DEX: Scalable Range Indexing on Disaggregated Memory

Baotong Lu[*]
Microsoft Research
baotonglu@microsoft.com

Kaisong Huang
Simon Fraser University
kha85@sfu.ca

Chieh-Jan Mike Liang
Microsoft Research
liang.mike@microsoft.com

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Eric Lo
The Chinese University of Hong Kong
ericlo@cse.cuhk.edu.hk

ABSTRACT

Memory disaggregation can potentially allow memory-optimized range indexes such as B+-trees to scale beyond one machine while attaining high hardware utilization and low cost. Designing scalable indexes on disaggregated memory, however, is challenging due to rudimentary caching, unprincipled offloading and excessive inconsistency among servers.

This paper proposes DEX, a new scalable B+-tree for memory disaggregation. DEX includes a set of techniques to reduce remote accesses, including logical partitioning, lightweight caching and cost-aware offloading. Our evaluation shows that DEX can outperform the state-of-the-art by 1.7–56.3×, and the advantage remains under various setups, such as cache size and skewness.

PVLDB Reference Format:
Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, Eric Lo. DEX: Scalable Range Indexing on Disaggregated Memory. PVLDB, 17(10): 2603 - 2616, 2024.
doi:10.14778/3675034.3675050

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

1 INTRODUCTION

Memory-optimized indexes [2, 3, 17, 27, 39] are crucial for accelerating OLTP. Their scalability and economy, however, are being limited by the traditional monolithic server architecture where CPU and memory (DRAM) are "bundled" together. Blindly scaling up can lead to high cost that often only pays off under the full load. Worse, as data size—and consequently index size—grow, the demand for memory capacity can go beyond what a single server could offer. Memory disaggregation [14, 22, 30, 43] has emerged to ease this problem by separating memory and compute into their own server pools and interconnecting the two resource pools via fast networks (e.g., InfiniBand (IB) [10] and CXL [23]). As the workload changes, we have the flexibility to independently scale compute threads and memory size, achieving high utilization and low cost.

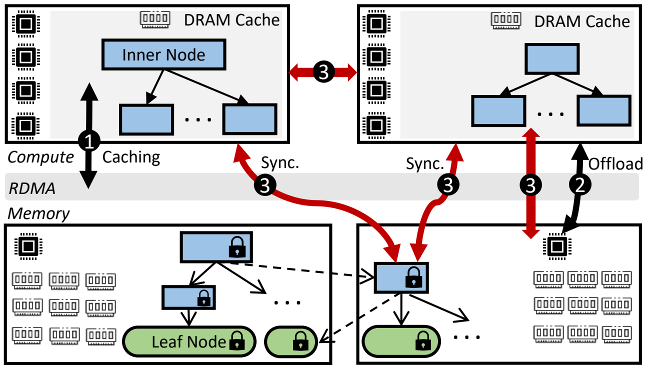


Figure 1: Desiderata of indexes on disaggregated memory. ❶ Caching should work with the smaller speed gap between local and remote memory, and limited local memory. ❷ Offloading should be aware of the scarcity of memory-side compute. ❸ Design should recognize potential data inconsistencies among servers (red arrows).

1.1 Range Indexes: Disaggregated ≠ Scalable

Unfortunately, naïvely deploying a tree index on disaggregated memory does not automatically achieve the aforementioned goals. With compute and memory decoupled, accessing the index inherently incurs remote memory accesses (e.g., over RDMA), which can add non-trivial latency (as compared to local DRAM accesses). This problem exacerbates, as we consider that an index operation (e.g., lookup) typically requires traversal from the root to the leaf node, necessitating at least one RDMA operation per tree level.

Memory disaggregation brings unique solution design space to the problem above. Although compute and memory are decoupled, there is actually a hidden resource hierarchy — compute servers can have some local memory (in addition to the larger remote memory), and memory servers can have some local compute (in addition to the more powerful compute servers).

At first glance, leveraging local resources can reduce remote memory accesses, through compute-side software-managed caching or offloading (aka computation pushdown) to memory servers.[1] However, we argue that disaggregation mandates a significant departure from existing caching and offloading approaches [25, 28, 37, 47], to address the unique challenges in Figure 1: ❶ rudimentary caching, ❷ unprincipled offloading, and ❸ excessive inconsistency.

Rudimentary Caching. Given the availability of compute-side memory, it is natural to cache frequently accessed index nodes on

[*] Work partially performed while at The Chinese University of Hong Kong and Simon Fraser University.

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675050

[1] We use offloading and computation pushdown interchangeably in this paper.

compute servers.[2] Many efforts have focused on improving cache-hit ratio [12, 28]. These past approaches, however, were mostly targeting conventional monolithic DBMSs. Disaggregation invalidates certain assumptions of traditional caching mechanisms.

First, the speed gap between local and remote memory (~10× through RDMA) is much narrower than that between memory and storage (e.g., ~1000× with SSDs). Therefore, software overheads associated with cache maintenance and synchronization become more prominent in the disaggregation setting.

Second, compute servers should not assume a certain amount of local memory, especially the industry has not converged to one particular disaggregation practice [6]. As such, if local memory is severely constrained, general-purpose caching mechanisms that do not exploit the properties of tree indexes might not perform well.

Unprincipled Offloading. Offloading takes advantage of the limited compute on memory servers for less RDMA communication. Conceptually, a compute thread simply sends *one* index operation to a memory server, instead of individual RDMA operations that incur multiple round trips. However, the disaggregation setting now requires us be resource-aware to avoid overwhelming the limited compute on memory servers. In other words, informed decisions (i.e., what, when, and how much to offload) become crucial, to avoid overloading and guarantee scalable performance.

Excessive Inconsistency. Memory disaggregation brings the challenge of handling different sources of data inconsistencies. Even without considering caching and offloading, we need synchronization (e.g., locks) to guard the index from being concurrently modified by compute threads. Unfortunately, existing RDMA-based locking for distributed synchronization is costly [46]. If we implement caching on compute servers, it becomes necessary to ensure the coherence among all compute-side caches. This complexity grows with the number of compute servers. The problem exacerbates if we consider both compute-side caching and memory-side offloading. Memory servers now become another potential source of data changes. Prior to serving an offloaded operation, the memory server needs to ensure its view of the tree index is consistent with all compute-side caches. Such global consistency needs to be guaranteed throughout offloading, with the use of locks or coherence messages. Finally, all compute-side caches that contain stale pages should be synchronized with memory servers.

1.2 DEX

This paper presents DEX, a new B+-tree designed to scale on disaggregated memory. DEX uniquely combines a set of new and existing techniques to effectively mitigate the aforementioned issues.

Compute-Side Logical Partitioning. DEX mitigates cross-compute consistency overhead using logical partitioning [15, 29] where each compute server logically "owns" a set of key ranges while the memory servers still present a globally addressable shared space. This way, different compute servers operate mostly on disjoint portions of the index, reducing the cost of cache coherence across compute servers and RDMA-based synchronization for remote memory accesses. Load balancing and adding/removing a compute server are simple and lightweight since logical partitioning only necessitates adjusting routing without physically re-partitioning data.

Optimized Caching. We propose a lightweight cache replacement strategy based on random sampling. By avoiding centralized data structures like FIFO queues, DEX's cache reduces the contention and achieves high scalability. To reduce cache misses, we leverage application-level information to do *path-aware* caching that tends to keep in the cache frequently accessed *index paths* from the root to lower level nodes. A child B+-tree node cannot be admitted to the compute-side cache unless its parent node has been cached. Similarly, in most cases, a parent B+-tree node is not evicted until all of its child nodes are evicted. This not only improves cache efficiency (as nodes closer to the root are hotter) but also enables more effective offloading with low consistency overhead between compute and memory servers (described below).

Opportunistic Offloading. DEX tracks resource availability on memory servers at runtime, and it offloads an index operation only if the completion time could be minimized. However, a challenge arises from the simultaneous use of compute-side caching above. Considering an index operation involving the tree traversal of $N_A \rightarrow N_B \rightarrow N_C$, a naïve caching policy (e.g., the widely used random eviction [35, 45]) may evict N_A and admit N_B and N_C into the compute-side cache. However, at this point, if there is another thread trying to offload the traversal of $N_A \rightarrow N_B \rightarrow N_C$ to the memory server after observing a cache miss on N_A, concurrent changes made on the compute-side cache and the memory-side data could result in data inconsistencies. Our design takes advantage of a property of path-aware caching, where a consecutive path from the root to lower level nodes are cached. This effectively prevents a tree traversal from being interleaved with caching and offloading, hence eliminating another source of potential inconsistencies.

