they need to cache an address for each KV item. In

contiguously: the KV item is stored within B+ tree leaf node, 索引是一个范围值，减少需要cache的索引数据量，但是KV-obj可以出现在leaf node的任意address上。 => access amp
discretely: each KV item is mapped to a unique memory address and access individually, reduce read-amplification. with bigger CM-side Cache.

contrast, storing KV items contiguously, *e.g.*, Sherman [56] and ROLEX [32], exhibits lower cache consumption since these approaches introduce an alignment between memory addresses and keys of KV items, *e.g.*, all KV items in a B+ tree leaf node are within a specific range of keys. Addresses of KV items in the cache can thus be compressed with the alignment. Nevertheless, these approaches suffer from severe read amplifications due to the imprecise and coarse-grained alignment, *e.g.*, a KV item can appear in all possible addresses within a B+ tree leaf node. An entire leaf node has to be fetched to locate a single item. Our experimental analysis shows that the trade-off degrades the state-of-the-art range indexes on DM by up to 7.5× under YCSB workloads [12].

In this paper, we propose to use B+ trees with hopscotch-hashing-based [23] leaf nodes to break the trade-off. This hybrid index contiguously stores KV items, achieving a low cache consumption similar to B+ trees. Besides, using hopscotch hash tables as leaf nodes can map each KV item to a precise location, mitigating read amplifications. While our idea resembles HT-tree [3], *i.e.*, a hardware-dependent conceptual design of combining trees with hash tables, several challenges still need to be addressed to make such an idea practical on DM with commodity hardware.

(1) Complicated optimistic synchronization. Data structures on DM [16, 32, 36, 56] typically employ optimistic synchronization techniques [22], *i.e.*, lock-free reads, to achieve high concurrency. However, the hybrid index has multiple read and write granularities. Optimistically coordinating these accesses is difficult since fine-grained reads cannot directly perceive coarse-grained concurrent writes.

(2) Extra metadata accesses. The hybrid index needs to maintain both metadata for B+ trees and hopscotch hashing to conduct the insert and search operations. Maintaining these metadata introduces extra remote memory accesses, resulting in low performance.

(3) Read amplifications of hopscotch hashing. The space efficiency of the entire hybrid index is a critical aspect after adopting hash tables as leaf nodes since a leaf node can not hold more data than the maximum load factor of the hash table. Hopscotch hashing maps each KV item to a small range of memory, named neighborhood, to achieve a high maximum load factor. The larger the neighborhood size, the higher the maximum load factor. This introduces additional read amplifications since we still need to fetch all items within the neighborhood.

To address the above challenges, we design **CHIME**, a **C**ache-efficient and **H**igh-performance hybrid **I**ndex on disaggregated **ME**mory. To achieve efficient optimistic synchronization, we present a *three-level optimistic synchronization* scheme to synchronize fine-grained reads and coarse-grained writes with cache line versions and bitmaps. To reduce the metadata access overhead, we propose an *access-aggregated metadata management* technique to avoid extra metadata accesses by piggybacking and replicating metadata. Finally, to mitigate the read amplification of hopscotch hashing, we adopt an effective *hotness-aware speculative read* mechanism to further boost the throughput.

We implement CHIME and evaluate its performance with the YCSB benchmark [12]. Compared with Sherman [56] and ROLEX [32], the state-of-the-art B+ tree and learned index on DM, respectively, CHIME achieves up to 4.3× higher throughput in read-only workloads and 2.6× higher throughput in write-intensive workloads, with similar cache consumption. Compared with SMART [36], the state-of-the-art radix tree on DM, CHIME achieves 5.1× higher throughput with the same cache size and similar performance with 8.7× lower cache consumption. The implementation of CHIME is available at https://github.com/dmemsys/CHIME.

In summary, this paper makes the following contributions:

- We reveal the trade-off between read amplifications and cache consumption for range indexes on DM, as well as the superiority of hopscotch hashing on DM, based on experimental analysis.
- We propose the idea of using a hybrid index combining the B+ tree with hopscotch hashing to break the trade-off. We also address the challenges of constructing the hybrid index on DM with CHIME.
- We show in our experiments that CHIME is a practical range index on DM. It achieves up to 5.1× higher throughput than the latest range indexes on DM.

2 Background

2.1 The Disaggregated Memory Architecture

DM is a novel architecture that can potentially address the resource utilization issues of datacenters [1, 35, 50, 55]. It physically decouples the computing and memory resources into two separate pools (*i.e.*, computing and memory pools) and interconnects them with a fast network. Compute nodes (CNs) in the computing pool have many powerful CPU cores to execute intensive tasks but only limited memory serving as local caches. In contrast, memory nodes (MNs) in the memory pool have masses of memory to store application data but only weak computing power for simple tasks, e.g., network connection and memory allocation.

CNs and MNs are interconnected via a high-speed CPU-bypass network, which allows clients on CNs to efficiently access data in the memory pool without involving weak CPU cores on MNs. Without loss of generality, this paper considers that the CNs access MNs through one-sided remote direct memory access (RDMA) verbs (*e.g.*, `READ`, `WRITE` and atomic `CAS`) like previous works [32, 36, 51, 52, 56, 63, 69].

2.2 Range Indexes on Disaggregated Memory

Range indexes are essential for storage systems supporting range queries, *e.g.*, databases [42, 45] and key-value stores [46, 47]. With more and more storage systems ported to DM [2, 30, 51, 56], constructing a high-performance range index on DM becomes increasingly important. We focus on shared indexing scenarios [30, 51–53], where masses of data

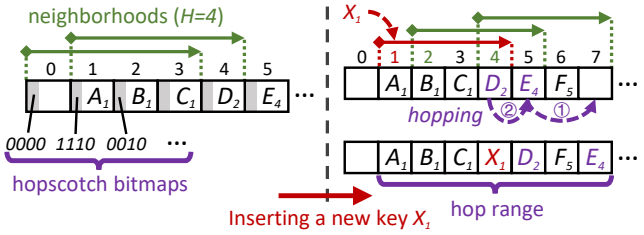


Figure 2. A hopscotch hash table with a neighborhood size of 4. The subscript of each key denotes its home entry.

are stored remotely for all CNs to share access, with only limited memory on each CN used for caching. In this scenario, there are currently three types of range indexes on DM, *i.e.*, B + trees, radix trees, and learned indexes.

B+ trees on DM. Sherman [56] is the state-of-the-art B+ tree on DM. It reduces lock-fail retries with shared *local lock tables* and mitigates the write amplification of B+ trees by enabling fine-grained modifications to the leaf node with *two-level versions*. Marlin [5] is a write-optimized B+ tree on DM designed for variable-length values. It enables clients to operate the same leaf node concurrently via CASes and prevents conflicts between index structure modifications and other concurrent accesses with *ternary-state node locks*. Despite the write optimizations, Sherman and Marlin suffer from the inherent read amplifications of B+ trees.

Radix trees on DM. SMART [36] is the state-of-the-art radix tree on DM that evades the read and write amplifications of B+ trees. It achieves efficient concurrency control with a *hybrid concurrency control scheme*. It also proposes a *read-delegation and write-combining (RDWC)* technique, which coalesces reads and writes on the same keys from the same CN to alleviate network congestion. However, since radix trees need to cache the address of each item, SMART exhibits high cache consumption.

Learned indexes on DM. ROLEX [32] is the state-of-the-art learned index on DM. It leverages machine-learning models on each CN as a cache to avoid caching the internal nodes of tree indexes. It decouples model retraining from one-sided data modifications by adding a bias and data-movement constraints to models. ROLEX also suffers from the read amplification issue since it needs to fetch several predicted leaf nodes during each point query.

2.3 Hopscotch Hashing

Hopscotch hashing [23] is a hash collision resolution algorithm for hash tables, as shown in Figure 2. It hashes each key to an entry in the hash table, *i.e.*, *home entry*, and stores the key within a neighborhood from the entry. A *neighborhood* is defined as H consecutive entries, where H is a pre-configured neighborhood size. A H-bit *hopscotch bitmap* is embedded inside each entry to efficiently track the occupancy status of the neighborhood from this entry. Each bit indicates whether each key is originally hashed to this entry.

When inserting a key X, hopscotch hashing finds the first empty entry starting from the key's home entry with linear probing. If the empty entry is outside the neighborhood, it searches previous $H-1$ items for the farthest one that can be swapped into the empty entry without violating the neighborhood constraint. The swap (*i.e.*, hop) is conducted and repeated until the empty entry is within the neighborhood of the home entry, where X can be directly inserted. If no empty entries or movable items exist, the hash table should be resized. We define the *hop range* as the smallest range of entries that is affected by the entire hopping process.

3 Analysis of Indexes Built on DM

This section motivates the idea (§ 3.1) and presents the challenges (§ 3.2) of building a hybrid index on DM.

Without explicit mention, all experiments in this section are conducted with 10 CNs (each with 64 clients) and 1 MN. Each CN and MN is equipped with a 100 Gbps Mellanox ConnectX-6 NIC. We use YCSB workloads [12] (including 60 million entries) with 8-byte keys and 8-byte values like previous studies [5, 32, 36, 56].

