Dolphin: A Resource-efficient Hybrid Index on Disaggregated Memory

Hang An, Fang Wang*, Dan Feng, Zefeng Liu
Wuhan National Laboratory for Optoelectronics,
Key Laboratory of Information Storage System, Ministry of Education,
Engineering Research Center of data storage systems and Technology, Ministry of Education,
Huazhong University of Science and Technology, Wuhan, China,
Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, China
{anhang, wangfang, dfeng, zefengliu}@hust.edu.cn

***Abstract*—The Disaggregated Memory (DM) architecture enables cloud providers to achieve unparalleled resource utilization and scalability. The clients operate on Compute Servers (CS) and access data on Memory Servers (MS) through DM-based indexes. However, due to inherent limitations in perspectives, existing DM-based indexes struggle to strike a balance between memory consumption, read performance (including Scan), and write performance. Some schemes even deviate from the original intent of reducing memory waste in pursuit of higher performance.**

In this paper, we present Dolphin, a DRAM-friendly and high-performance hybrid index on DM. To reduce memory consumption, Dolphin employs a hybrid architecture consisting of Adaptive Radix Tree (ART) internal nodes and B+-tree leaf nodes. It constructs a mapping between 1 index cache item and N remote data items, thereby reducing the number of index cache items. To quickly locate the target leaf nodes, Dolphin leverages a Hybrid-Index-friendly pathfinding algorithm, laying the foundation for improved read and write performance. To mitigate the synchronization overhead of write operations, Dolphin utilizes the features of RDMA_WRITE and proposes an MS-RNIC (RDMA Network Interface Card) Writing Synchronization mechanism, achieves high concurrency of Update operation and reduces the round trip time (RTT) of Update operation from 3 to 2. Compared to the state-of-the-art DM-based indexes, Dolphin achieves optimal performance across all metrics. Experimental results show that Dolphin reduces memory consumption by 1.44×-16.25×, achieving 1.16×-3.26× higher throughput under synthetic and real-world workloads. We will release the open-source codes for public use in GitHub.

***Index Terms*—Index structures, Disaggregated Memory, RDMA, Memory consumption**

I. Introduction

In modern data centers, the overall cost of DRAM stands as notably exorbitant, encompassing approximately 50% of the hardware expenditure [1]. Unfortunately, reports from industry giants like Google, Microsoft, and Alibaba indicate the existence of substantial memory underutilization within their data centers [2]–[4]. Consequently, academia and cloud providers have dedicated considerable efforts in recent years to investigate the realm of DM architectures [24], [25], [29], [34], [35], [38]. DM re-consolidates the CPU and DRAM resources to form Compute Servers (CS) and Memory Servers (MS). CSs and MSs are connected by fast networks (i.e., RDMA and CXL). The clients on CSs access data in MSs via a DM-based indexing system.

Building efficient index structures in DM is promising to offer high performance for in-memory databases. Current DM-based indexing systems prioritize optimizing either read or write operations, necessitating a sacrifice between the two and overlooking the optimization of memory consumption. Similarly, the "RUM Conjecture" states that we cannot have all three of read, update, and space optimized for a data structure [5], [6]. The issue is exacerbated further as half of the available memory of a DBMS might be consumed by index structures [7]. Consequently, achieving high performance while reducing memory consumption becomes imperative. However, this target poses significant challenges for write optimizations as conventional write synchronization mechanisms, such as the exclusive lock or CAS-based lock-free scheme, may not be applicable. In essence, designing DM-based indexes should strive to address the following challenges:

- *High Memory Overhead.* To quickly obtain the pointers to remote target leaf nodes without the need for remote index traversal, all DM-based indexes [15]–[18] need to build a key-pointer cache in CSs. However, existing DM-based indexes either cache remote pointers for all data items or cache complete keys, squandering precious DRAM in CSs. Furthermore, organizing data items independently is beneficial for enhancing concurrent write performance or alleviating read amplification, but it introduces additional metadata (data item pointer, checksum, etc.), exacerbating memory consumption.
- *Compromised Read (Scan) Operations.* To achieve high concurrency, some indexes [18], [19] adopt the out-of-place update scheme, connecting data items to the index via indirect pointers, thereby mandating at least two RTTs for read operations. Some indexes [17] organize leaf nodes by data item granularity to reduce read amplification, causing consecutive key-value pairs to lose spatial locality. Both schemes necessitate copious RDMA_READs for Scan operations, leading to a significant degradation in performance.
- *Tricky Remote Write Synchronization.* Existing solutions predominantly leverage CAS-based exclusive locks or CAS-based lock-free schemes to address write-write conflicts. Unfortunately, they either incur expensive synchronization overhead [16] or result in additional memory consumption [18]. Additionally, they unavoidably elevate the RTT for write

operations, consequently inducing higher latency.

To address the above challenges, we propose Dolphin[1], a resource-efficient hybrid index tailored for DM. Dolphin extensively amalgamates the advantages of ART and B+-tree while leveraging RDMA features for efficient in-place updates. Dolphin's primary objective is to strike a delicate balance between memory overhead, read performance (including Scan), and write performance, thereby providing support for systems based on DM.

To curtail DRAM consumption in CSs and MSs, Dolphin provides a hybrid architecture comprising ART internal nodes and B+-tree leaf nodes. CSs cache ART internal nodes, without storing complete keys. Data items reside in B+-tree leaf nodes, yielding two benefits: (1) a 1-N mapping between index cache item and remote data items, reducing the count of index cache items, and (2) shared metadata for data items in leaf nodes, thereby reducing metadata consumption in MSs.

To support high-performance Read (Scan), Dolphin stores data items in B+-tree leaf nodes and proposes a pathfinding algorithm conducive to the hybrid index. The clients in CSs can accurately and swiftly locate the remote pointers of the target leaf node, enabling the retrieval of leaf nodes within one RTT and reading tens of data items in a single sweep, ensuring optimal Read and Scan performance.

To reduce the synchronization overhead of remote write, we leverage two RDMA features: (1) RDMA_WRITEs are guaranteed to be performed in increasing address order; (2) Data atomicity per cacheline is guaranteed by RNIC [24]. Dolphin directly employs a single RDMA_WRITE, overwriting the target data items within leaf nodes. This approach is attributed to the first feature, ensuring the eventual result of concurrent updates is consistent. The second feature guarantees consistency in read-write results within a cacheline. Hence, we utilize RNIC on MSs to synchronize concurrent update operations instead of remote atomic operations (i.e., RDMA_CAS). Furthermore, we effectively address other types of write-write conflicts and read-write conflicts for objects larger than a cacheline.

