A High-performance RDMA-oriented Learned Key-value Store for Disaggregated Memory Systems

PENGFEI LI, YU HUA, PENGFEI ZUO, ZHANGYU CHEN, and JIAJIE SHENG, Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, China

Disaggregated memory systems separate monolithic servers into different components, including compute and memory nodes, to enjoy the benefits of high resource utilization, flexible hardware scalability, and efficient data sharing. By exploiting the high-performance RDMA (Remote Direct Memory Access), the compute nodes directly access the remote memory pool without involving remote CPUs. Hence, the ordered key-value (KV) stores (e.g., B-trees and learned indexes) keep all data sorted to provide range query services via the high-performance network. However, existing ordered KVs fail to work well on the disaggregated memory systems, due to either consuming multiple network roundtrips to search the remote data or heavily relying on the memory nodes equipped with insufficient computing resources to process data modifications. In this article, we propose a scalable RDMA-oriented KV store with learned indexes, called ROLEX, to coalesce the ordered KV store in the disaggregated systems for efficient data storage and retrieval. ROLEX leverages a retraining-decoupled learned index scheme to dissociate the model retraining from data modification operations via adding a bias and some data movement constraints to learned models. Based on the operation decoupling, data modifications are directly executed in compute nodes via one-sided RDMA verbs with high scalability. The model retraining is hence removed from the critical path of data modification and asynchronously executed in memory nodes by using dedicated computing resources. ROLEX efficiently alleviates the fragmentation and garbage collection issues, due to allocating and reclaiming space via fixed-size leaves that are accessed via the atomic-size leaf numbers. Our experimental results on YCSB and real-world workloads demonstrate that ROLEX achieves competitive performance on the static workloads, as well as significantly improving the performance on dynamic workloads by up to 2.2× over state-of-the-art schemes on the disaggregated memory systems. We have released the open-source codes for public use in GitHub.

CCS Concepts: • **Information systems** → **Network attached storage**; **Distributed storage;**

Additional Key Words and Phrases: Disaggregated memory system, learned index, key-value store

The preliminary version was published in the Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23), 99-114, as "ROLEX: A Scalable RDMA-oriented Learned Key-value Store for Disaggregated Memory Systems."
This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 62125202, U22B2022, and 61821003.
Authors' address: P. Li, Y. Hua (Corresponding author), P. Zuo, Z. Chen, and J. Sheng, Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, China, 430074; e-mails: {cspfli, csyhua, pfzuo, chenzy, jjsheng}@hust.edu.cn.

1553-3077/2023/09-ART30 $15.00
https://doi.org/10.1145/3620674

ACM Reference format:
Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. A High-performance RDMA-oriented Learned Key-value Store for Disaggregated Memory Systems. *ACM Trans. Storage* 19, 4, Article 30 (September 2023), 30 pages.
https://doi.org/10.1145/3620674

1 INTRODUCTION

Recent *disaggregated memory systems* separate memory, storage, and computing resources into independent pools [16, 36, 45] for high resource utilization, flexible hardware scalability, and efficient data sharing, which become prevalent in many datacenters and clouds [2, 3, 8]. The disaggregated system adopts the **Remote Direct Memory Access (RDMA)**-capable networks for communications due to the salient features, such as high throughput (40–400 Gbps), low latency (a few microseconds), and remote CPU/kernel bypassing [12, 44, 54], which are widely supported by InfiniBand, RoCE, and OmniPath [16, 31, 39, 44, 52].

The disaggregated memory systems become important infrastructures [1, 17, 34, 36, 42, 43, 45] for various applications, including databases [29, 43] and in-memory **key-value (KV)** stores [12, 42, 47, 56]. Among them, tree-based and learned indexes are two ordered structures for the key-value stores, which provide efficient range query performance via identifying items in a given range [7, 24]. In the disaggregated memory systems, the machines in compute and memory pools are respectively termed *compute and memory nodes*, which are specialized for computing and storage purposes.

Deploying tree-based structures in the disaggregated memory system becomes inefficient, since the inner nodes consume much memory space and fail to be fully cached, thus resulting in multiple network roundtrips for traversing the entire tree. Various *index caching* schemes [33, 46, 55] propose to alleviate the network penalty via locally caching partial data, which, however, still suffer from unavoidable capacity misses due to the rapid growth of data.

Unlike them, XStore [47] proposes to cache the learned indexes for remote data accessing, since the learned models consume less memory footprints than tree-based structures by up to several orders of magnitude [14, 24]. By locally holding the whole learned index structure, a one-sided RDMA READ is sufficient for compute nodes to fetch remote data in the context of static (i.e., read-only) workloads. However, the design goal of XStore is not to exploit the strengths of disaggregated memory systems. Instead, XStore relies on the monolithic servers to process dynamic (i.e., read-write and write-intensive) workloads. We construct existing XStore on the disaggregated memory systems (represented as XStore-D), rather than the conventional monolithic context, by transferring data modification requests to memory nodes via RPCs. We observe that in fact XStore-D becomes inefficient to handle intensive modification requests, since the computing resources in the memory nodes are insufficient to meet the intensive computation requirements [42, 56]. As a result, new models fail to be retrained in time and the stale models expand to a large prediction range to search dynamic workloads. The compute nodes have to consume more network roundtrips in determining the exact positions, since the positions dynamically change for data modifications. To avoid the penalty of large expansion, XStore-D transfers the subsequent requests to memory nodes until new models are retrained, which further increases the computing burden upon memory nodes. It is non-trivial to coalesce ordered KV stores in the disaggregated memory systems due to the following challenges:

(1) ***Limited computing resources on memory nodes.*** Existing ordered KV stores rely on the monolithic servers to process write-intensive modifications [23, 47]. However, the memory nodes in the disaggregated systems contain limited computation capability and fail to meet

the requirements of computing-intensive operations, e.g., modifying the large B-tree and frequently retraining models. The CPU access bottleneck on the memory nodes decreases the overall system performance. Moreover, simply adding more CPUs to the memory pool for data processing decreases the elasticity of the disaggregated memory systems, since the memory and computing resources fail to be independently scaled out [36, 55].

(2) ***Overloaded bandwidth for data transferring.*** Offloading data modifications to the compute nodes meets the computing requirements, which, however, rapidly fills up the entire bandwidth due to transferring massive data. Specifically, the compute nodes consume a large amount of network bandwidth to balance tree-based structures [7, 32], e.g., multi-level nodes splitting and merging, as well as fetching a large amount of data to retrain models for the learned indexes [10, 14, 40]. The network bandwidth becomes insufficient to enable high performance for various data requests.