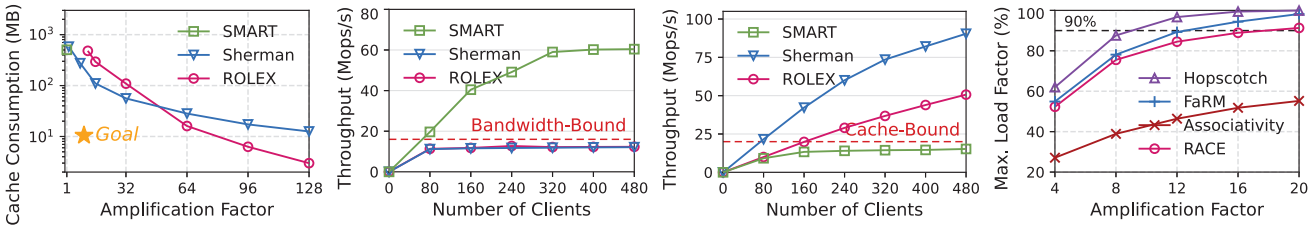
3.1 Motivation: Existing Indexes on DM

There are two design requirements for range indexes on DM. First, one-sided RDMA verbs are preferred on DM to enable clients to bypass the weak CPUs on MNs. This makes the network a performance bottleneck and requires range indexes on DM to ***reduce read amplifications*** [36]. Second, range indexes on DM [5, 32, 36, 56] cache part of the index structure and addresses of KV items on each CN. If the CN cannot cache the entire index, cache misses will result in multiple network round-trips on critical paths due to remote index traversals. Even if computing-side memory can hold the entire index, the valuable local memory should still be saved as much as possible to support larger datasets or preserved for other applications or indexes to use. Thus, range indexes on DM should ***reduce cache consumption*** [32]. However, there is a trade-off between these two requirements.

3.1.1 A trade-off between read amplifications and cache consumption for range indexes on DM. Range indexes on DM can be categorized into two types based on whether the addresses of KV items inside them exhibit continuity or not: *KV-contiguous* and *KV-discrete* range indexes.

KV-contiguous range indexes. B+ trees and learned indexes are KV-contiguous. They store KV items contiguously within large leaf nodes, which aligns the stored keys with the memory addresses of KV items. B+ trees and learned indexes can thus achieve low cache consumption with the alignment. Specifically, for B+ trees, caching the address of a leaf node and the range of keys within it enables clients to access all items in the node without needing to cache their exact addresses, as the items are stored contiguously. For learned indexes, we use machine-learning models to capture the relationship between keys and addresses and use the model as a cache.

However, B+ trees and learned indexes suffer from high read amplifications since entire leaf nodes are fetched to CNs



(a) The trade-off for range indexes. **(b)** Performance w/ limited bandwidth. **(c)** Performance w/ limited caches. **(d)** The trade-off for hashing schemes.

Figure 3. The comparisons of existing range indexes and hashing schemes on DM.

during point queries. Even worse, for learned indexes, more than one leaf node needs to be fetched due to the imprecise model predictions [32, 58]. The read amplification causes more bandwidth consumption, resulting in low throughput.

KV-discrete range indexes. Radix trees are KV-discrete since each leaf node holds only one KV item allocated at discrete addresses. This enables minimal read amplifications since only one KV item is fetched. However, using radix trees requires caching addresses of individual KV items, leading to high cache consumption. This prevents CNs from fully caching internal nodes, resulting in multiple network round-trips for traversing the remote index.

Analysis. We define the *amplification factor* as the theoretical ratio of bandwidth consumption from the server and bandwidth returned to the application. In the following, we do not consider the amplification caused by metadata, as its impact is minor [36]. For a radix tree, the amplification factor is 1 since its leaf node contains only one KV item. For a B+ tree, the amplification factor is the *span size*, *i.e.*, the number of entries in a node, of the leaf node. For a learned index, following the default settings of ROLEX, we keep the pre-defined model error equal to the span size. Therefore, the amplification factor is twice the span size of the leaf node since the learned index generally needs to fetch two leaf nodes for each search with the error.

Experiments. Figure 3a shows the trade-off between cache consumption and read amplification factors for DM range indexes. For Sherman and ROLEX, the lower the amplification factor (span size), the higher the cache consumption. SMART achieves the minimum read amplification but at the cost of high cache consumption.

Figures 3b and 3c show the performance comparison of DM range indexes under the read-only workload with limited bandwidth and caches, respectively. With 1 MN (*i.e.*, limited bandwidth) and 1000 MB caches on each CN, the peak throughputs of Sherman and ROLEX are 4.9× lower than SMART since they are bandwidth-bound due to high read amplifications. With 10 MNs and 100 MB caches (*i.e.*, limited caches) on each CN, the throughput of SMART under 480 clients is 5.9× and 3.3× lower than Sherman and ROLEX, respectively, since the limited caches fail to store all the internal nodes in SMART, resulting in poor performance.

A straightforward idea. To break the trade-off, a straightforward idea is to combine B+ trees with hash-table-based leaf nodes.[1] Ideally, hash-table-based leaf nodes can map each KV item to a specific address in a leaf node. Clients only need to fetch a single entry instead of the entire node, reducing read amplifications and preserving the low cache consumption of B+ trees. However, using hash tables introduces additional read amplification and space efficiency issues due to the unavoidable hash collisions.

3.1.2 A trade-off between read amplifications and space efficiency for hashing schemes on DM.

On DM, both space efficiency and read amplifications matter. A space-efficient hashing scheme can save the expensive memory capacity in the memory pool. Meanwhile, a lower read amplification enables higher performance. However, existing hashing approaches suffer from a trade-off between read amplifications and space efficiency.

The space efficiency for a hash table is determined by how hash collisions are resolved. Based on whether separate data structures are used to handle hash collisions, existing approaches can be classified into *closed-addressing* and *open-addressing* schemes. Closed-addressing schemes on DM typically adopt *associative buckets* to resolve hash collisions [51, 52, 59, 69]. Each key is hashed to an associative bucket containing multiple entries. When searching for an item, a client fetches all entries in a bucket, making the amplification factor the size of the bucket. For open-addressing schemes, collisions are resolved by finding an empty entry according to some probe sequences [23, 27, 44]. Read amplifications are induced since multiple entries are fetched iteratively [27] or parallelly [23, 44].

This trade-off urges us to widely choose a suitable hashing scheme that can achieve high space efficiency and low read amplifications. In our following experiments, we find that hopscotch hashing best fits our requirements.

Experiments. The space efficiency of hashing schemes is evaluated by the *maximum load factor*, *i.e.*, the ratio of the maximum number of stored items to the total number of entries in the hash table. We evaluate the maximum load factors of associativity, hopscotch hashing, RACE [69] and FaRM [16], with each hash table comprising 128 entries.

[1] Learned indexes are not considered since imprecise model predictions may introduce additional memory accesses, which will be verified in Section 5.3.

RACE is the state-of-the-art hash table on DM with three design choices, *i.e.*, associativity, two choices [41], and overflow colocation [68]. Its amplification factor is four times the associative bucket size since each item can be located in four buckets. FaRM proposes a chained associative hopscotch hashing. It fixes the neighborhood size to two associative buckets and chains an overflow block for each bucket. We disable the chained block design since it is unfriendly to DM. The amplification factor of FaRM is thus twice the associative bucket size. Other RDMA-based hashing schemes [39, 59] are not considered since they are unsuitable for DM [69].

Figure 3d shows the trade-off between the maximum load factor and the read amplification factor for the hashing schemes. The hopscotch hashing achieves the best space efficiency with low amplification factors.

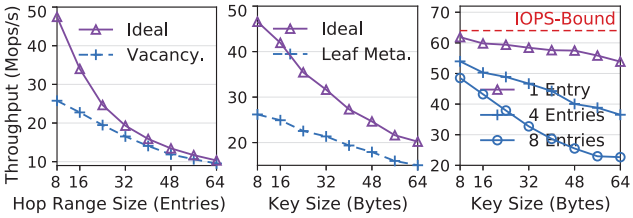
A viable idea. Based on the above analysis, we propose to break the trade-off between cache consumption and read amplifications by using **a B+ tree with hopscotch-hashing-based leaf nodes**, *i.e.*, *hopscotch tree*. The characteristic of the B+ tree helps achieve comparable cache consumption to large-span B+ trees, while hopscotch hashing reduces read amplifications with high space efficiency.

3.2 Challenges: The Hybrid Index on DM

The hopscotch tree retains the internal node structure of a B+ tree but replaces its leaf nodes with hopscotch hash tables, *i.e.*, *hopscotch leaf nodes*. For read operations, it first traverses the internal nodes like a B+ tree to find the hopscotch leaf node containing the target KV item. It then reads the neighborhood encompassing the item from the leaf node. For write operations, it locates the target leaf node in the same way, and then either directly updates the KV item or inserts a new one via a hopping process, as introduced in Section 2.3. If there is no feasible hopping, a node split and up-propagation are performed, similar to a B+ tree, to create new space in the hopscotch leaf node. Even though the hopscotch tree can meet both requirements of reducing read amplifications and cache consumption, three challenges still have to be overcome before it becomes a practical approach.

3.2.1 Complicated optimistic synchronization. The optimistic synchronization is preferred by DM since it allows lock-free `READ`s [22]. However, achieving it on a hopscotch tree is challenging due to various granularities in `READ`s and `WRITE`s, including reading a neighborhood (*i.e.*, *neighborhood read*), updating an entry (*i.e.*, *entry write*), writing a hop range to insert an item (*i.e.*, *hop range write*), and writing an entire node during a node split (*i.e.*, *node write*).

Specifically, optimistic synchronizations use verification information, *e.g.*, versions [16, 40, 56] and checksums [36, 39], to enable readers to identify conflicting write operations. For version-based approaches, version numbers are spread across the entire write region and updated altogether during each `WRITE`. For checksum-based approaches, a checksum in each write region is re-calculated and updated during each `WRITE`.



(a) Vacancy bitmap. **(b)** Leaf metadata. **(c)** Neighborhood.