Specifically, this paper mainly makes the following contributions:

- We propose a hybrid index amalgamating ART and B+-tree, which effectively reduces memory consumption in both CSs and MSs.
- We design a pathfinding mechanism tailored for our hybrid index, facilitating precise traversal to target leaf nodes and enabling efficient operations on these nodes.
- Leveraging RDMA features to implement the MS-RNIC Writing Synchronization mechanism, effectively mitigating synchronization overhead of write operations.
- We implement Dolphin[2] and evaluate it using synthetic and real-world workloads [8], [42]. The evaluation results demonstrate the low memory consumption and efficiency of Dolphin.

[1] We hope Dolphin to be as intelligent, energetic, and friendly as the dolphin.

[2] The source code is available at https://github.com/hust-anhang/Dolphin

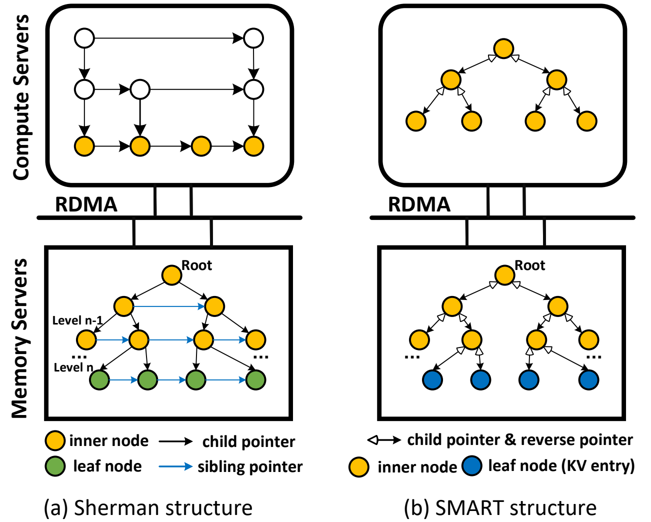

Fig. 1. Two typical DM-based indexes.

II. Background and Motivation

A. Disaggregated Memory

In the Disaggregated Memory architecture, CPUs are centralized within CSs, while DRAM is consolidated within MSs. These CSs and MSs are interconnected using high-speed communication protocols (i.e., RDMA, CXL), collectively constituting pools of compute and memory resources. Cloud providers allocate compute and memory resources according to user demands, enhancing resource utilization and lowering hardware costs. CSs typically feature a limited amount of DRAM, serving as a cache for MSs, while MSs are equipped with a small amount of CPU to handle control requests (connection establishment and memory allocation) from CSs.

Given the current non-commercialization status of CXL, the majority of DM-based approaches utilize RDMA for inter-server communication. Despite a performance gap of 10×, RDMA and CXL share some functional similarities: byte addressability, atomic operations, and cacheline atomicity. Our work integrates the characteristics of DM architecture and RDMA to optimize resource utilization while upholding system performance. Although CXL_WRITEs may not be performed in increasing address order, hybird index structure and pathfinding algorithm of Dolphin are also applicable to CXL-based systems. The index cache aids in increasing system throughput (local memory bandwidth + CXL memory bandwidth). Furthermore, CXL may encounter issues similar to RDMA: conducting atomic operations across multiple servers towards a single server might still lead to performance collapse [3]. Our exploration of RDMA atomic operations may provide some guiding insights for future CXL-based work.

B. Index Cache

The index cache in DM differs from the conventional cache issues. Traditional cache serves read-intensive workloads, stor-

[3] Private communication with authors of Sherman

TABLE I
COMPARISONS AMONG DOLPHIN AND STATE-OF-THE-ART DM-BASED INDEXES. (IN THE TABLE, "✓" INDICATES GOOD PERFORMANCE, AND "×" INDICATES BAD PERFORMANCE.)

	Memory Consumption		**Read Performance**		**Write Performance**	
	mechanism	consumption	mechanism	performance	mechanism	performance
RACE Hash	Hash	✓	Indirect Read	×	Lock-free	✓
Sherman	B+-tree leaf node	✓	Read leaf node	✓	Exclusive lock	×
Marlin	Independent data items	×	Indirect Read	×	Lock-free-like	✓
SMART	Independent data items	×	Read leaf node	✓ (× for Scan)	Write-combination	✓
Dolphin	B+-tree leaf node	✓	Reaf leaf node	✓	MS-RNIC synchronization	✓

ing hot data items. However, the index cache typically stores internal nodes of tree indexes, thus reducing remote index traversal and exhibiting structural correlation among index cache items. Hence, the index cache is directly associated with index structure design. Current index cache schemes primarily fall into two categories, exemplified by Sherman [16] and SMART [17], as shown in Figure 1.

Sherman caches the highest two levels of nodes (including the root) and the upper layer of internal nodes on the leaf node. Similar to the traditional B+-tree, Sherman's internal nodes store keys and remote pointers to leaf nodes. To expedite locating the upper-level internal nodes, Sherman constructs a skip list based on the key range represented by the internal nodes.

SMART endeavors to cache as many ART internal nodes as possible and constructs a local ART in CSs. The internal nodes in SMART contain a byte of the key and remote pointers to the next-level nodes. The last layer's internal nodes consist of the remote pointers pointing to leaf nodes in MSs.

Based on our observation, the essence of the index cache constitutes a mapping relationship of one cached leaf node pointer to a leaf node and n data items (1-1-n). A B+-tree leaf node typically contains dozens of data items, whereas an ART leaf node stores only one data item. With equivalent data volumes, a B+-tree inherently requires fewer cached leaf node pointers. However, for a B+-tree, the index cache needs to store complete keys to identify the key range in the leaf node. On the other hand, ART requires only a byte for each cached node pointer because it employs a byte-by-byte comparison path for representing complete keys. Hence, ART's upper-level structure conserves memory space more effectively.

C. *Performance-Resource Balance*

Several DM-based indexes aim to either avoid performance collapse caused by CAS-based exclusive locks, enhance write concurrency, or reduce read amplification. Although they achieve their specific objectives well, they often lead to a decrease in other system metrics. We will analyze each of them individually.

RACE Hash [19] is a DM-based lock-free hash. It caches directories of hash in CSs, while MSs' buckets store pointers to the latest data items. It uses RDMA_CAS for updating data item pointers and relies on RDMA_CAS's atomicity to resolve the conflicts of concurrent write operations. During subtable resizing, it employs RDMA_CAS to migrate data items individually, enabling concurrent execution of requests within subtables. Overall, its lock-free implementation relies on RDMA_CAS. Thus, data items are stored independently rather than within buckets. When the clients retrieve the target data item at MS, they require reading the bucket first, obtaining the data item pointer from the bucket, and then fetching the data item based on the pointer.