(3) ***Inconsistency issue among different nodes.*** Guaranteeing data consistency among different nodes during modification is essential to prevent data loss. However, the inconsistent states occur when different compute nodes fail to atomically complete the data and model modification operations; e.g., multiple compute nodes compete for the same space to insert data and the local cache becomes stale when the models are updated. The main reason is that the atomic granularity of an RDMA operation is 8B, which is much smaller than the size of each index operation. The compute nodes require multiple network roundtrips to guarantee data consistency, incurring high overheads for consistency.

To address the aforementioned challenges, we propose a scalable RDMA-oriented key-value store using learned indexes, called ROLEX, for the disaggregated memory systems, which processes data requests on the compute nodes via one-sided RDMA operations. The context of "scalable" means that ROLEX efficiently supports dynamic workloads and scales out to multiple disaggregated nodes. Although ROLEX adopts the similar idea as XStore on the static (i.e., read-only) operations, ROLEX is completely different from XStore in terms of the application scope, dynamic (i.e., data modification) operations, and the index structure on memory nodes. Specifically, ROLEX aims to efficiently support both static and dynamic workloads in the disaggregated memory systems. Unlike XStore, ROLEX does not maintain a B-tree on memory nodes to process modifications. Instead, ROLEX directly stores the sorted data in the assigned leaves (i.e., data arrays) on memory nodes. By judiciously decoupling the index operations and moving the retraining phase out of the critical path, the compute nodes efficiently modify the remote data via one-sided RDMA operations. When there are insufficient slots, ROLEX leverages a leaf-atomic shift scheme to atomically allocate a new leaf for accommodating more data. By using the retraining-decoupled index structure, ROLEX asynchronously retrains a model in the memory pool when there are sufficient computing resources. The compute nodes identify new models through a shadow redirection scheme and synchronize the retrained models from remote nodes during the next reading. It is worth noting that the memory node generally includes dedicated computing resources provided by FPGA or ARM cores to offload low-computing-requirement operations [17] (e.g., infrequent retraining in ROLEX), rather than all index operations.

We implement a prototype of ROLEX[1] and evaluate the performance via widely used YCSB [50], two real-world, and two synthetic workloads. Our experimental results show that ROLEX achieves competitive performance with XStore-D [47] on static workloads and outperforms state-of-the-art RDMA-based ordered KV stores by up to 2.2× on dynamic workloads. In summary, we have the following contributions:

[1]The source code is available at https://github.com/iotlpf/ROLEX

- *Scalable ordered KV store for disaggregated memory systems.* We propose ROLEX to directly process data requests on the compute nodes via one-sided RDMA operations, which efficiently explores and exploits the hardware benefits of the disaggregated memory systems, as well as avoiding the computing resources bottleneck in the memory pool.
- *Retraining-decoupled learned indexes for one-sided RDMA execution.* We decouple the insertion and retraining operations for the learned indexes and enable compute nodes to directly insert data without waiting for the model retraining. Non-retrained models are able to index newly inserted data using the proposed data movement constraints.
- *Atomic remote space allocation.* When there are insufficient slots, the compute nodes leverage a leaf-atomic shift scheme to atomically allocate data arrays in the memory pool for accommodating new data. In ROLEX, no collisions occur among different machines due to the atomic metadata management.

2 BACKGROUND AND MOTIVATION

2.1 Disaggregated Memory Systems

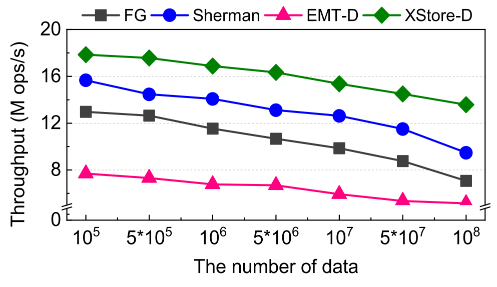
The disaggregated memory systems break monolithic servers into independent network-attached components, which meet various application requirements via independently scaling out the hardware resources. Different nodes communicate with each other via RDMA NICs, such as InfiniBand, RoCE, and OmniPath. The significant feature over the traditional network is that RDMA enables the compute nodes to directly access the memory nodes without involving remote CPUs via one-sided verbs, including `RDMA READ`, `WRITE`, and `ATOMIC` operations (e.g., **compare-and-swap (`CAS`)** and **fetch-and-add (`FAA`)**). It is worth noting that the granularity of the `ATOMIC` operation is 8B, and multiple `READ` and `WRITE` operations are completed via the doorbell batching [47] to reduce the network latency. Moreover, even though there are no powerful CPUs in the memory pool, each memory node generally includes dedicated computing resources provided by FPGA or ARM cores in NICs that are used for operation offloading [17], which efficiently supports the operation decoupling in ROLEX.

2.2 Network-attached Ordered KV Store

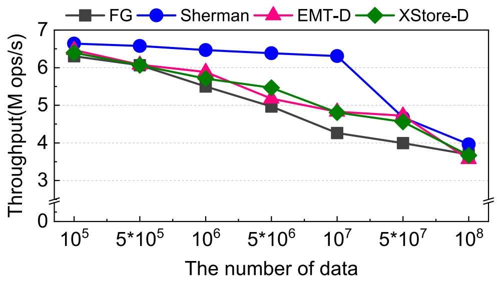
This article mainly focuses on the network-attached ordered key-value stores, including tree-based and learned indexes, which keep all data sorted and meet range query requirements.

Tree-based structures. Tree-based structures [7, 20, 32] (e.g., B^+-tree) store data in the leaf nodes and construct multi-level inner nodes to search the leaves. However, the tree-based structures become inefficient to leverage one-sided RDMA for accessing remote data [47], since the local machine fails to cache the whole index structure and has to consume multiple network **roundtrip times (RTTs)** for searching the inner nodes. Recent designs [33, 46, 55] cache top-level nodes on compute nodes to access the remote data. Among them, FG [55] designs a fine-grained B-link tree for the disaggregated systems, which distributes tree nodes across memory nodes and modifies trees with RDMA-based locks. Sherman [46] combines RDMA-friendly hardware and software features to deliver high write performance on the remote B-link tree, which optimizes the locking phase by constructing global locks on the on-chip memory of RDMA NICs. However, tree-based schemes inevitably incur multiple RTTs for retrieving inner nodes when the data overflow the limited local cache.