Figure 4. The effects of metadata accesses and the neighborhood size. "*Vacancy*", "*Leaf Meta*" represent reading: 1) the vacancy bitmap before the hop range (2 accesses), and 2) the leaf metadata and the neighborhood (2 accesses).

Both approaches require readers to fetch the entire write region to check if there are concurrent write operations. This introduces two problems to the hopscotch tree:

1) The verification information for coarse-grained writes is invisible to fine-grained reads. This is because a fine-grained read region cannot encompass the entire write region (*i.e.*, an entire node or a hop range), making the verification information unavailable to reads.

2) The verification information for the hop range writes is difficult to maintain. Hop ranges vary in size and overlap with each other in leaf nodes, which makes existing version-based or checksum-based approaches infeasible.

3.2.2 Extra metadata accesses. The hopscotch tree has to maintain the metadata for both the hopscotch hashing and the B+ tree, which introduces extra remote memory accesses on DM, decreasing the overall performance.

For insert operations on hopscotch hashing, to minimize read amplifications, only a hop range should be fetched on data accesses instead of an entire leaf node. To achieve this, a *vacancy bitmap*, where each bit represents the empty status of each entry, should be maintained for each leaf node to enable clients to identify the hop range before they probe the hash entries. This inevitably introduces extra remote memory accesses on the critical paths of insert operations to read and write the vacancy bitmap. We evaluate the overhead of maintaining the vacancy bitmap in Figure 4a by continuously issuing `READ`s. Compared with the ideal case where only the hop range needs to be fetched, the extra access of the vacancy bitmap decreases the throughput by up to 1.8×.

For the B+ tree, leaf metadata, *e.g.*, fence keys and sibling pointers, are maintained in the header of each leaf node for clients to validate the correctness of reads [19, 31]. Thus, the leaf metadata should be fetched during each query. However, as the neighborhood to read in the hopscotch leaf node is usually not adjacent to the in-header leaf metadata, clients have to either read the metadata via dedicated accesses or read the entire node. Figure 4b shows the impact of the extra leaf metadata access. With 8-byte keys and 8-byte values, the extra access brings a 1.8× reduction in throughput.

3.2.3 Read amplifications of hopscotch hashing. Hopscotch hashing can reduce the read amplification from the

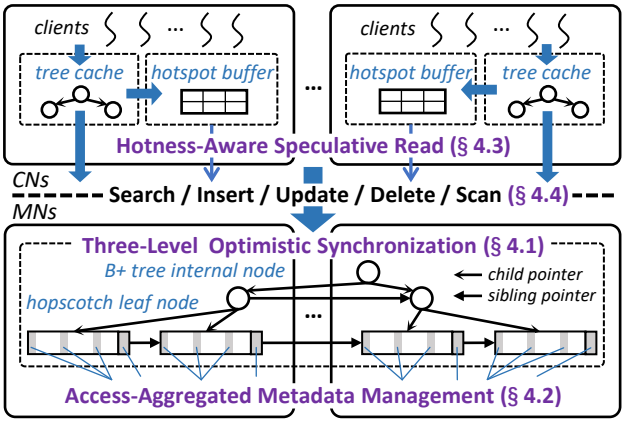


Figure 5. The overview of CHIME.

size of a leaf node to the size of a neighborhood. However, the neighborhood should still be configured more than a threshold (*e.g.*, 8 entries) to ensure an acceptable maximum load factor (*e.g.*, ≈ 90%), as shown in Figure 3d. This still incurs read amplifications compared with the optimal case where only a single KV item is read. We evaluate the performance decline caused by the read amplifications of hopscotch hashing by continuously issuing READs to the MN. As shown in Figure 4c, reading 8-entry neighborhoods shows at least 1.3× lower throughput than the optimal case, *i.e.*, the 1-entry neighborhood. This is because the network bandwidth is the bottleneck when reading 8-entry neighborhoods. With the same key sizes, 8-entry neighborhoods consume more bandwidth than 1-entry ones, resulting in lower throughput. Note that the IOPS upper bound of memory-side RDMA NICs becomes the new bottleneck when the data blocks are small enough [36]. Thus, reading 1-entry neighborhoods (IOPS-bound) cannot achieve an 8× throughput of reading 8-entry neighborhoods (bandwidth-bound).

4 The CHIME Design

We propose CHIME, a cache-efficient and high-performance range index for DM. Figure 5 shows the overview. To achieve better concurrency, we present a *three-level optimistic synchronization* scheme, including a two-level versioning and a bitmap check to detect B+-tree-related writes and entries hopping, respectively (§ 4.1). To avoid extra metadata accesses, we propose an *access-aggregated metadata management* technique to eliminate the accesses by piggybacking and replicating metadata (§ 4.2). To reduce read amplifications of hopscotch hashing, we adopt a *hotness-aware speculative read* mechanism to boost the throughput further (§ 4.3). Finally, we summarize operations CHIME supports (§ 4.4).

4.1 Three-Level Optimistic Synchronization

CHIME adopts B+-tree-like internal nodes for low cache consumption and hopscotch leaf nodes to reduce read amplifications with high space efficiency.

As shown in Figure 6, both internal and leaf nodes of CHIME consist of a header, an array of entries, and an 8-byte lock. In the header, a *level* byte is used to indicate the level of the node. Nodes closer to the root have higher level values. Leaf nodes are at level 0, and their parent nodes are at level 1, and so on. When the root node R is split, its level, *i.e.*, *level(R)*, remains unchanged, and the new root node's level is assigned to *level(R)+1*. A *valid* byte is used to indicate the deleted state. *Fence keys* [19] are the lower and upper bounds of keys in the node. Each node points to a sibling node via an 8-byte *sibling pointer* like a B-link tree [31]. Fence keys and sibling pointers are used to ensure the correctness of reads during node splits, which will be discussed in Section 4.2.2. Each internal node entry stores a *pivot key* to guide the search direction and an 8-byte *child pointer* pointing to a subsequent node to traverse. Each leaf node entry stores a KV item. A 2-byte hopscotch bitmap is used to support a maximum neighborhood size of 16.

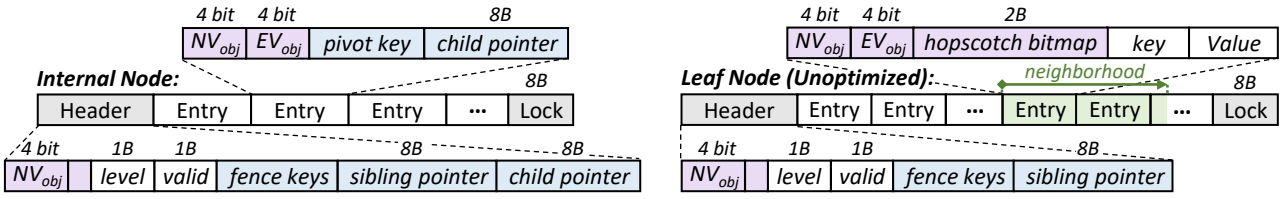
CHIME adopts lock-based writes and lock-free reads on each node. For write-write conflicts, the lock in each node is used to synchronize writes. For read-write conflicts, the verification information (*e.g.*, *NV* and *EV*) ahead of each node and entry is used to synchronize reads with a write.

As mentioned in Section 3.2.1, detecting read-write conflicts on leaf nodes is challenging since fine-grained reads cannot perceive coarse-grained writes, *i.e.*, node and hop range writes. We decompose the problem into detecting node and hop range writes and address them with *two-level cache line versions* and *reused hopscotch bitmaps*, respectively, which form the *three-level optimistic synchronization*. *NV* and *EV* are two levels, and the hopscotch bitmap is the third level.

4.1.1 Two-level cache line versions.

Cache line versioning is a widely adopted technique to resolve read-write conflicts among one-sided RDMA operations [16, 17, 66]. It places a version number at the start of each cache line and object. Concurrent modifications can be detected by checking whether versions within a fetched object are the same. Such a scheme also works when partially reading an object. However, directly applying cache line versions to tree nodes forces clients to update all versions and write back the entire node whenever a single entry is modified, resulting in severe write amplifications. The root cause is that all entries inside a node share the same cache line versions with the node. To address this, we design *two-level cache line versions* that decouple the versions of nodes and entries to achieve fine-grained writes.

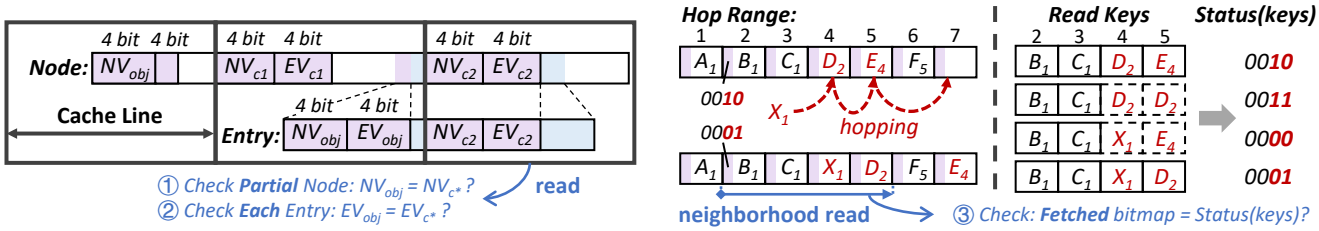
As shown in Figure 7a, we embed each cache line, node, and entry with a 1-byte two-level version field, each storing a 4-bit *node-level version (NV)* and a 4-bit *entry-level version (EV)*. When clients write a node, they increment all the node-level versions within the node (including those at the start of entries). When clients update an entry, they increment the entry-level versions within the entry and only write back the modified entry, eliminating the write amplification. When clients read part of a node, *e.g.*, a neighborhood, they retry



(a) The internal node.