Sherman [16] is a DM-based B+-tree. It uses CAS-based exclusive locks to address write-write conflicts. To prevent blind retries of the exclusive lock that could result in performance collapse [15], it introduces a hierarchical lock mechanism, reducing the frequency of exclusive lock acquisition in CSs.

Marlin [18] optimizes Sherman's concurrency of write operations. It presents an FAA-based ternary-state node lock to address concurrent conflicts between Structure Modification Operations (SMOs) and Insert, Delete, Update (IDU) operations. And achieving lock-free-like IDU operations. Similar to RACE Hash, it stores pointers to the latest data items in its leaf nodes and employs RDMA_CAS to update data item pointers.

SMART [17] constructs a DM-based ART to avoid read amplification caused by B+-tree leaf nodes. It caches leaf node (containing a data item) pointers in CSs. Additionally, to reduce redundant I/Os, it designs write-combining and read-delegation mechanisms.

Overall, Marlin and SMART have relatively high memory consumption. The former requires additional storage for data item pointers and data item metadata in MSs, while the latter needs to cache a substantial number of leaf node (data item) pointers in CSs. The independent storage of data items in RACE Hash, Marlin, and SMART results in very poor range query capabilities. The first two additionally require two RTTs to retrieve data items. Sherman exhibits poor write concurrency due to its reliance on the inefficient exclusive lock mechanism.

III. DESIGN

Before describing the specifics of our design, we'd like to outline the following goals for Dolphin:

- Reduce the memory overhead of the index cache in CSs and the memory consumption of the index in MSs.
- Devise an efficient pathfinding algorithm to retrieve the target data item within 1 RTT.
- Minimize synchronization overhead of write operation while ensuring consistency.

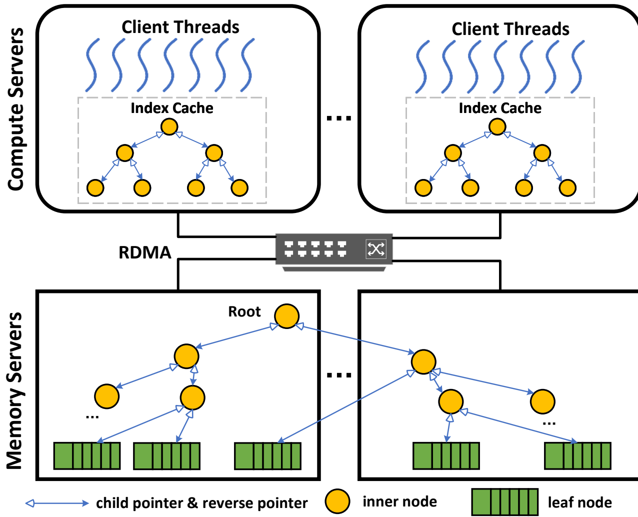

Fig. 2. Dolphin Structure.

To achieve these goals, we provide a detailed description of Dolphin's design and implementation in this chapter, (1) Dolphin's structural design, (2) pathfinding algorithm, and basic operations, as well as (3) read/write concurrency control.

A. The Dolphin Structure

Dolphin is a hybrid index comprising ART internal nodes and B+-tree leaf nodes. Similar to other DM-based tree indexes, Dolphin caches the upper layers of the index in CSs. The structure of Dolphin is shown in Figure 2. Our structural modifications are primarily focused on the B+-tree leaf nodes, allowing them to adapt to the upper-level ART's pathfinding algorithms while enabling more efficient write operations.

1) Internal Node: The internal nodes of Dolphin are classic ART internal nodes [9]. Each node stores partial keys and child pointers, linking the child nodes logically by byte-by-byte comparisons. This entire path from root to leaf node constitutes a complete key, thereby saving memory space for storing keys. To verify the invalidation of the index cache, we drew inspiration from the design of reverse pointer in SMART [17]. If a node splits and generates a new parent node, the reverse pointer will then point to the new parent node, indicating the invalidation of the reverse pointer in the index cache.

Dolphin's internal nodes possess a 'depth' attribute, indicating the position of the stored byte within the key. This attribute is crucial, and its usage will be explained in the pathfinding algorithm section (III-B).

2) Leaf Node: As shown in Figure 3, Dolphin's leaf nodes consist of rev_ptr [17], min, max, Key array, Value array, and lock. The rev_ptr points to the leaf node's parent node, validating the cache's validation. The min and max represent the range of keys contained in the current leaf node as [min, max). When initializing a leaf node due to the insertion of Key A, min is set to ((A>>8)<<8), and max is set to (A|255). Thus, data items sharing the same prefix as A (excluding the last byte) can be inserted into the current leaf node. Keys[n] represents the array of keys, typically 8 bytes in size. During data item insertion, RDMA_CAS is used to modify the target empty slot in the Key array. Values[n] represent the array of values. RDMA_WRITE is employed to update the value. If the value size is less than 64 bytes, RDMA_WRITE ensures atomicity and consistency of the value. For values larger than 64 bytes, Values[n] is combined with cacheline version [24] to verify the value's consistency. Dolphin adopts the ternary-state node lock of Marlin, resolving SMO and IDU operation concurrency conflicts with minimal overhead, allowing high concurrency for IDU operations.

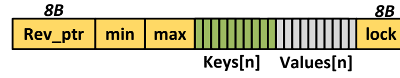

Fig. 3. The leaf node of Dolphin

Separating keys and values into two arrays has two advantages: (1) When dealing with value sizes larger than 64 bytes, it avoids the cacheline version affecting offsets of Key[i] (i.e., i=1,2,3...) because RDMA_CAS requires the target address to be 8-byte aligned. (2) For Update operations, clients read the leaf nodes, except for the Value array. Because the Key array provides the offset of the target value. The read amplification is reduced.

It's crucial to distinguish that PACTree [11], an ART and B+-tree hybrid index based on persistent memory. It organizes leaf nodes as a doubly linked list to ensure data recovery upon unexpected server crashes. This structure necessitates modifications to two sibling nodes during SMO operations, incurring substantial overhead. To mitigate SMO as a scalability bottleneck, it asynchronously updates the upper index. In contrast, Dolphin synchronously modifies the upper index, ensuring client visibility of the latest data. This decision stems from two reasons: (1) Dolphin avoids maintaining doubly linked pointers between leaf nodes, minimizing the complexity of SMO operations and eliminating expensive RDMA_CAS traffic introduced by remote pointer modifications. (2) Even if a client accesses old leaf nodes due to concurrent SMO, Dolphin's backtracking algorithm (III-C) swiftly locates the target leaf node, rather than accessing the sibling node via the doubly linked pointer.