Learned indexes. Learned indexes show significant advantages over tree-based structures in terms of search speed and memory consumption, due to the easy-to-use and small-sized learned models. Specifically, the learned indexes view the process of searching data as a regression model, which record the positions of all data by approximating the **cumulative distribution function**



(a) Read on static workloads.



(b) Write different numbers of data.

Fig. 1. The system performance for different schemes. *(a) Read and (b) write throughputs with different numbers of data, using one CPU core on memory nodes.*

(CDF) of the sorted keys [10, 14, 15, 24, 40]. The learned models achieve two to four orders of magnitude space savings than the inner nodes of the tree-based structures [14], which enables the local machine to cache the whole index structures and avoid the penalty of multiple RTTs to determine the remote data positions.

XStore proposes a hybrid index structure, i.e., maintaining a B-tree to process modifications and locally caching the learned indexes for remote data accessing. XStore [47] delivers high search performance due to only requiring one RTT to access the static workloads. For the dynamic workloads, XStore handles the data modification requests by modifying the B-tree on the memory nodes. At the same time, XStore expands the stale models to large prediction ranges to ensure that the newly inserted data are contained. However, such design becomes inefficient on the disaggregated memory systems, since the memory nodes have limited computing resources and fail to efficiently handle the intensive modification requests. The new models fail to be retrained in time and the stale models cause too low accuracy to search the remote data in one RTT due to the model expansion. As a result, the local cache becomes invalid and the subsequent data requests are transferred to the memory nodes via classic RPCs. The overall performance significantly decreases due to the limited computing resources of memory nodes.

2.3 Performance Analysis

We evaluate and analyze the performance of existing network-attached KV stores in the disaggregated memory system. Among them, FG [55] and Sherman [46] design RDMA-enabled B-link trees, enabling compute nodes to modify B-link trees via one-sided RDMA verbs. Moreover, we also equip the memory nodes with limited computing resources to analyze why RPC-based KV stores are inefficient for the disaggregated memory system, i.e., adopting the similar ideas of EMT-D (i.e., the Masstree [32] based on eRPC [23]) and XStore-D [47] on the computation-constrained memory nodes for evaluations.

Learned indexes outperform tree-based structures on large-scale static workloads. Figure 1(a) shows the search throughput on static workloads. As the datasets constantly increase, XStore-D shows higher throughput than tree-based structures, since the compute nodes cache the whole learned index structure, rather than caching partial inner nodes for tree-based structures, avoiding multiple RTTs to determine the data positions. XStore-D obtains remote data within one RTT according to the prediction results of the learned models, while other schemes fail.

Index cache becomes invalid on dynamic workloads. Figure 1(b) shows the throughput on write-intensive workloads. We observe that XStore-D delivers lower performance than Sherman, since XStore-D sends requests to memory nodes via eRPC and relies on the limited computing

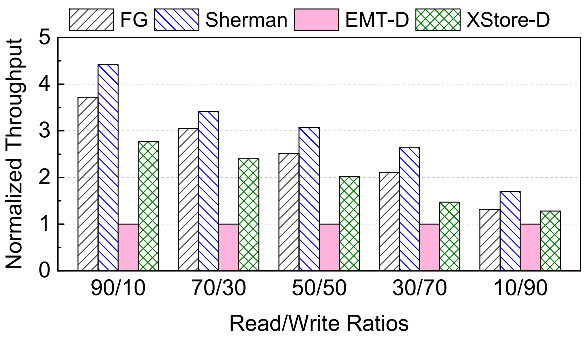


Fig. 2. Normalized throughput with respect to EMT-D for hybrid read/write workloads.

resources of memory nodes to process modifications. The local cache of XStore-D is not fully exploited and becomes invalid during the modification phase, while Sherman delivers higher throughput via one-sided RDMA. However, the performance of Sherman decreases when storing a large amount of data, since the increased inner nodes overflow the local cache.

Disaggregated system requires efficient one-sided RDMA operations. Figure 2 shows the throughputs of different schemes with respect to EMT-D when configuring various read/write ratios. FG and Sherman show significant advantages over EMT-D, since all index operations are completed via one-sided RDMA. The performance of XStore-D significantly deceases when configuring large write ratios, due to failing to handle writes via one-sided RDMA operations.

For existing schemes on the disaggregated memory systems, the learned indexes are limited by dynamic workloads, while the tree-based structures are limited by the increasing cache sizes. Unlike them, the design goal of ROLEX is to enable the ordered KV store to deliver high performance on both static and dynamic workloads in the disaggregated system.

3 ROLEX DESIGN

3.1 Overview

We present *a scalable RDMA-oriented key-value store using learned indexes (ROLEX)* for the disaggregated memory systems. Unlike existing schemes, ROLEX does not maintain a B-tree on the memory nodes to process data requests. Instead, ROLEX constructs the retraining-decoupled learned indexes on the stored data and processes data requests on compute nodes via the one-sided RDMA operations. The challenges are how to efficiently avoid the collisions of various index operations in different compute nodes, as well as enabling all compute nodes to correctly identify the modified data with low-consistency overheads. Our main insights are to execute index operations with atomic designs and asynchronously retrain models by decoupling the insertion and retraining operations with consistency guarantees.

Figure 3 shows the overview of ROLEX. In the memory pool, ROLEX stores all data into fixed-size leaves (i.e., arrays) and constructs a retraining-decoupled learned index based on these data, as shown in Sections 3.2 and 3.3. To process dynamic workloads, the compute nodes directly modify the remote leaves without retraining models, since we decouple the insertion and retraining operations. By adding a bias and some data movement constraints, the non-retrained models have the ability to correctly identify all data even after inserting new data. To construct sufficient data leaves for the new data with one-sided RDMA, we present a *leaf-atomic shift scheme* in Section 3.4, which also keeps all data sorted for range queries and avoids the collisions among different compute nodes. The stale models need to be retrained for high accuracy when a large amount of data