(b) The leaf node (Unoptimized).

Figure 6. The node structures of CHIME. "*NV*" and "*EV*" represent node-level and entry-level versions, respectively. For brevity, the cache line versions are omitted. The optimized leaf node structure will be introduced in Section 4.2.



(a) Two-level cache line versions for detecting node and entry writes.

(b) Reused hopscotch bitmaps for detecting hop range writes.

Figure 7. The three-level optimistic synchronization.

their read operations when they find that 1) the fetched node-level versions cannot match, or 2) for any entry they fetched, the entry-level versions within the entry cannot match.

We use the same 4-bit version as in Sherman [56], which is sufficient to handle read-write conflicts. This is because write operations on each node are synchronized by the lock. There will be only one write operation and multiple concurrent read operations on the version number at the same time.

4.1.2 Reused hopscotch bitmaps. The above versioning cannot detect concurrent hop range writes since the overlapping and variable sizes of hop ranges make versions hard to maintain. Figure 7b shows an example of a hop range. Keys E and D are hopping from entries 5 and 4 to 7 and 5. The hopscotch bitmap inside the home entry of D (*i.e.*, entry 2) is updated from 0010 to 0001 since D is moved from the third entry from its home entry to the fourth. The hop range with updated bitmaps is written back to the MN via a `WRITE`. Concurrent reads may fetch intermediate states of the write, resulting in an incorrect search. We define entries storing hopped keys as *hop entries*. As modifications to KV items within a hop range reside in hop entries, the hop range check can be decomposed into intra-entry and inter-entry checks. The former is solved by the entry-level versions proposed above. Thus, we only need to consider the latter, *i.e.*, detecting the location changes of hopped items.

To address this issue, we find that hopscotch hashing guarantees that **the home entry of a new key in a hop entry always differs from that of the original key, or the hop entry is originally empty**. This is because the hopscotch algorithm prioritizes swapping with a farther entry during each hop, as stated in Section 2.3. For example, the hop entries in Figure 7b are 4, 5, and 7. For entry 5, the home entry of D differs from that of E since D can otherwise directly hop to entry 7. For entry 4, the home entry of X differs from that of D since X can otherwise directly hop to entry 5. Based on this observation, we come up with reusing the *hopscotch bitmaps* to indicate the locations of hopped items.

The right part of Figure 7b shows all possible locations of keys fetched by a neighborhood read from entry 2. After fetching a neighborhood, readers re-construct a hopscotch bitmap of the target home entry, *i.e.*, *status(keys)*, according to the hash values of actual keys in the fetched neighborhood. Readers then compare the *status(keys)* with the fetched hopscotch bitmap in the home entry of the target key and re-read if the two bitmaps do not match. This enables clients to detect whether they read an intermediate state of the neighborhood, *i.e.*, the middle two rows in the right part of Figure 7b, where some KV items are unavailable.

4.2 Access-Aggregated Metadata Management

As described in Section 3.2.2, maintaining the vacancy bitmap and leaf metadata introduces expensive extra remote memory accesses on the operation's critical path. CHIME proposes an access-aggregated metadata management technique to eliminate these accesses. In this section, we first introduce how we avoid the extra accesses to the vacancy bitmap and leaf metadata. We then present a sibling-based validation mechanism to further mitigate the metadata overhead.

4.2.1 Vacancy bitmap piggybacking. The vacancy bitmap indicates the locations of empty entries in a node, which is used to avoid read amplifications during insertions. Before inserting a new key, the client first acquires the lock of the target node via a `CAS`. After acquiring the lock successfully, the client `READ`s the corresponding vacancy bitmap to identify the expected hop range. The client then `READ`s the range without read amplifications rather than fetching the entire node. However, reading the vacancy bitmap introduces an

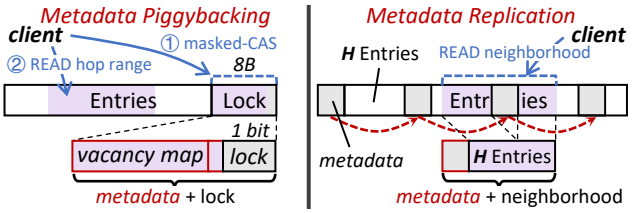


Figure 8. The access-aggregated metadata management.

extra remote memory access, which increases the latency and limits the throughput of the entire insert operation.

To address this issue, we find that we can leverage the unused bits in the lock to represent the vacancy bitmap. Specifically, we only need 1 bit to represent the lock state but have to construct locks with 8 bytes as required by RDMA atomic operations. Based on this observation, we **piggyback the vacancy bitmap read onto the lock acquirement** to eliminate the extra access. We achieve this with the *masked compare and swap* (`masked-CAS`), an advanced feature of RDMA, which is available on ConnectX-2 NICs and above [14, 15]. `Masked-CAS` enables a client to restrict the compare check and the swap to two portions of the 8-byte region, respectively. Two bitmasks (*i.e.*, *compare_mask* and *swap_mask*) indicate the two portions, where the masked-out bits will not get compared or swapped.

As shown in the left half of Figure 8, we embed the vacancy bitmap into the 8-byte *Lock* field, where only the last bit serves as a lock. The client issues a `masked-CAS` at the *Lock* field to acquire the lock, with a *compare_mask* value of `0x1` and a *swap_mask* value of `MAX_UINT64`. The *compare_mask* enables the client to acquire the lock without involving the vacancy bitmap. The *swap_mask* enables swapping the entire *Lock* field (including the vacancy bitmap) back to the client. In this way, the client gets the vacancy bitmap during lock acquirement with no extra access. Besides, the empty status of entries in a node only changes during insertion or deletion when the node is locked. Thus, any modifications to the vacancy bitmap can be piggybacked onto the lock releasing via a `WRITE`. As both reads and writes on the vacancy bitmap are piggybacked onto lock operations, the vacancy bitmap accesses are completely eliminated.

Note that the maximum size of the vacancy bitmap is 63 bits within the 8-byte *Lock* field. If the span size of the leaf node is larger than the vacancy bitmap size, we map each bit to several entries as evenly as possible.

4.2.2 Leaf metadata replication. CHIME adopts the same node split process as Sherman [56], moving KV items to a newly allocated sibling node on the right. Specifically, CHIME allocates a 16 MB memory chunk to each client via a remote procedure call (RPC) each time. When creating a new node, the client uses space from the pre-allocated chunk. If the space is exhausted, a new chunk is allocated via RPC. After allocating space for the new node, the client copies the items to be moved into the new node via a `WRITE`. After the new node is written back, the client deletes the moved items in the old node, sets the sibling pointer to the new node's address, and unlocks the old node via a single `WRITE`. Writing the new node before the old node ensures that the new node will not be accessed until the old node points to it. Thus, concurrent conflicts between `READ`s and `WRITE`s only occur on the old node, which can be managed well by the *three-level optimistic synchronization*. However, read operations may miss some moved KV items due to the node split process.

Existing approaches use fence keys to ensure the correctness of reads during node splits [19, 31, 56]. Specifically, two validations should be conducted after reading a node, *i.e.*, *cache validations* and *half-split validations*. The first is the cache validation. CHIME caches internal nodes on each CN. The cached nodes may be outdated due to node splits of the remote tree triggered by other CNs. To detect the cache invalidation, the client checks whether fence keys are consistent with the cached pivot keys. If not, the cached node will be invalidated. The second is the half-split validation. Although reads and node writes are well synchronized, a read may miss a key due to a *half-split* [60], where a node is split into two nodes with data moved, but the parent node update is pending. To detect it, a client checks whether the target key is within the range indicated by fence keys. If not, the client continues to `READ` the sibling node, where the target key has been moved to.

Everything performs well until the introduction of neighborhood reads for leaf nodes. The client has to read the neighborhood and the leaf metadata with two dedicated `READ`s since the neighborhood is usually not adjacent to the in-header metadata. To avoid this, we **replicate the metadata across the leaf node** to make it available to every neighborhood, as shown in the right of Figure 8. We insert a metadata replica at the position of every H entries, where H is the neighborhood size. For any neighborhood on the node, there is a replica that is either encompassed by it or adjacent to it. Thus, the leaf metadata can be fetched along with the neighborhood via a `READ`. Besides, since the leaf metadata is only changed during a node split or merge, all the replicas can be updated and written back along with the node write without introducing inconsistencies.

4.2.3 Sibling-based validation. Replicating fence keys to conduct validations introduces additional memory overhead, *i.e.*, an additional $\frac{2 \cdot key\_size}{H}$ bytes for each entry. To achieve better memory efficiency, we propose a *sibling-based validation* to achieve cache and half-split validations by reusing the sibling pointers in leaf nodes, as shown in Figure 9:

1) Cache validation. If a client reads a remote leaf node according to a **cached** pointer, it checks whether the sibling pointer in the leaf node equals the **next** child pointer in the cached parent node. If not, a mismatch is found, which indicates that a newly inserted sibling node (*i.e.*, $leaf_{A'}$) is

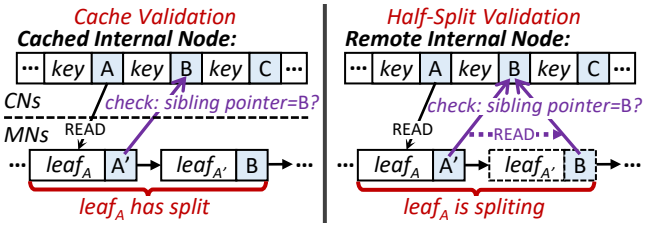


Figure 9. The sibling-based validation. For clarity, the address of leaf node $leaf_x$ is x, where $x = A, A', B$.

