applied sciences

MDPI

Article

Converting Concurrent Range Index Structure to Range Index Structure for Disaggregated Memory

Bonmoo Koo [1], Jaesang Hwang [1], Jonghyeok Park [2] and Wook-Hee Kim [1,*]

[1] Department of Computer Science and Engineering, Konkuk University, Seoul 05029, Republic of Korea
[2] Division of Computer Engineering, Hankuk University of Foreign Studies, Yongin 17035, Republic of Korea;
* Correspondence: wookhee@konkuk.ac.kr

Abstract: In this work, we propose the SPREAD approach, which tailors a concurrent range index structure to a range index structure for disaggregated memory connected via RDMA (Remote Direct Memory Access). The SPREAD approach leverages the concept of tolerating transient inconsistencies in a concurrent range index structure to reduce the amount of expensive RDMA operations. Based on the SPREAD approach, we converted B^{link}-tree, a concurrent range index structure, to a range index structure for disaggregated memory called RF-TREE. In our experimental study, RF-TREE shows comparable performance to Sherman, a state-of-the-art and carefully crafted range index structure for disaggregated memory.

Keywords: disaggregated memory; distributed memory; concurrency; scalability; index structures; RDMA

Citation: Koo, B.; Hwang, J.; Park, J.; Kim, W.-H. Converting Concurrent Range Index Structure to Range Index Structure for Disaggregated Memory. *Appl. Sci.* **2023**, *13*, 11130. https://doi.org/10.3390/app132011130

Academic Editor: Andrea Prati

Received: 21 August 2023
Revised: 2 October 2023
Accepted: 9 October 2023
Published: 10 October 2023

1. Introduction

In the memory-driven computing era, the demand for memory is increasing as memory provides high performance to the software. However, the cost of memory is still high, and the memory utilization of servers in the data center is still low.

Disaggregated memory is rising as a promising solution to fulfill the increasing demand for memory. Disaggregated memory separates computing and memory in data centers, and it manages the resources as connected computing and memory pools, respectively. The primary virtue of separating computing and memory pools lies in that it enables elastic resource management and reduces the TCO (Total Cost of Ownership) [1].

In-memory storage systems, such as in-memory database systems or an in-memory cache, can be major use cases of disaggregated memory as they require low-latency access to a large amount of physical memory.

However, simply exploiting the disaggregated memory cannot fully utilize the architecture's potential. To fully take advantage of disaggregated memory, the index structures, which are the primary data structures of in-memory storage systems, should be changed by considering the characteristics of disaggregated memory. For instance, disaggregated memory requires a high-performance interconnect network to provide low latency to support the memory pool. Most previous research for disaggregated memory [2–4] depends on RDMA (*Remote Direct Memory Access*) as RDMA provides high-performance and direct access to the memory in the remote server.

To directly access the remote memory using RDMA, RDMA one-sided verbs such as RDMA Read/Write/Atomic operations are used because they are memory semantics. RDMA one-sided verbs provide high performance by allowing a memory operation to be executed without the remote host's active involvement. Although RDMA one-sided verbs provide high performance, they still incur high overhead. This is because the latencies of RDMA one-sided verbs are relatively higher than local memory accesses. Previous studies [3,5] mainly exploit simple hash table-based storage systems because it is challenging to support complicated index operations using RDMA one-sided verbs.

There have been a few research efforts for designing range index structures that leverage RDMA one-sided verbs [2,6]. They explore the design of range index structures on disaggregated memory. The first range index structure to use RDMA one-sided verbs is called FG [6], a distributed B^{link}-tree that distributes the node among the memory servers in a fine-grained manner. However, it suffers a high overhead from the frequent RDMA one-sided verb calls. Sherman [2] is a state-of-the-art range index structure that uses RDMA one-sided verbs. Sherman leverages the compute server's local DRAM as a local cache. It also exploits the on-chip memory of NIC to reduce the overhead of atomic operations in the RDMA's one-sided verbs. However, these engineering efforts require a deep understanding of RDMA.

While recent studies require a steep learning curve for RDMA, we revisited their approaches and observed that they leverage RDMA write operation (i.e., one-sided verb), providing a strong ordering guarantee [2,7]. This strong ordering guarantee shows similar characteristics to the ordering guarantee of the memory operation in recent concurrent range index structures [8]. In recent concurrent range index structures, the readers can tolerate inconsistent transient states during index update operations by other writers. These inconsistent transient states are called *Transient Inconsistency*. Moreover, other recent studies [9,10] show that separating the search layer (the internal nodes) and the data layer (the data nodes) in a concurrent range index structure can improve scalability and performance. This structure also can take advantage of disaggregated memory, which is an emerging architecture in a distributed memory system.

Hence, in this work, we propose a new approach, called SPREAD, to convert a concurrent range index structure into a range index structure for disaggregated memory. Using the SPREAD approach, we implement RF-TREE, a range index structure for disaggregated memory, by converting B^{link}-tree, which is a concurrent range index structure.

- We develop a new approach called SPREAD, which converts a concurrent range index structure into a range index structure for disaggregated memory. The SPREAD approach exploits the concept of tolerating transient inconsistency in a concurrent range index structure.
- Following the SPREAD approach, we implement RF-TREE. We show that the existing concurrent range index structure can be converted for disaggregated memory without a deep understanding of disaggregated memory architecture.

The rest of the paper is organized as follows:

In Section 2, we explain the background of our work. Section 3 introduces the SPREAD approach and shows a case study with RF-TREE, a range index structure for disaggregated memory. Section 4 shows our experimental evaluation results, and Section 5 explains related works. In Section 6, we conclude the paper.

2. Background

2.1. Disaggregated Memory

Disaggregated memory [1] was proposed over the past decades to improve the efficiency of TCO (Total Cost of Ownership) by sharing idle resources in a server with other servers by disaggregating the computing and the memory resources.

Since there is currently no commercially available hardware for disaggregated memory, most previous research exploits environments where servers are connected via high-performance interconnect networks to emulate disaggregated memory. Disaggregated memory requires direct memory access to the remote memory in the remote servers. Currently, disaggregated memory leverages RDMA one-sided verbs [2–4,6], so we focus on exploiting RDMA one-sided verbs in disaggregated memory. Figure 1 presents the disaggregated memory that uses RDMA one-sided verbs. The server that uses the memory spaces and computes the data is called the *compute server*, and the server that provides the memory spaces is called the *memory server*. Compute servers typically have more computing powers (i.e., more powerful CPU cores), and memory servers typically have fewer computing powers but are equipped with more capacity.