The contributions of DEX lie in systematically realizing an unique combination of compute-side caching and memory-side offloading, in order to best minimize the scalability bottleneck on disaggregated memory (i.e., remote memory accesses). Evaluations on a four-server RDMA cluster show that DEX outperforms state-of-the-art, with 1.7–56.3× higher throughputs, under various workloads. DEX is open-sourced at https://github.com/baotonglu/dex.

2 BACKGROUND AND MOTIVATION

We give the necessary background on disaggregated memory and its impact to tree indexes that motivated our work.

2.1 Disaggregated Memory (DM)

Compute (CPU cores) and memory (DRAM) have been traditionally coupled to scale together in data centers. However, as shown by recent work [9, 33], this can lower memory utilization and waste compute resources. As networking technologies advance, it now becomes viable to decouple compute and memory respectively into compute and memory servers that can independently scale, similar to how storage is disaggregated in the cloud.

A typical disaggregated memory architecture consists of a set of compute servers and a set of memory servers, as Figure 1 shows (ignore the tree nodes for now). Compute servers focus on providing ample compute capabilities with high core count and high

[2] Unless otherwise noted, throughout this paper "cache" refers to the software-controlled DRAM cache on compute servers, instead of CPU caches.

CPU frequency; their memory capacity is usually limited. Memory servers focus on providing ample memory capacity, but their compute capabilities are limited. A high-speed interconnect such as InfiniBand and CXL allows compute servers to access data on memory servers using memory semantics. Currently, this is widely done using RDMA over InfiniBand, although other solutions (e.g., CXL.mem [23]) are being devised. We follow recent work [15, 21, 25, 37, 42, 43, 47] to focus on RDMA-enabled disaggregated memory.

RDMA allows participating servers to access each other's memory directly without involving the remote CPU and/or OS kernel, providing much lower latency than TCP/IP networks. RDMA is performed by "verbs" which can be one-sided (READ, WRITE and atomics such as compare-and-swap or CAS) or two-sided (SEND/RECV). One-sided verbs do not involve remote CPU, whereas two-sided verbs operate similarly to TCP/IP operations by requiring the remote CPU to participate. Due to limited memory-side compute power, one-sided verbs are preferable for many disaggregated memory settings. Nonetheless, offloading can be achieved by performing remote procedure calls (RPCs) using either one-sided [5] or two-sided verbs [13, 47]. Either approach requires memory-side threads to receive and process RPC requests. We use two-sided verbs for offloading, as it achieves better performance [13, 47].

2.2 Disaggregating Memory-Optimized Indexes

With the aforementioned architecture, now we discuss how software, in particular tree-based range indexes, can be adapted. Without losing generality, we focus on memory-optimized B+-trees that are designed for multicores assuming the tree fits in memory, and use memory-optimized layout and optimistic locking [18, 31] or lock-free concurrency [19]. Importantly, most of them are shared-everything where any thread can access any part of the tree, which recent DM-based B+-trees have inherited. As shown in Figure 1, these properties allow (1) using remote memory as "the main memory" to store tree nodes and (2) using the CPU cores in compute servers to perform tree operations.

To probe for a key, the compute server issues a one-sided RDMA READ to fetch the root node to its local DRAM. It then searches the node to find the next child node, which again is fetched to the compute server's local DRAM using RDMA READ. Depending on how the tree nodes are distributed across memory servers, a traversal may involve multiple memory servers. To coordinate accesses to shared data on a memory server, optimistic locks are replaced with RDMA-based locks built using atomics such as RDMA CAS. Also, since there is a non-trivial speed gap between accessing local DRAM and remote memory, it becomes important to cache frequently/recently accessed nodes in the compute server.

RDMA brings two major challenges. (1) RDMA does not provide off-the-shelf coherence among servers, leaving the responsibility of ensuring data consistency including data synchronization and cache coherence across servers to the implementation. As a result, after a compute server updates a node using RDMA WRITE, the cached node in other compute servers become stale and should be invalidated, which is typically done by explicitly sending coherence messages across compute servers. (2) RDMA exhibits higher latency (~2000ns) than local DRAM (~100ns). Both challenges require disaggregated indexes to avoid unnecessary remote accesses, discussed next.

2.3 State-of-the-Art and Motivation

Recent work goes beyond the naive adaptation to reduce RDMA operations. Section 1.1 has covered some of the issues, here we analyze in detail the design of two representative designs—Sherman [37] (a DM-optimized B+-tree) and SMART [25] (a DM-based trie [17])—and how they still do not scale well, which motivated our work.

Compute-Side Caching. A shared-everything disaggregated index normally needs to implement cache coherence by exchanging coherence messages across compute servers (e.g., to invalidate stale nodes). Both Sherman and SMART observed that exchanging coherence messages is costly as it can be as expensive as cache misses. To avoid such cost, they do not cache leaf nodes and only cache inner nodes, for which coherence is not strictly required: using stale cached inner nodes during a traversal will not read inconsistent data but lead to incorrect leaf nodes, which can be easily resolved by retrying the operation and fetching the up-to-date index nodes from the memory pool. Sherman only caches the lowest levels of inner nodes and builds an extra index for cached nodes in compute servers. As a result, it always incur one RDMA operation to access a leaf node, necessitating RDMA even when the cache has enough capacity to hold all the needed nodes. Maintaining the extra index also requires extra bookkeeping. Similar observations apply to SMART. In other words, existing work trades the benefits of caching leaf nodes for reducing coherence overheads.

In terms of caching, prior work [12, 16, 28] primarily focused on buffer pool solutions targeting the DRAM-SSD/HDD hierarchy. The large gap between DRAM and HDDs/SSDs means that data movement cost (storage I/O) is the major bottleneck, justifying the use of simple centralized data structures to maintain cache metadata (e.g., centralized LRU lists). However, in disaggregated memory, the latency gap between compute-side DRAM and remote memory is relatively small. Moreover, compute servers in the DM-setting should have even higher core counts, putting significant pressure on the caching data structure and replacement mechanisms.

We observe that, with lower latency gap between local and remote memory, cache replacement frequency becomes a critical factor of the synchronization cost. In our analysis, we denote the latency to access a cached and uncached page as T_c and T_d, respectively. For simplicity, we do not consider bandwidth limits. Suppose the cache miss ratio is R, the frequency can be computed as: $Replacements/second = \frac{1\ second}{T_d+(\frac{1-R}{R})\times T_c} \times N_t$, where N_t is the number of threads. We empirically set T_c as 400 ns for the access to a 1KB DRAM-resident cached page. T_d for SSDs is typically 100μs [32] while it takes 2μs for one RDMA READ. At the same cache miss ratio (e.g., 10%) and number of threads (e.g., 36), the replacement frequency for DRAM-SSD is 0.35×10^6 while it is 6.43×10^6 for disaggregated memory, such that there is over 18× higher. Even worse, given the limited memory capability in compute servers, we can easily get higher R and thus higher replacement frequency.

Existing DM-based indexes [25, 37] did not take these into consideration. For example, upon cache admission, SMART needs to update a centralized local counter to track cache usage and uses a centralized FIFO queue to organize cache entries. With high replacement frequency by concurrent threads, our evaluation in Section 8 shows that it exhibits severe contention and cannot scale.

Offloading. Although attractive, neither Sherman nor SMART considered offloading. Other existing approaches [47] can offload full or partial index operations, but do so by hard-coding policies. Without considering the actual capabilities and load of memory servers, one may easily overload the memory servers, defeating the purpose of offloading. It is crucial to selectively offload index operations and strike a balance between remote accesses and offloading.

Consistency. Besides cache coherence, to ensure correct concurrent accesses to the memory pool, Sherman and SMART use distributed optimistic locks [46] for synchronization. These locks are the same as their monolithic counterparts, except the low-level implementations are based on RDMA primitives. A write operation must obtain the lock in exclusive mode by atomically changing the lock word to the "locked" state using RDMA CAS, while a read operation only needs to verify that the protected node did not change after the read operation. While the adaptation is simple, it turns out that the RDMA-based verification process is very heavyweight by incurring two RDMA READ operations, significantly lowering performance. Existing work (including Sherman) has overlooked this issue, leading to unsafe implementations [46]. Moreover, DM-based indexes are more prone to performance collapse under skewed workloads [37] because RDMA-based locks incur high overheads due to multiple network roundtrips caused by retries. Thus, designing scalable synchronization remains an open challenge.

Although no existing DM-based indexes support both caching and offloading, some general frameworks [44] enable offloading through system calls. However, targeting fast memory-resident indexes, we prefer user-space solutions with low overhead.