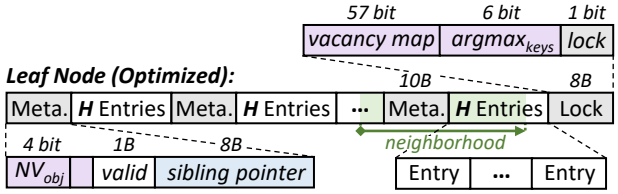


Figure 10. The optimized leaf node structure of CHIME.

invisible to the client due to the outdated cached node. The client then invalidates the cached node and retries the search.

2) Half-split validation. If a client reads a remote leaf node according to a **remote** pointer, it conducts the same sibling pointer checks. If the target key is not found in the leaf node and the sibling pointer is mismatched, it indicates that the target key may be moved to sibling nodes. The client then READs the sibling node. The process is repeated until the target key is found or the sibling pointer is matched.

A corner case happens when a half-split is found during an insertion. A client cannot determine whether to insert the new key into the split node (*i.e.*, $leaf_A$) or the sibling node (*i.e.*, $leaf_{A'}$), since the sibling pointer does not include the pivot key of the two nodes as fence keys do. To address this, we embed a $argmax_{keys}$ field inside the *Lock* field to indicate the position of the maximum key in the node and maintain it similarly to the vacancy bitmap. The client can READ the maximum key of the split node according to the $argmax_{keys}$ value and then determine which node to insert into. Figure 10 shows the final optimized leaf node structure with metadata refined. Apart from the fence keys, the *level* byte is removed since it is always zero for leaf nodes.

4.3 Hotness-Aware Speculative Read

CHIME adopts a hotness-aware speculative read scheme to reduce amplifications of reading neighborhoods. We use a hotspot buffer to store the locations of some entries and shortcut the index search when the target is in the buffer.

The hotspot buffer. We define *hotspots* as the frequently accessed entries in leaf nodes. A hotspot buffer on each CN is a small cache that stores descriptions of the hotspots. As shown in Figure 11, each buffer entry stores an 8-byte *leaf address*, a 2-byte *key index*, a 2-byte *fingerprint* and a 4-byte *counter*. A key index is the position of a hotspot in a leaf node. The pair of leaf address and key index can uniquely determine a hotspot. The fingerprint of the key in the hotspot is stored in the buffer to reduce the probability of incorrect speculations. We limit the size of the hotspot buffer to avoid

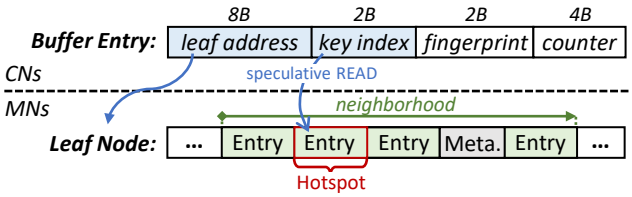


Figure 11. The hotness-aware speculative read.

high cache consumption. The counter is used to track how frequently the hotspot is accessed.

Whenever CHIME accesses a remote KV entry, it updates the hotspot buffer. Specifically, if the entry is already in the buffer (*i.e.*, it is a hotspot), CHIME checks if the fingerprint matches the key just read. If it does not match, it indicates that the buffer entry is outdated. In this case, update the fingerprint and reset the counter to 1. Otherwise, increment the counter to track the hotspot's frequency. If the entry is not in the buffer, insert its description. If the buffer is full, a buffer entry is evicted using the least frequently used (LFU) replacement strategy.

Speculative reads. The speculative read is a mechanism to greedily fetch the target entry rather than reading the neighborhood, aiming at reducing read amplifications further. It relies on the hotspot buffer to record the precise locations of hot entries for the speculation. Before conducting a neighborhood read for a key, the client first searches the buffer for hotspots within the neighborhood. The fingerprint is checked to exclude incorrect locations for the target key. If no matched hotspot is found, a normal neighborhood read is conducted. Otherwise, the client selects the hottest one by comparing counters and then speculatively READs the selected hotspot. If the target key is found in the entry, the speculative read succeeds, and the read amplification of hopscotch hashing is avoided. Otherwise, the neighborhood should be READ.

4.4 Operations

Put all designs together. We assume there are h layers of internal nodes. The number of round-trips for each operation is shown in Table 1, where for brevity, we do not count the round-trips caused by lock-fail retries, cache or half-split invalidations, and node splits or merges. In the best case, all internal nodes are cached on the CN. In the worst case, the internal nodes are not cached on the CN.

Operations on CHIME are described as follows. All operations first search the cache. A remote traversal starts from the deepest cached child pointer. A client can identify whether the child node is a leaf or an internal node according to the level of its parent node.

Search. If the child node is an internal node, the client directly READs it. Two-level cache line versions and fence keys are checked to detect concurrent writes and node splits. The client locates the next child pointer according to pivot keys. The process is repeated until the child node is a leaf node (0-h RTTs). The client determines the neighborhood to read

Table 1. Numbers of round-trips for each operation.

	Search	Insert	Update/Delete	Scan
Best	1 or 2	3	3 or 4	1
Worse	h+1 or h+2	h+3	h+3 or h+4	h+1

by hashing the target key. A speculative READ is conducted if a matched hotspot is found in the buffer. Otherwise, the client READs the neighborhood (1 RTT). In the case of incorrect speculation, which occurs infrequently, an additional neighborhood READ is required (1 RTT). For a *wrap-around* corner case, where the set of entries to probe (*e.g.*, a neighborhood) goes past the end of the hash table and continues from the beginning, the client READs the two segments with doorbell batching [24]. Two-level cache line versions, hopscotch bitmaps and sibling pointers are checked to detect concurrent writes, entries hopping and node splits, respectively. If the checks pass, search the neighborhood for the target key and return.

Insert. The client first identifies the leaf node like the search (0-h RTTs). Then, it locks the leaf node and gets the vacancy bitmap via a masked-CAS (1 RTT). After READing the hop range according to the vacancy bitmap (1 RTT), it inserts the new key into the fetched range via a hopping:

- If the hopping succeeds, increment the entry-level versions within each hop entry, WRITE the hop range back, unlock the node via another WRITE, and return (1 RTT, with the two WRITEs combined like Sherman [56]). If the hop range wraps around the leaf node, the client WRITEs the segments with doorbell batching.
- If the hopping fails, the client READs the rest of the node and conducts a node split. For leaf nodes, the split key is the median among the keys to hop during the failed hopping. This ensures a successful key insertion into the split or new node. The client moves items with keys larger than the split key to the new node's corresponding entries, updates hopscotch bitmaps, and WRITEs the new node to a newly allocated address. It then WRITEs back the old node, with the sibling pointer pointing to the new node, the vacancy bitmap updated, and the lock released.

After the node split, an up-propagation process is required to insert the split key and the new leaf address into the upper part of the tree. CHIME follows the same process as Sherman [56]. Let nodes A, A' and P denote the split node, the new node, and the parent node of node A, respectively:

- *(Step 1)* If node A is not the root node, execute *Step 2*; otherwise, execute *Step 3*.
- *(Step 2)* Lock node P, fetch it, and insert a new entry (*i.e.*, the split key and the address of A') into it. If node P is not full, write it back, unlock it, and return. Otherwise, split and unlock it. Let A=P and go back to *Step 1*.
- *(Step 3)* Allocate a new root node that points to nodes A and A'. Update a global pointer via CAS, which maintains the latest root address, and return.

Update. A client identifies (0-h RTTs) and locks (1 RTT) the leaf node like the insert and then gets the target entry like the search (1-2 RTTs). It modifies the entry and increments entry-level versions within it. Lastly, it WRITEs the entry back and unlocks the node via another WRITE (1 RTT, where the two WRITEs are also combined like the insert).

Delete. In case of no leaf node merging, a delete operation only needs to clear the target entry via the update process. Otherwise, a node merge is triggered like DM B+ trees [5, 56], where node-level versions are used to detect inconsistencies.

Scan. We conduct a scan like Sherman [56], where the client fetches target internal nodes (0-h RTTs) and leaf nodes (1 RTT) via parallel READs, respectively. Besides, to reduce read amplifications, we analyze neighborhood intersections of target keys and exclude unnecessary entries for each read.

4.5 Discussions

Supporting variable-length keys and values. By default, CHIME stores KV items in leaf nodes. To support variable-length values, CHIME stores a key and an 8-byte pointer in each leaf entry like previous works [5, 26, 32, 40, 58]. The remaining item content is stored in a block linked by the pointer. To further support variable-length keys, CHIME adopts a similar approach to PACTree [26]. The first 8 bytes of the key, used as a fingerprint, are stored in the leaf nodes, while the rest of the key and value are stored in the block linked by the pointer. The block needs to maintain information such as key length and value length. In the case of fingerprint collisions, CHIME simultaneously fetches all linked blocks that match the partial key. Since fingerprint collisions are rare, this overhead is acceptable. We will evaluate the performance of CHIME with variable-length KV items supported in Section 5.2.

The first one-sided RDMA-based hopscotch hashing. A hopscotch hash table based on one-sided RDMA can be formed with the *three-level optimistic synchronization*, where node-level versions can synchronize reads with the table resizing. To the best of our knowledge, this is the first hopscotch hash table that can be optimistically synchronized using only one-sided RDMA verbs.

Generality of techniques in CHIME. Some techniques in CHIME can also be applied to other kinds of indexes: 1) The *hopscotch leaf node* can benefit any KV-contiguous range index (*e.g.*, learned indexes) to mitigate the read amplification. 2) The *vacancy bitmap piggybacking* technique can piggyback any other small-sized metadata in lock-based structures, saving one RTT. 3) The *sibling-based validation* mechanism is applicable to all B+ trees to save metadata overhead.