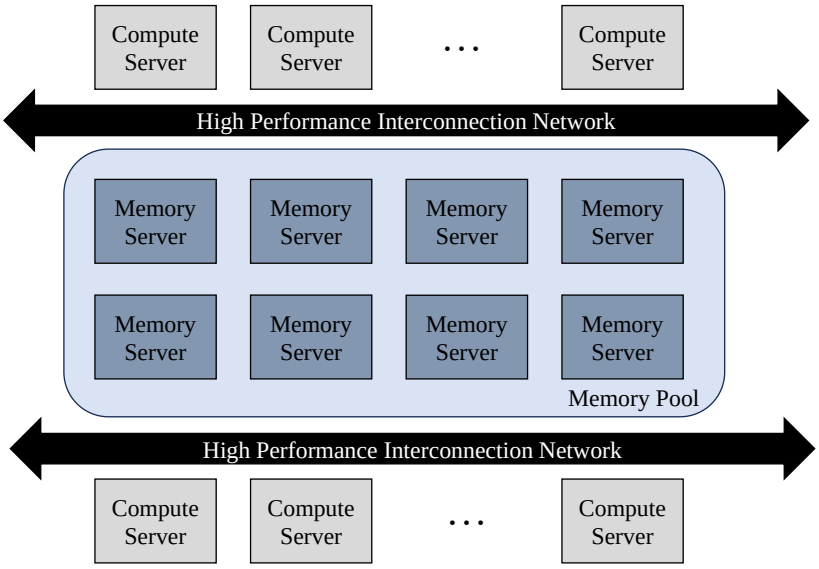

Figure 1. Overview of disaggregated memory.

2.2. RDMA

RDMA (*Remote Direct Memory Access*) provides high throughput with low latency. RDMA has two types of verbs: one-sided verbs and two-sided verbs. RDMA one-sided verbs are memory semantics and consist of RDMA Read, RDMA Write, and RDMA Atomic operations. RDMA two-sided verbs are composed of RDMA Send and RDMA Recieve, similar to the typical socket programming. The main difference between RDMA one-sided verbs and two-sided verbs is the involvement of the remote host. RDMA one-sided verbs do not involve the remote host, but the results of the RDMA two-sided verbs are posted by the remote host.

2.3. Range Index Structures on Disaggregated Memory

In this section, we explain the works most related to our paper, which exploits RDMA one-sided verbs for a range index structure on disaggregated memory.

FG [6] is the first B^{link}-tree to exploit only one-sided verbs and RDMA Atomic operations such as RDMA *Fetch-And-Add* and RDMA *Compare-And-Swap* for distributed B^{link}-tree operations. Since the one-sided verbs do not involve the remote host in disaggregated memory, the compute server performs the distributed B+-tree operation using these one-sided verbs. Even though RDMA one-sided verbs show low latency, the latencies of RDMA one-sided verbs are still much higher than memory accesses. Hence, FG leverages a prefetching technique to reduce the latency.

Sherman [2] is a B+-tree for disaggregated memory. Sherman analyzes the reason for slow one-sided verbs and adds engineering efforts to leverage hardware features such as on-chip memory for mitigating locking overhead. Sherman also actively exploits the compute server's local resources. For instance, it uses the compute server's local memory as a cache for the index structure to reduce the number of network round-trips. Moreover, the local lock table reduces the number of retries caused by contentions between clients located on the same server.

3. Method

Designing index structures for disaggregated memory entails both a time-consuming and labor-intensive process because the range indexes accompany complicated SMO (Structure Modification Operations). Moreover, RDMA hardware features are complicated and require deep knowledge. In this section, we present our SPREAD approach to convert a concurrent range index structure to a range index structure for disaggregated memory.

3.1. Search Layer and Data Layer in Range Index Structure

General range index structures are composed of the *search layer* and *data layer*, as shown in Figure 2. The search layer consists of internal nodes and provides a shortcut to the target node in the data node in the data layer. Thus, the nodes in the search layer do not include the data, and they only store the key and pointer to the child node. The data layer consists of data nodes (i.e., leaf nodes) in the range indexes. In the data node, the data are composed of key and value pairs where the value is typically the address of the data.

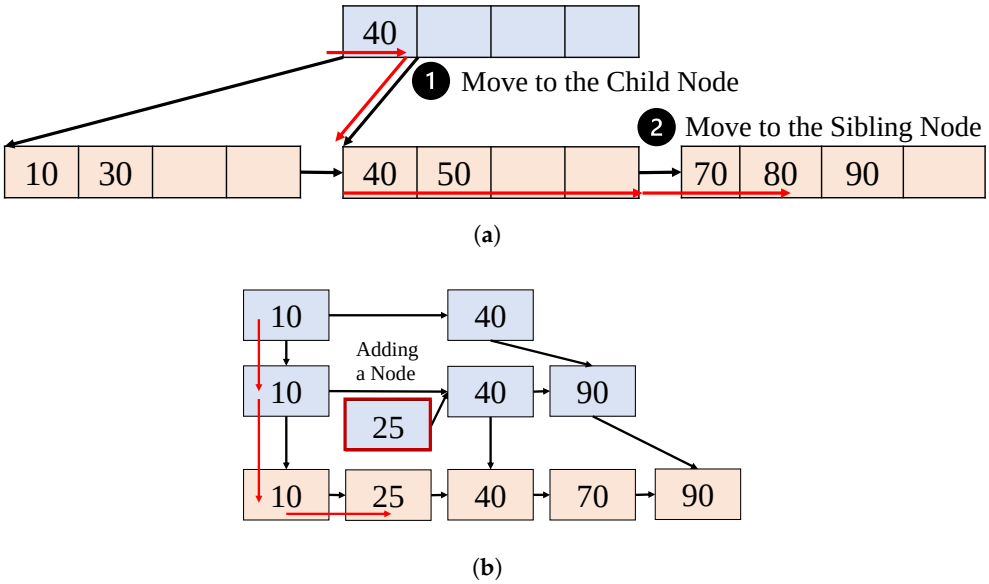

Figure 2. Tolerating Transient inconsistency in the concurrent range index structures. (**a**) B^{link}-tree: Example of lookup operation of key 80. When the node split occurs, the reader can tolerate the transient inconsistency by traversing the sibling node. (**b**) SkipList: Example of lookup operation of key 25. The reader can tolerate transient inconsistency by traversing the lower level when the key is added to the upper level.

There have been a lot of efforts to improve the concurrency level of range indexes in prior studies. B^{link}-tree [11] is the most widely known range index, improving the concurrency level by tolerating the transient inconsistencies of the range indexes. SkipList [12] is another index structure that can tolerate transient inconsistencies because of the insert and delete operation of internal nodes. Recently proposed hybrid range indexes [9,10] improve the performance and scalability by tolerating transient inconsistencies between the search layer and the data layer. We exploit the concept of tolerating transient inconsistencies between the search layer and the data layer. In the range index structure for disaggregated memory, the client always needs to read the data node from the memory server to retrieve the up-to-date version. However, the search layer does not need to be an up-to-date version because it only provides the route to the data node. Instead, the client can access the target data node by traversing the data layer using the sibling pointer of the data node.