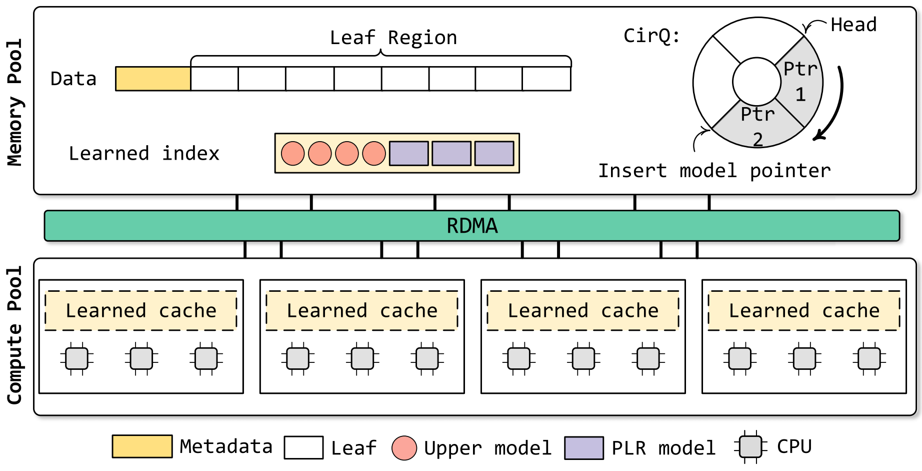


Fig. 3. The design overview of ROLEX.

are modified. Although the compute nodes have sufficient computing resources for retraining, obtaining all the pending retraining data from memory nodes consumes much network bandwidth. Instead, we observe that the retraining overheads mainly come from data merging and re-sorting, while the complexity of the training algorithm is only $O(N)$. We reserve limited computing resources in the memory pool (e.g., one CPU core on the memory node), which are sufficient to retrain models, especially after we have offloaded most index operations to the compute nodes and moved the retraining phase out of the critical path. With the aid of *leaf tables*, ROLEX *asynchronously* retrains models in place on the memory nodes, as shown in Section 3.5. After retraining, ROLEX updates the models in the memory pool using *the shadow redirection scheme*, while the compute nodes won't synchronize the retrained models until the next reading.

3.2 Retraining-decoupled Learned Indexes

The challenges of coalescing the learned indexes on dynamic workloads come from the high overheads of keeping all data sorted and avoiding data loss from the learned models during insertion. The reason for data loss is that the models record the positions of the trained data after training while failing to find the new positions after inserting many new data unless retraining. As shown in Figure 4, the red line represents a linear regression model that is trained on the black points (i.e., the trained data). All data are found in the prediction range, $[pred - \epsilon, pred + \epsilon]$ (i.e., the blue block), as long as the data are not moved out of this range, where ϵ is the predefined maximum model error. When some new data are inserted, point a moves backward to a', which is out of the prediction range. To record the new positions, the models are retrained via step-by-step operations, including re-sorting data, retraining models, and synchronizing models to all compute nodes. The system is blocked until the retraining and synchronization are completed, thus incurring a long latency and decreasing the overall system performance.

In fact, we observe that the learned indexes don't require frequent retraining as long as the non-retrained models can find all data. This observation offers an opportunity to address the dilemma in coalescing the learned indexes in the disaggregated memory systems; i.e., new data are written to the memory pool without waiting for retraining. To achieve this design goal, we modify the training algorithm and add some constraints to help the non-retrained models always find all data without retraining.

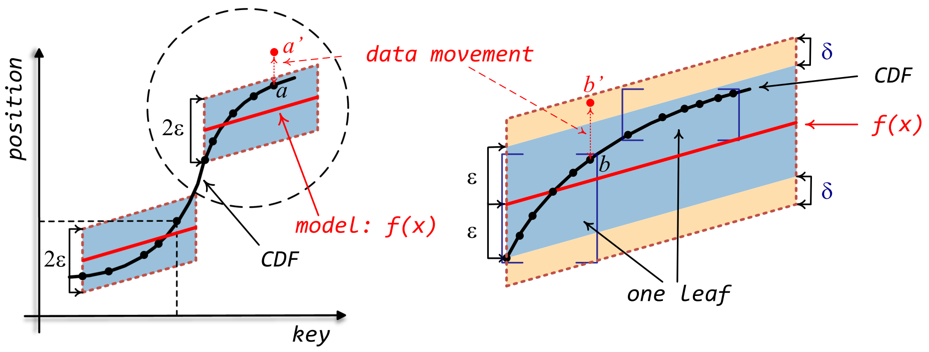


Fig. 4. The retraining-decoupled learned indexes.

Training algorithm. Leveraging multiple linear regression models is a common way to learn the data distribution due to the efficiency of training and memory savings [10, 14, 15, 24]. We use an *improved OptimalPLR* algorithm to train the **piece-wise linear regression (PLR)** models, since the OptimalPLR algorithm [49] has been proved to have the minimal number of PLR models while incurring small time and space complexity ($O(N)$). The key idea of OptimalPLR is to construct multiple optimal parallelograms with 2ϵ width on the trained data, where the optimal parallelogram is defined as a parallelogram of 2ϵ width in the vertical direction such that no trained data are placed outside of the parallelogram, as the blue blocks show in Figure 4. We thus obtain the linear regression model that intersects the two vertical sides and bisects the parallelogram.

$$\begin{aligned} \epsilon &>= max|f(X_i) - Y_i| \quad \forall i \in (0, N) \\ P_{range} &= [f(X_i) - \epsilon - \delta, f(X_i) + \epsilon + \delta] \end{aligned} \tag{1}$$

To ensure that the trained models find all data even after insertions, we improve the OptimalPLR algorithm by adding a bias (represented as δ) to the prediction calculation, as well as adding some constraints on the data movements. As shown in Equation (1), the optimal parallelogram is determined by guaranteeing that the distances between the predicted ($f(X_i)$) and true (Y_i) positions of all data are not larger than the predefined maximum model error (ϵ), while the prediction range (P_{range}) is calculated by adding an extra δ. Hence, the area covered by the prediction ranges of all data is larger than the determined optimal parallelogram; i.e., we extend the blue block to the yellow one, as shown in Figure 4. In this case, the models don't require retraining as long as the data move no more than δ positions, since the δ data movements won't exceed the prediction range.