Write amplifications in CHIME. For update operations, write amplifications are majorly caused by the versions. A 1-byte cache line version is embedded in every 63-byte data, and a 1-byte object version is embedded in every entry. Thus, for a 256-byte KV item, which is typical in real-world workloads [62], the size of versions is $1 + \frac{KV\_size}{63} = 5.1$ bytes,

which causes only a 1.02× write amplification. For insert operations, write amplifications result from the node and hop range writes. The former occurs infrequently. For the latter, the probability of hop ranges with sizes larger than H is not higher than α^H [13, 23], where α is the load factor of the hopscotch leaf node.

Remote memory consumption in CHIME. The consumption is determined by the KV data, metadata, and hash table load factor. Consider the hopscotch leaf nodes:

- Consumption caused by KV data. This is the same as that of other indexes under the same workloads.
- Consumption caused by metadata. Some techniques in CHIME introduce additional metadata overhead: 1) The *hopscotch leaf node* requires each entry to embed a 2-byte hopscotch bitmap. 2) The *two-level cache line versioning* adds $1 + \frac{KV\_size}{63}$ bytes of version data per entry. 3) The *metadata replication* embeds a 10-byte metadata replica to every H entries. With 256-byte KV items, the metadata overhead for each item is $3 + \frac{KV\_size}{63} + \frac{10}{H} = 8.3$ bytes, consuming only 3% of the memory used by the KV data.
- Consumption caused by load factor. As shown in Figure 3d, hopscotch hashing can achieve an acceptable maximum load factor of around 90% with a low amplification factor. This results in CHIME having a 1.1x higher memory consumption than other indexes. CHIME could consider narrowing this gap by increasing the neighborhood size. With a neighborhood size of 16, CHIME can reach a maximum load factor of 99.8%, which will be shown in Section 5.4.

Applicability of CHIME to large-scale datasets. CHIME can do well across different dataset sizes. As for cache consumption, the impact of dataset size is linear. The cache consumption of CHIME is close to large-span B+ trees. As for performance, the dataset size affects the tree height, which is $\lceil log_{span\_size} \frac{dataset\_size}{load\_factor} \rceil$ for CHIME. With a span size of 64 and a maximum load factor of 99.8%, the tree height remains not higher than 5 even as the dataset size grows to 1 billion.

Compatibility of CHIME to CXL. The design of CHIME should be adjusted when adopting CXL but the change is not significant. The *vacancy bitmap piggybacking* technique will no longer be applicable, as the `mask-CAS` is only specified by RDMA. Instead, CHIME should use the regular atomic command supported by CXL 3.0 [11]. This will decrease the performance under insert workloads since dedicated memory assesses are required to read and write the vacancy bitmap. In general, other designs remain feasible under CXL, as they do not rely on specialized operations restricted to RDMA.

5 Evaluation

5.1 Experimental Setup

Testbed. We conduct all experiments on 10 physical machines (10 CNs and 1 MN) on CloudLab [18].[2] Each machine has two 36-core Intel Xeon CPUs, 256 GB of DRAM, and one 100 Gbps Mellanox ConnectX-6 NIC. All machines are connected to a 100 Gbps Ethernet switch. Each CN contains 4 GB DRAM and 64 CPU cores, where each core can serve as a client. Each MN holds 64 GB DRAM and 1 CPU core.

Workloads. Unless stated otherwise, we evaluate CHIME with YCSB workloads [12] with 8-byte keys and 8-byte values, following previous works [5, 32, 36, 56]. We use 6 core workloads: A (50% search, 50% update), B (95% search, 5% update), C (100% search), D (latest-read, 95% search, 5% insert), E (95% scan with each accessing up to 100 items, 5% insert), and an additional LOAD (100% insert) workload. All workloads are generated with the default Zipfian distribution except for YCSB LOAD and D. For each workload, we populate 60 million KV items before conducting 60 million operations, except for the LOAD test.

Comparisons. We compare CHIME to three state-of-the-art DM range indexes: SMART [36] (radix tree), Sherman [56] (B+ tree), and ROLEX [32] (learned index). We enhance Sherman with two-level cache line versions since its original *bookend versioning* is incorrect [66]. We implement ROLEX with its open-source models.[3] We also evaluate SMART-Opt, a version of SMART with sufficient caches, to represent the performance upper bound with almost no read or write amplifications. Besides, we extend each range index to support variable-length items: CHIME-Indirect, SMART-RCU [36], Marlin [5], and ROLEX-Indirect [32]. For fairness, RDWC and coroutines are applied to all competitors.

Parameters. Without explicit mention, we use all the default configurations of all baselines (*e.g.*, a span size of 64 for Sherman and Marlin, a span size of 16 and a model error of 16 for ROLEX). For CHIME, we use a span size of 64, a neighborhood size of 8, and a hotspot buffer size of 30 MB. Except for the SMART-Opt test, we limit the cache size on each CN to 100 MB, shared by all clients on the CN.

5.2 Performance Comparison

Figure 12 presents the throughput-latency curves of the four DM range indexes under YCSB workloads.

The search-only workload (YCSB C). For YCSB C, CHIME outperforms Sherman and ROLEX by 4.3× in throughput since the hopscotch leaf node mitigates read amplifications. Although ROLEX has a lower theoretical amplification factor (*i.e.*, 32) than Sherman (*i.e.*, 64), its throughput is as low as Sherman's. This is because ROLEX stores overflow items of each leaf node in an additional one and must fetch both leaf nodes in each search operation. CHIME outperforms SMART by 5.1× in throughput and achieves 2.8× lower P99 latency since SMART fails to cache all internal nodes, resulting in expensive remote tree traversals.

Insert workloads (YCSB LOAD, D). For YCSB LOAD, CHIME outperforms Sherman by 1.6× in throughput since

[2] Like previous works [36, 56], we make one machine act as both CN and MN to save machines.

[3] For simplicity, we pre-train all KV items for ROLEX to avoid the model retraining. Thus, we do not evaluate ROLEX with YCSB LOAD.

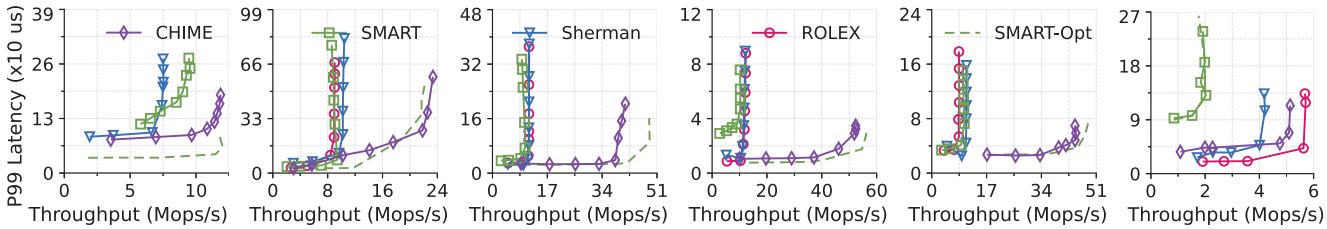


Figure 12. The performance comparison of range indexes on DM.

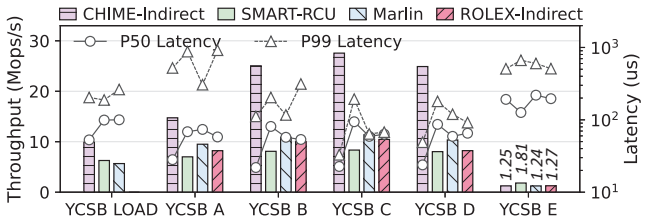


Figure 13. The performance comparison of range indexes on DM with variable-length KV items supported.

the vacancy bitmap enables reading a hop range rather than the leaf node. CHIME outperforms SMART by 1.2× in throughput and 1.5× in P99 latency since it caches KV items more efficiently. CHIME shows higher P99 latency than SMART-Opt since node splits occur recursively under the 100% insert.

For YCSB D, CHIME achieves 4.2×, 5.3×, and 4.4× higher throughput with similar P99 latency, compared with Sherman, ROLEX, and SMART. The improvement comes from the lower read amplifications and the lower cache consumption. Besides, with fewer node splits (under only 5% insert), CHIME achieves a similar P99 latency to SMART-Opt.

Update workloads (YCSB A, B). Read optimizations also benefit update workloads since each update first reads an object. Compared with Sherman, ROLEX, and SMART, CHIME reaches 2.2×, 2.6× and 2.5× higher throughput for YCSB A, and 3.6×, 3.6× and 4.1× higher throughput for YCSB B. Since the update operation of CHIME has almost no write amplifications, the throughput of CHIME is nearer to that of SMART-Opt in the update-intensive workload (*i.e.*, YCSB A) than the search-intensive one (*i.e.*, YCSB B).

The scan workload (YCSB E). For YCSB E, CHIME achieves 2.5× higher throughput and 2.5× lower P99 latency than SMART and SMART-Opt. This is because small-sized reads on KV-discrete range indexes rapidly saturate the IOPS upper-bound of RDMA NICs on MNs [36], resulting in low performance. ROLEX performs the best since it has the smallest span size and is least affected by the read amplification. CHIME outperforms Sherman by 1.2× in throughput since it excludes some unnecessary entries for each leaf fetch, thus mitigating read amplifications of scan operations, as mentioned in Section 4.4.