3 DEX OVERVIEW

DEX is a B+-tree optimized for disaggregated memory that combines a set of new and existing techniques in logical partitioning, compute-side caching and opportunistic offloading to mitigate the issues identified in previous sections.

Index Placement. As a B+-tree variant, DEX uses normal B+-tree nodes which are distributed onto different memory servers, as shown by Figure 2 (bottom). In particular, we group index nodes into sub-trees and ensure a subtree rooted at level M ($M = 0$ for the leaf level) is entirely stored on the same memory server. For example, as shown in Figure 2, all nodes in the subtrees rooted at level $M = 1$ are all stored in the same memory servers. As we describe later, this facilitates better offloading.

Node Layout and Addressing. Each tree node begins with a header area (including metadata such as the lock), followed by a key array and a pointer array (for inner nodes) or a value array (for leaf nodes). Different from monolithic B+-trees, the index nodes in DEX need to form a unified, global memory address space among multiple memory servers, such that compute servers can locate index nodes in memory servers. Similar to previous work [25, 37, 47], we address it using pointer tagging. A pointer is still 64-bit but is tagged with extra information in the format of [swizzled, memory-server-id, address], leveraging the fact that modern 64-bit x86 microarchitectures do not implement all the 64 bits [11]. The trailing 48-bit `address` carries a local memory address of a particular server identified by the 15-bit `memory-server-id`. The combination of `memory-server-id` and `address` form a unique global address in the memory pool. They are also stored in the header of each tree node to identify the node. The most significant `swizzled` bit indicates whether `address` is local to the current compute server, which is only used by the compute-side cache.

Index Node Accesses. With B+-tree nodes distributed across memory servers, each compute server may access tree nodes via one-sided RDMA and possibly cache them on the compute-side. Different from prior solutions, DEX is able to cache both inner and leaf nodes to better use the cache space. DEX departs from shared-everything architectures and adopts logical partitioning on the compute side to sidestep the vast majority of cross-server coherence issues. Synchronization is also mostly only needed within each compute server without involving distributed locks, except for certain index nodes crossing logical partition boundaries (e.g., the root node in Figure 2) that can be accessed by more than one compute server. As shown in Figure 2(top), regardless of how the index nodes are distributed across the memory servers, each compute server logically "owns" a disjoint range of keys and is responsible for handling all the requests in that range.

Upon a cache miss, the compute server will either fetch the missing node from or offload the remaining index operation to memory servers, depending on whether offloading is profitable. In the former case, DEX combines a scalable cache replacement mechanism, path-aware caching and selective cache admission to efficiently cache tree nodes. As mentioned earlier, index nodes are grouped into sub-trees. Therefore, offloading the operation of traversing a subtree rooted at level M will not cause remote pointer chasing across memory servers, improving offloading performance and reducing implementation complexity in memory servers.

Next, we elaborate how DEX realizes the above functionality by logical partitioning (Section 4), compute-side caching (Section 5), and opportunistic offloading (Section 6).

4 COMPUTE-SIDE LOGICAL PARTITIONING

As discussed in Section 2.3, prior works [25, 37] made various trade-offs to mitigate cache coherence overhead, yet they still exhibit excessive remote accesses to leaf nodes and require costly RDMA-based synchronization. Therefore, DEX uses logical partitioning to greatly reduce cross-compute consistency overhead, and to improve cache locality. As Figure 2 shows, each compute server owns a logical partition of keys. DEX can work with various range partitioning schemes (e.g., equal-width or workload-aware), as long as each leaf node is exclusively owned by exactly one partition (i.e., one compute server). This effectively limits the need for cross-server synchronization and cache coherence to only inner nodes, which are less frequently updated than leaf nodes. It is also easily achievable by picking partition boundaries using keys from the lowest inner node level because keys in these nodes indicate the possible key range of a leaf node (fence keys [7]).

Since certain nodes like the root are shared (i.e., crossing partition boundaries) by multiple compute servers while the others are not, we handle the concurrency control of them in different ways. Within a compute server, accesses to cached tree nodes are synchronized using in-memory optimistic lock coupling [18], as shown in Algorithm 1 (lines 19–20, 23–24, 37–38, 43–45). Upon a cache miss, DEX either offloads the remaining index operation to the memory

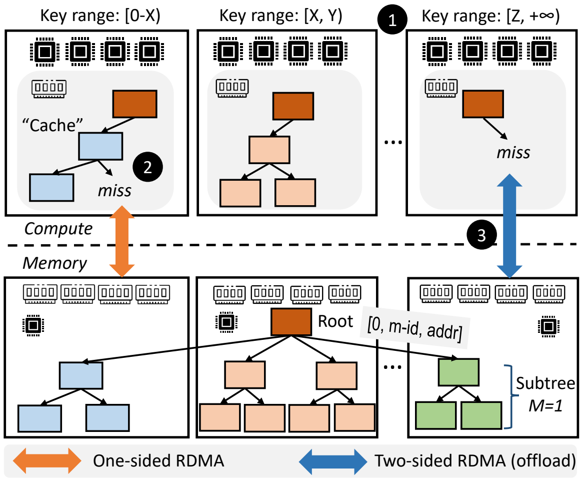


Figure 2: Overview of DEX. ❶ Each compute server “owns” a disjoint range of the key space and ❷ caches tree traversal paths in local DRAM. ❸ Upon cache misses, the compute server selectively offloads index operations when profitable. B+-tree nodes are distributed onto memory servers. However, subtrees under level *M* are all located in the same memory servers to avoid expensive pointer chasing across memory servers during offloading.

server (line 32) or retrieves the missing node from the memory pool to the compute side (line 35) through the `cache::remote_read` function. The `cache::remote_read` function determines whether to perform cross-server synchronization based on whether the target node crosses partition boundaries. Specifically, if the target node is shared by multiple compute servers, DEX utilizes RDMA-based optimistic locking [46] to synchronize concurrent accesses (lines 3–6). Otherwise, it simply reads the target node through a single RDMA READ operation (line 8).

Nonetheless, such cross-server synchronization is infrequent because only few inner nodes would cross the partition boundaries. Moreover, inner nodes that span across partition boundaries are typically closer to the root and are therefore more likely to be cached on the compute-side, largely eliminating the need for reading them with RDMA-based synchronization. However, for the updates to shared inner nodes, RDMA-based locking is still required and DEX writes back their updates to the memory pool to ensure the memory pool always has the most up-to-date version.

Cross-compute cache coherence is only required for inner nodes that cross a logical partition boundary *and* are cached. Since inner nodes store no data but guiding information, DEX follows prior work [37] to only bring in the fresh copy of an inner node from remote memory when its staleness lands its search to a wrong child node. To detect such cases, DEX records fence keys of the index nodes into their headers. Upon accessing a child node, DEX checks if the search key is within the fence keys in the header. If not, as shown in lines 39–40 of Algorithm 1, DEX will restart the search by bringing in fresh nodes on the search path from the remote root and invalidating stale cached nodes. Note that logical partitioning does not change the worst-case complexity of remote memory accesses. For instance, in an extreme scenarios where local resources (e.g., cache) are minimal or when refreshing cache from the remote root, DEX still necessitates $O(h)$ remote accesses, where h represents the height of the tree.

Although logical partitioning helps reduce cross-compute synchronization and cache coherence overhead, it can incur load imbalance in which certain compute servers are overloaded. However,

Algorithm 1 DEX lookup algorithm.

```
1  def cache::remote_read(node_addr, shared):
2    if shared is true:
3      version = RDMA_read(node_addr, 8B)
4      if is_lock_set(version): return NULL
5      node = RDMA_read(node_addr, node_size)
6      if version != RDMA_read(node_addr, 8B): return NULL
7    else
8      node = RDMA_read(node_addr, node_size)
9    cached_node = insert_to_cache(node)
10   return cached_node
11
12 def lookup(key):
13 retry:
14   parent = NULL, vp = 0
15   cur_node = cache.lookup(root)
16   if cur_node is NULL:
17     cur_node = cache.remote_read(root)
18     if cur_node is NULL: goto retry
19   vc = cur_node.version_lock
20   if is_lock_set(vc): goto retry
21
22   while(cur_node is inner):
23     if parent != NULL and vp != parent.version_lock:
24       goto retry
25     parent = cur_node, vp = vc
26     child_addr = parent.search(key)
27     cur_node = cache.lookup(child_addr)
28     if cur_node is NULL:
29       shared = is_child_shared(child_addr, parent)
30       # Consider operation offloading
31       if shared is false and deserve_offload() is true:
32           result = offload(child_addr, key)
33           return result
34       # Conduct caching
35       cur_node = cache.remote_read(child_addr, shared)
36       if cur_node is NULL: goto retry
37     vc = cur_node.version_lock
38     if is_lock_set(vc): goto retry
39     if cur_node.fence_keys is not valid:
40       refresh_from_root() and goto retry
41
42   result = cur_node.lookup(key)
43   if parent != NULL and vp != parent.version_lock:
44     goto retry
45   if vc != cur_node.version_lock: goto retry
46   return result
```