B. Pathfinding Algorithm and Basic Operations

1) Pathfinding Algorithm: While the approach of combining ART with B+-tree seems straightforward, the complexity arises from the fact that ART conducts precise searches for the target leaf node by byte-by-byte comparison, whereas B+-tree searches for potential leaf nodes containing the target data items based on the range of keys. Merging these two traversal methods is quite challenging. Additionally, the potential invalidation of the index cache may introduce uncertainties, leading to wrong paths in the upper layers.

Therefore, we attempt to employ precise byte-by-byte comparisons as much as possible in the upper layers of the

Fig. 4. Basic operations of Dolphin.

index. We introduce a ”special” internal node at a depth of Length(Key)[4], which integrates both byte comparison and range search methods. The internal node at a depth of Length(Key) represents at least Length(Key)-1 bytes of the key. Even if a wrong leaf node is accessed due to node splitting and cache invalidation, we can swiftly backtrack to locate the correct leaf node instead of restarting the search from the root. Premature use of range search could potentially continue down the wrong path, discovering the error only upon reaching the leaf node. In this situation, the clients lose context information, and require a full re-traversal from the root, leading to an unacceptable cost.

In Dolphin specifically, a node at depth of Length(Key) is not a leaf node but a specialized internal node (i.e., Node D in Figure 4 (d)). Its format remains consistent with an ART internal node but with an empty Partial Key. Each byte within this node is accompanied by a pointer directing to the leaf node (i.e., Node C/E in Figure 4 (d)). The leaf nodes store a subset of data items greater than min and less than max. The calculation method for min and max is explained in the leaf node design section. Furthermore, when a leaf node splits, the min and max of the new and old leaf nodes will be reset like B+-tree leaf node splitting.

Through the transition of Length(Key) layer internal nodes, we merge the ART traversal approach with the B+-tree traversal method. This ensures accurate targeting of the desired leaf node, which consists dozen data items. Consequently, it reduces the total memory consumption of leaf nodes and significantly reduces the number of remote pointers in the index cache.

In particular, the core of our pathfinding algorithm involves ”special” internal nodes. These nodes assist Dolphin in addressing the challenges posed by distributed cache consistency issues. The ”special” internal nodes ensure that the target leaf node obtained by the client shares the same prefix. This support enables the effectiveness of the backtracking algorithm in special cases and reduces the complexity of SMO operations. However, PACTree can be considered as a traditional ART. The B+-tree leaf nodes of PACTree are merely subservient to ART leaf nodes, representing partial data items greater than the Key represented by ART leaf nodes. Such an approach raises a challenging issue in DM: inconsistent prefixes within a leaf node complicate SMO operations, introduce additional network overhead, and ultimately result in SMO operations becoming a performance bottleneck.

In the subsequent Concurrency Control section (III-C), we will discuss various special cases caused by concurrent operations and how we efficiently address them.

2) Basic Operations: Like other indexes featuring index cache, Dolphin accesses the index cache in CSs before performing any read or write requests. To reduce round trips for remote index traversal, Dolphin uses the Hybrid-Index-friendly pathfinding algorithm to reach the ancestors of the leaf nodes as closely as possible. In the following, we present the details of several basic operations: Get, Scan, Insert, Update, Delete, and SMO:

Get: Benefiting from an efficient pathfinding algorithm, when a client’s Get operation reaches the target leaf node, it initiates an RDMA_READ operation directly, retrieving the desired data items from the leaf node. In the case of leaf node splits where the index cache hasn’t updated promptly, potentially leading to the retrieval of an old leaf node. In this situation, the client first reads the leaf node’s rev_ptr pointing to the parent node (i.e., Node D in Figure 4 (d)). Then, it locates the pointer to the newly split leaf node (i.e., Node E in Figure 4 (d)) from the parent node, and finally initiates an RDMA_READ to access the leaf node containing the desired data item. This process is known as backtracking.

[4]In key-value storage systems, keys are typically encoded to ensure a consistent length.

Scan: Similar to the B+-tree Scan operation, the client queries the index cache, obtaining pointers to all nodes (including internal and leaf nodes) within the specified range. These pointers are divided into groups based on the MSs they point to, and through Batched RDMA_READ, the client reads the target nodes into CS. For leaf nodes, the client records data items within the target range. For internal nodes, the client initiates Batched RDMA_READ to the next-level nodes until all target leaf nodes have been read.

Insert: The Insert operation consists of two scenarios. In the first, when the client locates the target leaf node with a common prefix, as shown in Figure 4 (a), it directly reads the target leaf node (Node B). Then, it searches for an empty slot (Keys[1]), using RDMA_CAS to set Keys[1] to the target key. Finally, the client uses RDMA_WRITE to overwrite Values[1]. It should be noted that to reduce the size of the Figure 4, we do not show the Values[n] in the leaf nodes. In the second scenario, there is no leaf node in the index representing the first Length(Key)-1 bytes of the target key. As depicted in Figure 4 (b), the client locates the internal node Node A with the longest common prefix and inserts a new child leaf node. For simplicity, we illustrate using 4-byte keys in Figure 4, while standard 8-byte keys are used in evaluation.

Update/Delete: As shown in Figure 4 (c), the client first locates the leaf node containing the target key, then searches for Keys[x] that matches the target key. The client uses RDMA_WRITE to set Values[x] to the target value. The Delete operation is similar to Update: upon finding the target Keys[x], it uses RDMA_WRITE to clear both Keys[x] and Values[x]. Unlike Sherman and SMART, Dolphin achieves Update/Delete requests with only two RTTs by leveraging the characteristics of RDMA WRITE.

SMO: In scenarios where leaf nodes are full or empty, requiring splitting or merging, the process is termed SMO. Take the example of a split when inserting a new key into a leaf node with only one empty slot. Depending on whether the leaf node's parent is a "special" internal node, SMO is divided into two cases. (1) When the parent node is not a "special" internal node, as shown in Figure 4 (d), it first evenly distributes the data items in the leaf node Node C, placing the larger portion into the new leaf node (Node E). Then, it writes Node E and a "special" internal node (Node D) to target MS. Node D contains pointers to its two child leaf nodes. After that, the client modifies the original pointer pointing to Node C to point to Node D. Finally, the client updates the original leaf node (Node C), modifying its rev_ptr and data items. (2) When the parent node is a "special" internal node, as shown in Figure 4 (e), the client writes the new leaf node (Node F), inserts a pointer (pointed to Node F) into Node D, and modifies the old leaf node (Node E).

C. Coordinated Concurrency Control

In terms of index structures, addressing write-write conflicts and read-write conflicts is crucial to ensure consistency. Existing methods to solve write-write conflicts primarily include CAS-based exclusive locks and CAS-based lock-free schemes.