3.2. Converting Concurrent Range Index Structure for Disaggregated Memory

Log-Structured Search Layer Region. In order to make the search layer RDMA-friendly, the search layer should be accessed with minimal RDMA Read operations. To do this, the internal nodes in the search layer are in consecutive memory space. In the memory server, the memory space is divided into two regions: the *search layer region* and the *data layer region*. The internal nodes are stored in the search layer region. The data node (i.e., leaf node) is stored in the data layer region. Storing the search layer in consecutive memory space is simple and straightforward. Basically, the internal nodes are allocated sequentially in the search layer region, similar to that of log-structured memory management [13,14].

Load Balancing. The disaggregated memory consists of memory servers connected via RDMA one-sided verbs. Hence, if the loads between the memory servers are unbalanced, the memory servers could become the performance bottleneck. As a result, the data should be distributed among the memory servers uniformly. However, as shown in a previous study [6], fine-grained node distribution can incur additional RDMA Read operations, which cancels out the benefit of the balanced load. Fortunately, the log-structure search layer region helps reduce the number of RDMA Read operations even when the nodes are distributed in a fine-grained manner. This is because the log-structured design enables reading the search layer as a large-sized chunk. Thus, in our SPREAD approach, each node is stored in the memory server randomly.

Higher Consistency Guarantee. Since the disaggregated memory is the remote memory, there can be a high possibility of transient inconsistencies between servers and clients. Previous studies [7,15] exploit a characteristic of *RDMA Writes*, which transfer data in the order of memory address.

However, a recent study [16] shows that the *RDMA Read* operation can still suffer from transient inconsistencies due to the concurrent multiple cache line retrievals. Hence, an additional consistency-guarantee mechanism is still required. There are several existing approaches, such as checksum [3,4,17,18] and versioning for each cache line [6,19]. These consistency-guarantee mechanisms improve consistency but incur additional overheads, such as computation and memory accesses. In our SPREAD approach, we add a CRC (Cyclic Redundancy Check) value as a checksum for each node to guarantee the consistency of the node.

Efficient Usage of Local Cache. As RDMA-based disaggregated memory still suffers from the high performance overhead from RDMA's relatively high latency, it is important to reduce the number of RDMA operations. In that sense, leveraging the compute server's local memory cache is a fundamental solution. Previous works [2,6] already have exploited the local cache. In a compute server, there can be multiple clients based on threads. These threads share the same local cache in the compute server, so the cached range index structure should support concurrent access. Originally, the concurrent range index structures were designed for multi-core scalability in the same machine. Thus, we can leverage the concurrency control of the range index structure.

Concurrency Control. The concurrency control protocol exploits atomic instructions to guarantee mutual exclusion in the critical section. The index structure can simply replace the atomic instructions, such as *Compare_And_Swap* (CAS) and *Fetch_And_Add* (FAA), with RDMA atomic opertions (i.e., RDMA CAS or RDMA FAA) [20]. However, RDMA atomic operations are more expensive than CPU atomic instructions. Thus, we must employ RDMA atomic operations while carefully avoiding excessive calling.

Pointer Representation. Since disaggregated memory consists of the memory from multiple servers, there is a need to add server information to the pointer representation. In order to distinguish the memory region of each memory server, the SPREAD approach embeds the *memory region ID*, which indicates the memory server in the address space. Note that current x86 systems only use six bytes, so we can use two bytes for storing memory region ID, similar to previous studies [2,10]. Moreover, the SPREAD approach uses offset information with a base address. The offset information of each object is stored in a six-byte space instead of the virtual address.

3.3. Case Study: RF-TREE

To demonstrate the benefit of the SPREAD approach, we present RF-TREE, an index structure for disaggregated memory, as shown in Figure 3. We judicially leverage B^{link}-Tree, which uses the Atomic Shift operation [8] to tolerate the transient inconsistencies.

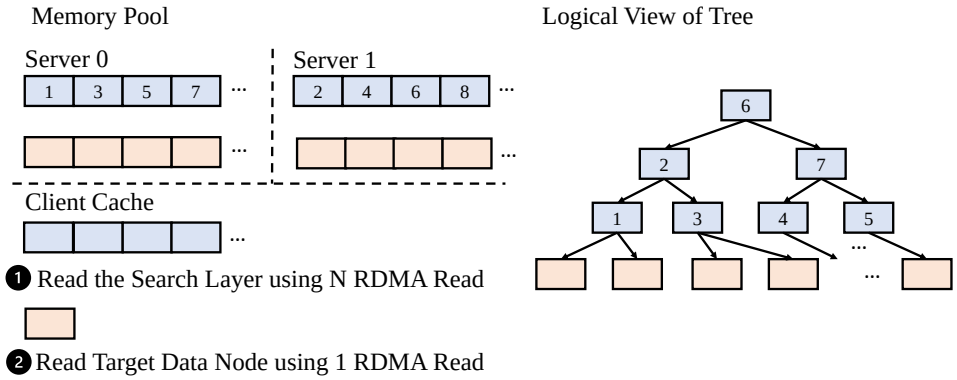

Figure 3. Overview of RF-TREE. When the SLC is empty, the client reads the search layer from the memory servers. When the search layer is cached and built, the client finds the address of the target data node. Using the address, the client reads the data node. In the figure, N is the number of the server.

3.3.1. Structure of RF-TREE

RDMA-Friendly Search Layer. In RF-TREE, we make the search layer RDMA-friendly by allocating the search layer in a log-structured manner in the memory server. Thus, RF-TREE can read multiple internal nodes using one RDMA Read operation. Moreover, RF-TREE divides the search layer into multiple chunks and distributes the chunks to multiple memory servers to balance the load of each memory server. Each internal node is appended to the end of memory space or added to the free page in the selected memory server. RF-TREE manages the free page using a linked-list-based *free list* (Section 3.3.2). Each client checks the free list, and, if there is any free page, then the client removes the free page from the free list and uses it.

Uniformly Distributed Data Nodes. The data layer is composed of data nodes (i.e., leaf nodes). The data node keeps the key and value pair. Since the data node and the target data will be accessed together, our approach stores the data node and the data in the same memory server to reduce the communication overhead. The memory server will be selected in a round-robin manner to uniformly distribute the data and data nodes.

3.3.2. Memory Server

Memory Management. RF-TREE leverages log-structured memory management to efficiently add and delete the node to the memory server's memory space. RF-TREE preallocates the memory space in the memory space in the memory server. The client updates the last index of the page in the memory server atomically when a new node is added to the memory space in the memory server.

The compute server manages its own local free page list called the *free list*. When a page is free, a client marks the page as free and adds the free page to the free list. Since the free list is located on the compute server's local memory, the free page information can be lost when the compute server is aborted. In that case, another compute server scans the memory spaces in the memory servers and reconstructs the free list by adding the free pages from the result of the scan operation.