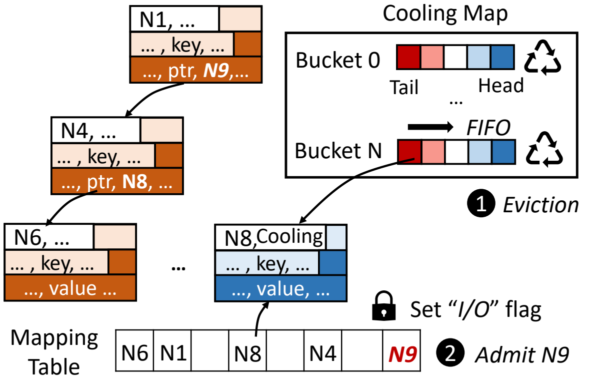


Figure 3: DEX caching in a compute server. ❶ Potential eviction candidates are first admitted to the cooling map which is a hash table of FIFO arrays to alleviate contention. ❷ To admit a new node (N9), the first thread that accesses it signals in-progress RDMA by atomically setting an `I/O` flag in the mapping table. Subsequent concurrent threads will then re-traverse the path from root to avoid repeatedly issuing RDMA by multiple threads for the same node.

this can be greatly mitigated through logical re-partitioning. Different from physical re-partitioning which requires expensive data movement and index rebuild, logical partitioning enables DEX to re-partition the overloaded compute server by simply re-adjusting the boundaries of logical partitions, without any physical data movement. Upon re-partitioning, the involved compute servers simply flush their dirty cache pages to the memory pool and adjust the partitioning boundaries. Experiments in Section 8 show that re-partitioning in DEX is lightweight and can be finished within seconds. We leave load balancing policy questions [40], such as when compute servers should be scaled, as interesting future work. Another advantage is that elasticity (scale-in and scale-out) can be supported very easily through lightweight logical re-partitioning.

5 COMPUTE-SIDE CACHING

DEX deploys a cache for B+-tree nodes (both inner and leaf) on each compute server using its local DRAM. Thanks to logical partitioning, communication between compute-side caches is rare, allowing us to localize the problem of optimizing caching on a single compute server. Since recent DBMS buffer pools optimized for fast SSDs share similar scalability concerns [1, 16], we design DEX cache based on these techniques to attain a reasonable baseline, on top of which we propose optimizations specific for disaggregated B+-trees.

5.1 Overall Structure

Like its monolithic counterparts, as Figure 3 shows, DEX tracks cached B+-tree nodes using a mapping table implemented using a concurrent hash table that maps node IDs (i.e., the global address of this node) to local node addresses in the cache. Each page frame in the cache is set to exactly the size of a B+-tree node, plus a header that embeds an optimistic lock to coordinate concurrent accesses from the same compute server. To access a node, a worker thread checks if it has been cached by probing the mapping table. Further, we follow representative buffer pool designs [16] to introduce a cooling status for nodes that are identified as potential eviction candidates. These nodes are indexed by a cooling map that is looked up when eviction is necessary.

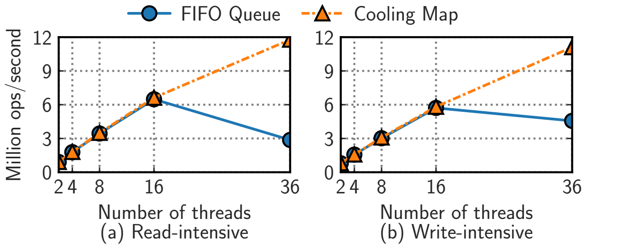


Figure 4: DEX's scalability with different cooling structures.

To reduce the overhead caused by probing and modifying the mapping table, DEX uses pointer swizzling [8, 16]. Upon the cache admission of a child node C, we record in its parent node P (which is already in the cache) C's local address with `swizzled` bit set. This way, subsequent traversals of the same (sub)path will primarily involve only local pointer chasing.

To reduce the overhead of selecting eviction candidates, DEX uses a randomized cooling policy [16] that differentiates hot and cold nodes in a coarse-grained manner. When a compute thread finds its thread-local free-page set is empty, it randomly samples a set of nodes in the cache (two in our implementation), unswizzles them from their parent nodes, writes back the dirty pages to the memory pool, and sets their state to `cooling`, indicating that the page is a candidate for future eviction.

If the target node is `cooling` (i.e., was chosen as a candidate for future eviction), we restore its status to `cached` and re-swizzle its pointer in the parent node. This gives a second chance for frequently accessed pages that are accidentally sampled for cooling to stay in the cache. In contrast, "truly" cold nodes will likely remain in the cooling state and are eventually evicted from the cache, improving cache hit ratio and reducing remote memory accesses.

With the overall structure laid out, in the rest of this section, we propose further optimizations specific to disaggregated B+-trees.

5.2 Scalable Cache Replacement

The analysis in Section 2.3 shows that the disaggregated memory setting can lead to much higher cache replacement frequency than DRAM-SSD settings. This in turn puts much more pressure on the data structure that tracks eviction candidates in the cache. Specifically, a fair eviction requires ranking hotness among cooling nodes. Yet prior approaches usually use a centralized shared FIFO list [16] protected by a single lock to achieve this goal. With high replacement frequency, threads need to frequently sample cooling pages to the shared FIFO queue or evict pages from it. This leads to intensive updates of the head/tail pointers of the queue, incurring severe cache-line pingpong between CPU cores. As Figure 4 (details in Section 8.1) shows, DEX with FIFO queue cannot scale due to high synchronization cost in the cooling process.

DEX solves this problem with an extremely simple but effective tweak that replaces the FIFO list with a concurrent hash table where each bucket includes a FIFO array (thus called "cooling map").

Because of the randomized nature of hash tables, this allows amortizing accesses over multiple memory locations, greatly alleviating contention. As shown in Figure 3, each bucket includes a CPU cacheline-sized FIFO array, where each slot stores the local pointer to a cooling node. Each bucket is protected by a lock for correct multi-threaded accesses. When a cached node is selected for cooling, it is hashed into one of the buckets using its node ID and inserted into the tail slot of the FIFO array by shifting existing entries. If the array is already full, shifting would cause the head page to be evicted from the cache. The evicted page is then inserted into a thread-local free page set. Therefore, with the cooling map, cache evictions in different buckets execute independently, significantly improving scalability. The cooling map design strikes a balance between maintaining FIFO ordering (within each bucket) and scalability. As shown in Figure 4, DEX with the cooling map scales well and outperforms the queue-based design. Finally, as stated in Section 5.1, DEX proactively unswizzles the pointer to the node once it is selected for cooling (e.g., N8 in Figure 3). This allows worker threads to avoid the unswizzling process for the eviction page, simplifying the eviction process.

5.3 Path-Aware Caching

DEX employs a path-aware strategy for cooling inner nodes where a tree path in a non-cooling state is always cached consecutively from the root node to low-level tree nodes. Specifically, when selected for cooling (e.g., N4 in Figure 3), an inner node will attempt to delegate the cooling command to one of its swizzled children (e.g., N6) instead. The cooling command will be recursively delegated until reaching a node with no swizzled child pointer (hence the leaf nodes are the base case). This way, we can avoid writing back nodes carrying swizzled pointers that are invalid in remote memory servers. More importantly, such path-aware caching ensures that only the node at the end of the cache path will be transferred to a cooling state (i.e., becoming an eviction candidate), even if a node in the middle of the cache path was initially sampled. While delegation is not new [16, 34], as we discuss later in Section 6, delegation in DEX also enables more efficient pushing down of the remaining operation to the memory pool.

5.4 (Selective) Cache Admission

When a cache miss occurs, the compute thread needs to fetch the missing node from the memory pool and admit it into the cache. The first thread that accesses the missing node will insert the node ID into the mapping table with an `I/O` flag as the value. Subsequent threads that see `I/O` in the mapping table entry will re-traverse from the locally cached root to avoid repeatedly trying to admit the same node from remote memory. For example, as shown in Figure 3, admitting the child node $N9$ of the root node $N1$ requires setting its `I/O` flag in the mapping table before cache admission. This ensures that other concurrent threads that are also trying to load $N9$ will not attempt to issue more RDMA operations, saving network bandwidth. To admit a new node, the compute thread also needs to obtain a free page in its local free page set. If there are no more available local free pages, the thread will randomly select a bucket in the cooling map, evict the oldest page in the array, and use it as the free page for accommodating the incoming node. Thanks to the aforementioned fine-grained locking in the cooling map, multiple threads can obtain free pages from the cooling map without scalability bottlenecks.