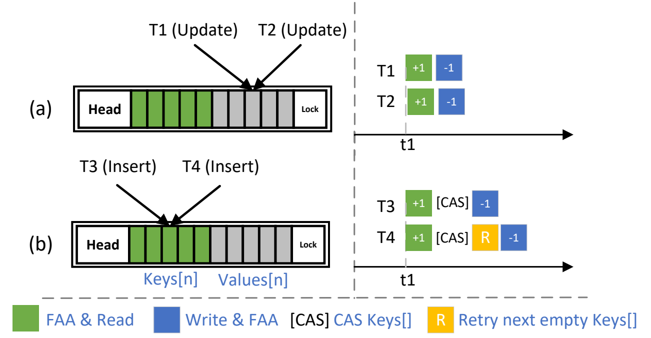

Fig. 5. Update-Update conflict and Insert-Insert conflict (The right part shows the timeing diagram of Dolphin.).

The logic for write operations based on exclusive locks generally follows a sequence of (1) lock, (2) read node, and (3) RDMA_WRITE & unlock. The process requires at least 3 RTTs and offers poor concurrency. CAS-based lock-free schemes support higher concurrency but entail separating data items and index, introducing substantial memory overheads in the form of node address pointers, validation bytes, etc.

Excitingly, we discover that RDMA_WRITE exhibits two features: (1) RDMA_WRITEs are guaranteed to be performed in increasing address order. (2) Data atomicity per cacheline is guaranteed by RNIC. Leveraging these characteristics, we designed the MS-RNIC Writing Synchronization mechanism, resolving write-write conflicts with minimal network overhead. Next, we explain the mechanism by addressing various write-write conflict scenarios.

Update-Update conflicts: In frequent updates to hot data, Dolphin only needs RDMA_WRITE to overwrite the target value. If the target value size is smaller than the cacheline size (64 bytes), the remote RNIC ensures the atomicity of RDMA_WRITE, ensuring consistency in all reads and writes to the target value. Even for values larger than the cacheline size, Dolphin ensures consistency of update operation because RDMA_WRITEs are performed in increasing address order. Consequently, the final target Value is always the expected value from the last update operation.

Insert-Insert conflicts: During insertion, the client selects an empty slot and uses RDMA_CAS to modify the empty Keys[x] from null to the target key. RDMA_CAS being atomic resolves Insert-Insert conflicts, where failed Insert operations retry inserting into a new empty Keys[y].

Update-Delete conflicts: The Delete operation sets both the target key and value to null in sequence, rendering the target data item invalid. If a concurrent Update operation considers the target Keys[x] still valid, it only sets the value field to the target value without modifying the Keys[x]. Therefore, Delete synchronizes with Update by manipulating the target Key[x].

SMO-IDU conflicts: SMO operations entail modifying the entire leaf node, while IDU operations only modify data items, rendering them exclusive. We implemented the Spear, a ternary-state node lock proposed by Marlin, to enforce exclusion between SMO and IDU operations. Spear essen-

tially functions as a read-write lock, with SMO operations resembling write operations and IDU operations resembling read operations. It synchronizes SMO and IDU operations and allows high-concurrency execution of IDU operations, enhancing the effectiveness of our MS-RNIC Write Synchronization mechanism. Unlike Marlin, to prevent erroneous modifications to Spear, we solely use RDMA_FAA operations to modify Spear. While SMO requires RDMA_WRITE to overwrite old leaf nodes, RDMA_WRITE does not encompass the memory area where Spear resides.

The resolution for Update-Update conflicts and Insert-Insert conflicts is illustrated in Figure 5. Update and Insert operations are performed under the protection of Spear without introducing additional RTT. As shown in Figure 5 (a), concurrent Updates achieve synchronization when modifying the target Value[x], and the synchronization process occurs in the RNIC of MS, resulting in minimal synchronization overhead. Concurrent Insert operations synchronize through RDMA_CAS operations on the target Keys[x]. Failed Insert operation will choose a new empty Keys[y] to retry the RDMA_CAS operation.

As previously outlined, if the size of Values[x] is smaller than the cacheline size, the read and write operations to Values[x] are atomic. However, when the size of Values[x] is larger than the cacheline size, read operations may retrieve an inconsistent state caused by concurrent write operations. Additionally, during concurrent SMO operations, read operations may access old leaf nodes, making them fail to locate the target data item.

WRITE-READ conflicts: If the size of the value is larger than the cacheline size, we append the cacheline version mechanism [24], embedding a version field in each cacheline. While reading the target value, the cacheline version field is verified for equality. If equal, it signifies a consistent read of the value.

READ-SMO conflicts: During SMO processes, the client allocates the target data item to the new leaf node potentially, concurrent read operations may access the old leaf nodes. Fortunately, leaf nodes of Dolphin store min and max keys in their first cacheline, allowing read operations to identify whether the target data item is in the current leaf node or not. These read operations then adopt a backtracking approach to locate the correct leaf nodes.

IV. Performance Evaluation

A. Experimental Setup

Testbed. We run all experiments on a cluster with 4 Compute Servers and 4 Memory Servers, and each server is equipped with 128 GB DRAM, a 100 Gbps Mellanox ConnectX-5 IB RNIC, and two 10-core Intel Xeon CPUs E5-2640. All RNICs are connected to a 100 Gbps Mellanox IB Switch. During the initialization, we register a large amount of RDMA Memory with huge pages in MSs in advance to avoid the performance impact of the memory registration process. All CSs run with 18 threads by default.

Workloads. We use YCSB [42] with both uniform and Zipfan distributions to evaluate the performance, which contains 5 workloads, including (1) YCSB A (50% read and 50% update), (2) YCSB C (100% read), (3) YCSB LOAD (100% insert), (4) range-only (100% scan accessing up to 100 items), (5) range-write (50% scan and 50% insert). Skewed workloads follow the Zipfian request distribution (θ = 0.99), which is commonly observed in production environments. Unless otherwise stated, we warm up the tree with 512 million items 50% full, except for the LOAD test. Apart from these workloads, we also evaluate the performance with cluster8 workloads (write-intensive) and cluster9 workloads (read-only) in Twitter cache trace [8]. For all experiments, we set the size of the leaf nodes is 1 KB, except for SMART. The default value size is 8 bytes, which is consistent with prior work [10], [12], [14], [16], [17].

Comparisons. We compare Dolphin with three state-of-the-art DM-based indexes. Specifically, Sherman [16] is a write-optimized B+-tree based on DM. Marlin [18] is the latest concurrency-optimized B+-tree based on DM. SMART [17] is the first DM-based ART, which reduces the read amplification and redundant I/Os. Sherman and SMART are open-sourced. And we get the source code from the author of Marlin. For a fair comparison, we allocated the same compute and memory resources for the four indexes in each experiment.