Concurrency Control. As noted in the previous study [16], the version-based locking protocol with one RDMA Read has inconsistencies because it can prevent inconsistencies from the multiple concurrent cache line retrievals. Hence, RF-TREE exploits a version-based locking protocol with a checksum protocol to guarantee the consistency of the data structure. RF-TREE only acquires a write lock prior to updating both internal and data nodes since it allows a non-blocking read operation. We will discuss the Read operation without locking in the following section (Section 3.3.3).

3.3.3. Read Operation

The Read operation in the range index structure is important as every index operation should traverse the tree and do its own work. The Read operation is composed of a *Non-Blocking Read* operation in the search layer and an optimistic Read operation in the data layer.

Search Layer Cache in the Compute Server. Since the one-sided verbs have relatively higher latency than local memory access, exploiting the local memory cache in the compute server is essential. In the compute server, a client reads the search layer from the memory servers and reconstructs the search layer in the *Search Layer Cache* (SLC). The search layer is shared with multiple threads in the same server. RF-TREE uses the version-based concurrency control mechanism, similar to the previous studies [4,9,10].

Non-Blocking Read in the Search Layer. RF-TREE provides a Non-Blocking Read in the search layer to provide high performance. To find the target data node, the client should traverse the search layer. The client reads the search layer from the memory servers using the RDMA Read operation when the compute server's SLC is empty. Since the client copies the search layer without blocking, some internal nodes experience an inconsistent state. To tolerate the inconsistent states of some internal nodes, RF-TREE leverages the x86 atomic primitives that can guarantee that 8-byte write is atomic [8,10,21,22]. Moreover, RDMA Write guarantees the order that follows the order of the memory address. RF-TREE's Read operation considers the RDMA Write operation's ordering to tolerate the transient inconsistencies, as shown in the following cases.

- *Case 1: Inconsistent Search Layer.* When the internal nodes are inconsistent, the internal nodes may shift the key-value pairs. For the Write operation, RF-TREE shifts the pointer first and then the key. Figure 4 shows an example of an inconsistent search layer. When the client finds a proper key, it first checks whether the pointer value of the key is duplicated. If it is, then it returns the pointer value of the node to the right before the key. If it is not, it returns the pointer of the key, similar to the previous study [8]. The child node may have lower keys and does not have the target key. However, the client tolerates the transient inconsistencies by traversing the child nodes.
- *Case 2: Stale SLC.* Since the search layer is stored in the memory server and cached in the local cache of the compute server, the SLC may be stale. RF-TREE can endure the stale SLC by traversing the data layer using the *sibling pointer* of the data node, as shown in Figure 5. However, traversing the data layer incurs additional expensive *RDMA Read* operations. Thus, RF-TREE updates the SLC when the client traverses more than the threshold. The SLC update is performed using the COW (*Copy-On-Write*). The client reads the SLC from the memory server to another memory space and then updates the pointer of the SLC to the new one.

Read in the Data Node. RF-TREE reads the data node from the memory server using an optimistic locking protocol. Since there can be an inconsistent state in the leaf node, RF-TREE checks the checksum after reading the data node. When the checksum value is not the same, RF-TREE reads the data node from the remote memory server again. Note that the data nodes are not cached in the compute server's SLC. RF-TREE always reads the data node from the memory server and stores it in the thread's local buffer. This is because the data node keeps the pointer to the data, so RF-TREE needs a consistent and latest version of the data node.

Range Query. In the range query operation, the client traverses the search layer in the SLC and finds the pointers of data nodes, which have the keys to the result of the given range query. Note that the range query of RF-TREE has low-level isolation as RF-TREE does not keep the locks for the data nodes for the range query.

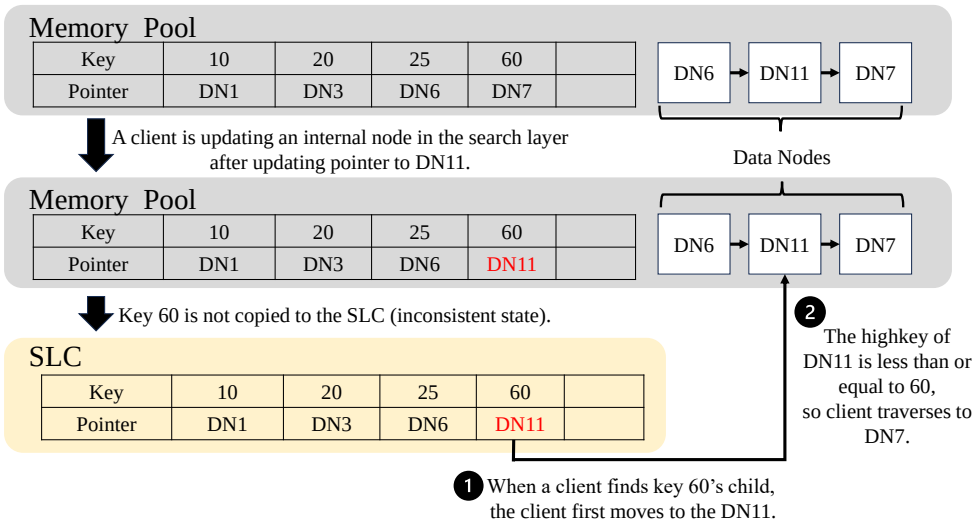

Figure 4. An example of an inconsistent search layer. When RF-TREE updates a node, RF-TREE writes a pointer (value) and then writes the key. ❶ Since DN11 has smaller keys than DN7, ❷ the client moves to the DN7.

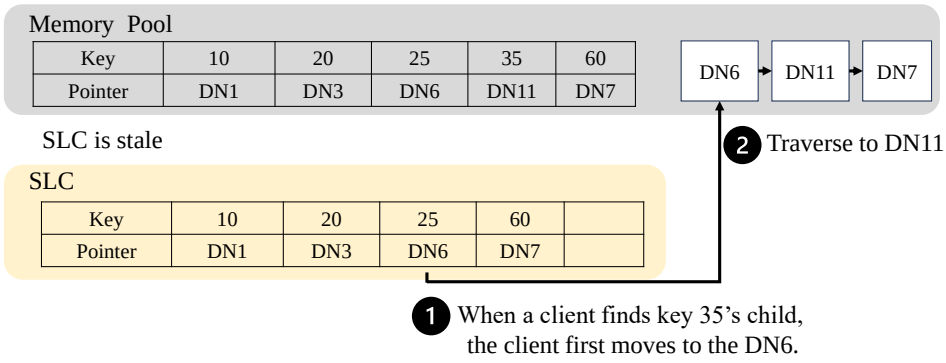

Figure 5. An example of a stale SLC. ❶ A client moves to the child node when it finds a key that is larger than the target key. ❷ The client traverses the child nodes to find the target key.