Given that remote accesses are expensive and cache sizes are limited, it is important to keep hot nodes in the cache as long as possible. Yet previous disaggregated indexes [25, 37] employ an eager admission policy that unconditionally admits all the accessed nodes to the cache. This can result in the eviction of hot nodes in favor of newly retrieved nodes whose hotness has not yet been confirmed. Moreover, admitting newly retrieved nodes eagerly may introduce unnecessary overhead since provisioning free cache pages may trigger RDMA writeback for page eviction.

DEX proposes a lazy admission policy to mitigate this issue. We assign each newly retrieved node a probability (P_A) of being admitted into the cache. Pages that cannot be admitted into the cache are discarded (or written back if dirty) immediately after use. Based on experimental results, we empirically set P_A to 0.1 for leaf nodes but set P_A to 1 for inner nodes. In other words, inner nodes are still always admitted into the cache. We made this design choice because inner nodes are generally hotter than leaf nodes. Besides, since DEX adopts the lightweight optimistic locking for tree traversal, missing some nodes on a cached path would cause unnecessary complexity and overhead. Exploring auto-tuning/learned techniques [20, 41] to determine P_A is future work.

6 OPPORTUNISTIC OFFLOADING

DEX improves resource utilization and reduces unnecessary remote accesses by offloading selected index operations to memory-side CPUs. This is enabled by DEX's path-aware caching which exhibits an important property: upon a cache miss during a traversal at level L of the tree, it is likely that subsequent node traversals from level $L-1$ down to the leaf (level 0) will also encounter cache misses. This is because DEX employs cooling delegation (Section 5.3) where a parent becomes an eviction candidate (i.e., in cooling) after all its children. Therefore, DEX makes the offloading decision upon a cache miss on node N at level L, then the memory-side thread will take over and finish the remaining index operation including the local traversal from level L to level 0 and return the operation result to the compute server. However, DEX does not offload (1) the traversal of inner nodes that do not exclusively belong to one partition or (2) when the node is at level $L > M$. Condition (1) ensures that the missed subpath from N to the leaf level is dedicated to one compute server. For condition (2), recall that DEX enforces nodes in the sub-tree rooted at level M are all stored in one memory server (Figure 2). These two conditions guarantee that offloading is confined between one compute server and one memory server, avoiding much complexity and remote pointer chasing across memory servers. For the same reason, DEX will fall back to the normal path when an offloading attempt returns and reports that it would trigger a structural modification operation (SMO) on the memory server because SMOs (such as leaf node splits) can propagate up to nodes at level $M + 1$ or higher.

It is important to note that offloading is only beneficial when there is spare compute capacity available in the memory pool. Previous offloading policies [47] that always offload index operations, can easily saturate limited memory-side CPU. We show this effect using an Offload-only variant that caches inner nodes above level M and always offloads the remaining index operation. As shown in Figure 5 (details in Section 8.1), Offload-only cannot scale due to high contention in the memory server caused by excessive requests, whereas our cost-aware approach (described below) can truly benefit from offloading and outperform other baselines.

Next, we discuss how DEX makes the offloading decision upon a cache miss, followed by our approach to ensuring consistency between compute-side cache and nodes in memory servers that are simultaneously modified by offloaded operations.

6.1 Load and Cost Aware Offloading

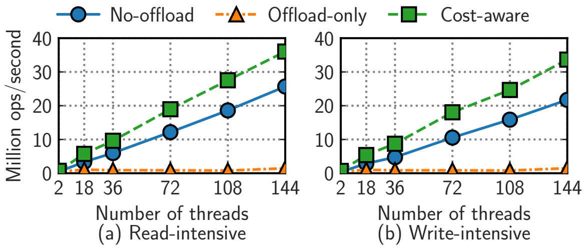
DEX determines whether a sub-path traversal is worth being pushed to remote memory using statistics collected at runtime. While more sophisticated methods (e.g., those that leverage machine learning) exist, DEX strikes a balance between runtime overhead and decision-making accuracy.

Upon a cache miss on a node N at level L, DEX compares the estimated latencies of conducting the access via one-sided RDMA vs. offloading. The former is estimated as $(L+1)\times(l_o+l_s)\times c$, where l_o is the latency of an RDMA READ, l_s is the latency of a local node search in the compute-side cache, and c is an empirically determined coefficient (>1) that accounts for the operational cost of using the compute-side cache (e.g., the cost of free page provisioning). Our experiments show that l_s and c are relatively stable and are therefore specified upon tree initialization. The latter (offloading latency, l_p) and l_o are empirically determined based on the moving average of a predefined number (e.g., 50) of recent samples of the corresponding actions (i.e., two-sided RPC for offloading and RDMA READ). To cope with workload changes, DEX ensures that it has a small probability q (e.g., 1%) of taking the contrary action, allowing for regular updates. With these, an offloading request is issued only when offloading takes shorter latency than the accesses via RDMA READ, i.e., when $l_p < (L+1)\times(l_o+l_s)\times c$.

6.2 Compute-Memory Data Coherence

With concurrent updates from compute-side worker threads and memory-side offloading threads, tree nodes in the cache and memory pool may diverge, jeopardizing correctness. Therefore, data coherence needs to be resolved vertically [44] between compute and memory servers. DEX first simplifies the problem by ensuring that offloading occurs exclusively between a pair of compute and memory servers, as stated earlier in this section. We then focus on ensuring coherence between one compute and one memory server.

A pushdown thread in a memory server by design always can operate on the latest subtree T_N whose root is N, with no dirty pages of T_N in the compute server's cache. That is because when N is not in the cache, it implies all nodes in T_N are either evicted or under cooling (with their dirty pages written back to the memory pool already). This mandates two conditions for correct offloading. (1) No concurrent compute threads are working on the cached (cooling) nodes in T_N or using RDMA to retrieve any missing node in T_N on the compute side when offloading is in-progress. (2) Any node update during offloading is propagated back to the compute node to invalidate the corresponding cached copies.



(a) Read-intensive
(b) Write-intensive

Figure 5: DEX's throughput under different offloading policies, the cache size is set to 1% of the data.

To satisfy condition 1, before the offloading operation starts, DEX pins N's parent in a hot state to safeguard it against eviction. In addition, we insert N into the mapping table with the I/O flag as the value. Concurrent compute threads seeing the I/O flag when looking up N in the mapping table would restart from the root to avoid conflicts with the concurrent ongoing pushdown request, similar to how we prevent concurrent threads from issuing repeated RDMA operations earlier. To satisfy condition 2, memory-side offloading threads return the global address of the updated node upon success (or return failure status on encountering any SMO). After an offloading request returns, the compute-side worker thread checks the mapping table to determine if there are any cached copies of the updated nodes and if so, invalidates them by removing their references from the mapping table. Such invalidation is rare because the subpath from node N to the leaf node has a high probability of not being in the cache, thanks to path-aware caching and cooling delegation. Finally, DEX unpins N's parent node and removes N from the mapping table, completing this pushdown request.

7 INDEX OPERATIONS

Now we describe how DEX performs common index operations.

Lookup and Update. As described in Algorithm 1, for any index operation, the compute thread initially navigates the cached tree path, employing optimistic lock coupling [18] for synchronization. When reaching the leaf node, DEX also only requires in-memory locks instead of RDMA-based locks thanks to logical partitioning.

Insert. Insert operations may cause node splits to accommodate new keys. We use an eager split policy that any full index node encountered during the top-down traversal will be immediately split. However, since the cached shared nodes (e.g., root) crossing partition boundaries may not be up-to-date, we only apply this policy to index nodes dedicated to the current compute server. Node splits may propagate up to upper level nodes that are shared. We determine whether to split those shared nodes or not by considering their freshness. Consider the case where an index node N within the current partition is full and its parent P is shared, we acquire the global lock of P and retrieve its up-to-date version from the memory pool. DEX proceeds with and completes the current node split if two conditions are met: (1) cached P is up-to-date and (2) P is not full. If not, it means the shared nodes in this traversal path should be first refreshed and possibly split. As a result, we abandon the ongoing node split and trigger a cache refresh from the remote root with immediate node splits for the shared nodes in the path, if they are full. Finally, DEX will retry its insert operation and possibly continue the abandoned split for N.

Delete. Similar to node splits, merging nodes may also propagate up to shared nodes. Such merging operation proceeds as long as the cached shared nodes are up-to-date. Otherwise, a cache refresh from the remote root would be triggered.