B. Index Cache Size Analysis

To investigate the adaptability of different indexes when the memory resources of CSs are insufficient, we allocated the size of the index cache ranging from 50 MB to 800 MB for different indexes. The efficacy of index cache remains consistent across all operations, hence, we utilize the YCSB A workload (50% read and 50% update) which can effectively demonstrate both read and write optimizations simultaneously. Marlin uses the same index cache scheme as Sherman and is primarily optimized for concurrent write operations. In our experiment, there is little performance difference between Marlin and Sherman, and we do not separately present the results for Marlin.

Specifically, as shown in Figure 6 (a), in the case of skewness=0.99 where data is highly skewed, a small amount of index cache can hit most of the hot internal nodes. Therefore, the throughput of all schemes rapidly increases as the index cache size grows from 0 to 200 MB. Beyond 200 MB, the throughput of Dolphin and Sherman plateaus since they have fully cached the internal nodes of the index. Ultimately, at an index cache size of 800 MB, the throughput of SMART continues to rise slowly because many internal nodes are still not loaded into the index cache. In this scenario, the throughput of Dolphin is 1.51×-1.32× higher than that of Sherman and SMART, respectively. This is because the Update of Dolphin only requires 2 RTTs, while other schemes require 3 RTTs. From another perspective, in Figure 6 (a), to achieve the same throughput, Dolphin requires less than 50 MB of index cache, while Sherman and SMART need 200 MB, which is 4 times that of Dolphin.

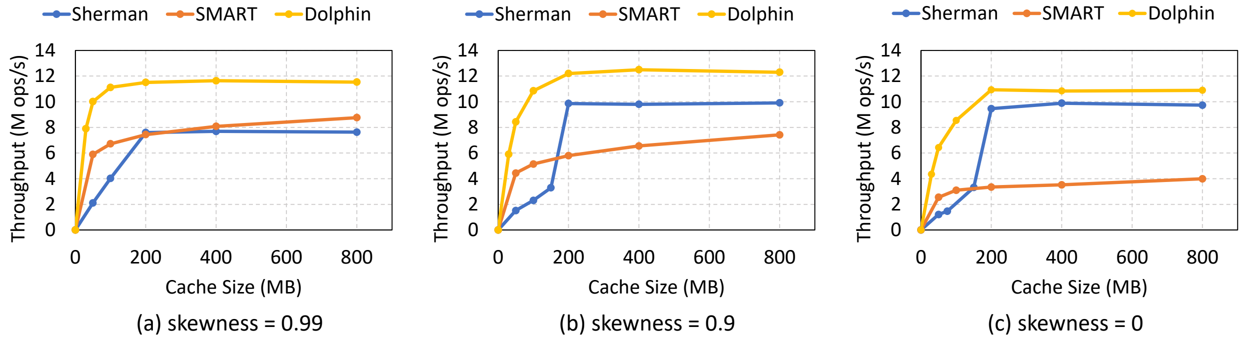

Fig. 6. The performance comparison of tree indexes on DM under different index cache sizes.

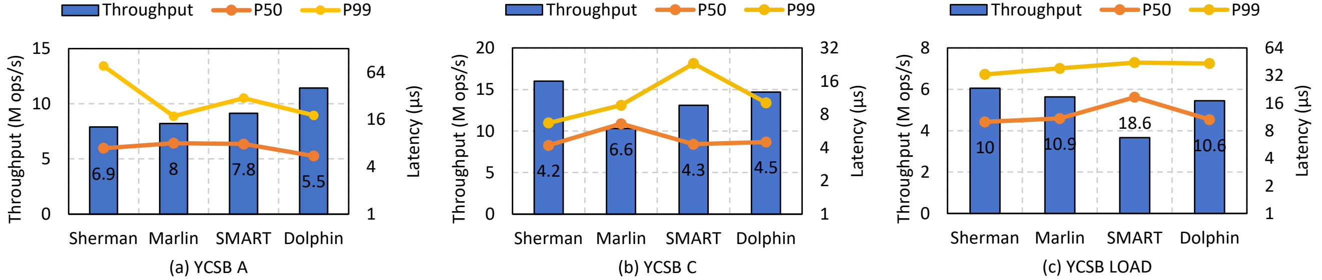

Fig. 7. The performance comparison of tree indexes on DM under YCSB workloads (index cache size = 2000 MB).

As shown in Figure 6 (b) and (c), the throughputs of Dolphin remain the highest. Additionally, the throughput of Sherman increases because the data becomes less skewed, reducing write-write conflicts. In Figure 6 (c), the throughput of SMART noticeably decreases compared to Figure 6 (a) because the workloads are uniform. The effectiveness of the index cache is directly determined by the number of cached objects (internal nodes). We calculate that caching all internal nodes of SMART requires at least 4600 MB of DRAM, which far exceeds our testing range.

Overall, at an index cache size of 200 MB, Dolphin's throughput is 1.16×-3.26× higher than other schemes. This indicates that Dolphin can save valuable cache space and achieve higher performance.

C. Overall Performance in YCSB

To comprehensively analyze the performance of different schemes in terms of update, read, and insert operations, we use three YCSB workloads and test throughput, P50 latency, and P99 latency. Considering that the index cache of SMART requires a large amount of DRAM, we set the index cache size for all tests to 2000 MB.

As shown in Figure 7 (a), Dolphin's throughput is 1.25×-1.45× higher than other schemes, with a P50 latency of 5.5 μs, which is 1.4-2.5 μs lower than other schemes. This is because Dolphin's Update operation only requires 2 RTTs, while other schemes require at least 3 RTTs. The P99 latency of Sherman is higher, attributed to its use of inefficient RDMA_CAS-based exclusive lock to resolve write-write conflicts. Marlin implements lock-free-like IDU operations, hence having lower P99 latency. Dolphin employs the MS-RNIC Write Synchronization mechanism to resolve concurrent conflicts of Update operations, achieving lower P99 latency as well.

The performance of read operations (YCSB C) is shown in Figure 7 (b), where Sherman and Dolphin exhibit the best performance as they both require only one RTT to fetch the target data items from leaf nodes. The P50 latency of Marlin is 2.4 μs higher than Sherman and Dolphin. After obtaining the target leaf node, Marlin needs to read the data items from MS based on the data item pointers in the leaf node. Additionally, the P99 latency of SMART is around 10 μs higher than other schemes, as even with an index cache size of 2000 MB, there are still internal nodes not cached, necessitating remote traversal of the index.