3.3.4. Write Operation

Insert/Update/Delete. In common Write operations, such as the Insert/Update/Delete operation, the client first finds the target node's address by traversing the search layer in the compute server's SLC. The client reads the target node from the memory server using the target node's address. Next, the client checks the checksum value to guarantee the consistency of the data node. The client performs the Write operation after acquiring the write lock. When the data node is successfully updated, the client writes the unlocked data node to the memory server. Note that the lock of the data node is located on the last 8-byte of the data node.

Split. When the node does not have enough space to store a given key and value pair, the client conducts a split operation of the node. The split operation is also a part of the insert operation, so the overflown data node is already locked. The client allocates a new node and copies half of the key-value pairs to the new node. The client finds the free node from the local *free list* and uses it. The node is newly allocated when the free list is empty. After copying the data and connecting the new node, the client atomically updates the variable

for the number of data to half. The client writes the nodes and inserts the parent entry to the parent node of the overflown node.

Merge. When the utilization of the node is too low, RF-TREE triggers the merge operation. The client first checks the size of the data of the sibling node. If the sum of the data size of the underutilized node and the sibling node is smaller than the node size, the merge operation is triggered. The client copies the key-value pairs of the sibling node to the underutilized node. The client writes the underutilized node to the memory server. After writing the underutilized node, the client reads and locks the parent node, qnd then removes the sibling's parent key-value pair. The sibling page is added to the *free list* of the compute server.

4. Experimental Results

4.1. Experimental Environment

Hardware. We used two servers that are connected directly via infiniband. Each server is a dual-socket machine with two Intel Xeon Gold 5318Y CPUs having 24 physical cores at 2.1 GHz and 256 GB of DRAM. Since there is no publicly available disaggregated memory hardware, we emulate the memory disaggregation using the two servers. Each server launches one memory server and one compute server as in previous work [2]. Each compute server launches multiple threads, and the threads can share the same cache in the server. We set the size of the local cache of the compute server to 1 GB, and the size of memory space is set to 200 GB. Each memory server provides a 100 GB memory size to the memory pool. For all experiments, we follow the prior work [2] to enable huge pages. The main performance overhead in the range index structures for disaggregated memory comes from frequent RDMA Read/Write/Atomic operations because of complicated index operations. In our experiment, each compute server and memory server are distributed among the servers.

Workload. We used YCSB workloads [23], which are representative workloads of key-value stores. Table 1 shows the characteristics of each workload. We performed LOAD A and YCSB A-E to evaluate RF-TREE against other index structures for disaggregated memory. Table 1 shows the ratio of operations of each YCSB workload. In the evaluation, we inserted 10 M key-value pairs and conducted the 10 M operations of workloads after inserting the key-value pairs. We used an 8-byte integer key and 8-byte value pairs.

Table 1. The operation ratio of each YCSB workload.

	Lookup	Insert	Update	Scan
Load A	0%	100%	0%	0%
Workload A	50%	0%	50%	0%
Workload B	95%	0%	5%	0%
Workload C	100%	0%	0%	0%
Workload D	95%	5%	0%	0%
Workload E	0%	5%	0%	95%

Competitors. We compared the performance of RF-TREE against other range indexes for disaggregated memory. We used the open-sourced version of Sherman [24], and we implemented a B^{link}-tree for disaggregated memory that is similar to the FG [6].

The implementation of RF-TREE and B^{link}-tree use the same baseline for the fair performance evaluation. Since FG did not exploit the local cache in the compute server, the B^{link}-tree also did not use the local cache. In the B^{link}-tree, we also added the CRC32 value as a checksum and optimization for the Write operation, such as exploiting the RDMA Write ordering for version updates. Note that Sherman has an inconsistency problem [24], so we fixed the inconsistencies by adding the CRC as a checksum protocol.

4.2. Throughput

We measured throughput for YCSB workloads while varying the number of clients. We used YCSB workloads with both Zipfian distribution and uniform distribution.

Zipfian distribution. Figure 6 shows the throughput of the range index structures for disaggregated memory. In Write-intensive workloads, such as load A and workload A, Sherman shows 16% higher throughput than RF-TREE. This is because Sherman optimizes RDMA one-sided verbs, such as RDMA Write batching and on-chip lock management, which require lots of engineering efforts. Note that RF-TREE also can apply Sherman's optimization. When the Read ratio is increased, as in workloads A–D, RF-TREE shows better throughput than Sherman. This is because RF-TREE has a sorted node layout, and RF-TREE's internal nodes are sequentially stored. In workload E, RF-TREE and Sherman show similar performance. Both experience performance degradation, due, however, to different reasons. Sherman can reduce RDMA Read operations by batching RDMA Read operations, but it suffers huge overhead of in-memory operations, such as cache management and unsorted node layout. Meanwhile, RF-TREE suffers higher RDMA Read operation overhead than Sherman, as it does not provide the batching RDMA Read operation. B^{link}-tree suffers high overhead from the RDMA Read operation because it does not use the local cache of the compute server.

Uniform distribution. The YCSB workload evaluation with uniform distribution shows similar performance trends to the YCSB workload evaluation with Zipfian distribution, as shown in Figure 7. In Write-intensive workloads, RF-TREE shows comparable performance to Sherman because a uniform distribution reduces the possibility of contention.

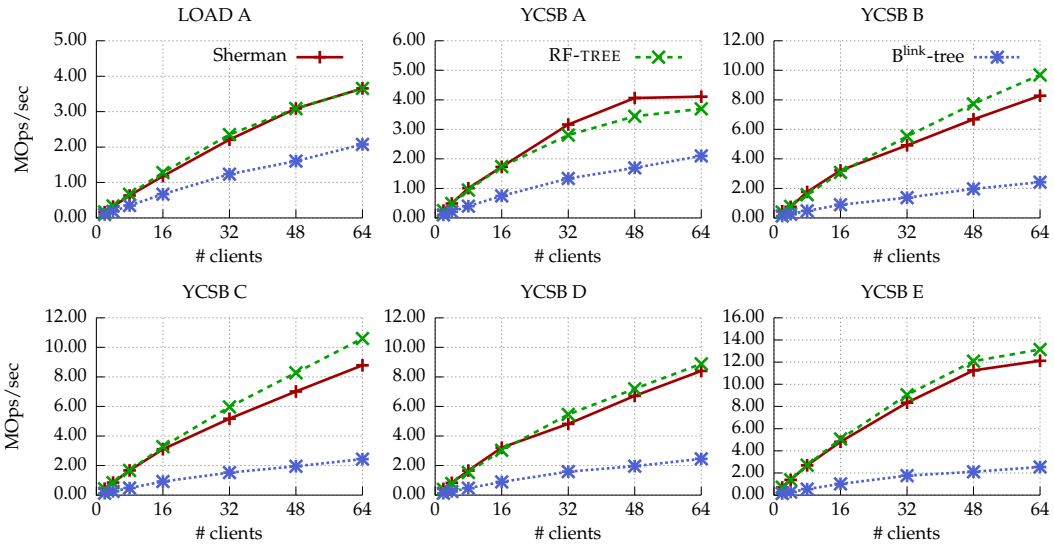

Figure 6. Throughput of range index structures for disaggregated memory (Zipfian distribution).