Range Query. To maintain simplicity for the pointer unswizzling process, DEX does not maintain links between leaf nodes, ensuring that unswizzling only operates on references from the parent node. Consequently, range scans that span multiple leaf nodes are subdivided into multiple lookups by employing fence keys. With lightweight concurrency control, the tree traversal of multiple lookups remains efficient. After the initial lookup, the traversal of subsequent lookups is generally cached in the CPU cache. We do not support operation offloading for range queries because a range query may require scanning multiple leaf nodes, whose computation load is hard to estimate. Our evaluation shows that DEX solely with caching and one-sided verbs already significantly outperforms state-of-the-art indexes. The latency of scanning multiple leaf nodes using one-sided verbs can be alleviated via prefetching, using the remote leaf pointers stored in the last-level inner nodes, which is interesting to be explored in the future.

8 EVALUATION

We evaluate and compare DEX with two state-of-the-art range indexes: Sherman [37] and SMART [25]. Major results include:

- DEX achieves 2.5–8.2× and 4.4–9.6× higher throughput than baselines for read- and write-intensive workloads, respectively.
- DEX's superior performance comes from systematically designed techniques. With logical partitioning, our cache design improves DEX's throughput by up to 15×, and opportunistic offloading for constrained cache attributes to 55% further improvement.
- We quantify the cost of logical repartitioning, which can finish in <2s even for compute servers with large caches.

8.1 Experimental Setup

Testbed. We conduct experiments in a cluster of four servers. Each server has two 20-core Intel Xeon Gold 6242R CPUs clocked at 3.1GHz (each with 35.75MB of caches), 384GB DRAM (32GB×12) across two sockets, and a 100Gbps Mellanox ConnectX-5 InfiniBand NIC connected to a 100Gbps InfiniBand switch.

To increase the scale of our experiments and stress test DEX, we follow prior work [25, 37] to configure each machine to act as one compute server and one memory server.[3] On each machine, we allocate 36 cores for the compute server and use the remaining 4 cores for the memory server. Therefore, the compute power ratio between the compute pool and memory pool is 9:1, similar to settings used by previous work [44]. The exact deployment shape (capabilities of each infrastructure group) is still being actively explored and to the best of our knowledge, the industry has not converged on one particular design (e.g., split vs. pool [6]). We study different degree of memory disaggregation by varying the the CPU core ratio from 36:1 to 9:1 in Section 8.4. We allocate 256MB of DRAM cache in each compute server (i.e., ~8% of the bulk-loading dataset size in our evaluations, detailed later in this section) and 64GB of DRAM for each memory server. In our experiments, *cache size* refers to the amount of DRAM used as the compute-side cache in one compute server. Unless explicitly stated, our experiments use all the 144 (36×4) compute-side threads and 16 (4×4) memory-side threads in the cluster. Each thread is pinned to a physical core. Servers run Arch Linux with kernel 6.3.2. All the code is compiled with GCC 13.1.1 with all the optimizations enabled.

Table 1: Microbenchmarks used in our experiments.

Workload	Insert	Lookup	Update	Scan
Read-only	0	100%	0	0
Read-intensive	0	95%	5%	0
Write-intensive	0	50%	50%	0
Insert-intensive	50%	50%	0	0
Scan-intensive	5%	0	0	95%

Implementation and Parameters. We developed DEX using C++. For Sherman and SMART, we use the open-sourced code from their original authors.[4] Since Sherman's original lock-free search using versions [46] is not fully correct, we implemented an RDMA-based optimistic locking for correct synchronization [46].

For fair comparison, we configure Sherman and SMART with the parameters as recommended by their original papers [25, 37]. Sherman uses 1KB fixed-size tree nodes, and SMART uses variable node sizes. For DEX, we use 1KB node size. We also set DEX's cooling map's capacity to 10% of the cache. Each cooling map bucket occupies 64-byte (one cacheline) to include six FIFO slots. Subtrees from level 0 to $M = 3$ are grouped within the same memory servers. For logical partitioning, we range-partition the key space such that each compute server owns an equal key range.

Benchmarks. We stress test the indexes with microbenchmarks modeled after YCSB [4]. Table 1 describes our five workloads. The range scan is performed by reading 100 records in ascending order from the initial key. The scan-intensive workload evaluates range query performance. Unless explicitly specified, we generate skewed keys for all the workloads, following a Zipfian distribution (theta=0.99) which is the default in YCSB. For scalability experiments, we also include tests with uniform workloads.

For all runs, we first bulk-load the index with 200 million key-value records, execute a warmup phase comprising 10 million workload operations, and then benchmark 200 million workload operations. For any benchmark exceeding 60 seconds, we collect results from the initial 60 seconds. Unless otherwise specified, we use 8-byte keys and 8-byte values which can be either an inlined payload or a pointer to an actual record.

8.2 Performance and Scalability

We examine how each index performs and scales with an increasing number of compute threads under different workloads. As new compute threads are added, we first exhaust the available cores on existing compute servers, before adding a new one. We use all four memory servers to store index nodes. Since logical partitioning is

[3] RDMA between a compute and a memory server co-located on the same physical machine may not go through the switch. However, we observe such accesses already exhibit ~90% of the cross-server latency going through the switch. We therefore believe it is a reasonable tradeoff for larger-scale experiments.

[4] http://github.com/thustorage/Sherman and http://github.com/dmemsys/SMART.

Table 2: RDMA statistics per index operation in skewed read-only (RO)/write-intensive (WI) workloads under 144 threads.

Index	Reads	Writes	Atomics	Two-sided	Traffic (B)
DEX (RO)	0.33	0	0	0.0002	333.9
Sherman (RO)	3.02	0	0	0	1064.69
SMART (RO)	1.44	0	0	0	996.99
P-Sherman (RO)	1	0	0	0	1025.04
P-SMART (RO)	1.15	0	0	0	397.41
DEX (WI)	0.33	0.19	0	0.0001	524.1
Sherman (WI)	2.71	0.99	0.59	0	1078.95
SMART (WI)	1.45	0.11	0.11	0	1002.88
P-Sherman (WI)	1.02	0.5	0	0	1054.39
P-SMART (WI)	1.16	0.13	0	0	404.207

also applicable to Sherman and SMART, we enable it for them to better understand its benefits (denoted as P-Sherman/P-SMART). Specifically, we range-partition the key space such that non-shared nodes do not require RDMA-based synchronization.

Skewed Workloads. As shown in Figure 6(a), for read-only workloads, DEX scales better and outperforms Sherman/SMART/P-Sherman/P-SMART by 3.6×/9.6×/2.5×/7.1×, respectively. This superiority stems from DEX's ability to cache hot tree paths including leaf nodes, with very low coherence overhead among compute servers. Conversely, competitors need to retrieve leaf nodes from remote memory through costly RDMA operations, leading to diminished performance. P-Sherman and P-SMART exhibit higher performance than Sherman and SMART, respectively, thanks to the reduced RDMA-based optimistic reads and better cache locality. Table 2 lists the corresponding RDMA statistics for all indexes under read-only workloads with 144 compute threads. Leveraging efficient caching, DEX incurs much lower RDMA costs: on average it cuts 89%/77%/67%/71% RDMA operations and 69%/67%/67%/16% of RDMA traffic per index operation compared to Sherman/SMART/P-Sherman/P-SMART, respectively. Furthermore, we note that DEX incurs very few two-sided operations in Table 2. The reason is that having sufficient cache capacity in compute servers reduces the need for offloading (more discussions in Sections 8.3 and 8.4).

As Figure 6(b) shows, DEX outperforms Sherman/SMART/P-Sherman/P-SMART by 3.4×/8.2×/2.5×/5.3× under read-intensive workloads. SMART exhibits the poorest scalability due to the unscalable FIFO-based caching policy. Our profiling result shows that its cache admission/eviction takes 49% CPU cycles due to severe contention. For write-intensive workloads depicted in Figure 6(c), DEX outperforms Sherman/SMART/P-Sherman/P-SMART by 9.6×/7×/7×/4.4×, respectively. RDMA statistics in Table 2 reveal that DEX benefits from logical partitioning, avoiding global synchronization overhead (i.e., RDMA atomics) by dedicating each leaf node to a single compute server. In contrast, Sherman and SMART incur more RDMA atomics and writes, due to the manipulation of RDMA-based locks and the immediate write-back of updated leaf nodes to the remote memory pool, hindering their performance and scalability. Although P-Sherman and P-SMART avoid RDMA-based synchornization for leaf nodes, they still lag behind DEX due to RDMA traffic of leaf nodes. The results in insert-intensive workloads depicted in Figure 6(d) follow the similar trend as write-intensive workloads.