Data movement constraints. Simply adding a bias to the prediction calculation is insufficient to achieve the design goal of operation decoupling, since the data move more than δ positions when inserting/deleting a large amount of data. To further address these issues, we add some constraints on the data movements:

- *Moving data within fixed-size leaves.* We store the data into fixed-size arrays (termed *leaves*) in the training phase, and each leaf contains at most δ data. All data are only allowed to be moved within their assigned leaves. In this case, we identify all data via existing trained models since no data move out of P_{range} calculated from Equation (1). Furthermore, we transfer the position prediction to the leaf prediction; i.e., the learned models provide a range of leaves that may contain the queried data via Equation (2). Due to not moving out of the assigned leaves, no data are lost. In the disaggregated memory systems, the leaves in L_{range}

are easily obtained via one-sided RDMA verbs.

$$L_{range} = \left[\frac{f(X_i) - \epsilon}{\delta}, \frac{f(X_i) + \epsilon}{\delta}\right] \quad \forall i \in (0, N) \tag{2}$$

- *Synonym-leaf sharing.* We allocate a new leaf (*nl*) to accommodate more data when a leaf (*l*) has insufficient slots, where *nl* shares the same positions (i.e., the labels used for training) with *l*. We define *nl* as a *synonym leaf* of *l*, which is linked via a pointer. The data of synonym leaves move within each other to facilitate data sorting. Since *nl* doesn't change the positions recorded by models, the learned indexes still calculate L_{range} via Equation (2). Moreover, we need to search the synonym leaves referred by L_{range}, since the data may locate in the predicted and synonym leaves.

The non-retrained models have the ability to find all data without retraining, since no data move out of the predicted leaves. We hence decouple the insertion and retraining operations for the learned indexes.

3.3 ROLEX Structure

To exploit the hardware benefits of the disaggregated memory systems, ROLEX stores data on the memory nodes while processing requests on the compute nodes, as shown in Figure 3.

Memory pool stores data. Driven by the operation decoupling, we store all data into fixed-size leaves and train a learned index on these data using our improved training algorithm. All leaves are stored in a continuous area (termed a *leaf region*) allocated from an RDMA-registered memory region. The structure of the leaf region is shown in Figure 3, where the first two 8B data are respectively used to indicate the number of leaves that have been allocated (*alloc_num*) and the total number that the leaf region can allocate. The remaining leaf region stores a large number of leaves, and each leaf contains δ pairs of keys and values.[2] To allocate a new leaf, we read *alloc_num* and write it back with (*alloc_num*+1) via the atomic FAA. We store data in the leaf pointed by the obtained *alloc_num*. The leaves are accessed via adding offsets to the start position of the leaf region.

Some leaves become empty when removing all the contained data, which further become fragmented in the disaggregated memory systems when being removed from the trained models. In this case, the fragmentation and garbage collection can be efficiently mitigated in ROLEX, since ROLEX allocates and reclaims space via fixed-size leaves that are accessed via the atomic-size leaf numbers. Specifically, the memory pool maintains a reclaiming queue to contain the numbers of the deleted empty leaves. To avoid the collisions among different compute nodes, the first 8B of data (*empty_num*) in the queue is preserved to indicate the number of slots that have been occupied by the empty leaf numbers, and the remaining slots store the leaf numbers. During the runtime, some leaves become empty after removing all the contained data, and the compute node inserts the corresponding leaf numbers into the reclaiming queue. In this case, the compute node first fetches *empty_num* and writes it back with (*empty_num*+1) via the atomic FAA, and then stores the empty leaf number into the slot pointed by the obtained *empty_num*. Finally, the empty leaves in the reclaiming queue are further reused to improve space utilization. It is worth noting that the process of inserting empty leaf numbers is asynchronously conducted in the background, since reclaiming empty leaves is not a performance-critical operation. Moreover, to prevent the compute nodes from accessing the reused leaves via the stale models, ROLEX updates the cached models when identifying that the model pointers change, as shown in Section 3.5.

[2]Similar to prior RDMA-based schemes [33, 46, 47], ROLEX stores 8B values or 8B pointers for variable-length values.

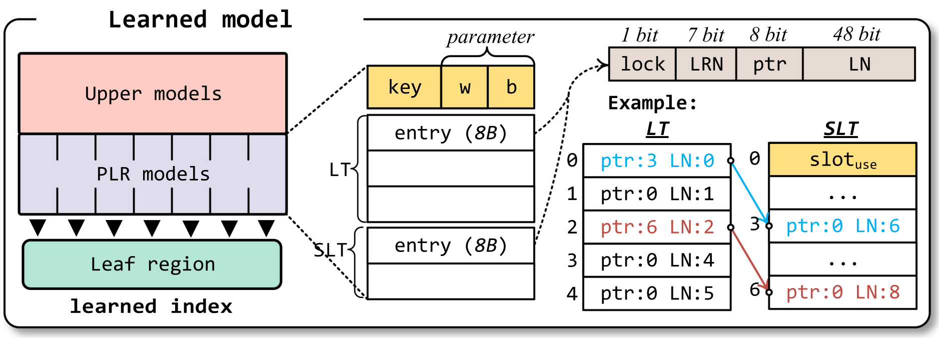


Fig. 5. The structure of the learned models.

We train multiple PLR models on the stored leaves, and each model consists of four parts, including the covered smallest key, the model parameters, a **leaf table (LT),** and a **synonym leaf table (SLT)**, as shown in Figure 5. LT and SLT store the leaf numbers (i.e., the *alloc_num* when being allocated) to access leaves. It is worth noting that different models independently record the data positions for training, which become easy to be updated since no position dependency exists among models. The obtained PLR models are indexed by training *upper models* on the smallest keys, where the upper models are represented as linear regression models that don't contain leaf tables. We repeat this procedure and construct multi-level models like PGM-index [14] due to the small space consumption, which are fully cached in the compute nodes. Moreover, we store the models with pointers, which efficiently support our shadow redirection scheme to update models, as shown in Section 3.5.

Compute pool caches indexes. The memory pool is shared across compute nodes, which supports the system scalability. Specifically, the newly added compute nodes identify the shared memory pool via the RNIC, which obtain the starting addresses of the model and leaf regions. After reading the learned models from the model region, the new compute nodes efficiently access the remote data according to the prediction range of the learned models, where the entry in the prediction range contains the leaf region number and the leaf number, thus indicating the locations of the required data in the memory pool. ROLEX processes various data requests (e.g., search, update, insert, and delete) on compute nodes with one-sided RDMA operations.

3.4 One-sided Index Operations

Simply executing data modification operations on compute nodes incurs two challenges, i.e., long latency of multiple remote operations and inconsistency issues among different machines. For example, on dynamic workloads, conflicts occur when different compute nodes write data at the same address in the memory pool, and inconsistencies occur when one node constructs new leaves while not notifying others. The 8B-atomic RDMA verbs fail to guarantee the data consistency among different machines, since the moved data during insertion are larger than 8B. An intuitive solution is to modify data leaves and LTs with locks, as well as broadcasting other nodes to synchronize their indexes after modifications. However, other nodes could not access or insert data due to the consistency requirement from the locks until the modification completes, which blocks the systems for a long time.