4.3. Performance Breakdown

We analyzed the performance of range index structures for disaggregated memory, as shown in Figure 8. The performance trends on the Zipfian distribution and uniform distribution were similar. B^{link}-tree spends the most time on RDMA operations, such as RDMA Read, RDMA Write, and RDMA CAS. Sherman and RF-TREE take advantage of their local cache, so the elapsed time for RDMA operation was drastically reduced. As shown in Figure 8, both Sherman and RF-TREE similarly experience significant performance overhead due to the CRC (checksum) computation (i.e., 7–50%). Moreover, RF-TREE showed less in-memory computation overhead than Sherman because RF-TREE stores the search layer in the consecutive memory space, and RF-TREE has a sorted layout. Since the in-memory computation overhead is low, RF-TREE is more efficient for disaggregated memory. Sherman shows low overhead from RDMA Write operations because it only writes key-value pairs to the target data node instead of the whole data node. More-

over, Sherman incurs higher overhead for in-memory overhead, which is inefficient for disaggregated memory.

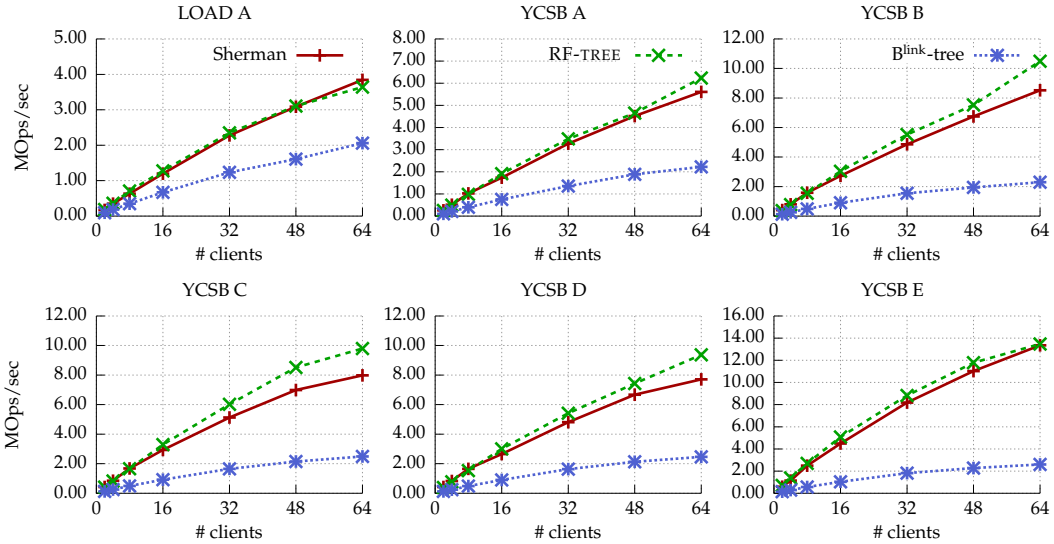

Figure 7. Throughput of range index structures for disaggregated memory (uniform distribution).

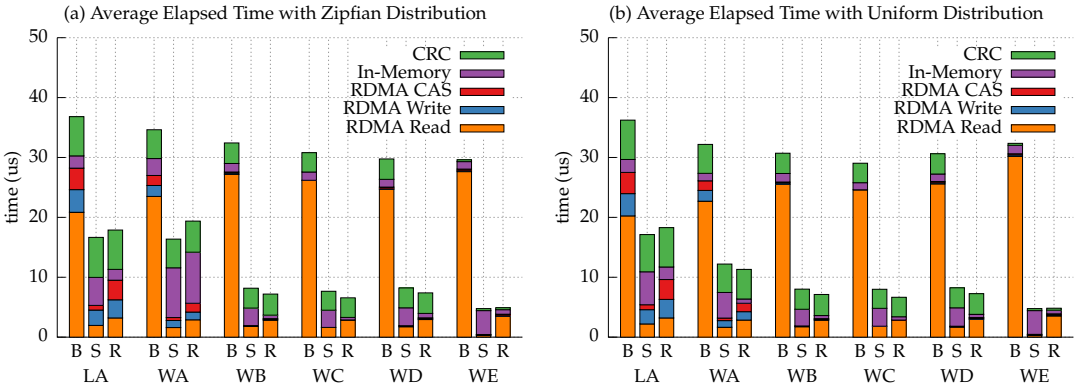

Figure 8. Performance breakdown of range index structures for disaggregated memory with YCSB workload. In this experiment, B stands for B^{link}-tree, S stands for Sherman, and R stands for RF-TREE. For workload, LA stands for load A, and WA–WE stand for workload A–E.

In workload E, Sherman batches RDMA Read operations for reading the data nodes. RF-TREE traverses data nodes using a sibling pointer. Sherman spends more time on in-memory operation than RF-TREE because Sherman has an unsorted data node layout and SkipList-based local cache of the compute server.

4.4. Tail Latency

We performed tail latency evaluation for the range index structures for disaggregated memory, as shown in Figures 9 and 10. Overall, RF-TREE shows lower latency than other range index structures in medium latency (50%) and 99th latency. While RF-TREE might experience high tail latency at 99.9 and 99.99th, this can be further optimized by adopting optimization techniques in Sherman (e.g., batching RDMA operations and leveraging on-chip memory). We leave this for future work considering that a high tail latency in Write-intesive workloads represents the worst-case scenario and that RF-TREE has a performance benefit for a Read-intensive workload.