For scan-intensive workloads in Figure 6(d), DEX outperforms Sherman/SMART/P-Sherman/P-SMART by 2.8×/56.3×/1.6×/48.4×, respectively. SMART and P-SMART lag because each leaf node stores only one key-value record, necessitating excessive RDMA operations for range scans.

Uniform Workloads. Most real-world workloads are skewed [26, 36]; we use uniform workloads to study the worst case for caching and DEX's performance lower bound. Figure 7 illustrates that DEX consistently outperforms Sherman, SMART and P-SMART across all uniform workloads. Compared to the results under skewed workloads, the performance gap between DEX and other indexes is smaller. Since uniform workloads inherently exhibit much less locality, caching becomes less effective and DEX performs similarly to P-Sherman. Nevertheless, DEX can still leverage the limited locality with larger caches, whereas it is impossible for P-Sherman to do so because it by design does not cache leaf nodes. For example, with 512MB cache, DEX improves performance by ~20%.

8.3 Effect of DEX Design Choices

This section quantifies the impact of individual design choices in DEX, including the relative contribution of each optimization, effect of our cache design and the cost of logical repartitioning. We compared DEX with a non-distributed B+-tree on a single machine to assess distribution overhead. DEX shows only an 8% performance decrease, as detailed in the extended version [24].

Ablation Study. We study the effect of each optimization under write-intensive workloads. Starting from a baseline RDMA B+-tree (described in Section 2.2), we add logical partitioning, caching and opportunistic offloading to show the throughput improvement. We first enable logical partitioning because it serves the foundation of our caching and pushdown design. We set a small cache size (e.g., 1% of the working set or 31MB) to trigger offloading.

As Figure 8(a) shows, the baseline cannot scale under skewed workloads due to excessive remote accesses. Under two compute threads (one in each NUMA node), logical partitioning improves the throughput by 2.4× (i.e., from 0.04 to 0.096 Mops/s), thanks to the elimination of RDMA-based synchronization on non-shared nodes. However, with more threads, network bandwidth becomes the bottleneck again, and the speedup is limited. Figure 8(b) shows similar observations under uniform workloads. Further adding caching improves throughput by 21.2×/6.9× under skewed/uniform workloads because hot tree paths are cached. Finally, adding offloading increases throughput by 55%/34% under skewed/uniform workloads, benefiting from near-data processing.

Cache Design. We evaluate the effectiveness of the cooling map and leaf admission control. We disable opportunistic offloading here to isolate the effect of these two design choices. We experiment with two cache sizes: 64MB and 256MB. The former stresses our replacement algorithm, given its higher cache replacement frequency; the latter is the default size. Starting with a baseline that (1) uses a single lock to protect the cooling map and (2) employs an eager cache admission policy, we incrementally introduce other features and measure throughput under 144 compute threads.

The results are shown in Figure 9. Compared to the baseline, using the cooling map improves DEX's throughput by 12× and 10× for 64MB and 256MB cache, respectively. This improvement stems

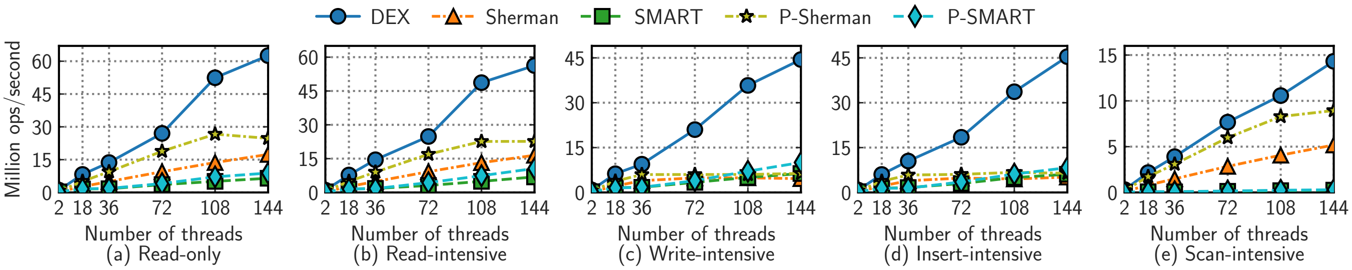


Figure 6: Throughput under skewed workloads with a varying number of compute threads.

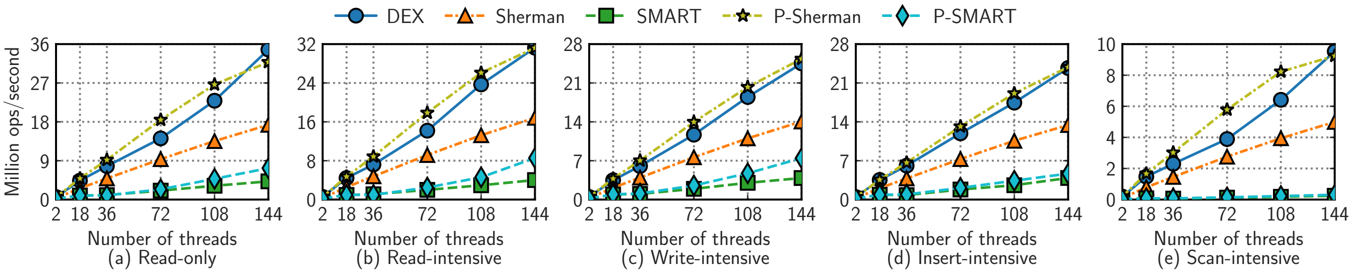


Figure 7: Throughput under uniform workloads with a varying number of compute threads.

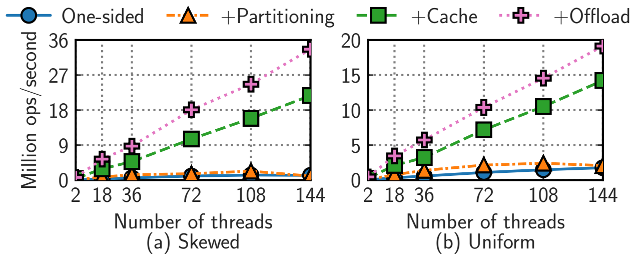


Figure 8: Effect of each optimization in DEX under different cache sizes and write-intensive workloads.

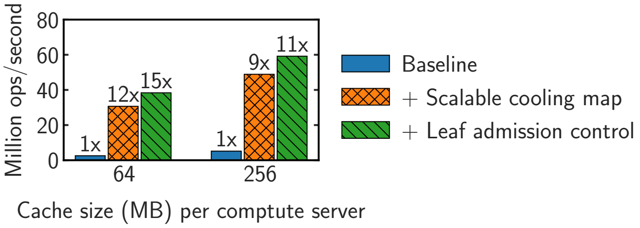


Figure 9: Effect of cache design choices in skewed read-intensive workloads under different cache sizes.

from the use of fine-grained bucket-level locks in the cooling map, significantly reducing contention upon cache replacement. On top of that, adding leaf admission control enables DEX to filter out potentially cold pages. This further improves throughput by 25% and 21% for 64MB and 256MB cache, respectively. This experiment underscores the efficacy of our caching design, even in scenarios with highly constrained caches.

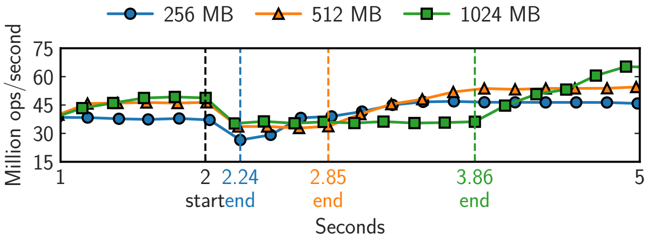


Figure 10: Throughput changes during logical repartitioning (started at second 2) under skewed write-intensive workloads.

Cost of Logical Repartitioning. As Section 4 describes, DEX can promptly repartition to satisfy the scaling requirement or resolve load imbalance on compute servers. We demonstrate this point by observing DEX's throughput changes over time during the repartitioning process. We start with write-intensive workloads across three compute servers, and then select one for repartitioning. For a compute server, the cost of repartitioning includes (1) flushing its dirty cache to the memory pool, and (2) transferring a portion of its key range to another compute server.

As shown in Figure 10, repartitioning begins after the benchmark has run for two seconds. DEX completes repartitioning within two seconds for cache sizes ranging from 256MB to 1024MB, with larger caches requiring longer time to flush dirty cache pages. Notably, these results are based on a single compute thread for dirty cache flushing; employing more compute threads could further accelerate the repartitioning process. After repartitioning, both the repartitioned compute server and the scale-out (new) server undergo cache warm-up, gradually ramping up the throughput to normal levels.