When a storage system initializes, a large amount of data needs to be loaded into the index. As shown in Figure 7 (c), Sherman, Marlin, and Dolphin have similar performances because their insert operations all need 3 RTTs. However, the throughput of SMART is approximately 1.54× lower than other schemes, as each insert operation in SMART inserts a new leaf node pointer into the internal node, causing the cache invalidation of that internal node in CSs. The clients in CSs need to repeatedly evict invalidated internal nodes from the index cache and load the valid internal nodes into the index cache. This results in a P50 latency for SMART that is 8 μs higher than other schemes, with a noticeable decrease in throughput.

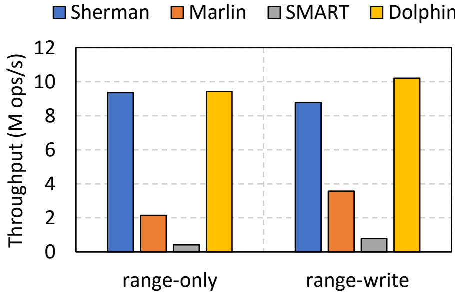

Fig. 8. Performance of Scan operation.

D. Range Query Performance

In this experiment, we use uniform workloads and conducted experiments for range-only and range-write operations with a range size of 100. The index cache size is set to 2000 MB. As shown in Figure 8, in the range-only experiment, the throughputs of Dolphin and Sherman are close and higher than that of Marlin and SMART by 4.38×-22.4×. This is because the B+-tree leaf nodes of Sherman and Dolphin provide spatial locality for adjacent data items, allowing a client in the CS to use a single RDMA_READ to read dozens of data items. However, Marlin and SMART store individual data items separately, requiring one RDMA_READ per data item. When scanning 100 data items, Sherman and Dolphin may only need 2-3 RDMA_READs, while Marlin and SMART need at least 100 RDMA_READs. Furthermore, Marlin's performance is 5.11× higher than that of SMART. Marlin can retrieve a target leaf node (containing the addresses of dozens of target data items) through one index cache lookup and one RDMA_READ, then use dozens of RDMA_READs to read the target data items. For SMART, each leaf node has a different path, so obtaining 100 data items requires 100 index cache lookups and 100 RDMA_READs. Therefore, SMART requires more index lookup operations compared to Marlin. We obtained that the time for an additional index cache lookup is approximately 1-2 μs. Thus, besides lacking spatial locality, SMART's unique path traversal in ART also results in poor scan performance.

Additionally, in the range-write test, the performance of Marlin, SMART, and Dolphin slightly increases. Only Sherman slightly decreases due to its inefficient update operation.

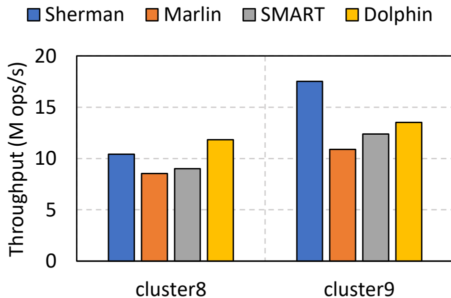

Fig. 9. Performance under real-world workloads.

E. Performance Under Real-world Workloads

In addition to synthetic workloads, we also utilized real-world workloads [8], which describe the traces from Twitter's in-memory caching clusters. Cluster8 workloads represent write-intensive workloads, while cluster9 represents read-only workloads. As shown in Figure 9, in the cluster8 experiment, Dolphin exhibits optimal performance, which can be attributed to our efficient pathfinding algorithm and the MS-RNIC Write Synchronization mechanism. In the cluster9 experiment, Sherman achieves slightly higher throughput, possibly influenced by the distribution of keys in the workloads.

TABLE II
THE DRAM CONSUMPTION ON ONE CS AND ALL MSS.

	CS (MB)	MS (GB)
Sherman	460	12
Marlin	460	26
SMART	5200	156
Dolphin	320	14

F. Memory Consumption Analysis

To analyze the memory overhead in CSs in detail, we measured the index cache sizes required by different schemes to achieve optimal performance. We load 512 million data items into the indexes and then conduct the statistics. As shown in Table II, Dolphin requires a 1.44×-16.25× smaller index cache compared to other schemes. This indicates that Dolphin can significantly save valuable DRAM in CSs. Moreover, we measured the total DRAM size needed to store complete indexes in MSs. As shown in Table II, We find that SMART requires at least 156 GB of DRAM, possibly due to issues with memory management. The memory consumption of Marlin is about 2× higher than Sherman and Dolphin. In addition, Dolphin has a slightly higher memory consumption than Sherman in MSs. However, we believe saving precious DRAM in CSs is more crucial.

V. RELATED WORK

Disaggregated Memory. The DM architecture is widely discussed recently [20], [21], [24]–[26], [29], [34], [35]. Existing schemes provide solutions for DM from various perspectives, including operating system [30]–[32], hardware architecture [20]–[22], [26], [27], [36], [37], database [23], [33], [38], [39], distributed lock [40], and transaction [29], etc. Legoos [32] mainly focus on efficient research in resource management based on DM, providing a foundation for runtimes running on DM. Hardware-related research, such as Clio [36], explores how hardware components like SmartNIC, programmable switch, and CXL can be organized and applied in DM-based systems. Polardb [38] practices DM in commercial databases. FUSEE [28] designs a fully DM-based key-value store that introduces disaggregation to metadata management. Ford [29] proposes an efficient DM-based transaction system. ROLEX [41] proposes a scalable RDMA-based learned key-value store that separates model retraining from data modification operations. Citron [40] provides distributed range

locks suitable for DM, supporting range query operations in databases. Dolphin focuses on the DM-based index that is orthogonal to these studies.

RDMA-based Tree Indexes. With the popularity of RDMA in data centers, there are increasing studies focusing on RDMA-based tree indexes [12], [13], [15]–[18]. Some of them utilize RDMA-based remote procedure calls (RPCs) for communication between servers. However, these indexes are not suitable for DM since the MSs lack sufficient CPU resources to provide RPC services. Sherman [16] employs local locks to reduce unnecessary retries of exclusive locks. Marlin [18] uses customized read-write locks to address concurrency conflicts between SMO and IDU, enhancing the concurrency of IDU operations. SMART [17] leverages the advantage of ART where leaf nodes contain only the target data items, solving the read amplification problem. SMART also introduces software techniques such as write-combination and read-delegation to improve peak throughput. In contrast, Dolphin addresses the challenge of balancing memory overhead, read performance, and write performance while utilizing fewer memory resources to achieve higher throughput.