To address these problems, we propose a *leaf-atomic shift scheme* that provides consistency guarantees for concurrently modifying data via compute nodes while requiring few remote RDMA operations. The key insights are to atomically assign the write regions in the shared memory pool for different compute nodes and enable each compute node to access data via the stale index structure. Specifically, we first show the structures of LT and SLT that are designed for the leaf-atomic shift scheme and then respectively elaborate how different index operations coalesce with this scheme.

The structures of LT and SLT. We leverage the 8B *alloc_num* in the leaf region to enable the lock-free leaf allocations via FAA, as well as using 8B entries in LT to enable the consistent leaf modifications. The structures of LT and SLT are shown in Figure 5. The first slot in SLT is preserved to indicate how many slots ($slot_{use}$) of SLT have been used, which is modified when constructing new synonym leaves. Other slots of LT and SLT store 8B entries, each of which consists of a lock (1 bit), a leaf-region number (7 bits), a pointer (8 bits), and a leaf number (48 bits). The lock is lightweight and fine-grained due to only locking the current leaf rather than all leaves under the model. We use the leaf region and leaf numbers to determine the leaves, while the pointer points to an offset of SLT to link the synonym leaf. For example, as shown in Figure 5, the pointer of leaf 0 points to 3, indicating that leaf 0 has a synonym leaf stored in the third position of SLT, while this synonym leaf is stored in the sixth position in the leaf region. The size of LT is determined in the training phase, while the size of SLT is fixed to contain 2^8 slots. In our design, each leaf region registers up to 2^{48} leaves, while a model is able to construct up to (2^8-1) synonym leaves. It is worth noting that the max number of each field can be adjusted by specifying the bits in the entry of LT.

Point query. For a given key, the compute node searches remote data via the following steps: ① Predict L_{range} with the local learned indexes according to Equation (2). ② Translate the leaf positions into physical addresses by looking up LT and SLT. As shown in Figure 5, we look up the first through third entries in LT when L_{range} predicts $[1, 3]$, and further read the synonym leaf number in the sixth slot of SLT when the second entry points to 6. The physical address (*phy_addr*) of a remote leaf is calculated via Equation (3), i.e., multiply the leaf number (l_{num}) by the leaf size (l_{size}) and add the address of the first leaf in the leaf region (LR_{addr}). ③ Read leaves with doorbell batching according to the physical addresses. ④ Search the fetched leaves and further read the value according to the value pointer. ROLEX leverages the checksum-based schemes like existing KV stores [12, 47, 48] to guarantee the consistency of the read leaves.

The LT and SLT change when constructing new leaves in the memory pool, which is identified by the compute nodes when the first slot (i.e., $slot_{use}$) of SLT changes in the doorbell-batch reading. The compute nodes synchronize remote LT and SLT and read the new leaves for data consistency.

$$phy\_addr = l_{num} * l_{size} + LR_{addr} \tag{3}$$

Range query. A range query for $[K, N]$ requires N items starting from K. Apart from the leaves in L_{range}, ROLEX reads another (N/δ) adjacent leaves to ensure that at least N items after K are fetched. Like point query, ROLEX reads all required leaves via a doorbell batching.

Insert. ROLEX executes the insertion operation on compute nodes via the following phases:

❶ *Fetching.* The compute node (represented as C_{node}) fetches the remote leaves like point query without reading synonym leaves in this phase, since the latest synonym leaves will be fetched after acquiring the lock.

❷ *Fine-grained locking.* C_{node} determines the leaf to be inserted (represented as L_{insert}) according to the data order and locks L_{insert} by changing the lock bit of the LT entry to 1 with CAS. After locking, C_{node} reads L_{insert} and its synonym leaves to ensure that the data are up to date. The synonym leaves share the same lock with the trained leaf to enable the atomic lock. Even if L_{insert} and its synonym leaves are modified by other compute nodes before

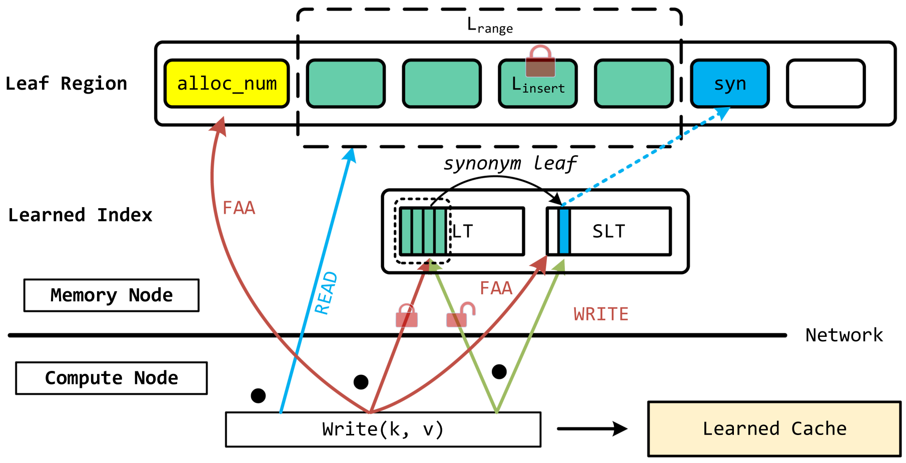


Fig. 6. The insertion operation of ROLEX.

being locked by C_{node}, inserting data into these leaves still keeps all data sorted, since the data of L_{insert} are only allowed to move within L_{insert} and its synonym leaves.

❸ *Writing and unlocking.* C_{node} inserts data into the fetched leaves according to the data order and unlocks L_{insert} via CAS.

When the fetched leaves have insufficient empty slots, C_{node} constructs a new synonym leaf as shown in Figure 6. Within one doorbell batching, C_{node} fetches and increases the *alloc_num* of the leaf region and $slot_{use}$ of SLT by 1 via FAA. Furthermore, C_{node} writes the new synonym leaf in the memory pool according to the physical address calculated by Equation (3) and inserts the *alloc_num* of the newly constructed synonym leaf into SLT at position $slot_{use}$. C_{node} also changes the pointer field of L_{insert} to the new leaf and unlocks L_{insert} via CAS.

For optimizations, other threads of C_{node} can leverage the acquired lock to modify the same leaves, and the operations of writing leaves and modifying leaf tables are completed in one doorbell batching to improve the performance.