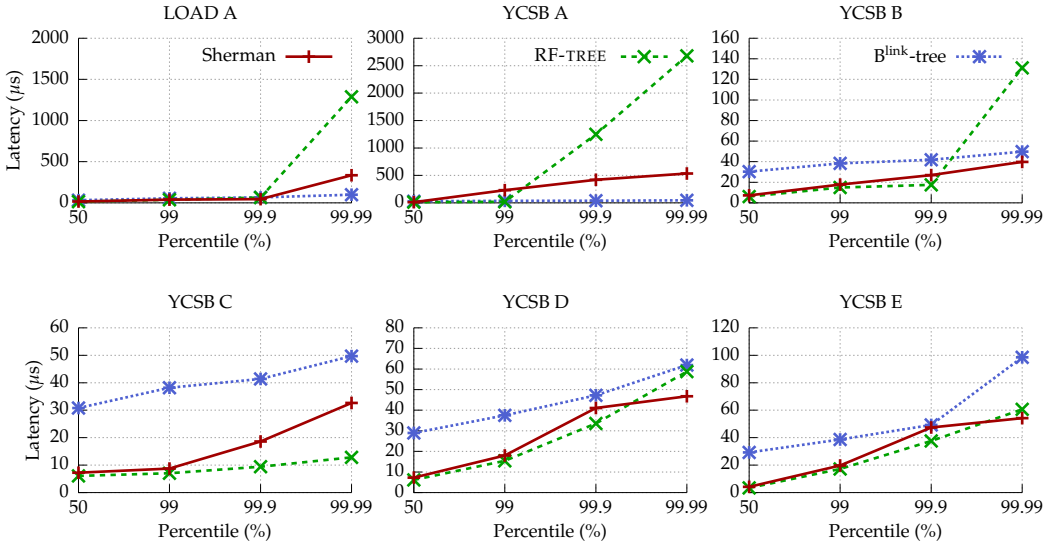

Figure 9. Tail latency of range index structures for disggregated memory with YCSB workload (Zipfian Distribution).

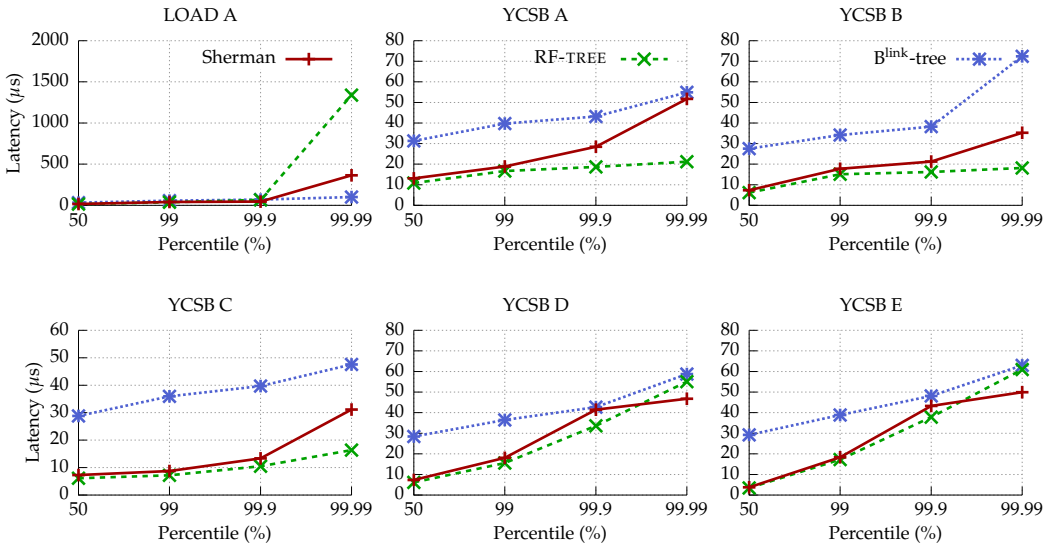

Figure 10. Tail latency of range index structures for disggregated memory with YCSB workload (Uniform Distribution).

Zipfian distribution. In load A, RF-TREE shows higher latency at the 99.99 percentile, as it frequently reads the search layer from the memory server. Note that the load A workload includes insert operation only and starts from scratch. In that case, the search layer is updated frequently, and thus, it worsens the performance of RF-TREE.

In workloads A–B, Sherman shows lower tail latency than RF-TREE, which translates to Sherman's optimization techniques for RDMA Write/CAS operation. The workloads include Write operations, and there are contentions between clients with the Zipfian workload, so RDMA CAS overhead was increased. Moreover, retrying the search layer Read operation incurs additional CRC overhead, so the tail latency is larger. In workloads C–E, RF-TREE shows comparable performance to Sherman, as it efficiently leverages the SLC.

Uniform distribution. In the tail latency evaluation with uniform distribution, RF-TREE exhibits lower latency than Sherman in most workloads. This is because the effect of Sherman's optimization techniques is not significant where there is low contention between clients.

5. Related Work

In this section, we briefly introduce the related work to our work.

5.1. RDMA One-Sided Verb-Based Key-Value Stores

There are previous studies that exploit the RDMA one-sided verbs for key-value stores. Many previous works are based on distributed hash tables because of their simpler structure than range index structures.

Clover [4] is a key-value store working with disaggregated persistent memory. Clover is a distributed hash table-based key-value store and employs multi-version-based concurrency control. Clover leverages RDMA one-sided verbs to access the data in the disaggregated persistent memory.
RACE Hashing [3] is a distributed lock-free hash table that exploits RDMA one-sided verbs. Race Hashing exploits the computing server's local cache and provides a protocol to tolerate inconsistencies in the cached data in the local cache.
Herd [25] is a key-value store that carefully optimizes RDMA operations to obtain high performance.

5.2. Conversion Techniques for Emerging Hardware

Emerging persistent memory opens up new challenges as it has unique characteristics. However, legacy storage systems still depend on volatile index structures. Moreover, newly developed persistent indexes are less stable than mature volatile indexes [26–30]. Hence, the researchers focus on leveraging the existing concurrent indexes for persistent memory.

Recipe [26] proposes the protocols to convert volatile concurrent data structure to persistent indexes by adding persistence instructions, such as *clwb and sfence*, *after* store instruction. Recipe supports both lock-based and lock-free data structures.
NVTraverse [31] and **Log-Free Concurrent Data Structure** [30] propose approaches to convert the volatile lock-free data structure to a persistent lock-free data structure. NVTraverse adds persistence instructions after every *store* and *load* instruction to guarantee the persistence of the data structure. Log-Free Concurrent Data Structure proposes the *link and persist* concept to guarantee persistence using pointer marking. In link and persist, not only does the writer persist the data structure but the reader also persists the data structure when it detects unpersisted data.

Previous studies require in-depth knowledge of the volatile indexes as they have to add persistence instructions manually. There have been research efforts to develop the framework to convert the volatile indexes to persistent ones systemically.

Pronto [28] is a framework to convert a legacy volatile index structure to a persistent index structure. In Pronto, the volatile index structures are stored in both volatile DRAM and persistent memory. The foreground threads access the index structure in the DRAM. If the threads update the index structure, the update will be written to the operation log. A background thread updates the index structure in the persistent memory using the operation log.
TIPS [29] is another framework to make volatile index structures persistent. TIPS stores the index structure on persistent memory. In DRAM, TIPS keeps a hash-table cache to provide high performance. TIPS also uses an operation log to guarantee persistence and leverages background threads to update the index structures on persistent memory asynchronously.