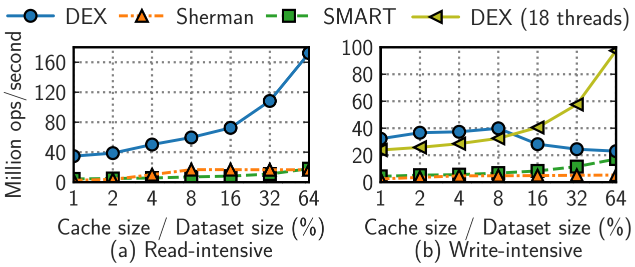


Figure 11: Throughput under different cache sizes.

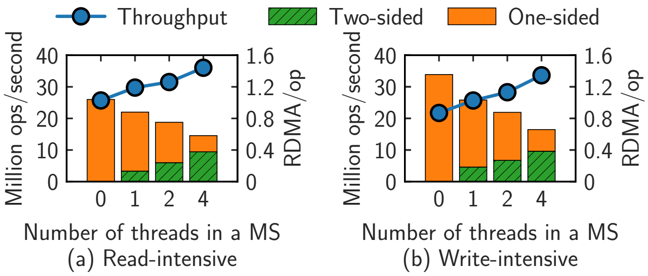


Figure 12: Effect of opportunistic offloading under skewed workloads with varying threads in each memory server (MS).

8.4 Sensitivity Analysis

Now we study how different cache sizes and memory-side compute power impact index performance.

Cache Size. Varying cache size changes cache ratio ($\frac{cache\ size}{dataset\ size}$). Figure 11(a) shows that DEX's performance significantly improves as the cache ratio increases under skewed read-intensive workloads. Having a sufficiently large cache enables DEX to cache more tree paths, thus incurring fewer remote accesses. In contrast, Sherman and SMART do not exhibit the same benefit from large caches, as they do not cache leaf nodes at all. Figure 11(b) highlights DEX's throughput in skewed write-intensive workloads under different cache ratios. Its performance improves as the cache ratio increases from 1% to 8% because of reduced remote accesses. Interestingly, using larger caches (cache ratio > 8%) lowers performance. The reason is that local synchronization (using optimistic locking) in the cache becomes a scalability bottleneck since the workload is skewed. This becomes particularly severe when we use more than one NUMA node. We verified this by re-running the experiment with only 18 compute threads pinned to the same socket (labeled as DEX (18threads) in the figure). With 18 threads all in one socket, DEX scales well without cross-NUMA synchronization. We observe the culprit is that the optimistic lock used here is based on centralized spinlocks that are known to be vulnerable to high contention. We leave it as future work to address this issue using more recent robust optimistic locks [31] in disaggregated B+-trees.

Impact of Memory-Side Compute Power. Assessing the effectiveness of opportunistic offloading involves varying the number of memory-side threads serving offloading requests. To trigger offloading, as done in Section 8.3, we set the cache size in each compute server to 1% (i.e., 31MB) of the data size. Figure 12 shows the throughput and RDMA statistics under 144 computing threads. As we use more threads to serve offloading requests, the number of RDMA operations including both one-sided and two-sided (incurred by offloading) is reduced by 56%/49% in read-intensive/write-intensive workloads. This then leads to a throughput increase of 40%/55%. As more compute power becomes available in memory servers, DEX can dynamically offload more index operations to the memory pool, effectively reducing overall RDMA costs.

9 RELATED WORK

Range Indexes for Disaggregated Memory. Most DM-optimized indexes are shared-everything. FG [47] is the first DM-based B+-tree that entirely relies on one-sided RDMA. Sherman [37] and SMART [25] cache inner nodes of tree indexes and necessitate remote accesses to leaf nodes. dLSM [38] is a DM-optimized log-structured merge tree which adopts a shared-nothing architecture (physical sharding). DEX takes a different approach that is based on logical partitioning for reduced consistency overhead.

Modern Database Caching. While DEX's cache design is inspired by sampling-based caching approaches, we study and optimize it for scalable range indexing on disaggregated memory. LeanStore [16] randomly samples pages and puts them into a shared FIFO list for cooling but exhibit severe contention on disaggregated memory. A more recent improvement is a simpler second-chance strategy [1] where pages already sampled two times are immediately evicted. It requires dedicated background threads for sampling and timely free-page provision. Other sampling-based caching approaches [35, 45] rank page hotness using epoch information embedded in each cache page and also use page-provider threads to avoid stalling worker threads. DEX uses a cooling map for hotness ranking and scalable eviction on disaggregated memory, without employing dedicated threads.

10 SUMMARY

Disaggregated memory poses unique challenges for building scalable range indexes. We observe that achieving high scalability requires a holistic design to efficiently utilize limited memory(compute) in compute(memory) pool with low consistency overhead. We present DEX, which systematically combines three techniques to reduce remote memory accesses and maintain good scalability, as demonstrated through extensive evaluations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. We extend our thanks to Fan Yang (Microsoft Research) for his insightful discussions on paper writing, and to Qianxi Zhang and Ran Shu from Microsoft Research for their extensive discussions during the early stages of this work. This work is partially supported by Hong Kong General Research Fund (14208023), Hong Kong AoE/P-404/18, and the Center for Perceptual and Interactive Intelligence (CPII) Ltd under InnoHK supported by the Innovation and Technology Commission, an NSERC Discovery Grant, Canada Foundation for Innovation John R. Evans Leaders Fund and the B.C. Knowledge Development Fund.

REFERENCES

[1] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI, Vol. P-331)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.). Gesellschaft für Informatik e.V., 259–281. https://doi.org/10.18420/BTW2023-13

[2] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1626–1639. https://doi.org/10.1145/3514221.3526121

[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 521–534.

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

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

[6] Mohammad Ewais and Paul Chow. 2023. Disaggregated Memory in the Datacenter: A Survey. *IEEE Access* (2023).

[7] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26.

[8] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *Proc. VLDB Endow.* 8, 1 (2014), 37–48.

[9] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019*. ACM, 39:1–39:10.

[10] Infiniband™. 2018. About Infiniband™. https://www.infinibandta.org/about-infiniband/ Accessed: 2023-10-24.

[11] Intel Corporation. 2016. Intel 64 and IA-32 Architectures Software Developer Manuals. (Oct. 2016).

[12] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 439–450. http://www.vldb.org/conf/1994/P439.PDF

[13] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[14] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2015, Portland, OR, USA, June 16, 2015*, Torsten Hoefler and Kamil Iskra (Eds.). ACM, 1:1. https://doi.org/10.1145/2768405.2768406

[15] 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.

[16] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196.

[17] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[18] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*. ACM, 3:1–3:8.

[19] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[20] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[21] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

[22] 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*, Stephen W. Keckler and Luiz André Barroso (Eds.). ACM, 267–278. https://doi.org/10.1145/1555754.1555789

[23] Compute Express Link™. 2022. About CXL™. https://www.computeexpresslink.org/about-cxl Accessed: 2023-10-01.

[24] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory [Extended Version]. *arXiv preprint arXiv:2405.14502* (2024).

[25] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 553–571.

[26] Clifford A Lynch. 1988. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values.. In *VLDB*. 240–251.

[27] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[28] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 297–306. https://doi.org/10.1145/170035.170081

[29] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1 (2010), 928–939.

[30] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos K. Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 868–880. https://doi.org/10.1109/MICRO50266.2020.00075

[31] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proc. ACM Manag. Data* 1, 3 (2023), 216:1–216:26. https://doi.org/10.1145/3617336

[32] Radu Stoica, Roman Pletka, Nikolas Ioannou, Nikolaos Papandreou, Sasa Tomic, and Haris Pozidis. 2019. Understanding the design trade-offs of hybrid flash controllers. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 152–164.

[33] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 30:1–30:14.

[34] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1541–1555.

[35] Demian Vohringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3323–3334.

[36] Christopher B Walton, Alfred G Dale, and Roy M Jenevein. 1991. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins.. In *VLDB*, Vol. 91. 537–548.

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

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

[39] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference*

on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 473–488. https://doi.org/10.1145/3183713.3196895

[40] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2021. Autoscaling tiered cloud storage in Anna. *VLDB J.* 30, 1 (2021), 25–43.

[41] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 415–432. https://doi.org/10.1145/3299869.3300085

[42] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.

[43] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (2020), 1568–1581. https://doi.org/10.14778/3397230.3397249

[44] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1345–1359.

[45] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 685–699.

[46] 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.

[47] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 741–758.