Update. C_{node} fetches the remote leaves like point query. When the given key is matched in one of the fetched leaves, C_{node} locks and re-reads the corresponding leaf to ensure that the data are up to date. The compute node updates the key-value item and unlocks the remote leaf.

Delete. To delete the data K, C_{node} ❶ fetches and ❷ locks the remote leaves like insertion operations; e.g., C_{node} fetches the leaf L_1 and its synonym leaves L_{5-8}. When K is identified in one of the fetched leaves, e.g., L_6, C_{node} removes K in L_6, while other leaves are not modified. When L_6 becomes empty after deleting K, C_{node} removes L_6 by modifying the leaf table, i.e., changing the leaf pointers of $L5$ to $L7$ to unlink $L6$. ❸ C_{node} writes L_6 to memory nodes and unlocks the leaves. Moreover, the empty trained leaf L_1 is not removed until the next retraining to avoid the prediction error, as shown in Section 3.5. Other compute nodes identify the deleted leaf when observing that the data in the synonym leaves are not sorted, which further synchronize the leaf tables and read the remote data.

3.5 Asynchronous Retraining

The retraining overheads come from the data re-sorting and retraining algorithms [40, 49]. An intuitive solution is to conduct retraining on compute nodes, which, however, consumes a large amount of available network bandwidth for transferring the pending retraining data. Instead, we

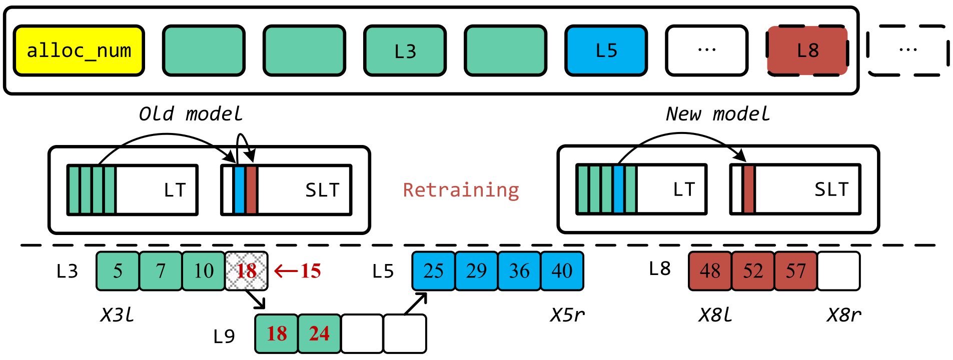


Fig. 7. The consistency guarantee of concurrent retraining.

observe that all data have been sorted by the leaf tables (i.e., LTs and SLTs) during the runtime, and the OptimalPLR algorithm has a low complexity (i.e., $O(N)$) [49] to train data, where N represents the number of the training data. Hence, ROLEX asynchronously retrains data in place on the memory nodes to achieve an efficient tradeoff between the network consumption and computing resource utilization. After offloading most index operations to the compute nodes, our experimental results show that the limited computing resources (e.g., one CPU core) on memory nodes are enough for retraining, as shown in Section 5.5.

ROLEX maintains a **circular queue (CirQ)** to identify the pending retraining models and concurrently retrains models using the shadow redirection scheme without blocking the systems. Specifically, the compute nodes insert the pointer of a model at the end of CirQ when the model consumes 2^7 slots of SLT. The memory nodes periodically check the head of CirQ for retraining, which retrains models in the background and constructs a new LT to merge the old LT and SLT, while the compute nodes concurrently access the old models. Both new and old models access the same data via their own leaf tables. After retraining, the memory nodes replace the models with consistency guarantees.

Consistency guarantee. Figure 7 shows the consistency guarantee when the memory nodes concurrently retrain the leaves L_{1-5}, where L_5 is a synonym leaf of L_3. During retraining, the compute nodes concurrently modify the data, which leads to inconsistency when the positions of the data are not retrained by the new model, e.g., (1) constructing a new synonym leaf L_8 of L_5 and (2) moving data within the synonym leaves. ROLEX ensures the data consistency by redirecting the non-retrained data into a new SLT for the new model.

(1) ROLEX identifies the newly constructed leaf (e.g., L_8) by checking the leaf tables of both old and new models, where the entry appearing in the old LT or SLT but not appearing in the new LT is identified as a non-retrained leaf. When replacing the old model with the new one after retraining, ROLEX locks the old model and inserts L_8 to the new SLT, as well as changing the model pointer to the new model before unlocking, as shown in Figure 7. Hence, the new model correctly identifies L_8 by accessing the new leaf tables, and the compute nodes correctly identify the new model by checking the model pointer. Similarly, the removed leaves are identified by checking both old and new leaf tables.

(2) ROLEX identifies the new positions of the moved data by checking the previous trained leaf. As shown in Figure 7, before the retraining begins, we respectively represent the leftmost and rightmost data in each leaf as X_l and X_r; e.g., X_{3l} represents the leftmost data of L_3. During retraining, the old model inserts the new item 15 in L_3 and inserts the items 18 and 24 into the newly constructed synonym leaf L_9. The challenge is to ensure that the new model correctly identifies

the data modified by the old model, including the trained data in the leaves (e.g., the data between X_{3l} and X_{3r}) and the new data between two sorted leaves (e.g., the data between X_{3r} and X_{5l}). According to Equation (2), the new model predicts the data between X_{3l} and X_{3r} in L_3 due to recording these data in L_3 when the retraining begins. The new model correctly identifies these modified data in the synonym leaves by checking the new SLT. However, the inconsistent state occurs for the data between X_{3r} and X_{5l} (e.g., 24), since the new model may predict these data in L_5 but overlook L_3 and L_9. To avoid such error, ROLEX checks the previous leaf (i.e., L_3) to correctly identify the modified data.

ROLEX doesn't need to re-sort or move any data for retraining, since all data have been sorted by the leaf tables during the runtime. No data are lost during retraining, since all leaves are either retrained by the new model or being inserted into the new SLT.

ROLEX inserts the new data in the synonym leaves, which triggers retraining when the synonym leaves consume half of (i.e., 2^7) the slots in the SLT. Before the retraining completes, SLT still contains the space to create 2^7 more synonym leaves to insert new keys. After retraining, the new models include new SLTs to accommodate more data. In our experiments, each leaf contains 16 slots and the model totally inserts 2,048 data before being retrained, while a model covers on average 1,465 trained data. Hence, a retraining is triggered when inserting about 1× new data more than the trained data, having a low retraining frequency. The speed of retraining models is much faster than that of filling all synonym leaves. Moreover, ROLEX has a priority queue to identify and train the model with almost full SLT to avoid the scenario where a model has insufficient slots in SLT.