6. Conclusions

We presented the SPREAD approach, which is an approach to convert concurrent range index structures to range index structures for disaggregated memory. The SPREAD approach leverages the concept of tolerating transient inconsistencies in concurrent range index structures. This approach makes it easy to convert concurrent range indexes to range index structures for disaggregated memory. Following the SPREAD approach, we

built RF-TREE, which converts the B^{link}-tree to a range index structure for disaggregated memory. In our experimental evaluation, RF-TREE shows comparable performance to Sherman, a carefully engineered range index structure for disaggregated memory while providing easy conversion without complex engineering tasks.

Author Contributions: Conceptualization, B.K. and W.-H.K.; methodology, B.K. and W.-H.K.; software, B.K. and J.H.; validation, B.K., J.H., J.P. and W.-H.K.; writing—original draft preparation, B.K. and W.-H.K.; writing—review and editing, B.K., J.P. and W.-H.K.; supervision, W.-H.K.; project administration, W.-H.K.; funding acquisition, W.-H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2022R1F1A1076373) and by Institute of Information & communications Technology Planning & Evaluation (IITP) under the metaverse support program to nurture the best talent (IITP-2023-RS-2023-00256615) grant funded by the Korea government (MSIT).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data used in the paper are available from the corresponding author upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lim , K.; Chang, J.; Mudge, T.; Ranganathan, P.; Reinhardt, S.K.; Wenisch, T.F. Disaggregated Memory for Expansion and Sharing in Blade Servers. *SIGARCH Comput. Archit. News* **2009**, *37*, 267–278. [CrossRef]
2. Wang, Q.; Lu, Y.; Shu, J. Sherman: A Write-Optimized Distributed B + Tree Index on Disaggregated Memory. In *SIGMOD'22: Proceedings of the 2022 International Conference on Management of Data*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1033–1048. [CrossRef]
3. Zuo, P.; Sun, J.; Yang, L.; Zhang, S.; Hua, Y. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 21), Virtual, 14–16 July 2021; pp. 15–29.
4. Tsai, S.Y.; Shan, Y.; Zhang, Y. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20), Virtual, 15–17 July 2020; pp. 33–48.
5. Dragojević, A.; Narayanan, D.; Castro, M.; Hodson, O. FaRM: Fast Remote Memory. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), Seattle, WA, USA, 2–4 April 2014; pp. 401–414.
6. Ziegler, T.; Vani, S.T.; Binnig, C.; Fonseca, R.; Kraska, T. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In Proceedings of the ACM SIGMOD/PODS Conference, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 741–758.
7. Mitchell, C.; Montgomery, K.; Nelson, L.; Sen, S.; Li, J. Balancing CPU and Network in the Cell Distributed B-Tree Store. In Proceedings of the USENIX Annual Technical Conference (ATC), Denver, CO, USA, 22–24 June 2016; pp. 451–464.
8. Hwang, D.; Kim, W.H.; Won, Y.; Nam, B. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST), Oakland, CA, USA, 12–15 February 2018; pp. 187–200.
9. Mathew, A.; Min, C. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In Proceedings of the 46th International Conference on Very Large Data Bases (VLDB), Tokyo, Japan, 31 August–4 September 2020.
10. Kim, W.H.; Krishnan, R.M.; Fu, X.; Kashyap, S.; Min, C. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP), Online, 26–29 October 2021; pp. 424–439.
11. Lehman, P.L.; Yao, S.B. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* **1981**, *6*, 650–670. [CrossRef]
12. Pugh, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* **1990**, *33*, 668–676. [CrossRef]
13. Hu, Q.; Ren, J.; Badam, A.; Shu, J.; Moscibroda, T. Log-Structured Non-Volatile Main Memory. In Proceedings of the ATC17, Santa Clara, CA, USA, 12–14 July 2017.
14. Rumble, S.M.; Kejriwal, A.; Ousterhout, J. Log-structured Memory for DRAM-based Storage. In Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14), Santa Clara, CA, USA, 17–20 February 2014; pp. 1–16.
15. Zamanian, E.; Binnig, C.; Harris, T.; Kraska, T. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* **2017**, *10*, 685–696. [CrossRef]

16. Ziegler, T.; Nelson-Slivon, J.; Leis, V.; Binnig, C. Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA. *Proc. ACM Manag. Data* **2023**, *1*, 1–26. [CrossRef]
17. Mitchell, C.; Geng, Y.; Li, J. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 13), San Jose, CA, USA, 26–28 June 2013; pp. 103–114.
18. Singhvi, A.; Akella, A.; Anderson, M.; Cauble, R.; Deshmukh, H.; Gibson, D.; Martin, M.M.K.; Strominger, A.; Wenisch, T.F.; Vahdat, A. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *SIGCOMM'21: Proceedings of the 2021 ACM SIGCOMM 2021 Conference*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 93–105. [CrossRef]
19. Wei, X.; Chen, R.; Chen, H. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Virtual, 4–6 November 2020; pp. 117–135.
20. Mellanox Technologies. RDMA Aware Networks Programming User Manual. Available online: https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17 (accessed on 30 September 2023).
21. Liu, J.; Chen, S.; Wang, L. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* **2020**, *13*, 1078–1090. [CrossRef]
22. Chen, Y.; Lu, Y.; Fang, K.; Wang, Q.; Shu, J. UTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* **2020**, *13*, 2634–2648. [CrossRef]
23. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.
24. Sherman Github. Available online: https://github.com/thustorage/Sherman (accessed on 20 August 2023).
25. Kalia, A.; Kaminsky, M.; Andersen, D.G. Using RDMA Efficiently for Key-Value Services. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 295–306. [CrossRef]
26. Lee, S.K.; Mohan, J.; Kashyap, S.; Kim, T.; Chidambaram, V. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, ON, Canada, 27–30 October 2019.
27. Fu, X.; Kim, W.H.; Shreepathi, A.P.; Ismail, M.; Wadkar, S.; Lee, D.; Min, C. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *SOSP'21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*; Association for Computing Machinery: New York, NY, USA, 2021; pp. 100–115. [CrossRef]
28. Memaripour, A.; Izraelevitz, J.; Swanson, S. Pronto: Easy and Fast Persistence for Volatile Data Structures. In Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland, 16–20 March 2020.
29. Krishnan, R.M.; Kim, W.H.; Fu, X.; Monga, S.K.; Lee, H.W.; Jang, M.; Mathew, A.; Min, C. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21), Virtual, 14–16 July 2021, pp. 773–787.
30. David, T.; Dragojevic, A.; Guerraoui, R.; Zablotchi, I. Log-free concurrent data structures. In Proceedings of the 2018 USENIX Annual Technical Conference (ATC), Boston, MA, USA, 11–13 July 2018.
31. Friedman, M.; Ben-David, N.; Wei, Y.; Blelloch, G.E.; Petrank, E. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, UK, 15–20 June 2020.