VI. Conclusion

This paper systematically analyzes the strengths and weaknesses of existing DM-based indexes and summarizes the performance bottlenecks they encounter in terms of memory overhead, read performance, and write performance. In response, we propose Dolphin, a resource-efficient DM-based hybrid index. Dolphin achieves a well-balanced trade-off between read and write performance, introducing a hybrid index structure with ART internal nodes and B+-tree leaf nodes to reduce memory overhead in CSs and MSs. We also design efficient pathfinding algorithms and low-overhead MS-RNIC Write Synchronization mechanism, enabling efficient READ/WRITE/Scan operations. Experimental results show that, under synthetic and real-world workloads, Dolphin reduces memory overhead by 1.44×-16.25× and improves throughput by 1.16×-3.26× compared to the state-of-the-art DM-based indexes.

Acknowledgement

This work was supported by NSFC (No. U22A2027, 61832020 and 61821003), Project of Shenzhen Technology Scheme (JCYJ20210324141601005). We are grateful to our anonymous reviewers for their constructive comments and suggestions.

References

[1] CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-ziron-out-a-coherent-interconnect-strategy/, 2020.

[2] M. Tirmazi, A. Barker, N. Deng, Md E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter and J. Wilkes, "Borg: the next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, New York, NY, USA, 2020, pp. 1–14.

[3] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*, New York, NY, USA, 2023, pp. 574–587.

[4] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, Phoenix, AZ, USA, 2019, pp. 1–10.

[5] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki and M. Callaghan, "Designing Access Methods: The RUM Conjecture," in *International Conference on Extending Database Technology (EDBT 2016)*, 2016, pp. 461–466.

[6] C. Anneser, A. Kipf, H. Zhang, T. Neumann and A. Kemper, "Adaptive Hybrid Indexes," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*, New York, NY, USA, 2022, pp. 1626–1639.

[7] H. Zhang, G. Chen, B. C. Ooi, K. -L. Tan and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," in *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2015, pp. 1920–1948.

[8] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 191-208.

[9] V. Leis, A. Kemper and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, Brisbane, QLD, Australia, 2013, pp. 38-49.

[10] A. Kalia, M. Kaminsky and D. G. Andersen, "Datacenter RPCs can be general and fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019)*, Boston, MA, February 26-28, 2019, pp. 1–16.

[11] WH. Kim, R. M. Krishnan, X. Fu, S. Kashyap and C. Min, "PACTree: A High Performance Persistent Range Index Using PAC Guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, New York, NY, USA, 2021, pp. 424–439.

[12] C. Mitchell, K. Montgomery, L. Nelson, S. Sen and J. Li, "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, pp. 451–464.

[13] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan and M. Castro, "Fast General Distributed Transactions with Opacity," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*, New York, NY, USA, 2019, pp. 433–448.

[14] X. Zou, W. Fang, D. Fen, J. Chen, C. Liu, F. Li and N. Su, "HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory," in *Proceedings of the 36rd International Conference on Massive Storage Systems and Technology, MSST*, 2020.

[15] T. Ziegler, S. T. Vani, C. Binnig, R. Fonseca and T. Kraska, "Designing distributed tree-based index structures for fast rdma-capable networks," in *Proceedings of the 2019 International Conference on Management of Data*, Amsterdam, Netherlands, 2019, pp. 741–758.

[16] Q. Wang, Y. Lu and J. Shu, "Sherman: A write-optimized distributed b+ tree index on disaggregated memory," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, Philadelphia, PA, USA, 2022, pp. 1033–1048.

[17] X. Luo, P. Zuo, J. Shen, J. Gu, X. Wang, M. R. Lyu and Y. Zhou, "SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory," in *the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023)*, July 10-12, 2023, pp. 553–571.

[18] H. An, F. Wang, D. Feng, X. Zou, Z. Liu and J. Zhang, "Marlin: A Concurrent and Write-Optimized B+-tree Index on Disaggregated Memory," in *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23)*, New York, NY, USA, pp. 695–704.

[19] P. Zuo, J. Sun, L. Yang, S. Zhang and Y. Hua, "One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 15-29.

[20] Eric Hooper. Intel rack scale design: Just what is it? https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it, 2018.

[21] HP Labs. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine, 2014.

[22] S. -S. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong and A. Bhattacharjee, "MIND: In-network memory management for disaggregated data centers," in *28th Symposium on Operating Systems Principles*, October 26-29, 2021, pp. 488–504.

[23] J. Wang and Q. Zhang, "Disaggregated Database Systems," in *Companion of the 2023 International Conference on Management of Data (SIGMOD '23)*, New York, NY, USA, 2023, pp. 37–44.

[24] T. Ziegler, J. Nelson-Slivon, V. Leis and C. Binnig, "Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA," in *Proceedings of the ACM on Management of Data*, 2023, pp. 1–26.

[25] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[26] S. Grant and A. C. Snoeren, "In-network Contention Resolution for Disaggregated Memory," in *Proceedings of the Workshop on Resource Disaggregation and Serverless (WORDS)*, 2021.

[27] Y. Shan, W. Lin, Z. Guo and Y. Zhang, "Towards a fully disaggregated and programmable data center," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022, pp. 18–28.

[28] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou and M. R. Lyu, "FUSEE: A Fully Memory-Disaggregated Key-Value Store," in *21th USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 81-98.

[29] M. Zhang, Y. Hua, P. Zuo and L. Liu, "FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 51–68.

[30] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury and K. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[31] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian and M. Wei, "Remote regions: a simple abstraction for remote memory," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 775–787.

[32] Y. Shan, Y. Huang, Y. Chen and Y. Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 69–87.

[33] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu and B. T. Loo, "Understanding the effect of data center resource disaggregation on production dbmss," in *Proceedings of the VLDB Endowment*, 2020, pp. 315–344.

[34] S. -Y. Tsai, Y. Shan and Y Zhang, "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 33–48.

[35] Z. Ruan, M. Schwarzkopf, M. K. Aguilera and A. Belay, "AIFM: High-Performance, Application-Integrated Far Memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 315–332.

[36] Z. Guo, Y. Shan, X. Luo, Y. Huang and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 417–433.

[37] D. Gouk, S. Lee, M. Kwon and M. Jung, "Direct access, high-performance memory disaggregation with directcxl," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[38] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang and J. Tong, "Polardb serverless: A cloud native database for disaggregated data centers," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2477–2489.

[39] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. Milojičić and G. Alonso, "Farview: Disaggregated memory with operator off-loading for database engines," in *Proceedings of Conference on Innovative Data Systems Research*, 2022.

[40] J. Gao, Y. Lu, M. Xie, Q. Wang and J. Shu, "CITRON: distributed range lock management with one-sided RDMA," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, USA, 2023, pp. 297–314.

[41] P. Li, Y. Hua, P. Zuo, Z. Chen and J. Sheng, "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, pp. 99–114.

[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. Indianapolis, Indiana, USA, 2010, pp. 143–154.