Variable-length KV items. Figure 13 shows the performance comparison under 320 clients with variable-length KV items supported. CHIME-Indirect still performs the best in most workloads due to its lower read amplifications than ROLEX-Indirect and Marlin, and lower cache consumption than SMART-RCU. Besides, there are two differences compared with previous results: ***1) YCSB A.*** Marlin exhibits the lowest P99 latency in YCSB A, as it reduces write conflicts by enabling concurrent KV updates within the same leaf node. This design is orthogonal to the CHIME design. ***2) YCSB E.*** SMART-RCU performs the best in YCSB E, as it does not store items out of leaf nodes, saving one RTT.

Cache consumption. Figure 14 shows the cache consumption of the four range indexes with sufficient caches. With the number of loaded items growing from 40 to 120 million, the cache consumption of KV-contiguous range indexes remains below 100 MB. With 60 million items loaded, CHIME, Sherman, ROLEX, and SMART consume 27.6 MB, 23.6 MB, 31.2 MB, and 503.2 MB, respectively, in their computing-side caches. Together with the 30 MB hotspot buffer, CHIME achieves an 8.7× lower cache consumption than SMART. As the cache consumption increases linearly with the dataset size, CHIME's cache consumption will be approximately 450 MB for 1 billion items.

5.3 Factor Analysis

Figure 15 presents the factor analysis for techniques in CHIME. We apply each proposed technique one by one to Sherman and ROLEX, respectively. The following starts with Sherman.

+ Hopscotch leaf node. The hopscotch leaf node contributes to all workloads except for YCSB LOAD. The throughput improves by 2.3×, and P50/P99 latency decreases by 2.4×/2.0× with YCSB C since clients only need to read neighborhoods rather than entire leaf nodes.

+ Vacancy bitmap piggybacking. The vacancy bitmap piggybacking contributes to YCSB LOAD. The throughput improves by 1.6×, and P50/P99 latency decreases by 1.7×/1.4× since clients only need to read hop ranges rather than entire leaf nodes, with no extra metadata access.

+ Leaf metadata replication. The leaf metadata replication contributes to all workloads. It eliminates the dedicated `READ` for the leaf metadata, saving one remote memory access for each request. For YCSB C, it brings a 1.6× improvement in throughput and a 1.6×/1.4× reduction in P50/P99 latency.

+ Sibling-based validation. Figure 16 shows how sibling-based validations (SV) optimize the metadata overhead. As

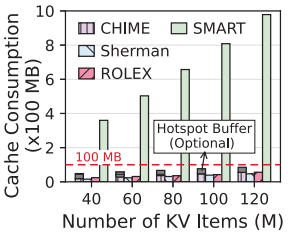
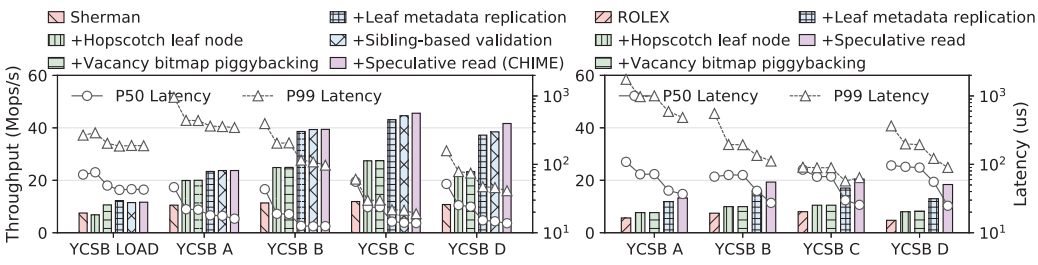


Figure 14. The comparison of cache consumption of range indexes on DM.



(a) Start with the B+ tree (Sherman). **(b)** Start with the learned index (64-span ROLEX).

Figure 15. The factor analysis for techniques in CHIME with 320 clients.

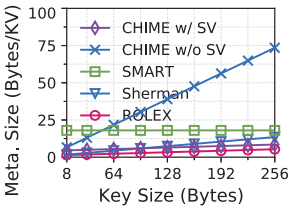


Figure 16. The contribution of sibling-based validations.

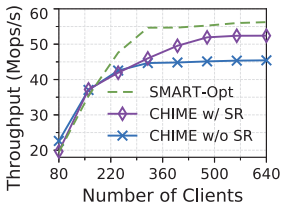


Figure 17. The contribution of speculative reads.

the key size increases from 8 to 256 bytes, the optimization increases from 1.4× to 8.6×, allowing CHIME to achieve a comparable metadata size with other indexes.

+ Speculative read. The speculative read mechanism brings only a negligible improvement in Figure 15 since the 320 clients do not saturate the network bandwidth. As shown in Figure 17, with more than 500 clients, the network bandwidth is saturated and thus the speculative read (SR) mechanism improves the peak throughput by up to 1.2× with YCSB C, allowing CHIME to achieve a closer performance to the optimal case. This is approximately consistent with the optimization space (1.3×) stated in Section 3.2.3.

CHIME vs. CHIME-Learned. We call the final learned index in Figure 15b as CHIME-Learned. CHIME outperforms CHIME-Learned since the latter may fetch multiple neighborhoods (one for each leaf node) for each search due to the unavoidable model error. Thus, the combination of the B+ tree and hopscotch hashing is preferred in our design.

5.4 Sensitivity Analysis

In this section, we investigate how the parameters of CHIME affect its performance. Without explicit mention, we use 640 clients and YCSB C.

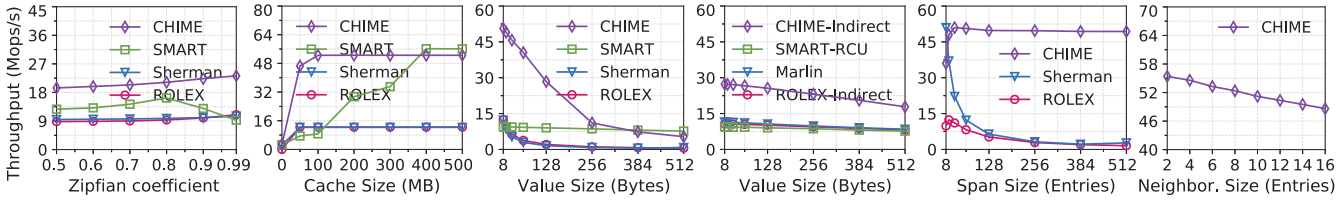
Workload skewness. Figure 18a shows the performances of range indexes on a generated Zipfian workload [33] (50% search + 50% update) with different skewness. As the skewness grows from 0.5 to 0.99, the performances of CHIME, Sherman, and ROLEX slightly increase thanks to the RDWC technique, where the more skewed the workload is, the better RDWC performs in combining requests. SMART's performance decreases in highly skewed workloads since lock-fail retries and remote traversals exacerbate its IOPS bottleneck.

Impact of cache size. As shown in Figure 18b, CHIME, Sherman, and ROLEX only require small cache sizes (< 100 MB) to reach peak throughput, while SMART requires 400 MB caches. The peak throughput of CHIME is 4.3× higher than Sherman and ROLEX and comparable with SMART due to the low read amplification.

Impact of value size. Figures 18c and 18d show how the value size affects performances with values stored inside (*inline value*) and outside leaf nodes (*indirect value*). As the inline value size grows from 8 to 512 bytes, CHIME, Sherman, and ROLEX show more severe performance declines (9.4×, 15.5× and 23.0×) than SMART (1.2×) since neighborhood and leaf node sizes grow with value size, causing the rapidly increasing bandwidth consumption. CHIME performs similarly to SMART with large inline value sizes since the 30 MB hotspot buffer cannot completely eliminate the read amplifications of hopscotch hashing. The larger the value size, the more network bandwidth is wasted by the read amplifications. This problem is addressed with indirect values since values are outside the leaf node and thus larger value sizes will not waste more bandwidth. In CHIME, the key and value are always stored contiguously and accessed together. Thus, the impact of key size is similar to that of value size.

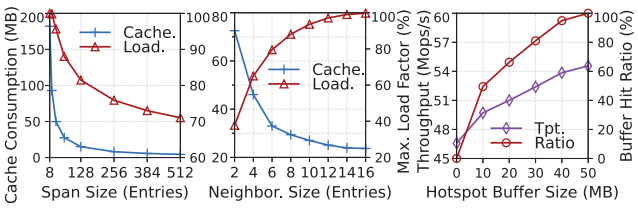
Impact of span size. As shown in Figure 18e, CHIME's performance is almost unaffected by the span size since CHIME does not read the entire leaf node. As the span size grows from 8 to 512, the performances of Sherman and ROLEX decrease by 18.6× and 6.4×, respectively. Note that the performance of CHIME declines when the span size is less than 32 since small span sizes increase the frequency of the wrap-around corner case. As shown in Figure 19a, the larger the span size of CHIME, the lower the cache consumption and the lower the maximum load factor. We set CHIME's span size to 64 to achieve low cache consumption (27.6 MB) and an acceptable maximum load factor (88.1%).

Impact of neighborhood size. As shown in Figures 18f and 19b, as CHIME's neighborhood size grows from 2 to 16, the performance decreases by 1.1×, and the maximum load factor increases from 37.7% to 99.8%. We set the neighborhood size to 8 to maximize performance while maintaining an acceptable maximum load factor. We also identify an interesting phenomenon: smaller neighborhood sizes increase cache consumption. This is because the smaller neighborhood leads to more hash collisions, resulting in more node splits and thus more internal nodes to cache.



(a) Skewness. **(b)** Cache size. **(c)** Value size (Inline). **(d)** Value size (Indirect). **(e)** Span size. **(f)** Neighborhood size.

Figure 18. The sensitivity analysis for overall performance.



(a) Span size. **(b)** Neighborhood size. **(c)** Hotspot buffer size.

Figure 19. In-depth analyses of CHIME with other metrics.

Impact of hotspot buffer size. Figure 19c shows the impact of hotspot buffer size on the performance. As the buffer size grows from 0 to 50 MB, the throughput of CHIME increases by 1.2×. Under the skewed YCSB workload, the 30 MB buffer achieves an 81.0% hit ratio, with a rate of correct speculations nearing 100% thanks to the fingerprints.

6 Related Work

6.1 Disaggregated Memory

Existing studies explore DM in many areas, *e.g.*, hardware designs [21, 28, 34, 57], operating systems [4, 20, 50, 54, 64], software runtimes [8, 37, 48, 55, 65], storage systems [2, 30, 51–53, 63], and data structures [5, 32, 36, 56, 69]. The studies most related to CHIME are DM range indexes [5, 32, 36, 56]. However, existing approaches all suffer from the trade-off between memory-side read amplifications and computing-side cache consumption. CHIME is a hybrid index that exhibits both low read amplifications and low cache consumption. PolarDB Serverless [9], a real database product for disaggregated datacenters, uses a memory-disaggregated B+ tree to support range queries. DINOMO [30], a KV store on disaggregated persistent memory, proposes ownership partitioning to reduce coordination overheads, which is transparent to indexing. It uses RECIPE [29] for its index, which converts in-memory indexes to persistent ones and is orthogonal to CHIME's design. CHIME focuses on optimizing indexing data structures rather than complete database designs or data persistence.

6.2 RDMA-Based Indexes

RDMA has attracted increasing research attention in terms of the design of data indexes for distributed systems. Many studies among them conduct operations via DM-unfriendly RPCs [16, 39, 40, 58, 59] or customized hardware [3, 7, 25, 49]. CHIME focuses on building DM-friendly indexes with commodity RDMA NICs.

As for hash indexes, RACE [69] is a one-sided RDMA-based closed-addressing hashing with lock-free remote concurrency control. FaRM [16] proposes an RDMA-based hopscotch hashing with high space efficiency. It only supports a neighborhood size of two and RPC-based index modifications, which is unsuitable for DM. CHIME includes a fully one-sided RDMA-based hopscotch hashing design.

As for range indexes, FG [67] is the first one-sided B-link tree index on DM. Sherman [56] and Marlin [5] are B+ trees on DM with several RDMA-friendly write optimizations. ROLEX [32] is the state-of-the-art learned index on DM that uses machine-learning models as computing-side caches to achieve lower cache consumption. However, they all suffer from inherent read amplifications. SMART [36] uses a radix tree as a range index on DM with almost no read amplifications. However, it exhibits high cache consumption. CHIME breaks the trade-off between read amplifications and cache consumption with a hybrid tree design.

6.3 Hybrid Indexes

Constructing a hybrid index with the B+ tree and the hash table is not a new idea [3, 10, 38, 61]. Among them, HT-tree [3] is the most related one to DM. It proposes a new tree structure where each leaf node stores base pointers of hash tables to enable caching most levels of large trees in far memory. Unlike HT-tree, CHIME is specifically designed for DM and addresses the challenges, e.g., concurrency control and metadata management, of constructing such a hybrid index on DM.

7 Conclusion

This paper identifies the trade-off between the memory-side read amplifications and the computing-side cache consumption for range indexes on DM. We propose to use a hybrid index, CHIME, that combines B+ trees with hopscotch hashing to break the trade-off. Our evaluation results verify the efficacy and efficiency of CHIME.

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their constructive comments and suggestions. This work is supported by the National Natural Science Foundation of China (Project No. 623B2026) and the Open Fund of PDL (Project No.WDZC20245250106). Yangfan Zhou is the corresponding author (zyf@fudan.edu.cn).

References

[1] Marcos K. Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory disaggregation: why now and what are the challenges. *ACM SIGOPS Oper. Syst. Rev.* 57, 1 (2023), 38–46. https://doi.org/10.1145/3606557.3606563

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 862–877. https://doi.org/10.1145/3575693.3575732

[3] 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, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 120–126. https://doi.org/10.1145/3317550.3321433

[4] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 14:1–14:16. https://doi.org/10.1145/3342195.3387522

[5] Hang An, Fang Wang, Dan Feng, Xiaomin Zou, Zefeng Liu, and Jianshun Zhang. 2023. Marlin: A Concurrent and Write-Optimized B+-tree Index on Disaggregated Memory. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*. ACM, 695–704. https://doi.org/10.1145/3605573.3605576

[6] InfiniBand Trade Association. Accessed: 2024. Enabling the Modern Data Center – RDMA for the Enterprise. https://www.infinibandta.org.

[7] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 228–242. https://doi.org/10.1145/3477132.3483587

[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 *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 79–92. https://doi.org/10.1145/3445814.3446713

[9] 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*. ACM, 2477–2489. https://doi.org/10.1145/3448016.3457560

[10] Hokeun Cha, Xiangpeng Hao, Tianzheng Wang, Huanchen Zhang, Aditya Akella, and Xiangyao Yu. 2023. Blink-hash: An Adaptive Hybrid Index for In-Memory Time-Series Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1235–1248. https://www.vldb.org/pvldb/vol16/p1235-cha.pdf

[11] CXL Consortium. Accessed: 2024. Compute Express Link. https://www.computeexpresslink.org.

[12] 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*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company.

[14] NVIDIA Corporation. Accessed: 2024. Advanced Transport. https://docs.nvidia.com/networking/display/ofedv502180/advanced+transport.

[15] NVIDIA Corporation. Accessed: 2024. RDMA Aware Networks Programming User Manual v1.7. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17.

[16] 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*. USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[17] Aleksandar Dragojevic, 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, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 54–70. https://doi.org/10.1145/2815400.2815425

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[19] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26. https://doi.org/10.1145/1806907.1806908

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: a hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 417–433. https://doi.org/10.1145/3503222.3507762

[22] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[23] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5218)*. Springer, 350–364. https://doi.org/10.1007/978-3-540-87779-0_24

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. USENIX Association, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 756–771. https://doi.org/10.1145/3477132.3483565

[26] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 424–439.

https://doi.org/10.1145/3477132.3483589

[27] Donald Ervin Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley. https://www.worldcat.org/oclc/312910844

[28] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* ACM, 488–504. https://doi.org/10.1145/3477132.3483561

[29] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 462–477. https://doi.org/10.1145/3341301.3359635

[30] 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://www.vldb.org/pvldb/vol15/p4023-lee.pdf

[31] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[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 2023, Santa Clara, CA, USA, February 21-23, 2023.* USENIX Association, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[33] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014.* USENIX Association, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[34] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA.* ACM, 267–278. https://doi.org/10.1145/1555754.1555789

[35] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. 2019. Memory Disaggregation: Research Problems and Opportunities. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019.* IEEE, 1664–1673. https://doi.org/10.1109/ICDCS.2019.00165

[36] 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.* USENIX Association, 553–571. https://www.usenix.org/conference/osdi23/presentation/luo

[37] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022.* ACM, 92–107. https://doi.org/10.1145/3519939.3523441

[38] Linsen Ma, Rui Xie, and Tong Zhang. 2023. ZipKV: In-Memory Key-Value Store with Built-In Data Compression. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, ISMM 2023, Orlando, FL, USA, 18 June 2023.* ACM, 150–162. https://doi.org/10.1145/3591195.3595273

[39] 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.* USENIX Association, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[40] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* USENIX Association, 451–464. https://www.usenix.org/conference/atc16/technical-sessions/presentation/mitchell

[41] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distributed Syst.* 12, 10 (2001), 1094–1104. https://doi.org/10.1109/71.963420

[42] MySQL. Accessed: 2024. https://www.mysql.com.

[43] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018.* ACM, 16:1–16:12. https://doi.org/10.1145/3190508.3190537

[44] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[45] PostgreSQL. Accessed: 2024. https://www.postgresql.org.

[46] Redis. Accessed: 2024. http://redis.io.

[47] RocksDB. Accessed: 2024. https://rocksdb.org.

[48] 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 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[49] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021.* ACM, 740–755. https://doi.org/10.1145/3477132.3483555

[50] 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.* USENIX Association, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[51] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023.* ACM, 675–691. https://doi.org/10.1145/3600006.3613144

[52] 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.* USENIX Association, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[53] 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 2020, July 15-17, 2020.* USENIX Association, 33–48. https://www.usenix.org/conference/atc20/presentation/tsai

[54] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark

Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 352–367. https://doi.org/10.1145/3492321.3519569

[55] 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 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

[56] 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*. ACM, 1033–1048. https://doi.org/10.1145/3514221.3517824

[57] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. USENIX Association, 277–292. https://www.usenix.org/conference/fast21/presentation/wang

[58] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 117–135. https://www.usenix.org/conference/osdi20/presentation/wei

[59] 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*. ACM, 87–104. https://doi.org/10.1145/2815400.2815419

[60] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann.

[61] Haichang Yang, Zhaoshi Li, Jiawei Wang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. HeteroKV: A Scalable Line-rate Key-Value Store on Heterogeneous CPU-FPGA Platforms. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 834–837. https://doi.org/10.23919/DATE51398.2021.9474088

[62] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[63] 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*. USENIX Association, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[64] 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*. ACM, 1345–1359. https://doi.org/10.1145/3514221.3517856

[65] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhouyang

[66] 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

[67] 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*. ACM, 741–758. https://doi.org/10.1145/3299869.3300081

[68] 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*. USENIX Association, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

[69] 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 2021, July 14-16, 2021*. USENIX Association, 15–29. https://www.usenix.org/conference/atc21/presentation/zuo