3.6 Discussion

Scalability. ROLEX distributes large datasets across multiple memory nodes by constructing multiple leaf regions. Specifically, 2^7 leaf regions form a *group* and each region contains at most 2^{48} leaves to store data. A leaf group hence contains 2^{55} leaves and is sufficient to construct a large number of learned models. By training data in the same group, the learned models become efficient to determine the location of a leaf via the leaf region (7 bits) and leaf numbers (48 bits) of the entry in LT and SLT. Moreover, ROLEX constructs multiple groups to scale across multiple memory nodes and becomes efficient to accommodate a large amount of data.

Durability and fault tolerance. Existing disaggregated memory systems enable the durability and fault tolerance in different ways, such as the persistent memory [42, 53], battery backup system [12], and logging writes [47], while achieving efficient performance. All these solutions are orthogonal to ROLEX for efficient durability and fault tolerance.

Emerging heterogeneous technology. ROLEX benefits from the technology integrating emerging accelerators and specialized hardware into the disaggregated memory nodes [17], due to the sufficient computing resources. Moreover, the powerful network technology [34] incurs low network penalty on remote data accessing. In this case, ROLEX needs a fallback mechanism to avoid the lock contention among many compute nodes, which is our future work.

4 IMPLEMENTATION DETAILS

In this section, we present the implementation details of ROLEX in the disaggregated memory systems, including the initial training and asynchronous retraining phases on the memory nodes, and the one-sided search, insert, update, and delete operations on the compute nodes.

The typical memory pools [17, 46] have limited computing resources to conduct some non-performance-critical operations. In ROLEX, the memory pool registers memory with huge pages during the initialization phase to avoid the penalty of the cache misses in page translation. The registered memory consists of the model and leaf regions to respectively maintain the learned

```
ALGORITHM 1: TRAIN(dataset)
// Add a range of data that has the same linear distribution pattern
subdataset = [];
pos = 0;      // The logical position of data starting from 0
// Sequentially train data in the training dataset
for each key in dataset   // Dataset contains all the training data
    // Check if the added key violates the linear distribution pattern
    // Successfully add one data and continue to process the next one
    if subdataset.AddData(key, pos++)==SUCCESS
        continue;
    // Otherwise, obtain the trained model and clear the subdataset for next training
    else
        model = subdataset.TrainModel();
        [slope, intercept] = model.GetSlopeIntercept();
        // respectively store the model and data in the model and leaf regions
        StoreModel(slope, intercept, subdataset);
       // Train the next model
        pos = 0;
        subdataset.Clear();
// Store the last trained model and the covered training data
model = subdataset.TrainModel();
[slope, intercept] = model.GetSlopeIntercept();
StoreModel(slope, intercept, subdataset);
return SUCCESS;
```

Fig. 8. The pseudo-code of training models.

models and data. Existing RNIC hardware does not support remote memory allocation [56], and we hence pre-allocate memory for the leaf regions to support our proposed atomic-leaf shift scheme. We initially store the training data and the trained models on the memory nodes. The compute nodes connect to the memory pool using the RDMA QC pairs and then synchronize the learned index structures from the memory nodes to further conduct index operations.

4.1 Train Models

ROLEX stores all data in the leaves on the memory nodes and trains models based on these leaves. Figure 8 presents the pseudo-code of the improved OptimalPLR training phase in ROLEX. The improved OptimalPLR algorithm divides all data into non-overlapping subdatasets and ensures that the linear regression model trained from each subdataset is smaller than the predefined prediction error ϵ.

The improved OptimalPLR algorithm sequentially trains data, which adds each data into the subdataset until identifying one data violating the linear distribution pattern of the subdataset. The linear distribution pattern is determined by obtaining a linear regression model on the sorted keys and logical positions, where the logical position is the order number of each data in the subdataset. The data belong to the same linear distribution pattern, indicating that the distances between the logical positions of these data and the positions calculated by the obtained linear regression model are smaller than the predefined threshold. The subdataset successfully adds one data only when the prediction error of the obtained model is smaller than ϵ. Otherwise, ROLEX stores the data of the subdataset into the fixed-size leaves pre-allocated by the RNIC and maintains the leaf numbers of the involved leaves into the **leaf table (LT)** for the model. All leaves have δ slots to accommodate

```
ALGORITHM 2: SEARCH(K)
// Search and determine the model covering the given key
model = ModelForKey(key);
// Show a prediction range for the given key
predict_range =  model.Predict(key);
remote_addresses = leaf_table.Translate(predict_range);
// Remotely read the data leaves
leaves = RdmaRead(remote_addresses);
if HaveNewSynonymLeaf(leaves)==SUCCESS
    leaves.Append(RdmaRead(leaves.new_synonym_leaf));
// Locally search the data
for each leaf in leaves
    if leaf.Find(key)==SUCCESS
        return leaf.ObtainValue(key);
return NULL;
```

Fig. 9. The pseudo-code of the search operation.

new data. Moreover, the improved OptimalPLR algorithm clears the subdataset and continues to train the subsequent data with the same training mechanism.

The obtained models are stored in the model area pre-allocated by RNIC, and the model area is efficiently shared within the compute pool. The compute nodes synchronize the learned indexes from the model area and leverage the cached index structures to access and modify the remote data.

4.2 Search

Figure 9 presents the pseudo-code of the one-sided search operation on the compute nodes. To access the value of the given key, the compute node leverages the cached learned models to calculate the physical position and obtains the data from the remote memory nodes using one-sided `RDMA READ` verbs.

Specifically, the compute node searches the cached learned index structure to determine the model that covers the given key. In ROLEX, each key only corresponds to one model, since different models cover different ranges of data, and the corresponding model is determined by comparing the covered ranges. The model shows a prediction range for the given key using Equation (2) and then translates these logical positions into physical ones by checking the leaf tables, where the physical positions are calculated via Equation (3), i.e., multiply the leaf number by the leaf size and add the start address of the first leaf in the leaf region. Finally, the compute node obtains the remote data from the calculated physical positions using one-sided RDMA verbs and locally searches the request data from the obtained data. At the same time, the compute node identifies the newly constructed synonym leaves when obtaining synonym pointers from the predicted leaves.

The range query request $SCAN(K, N)$ needs to search N data starting from K. In this case, the compute node leverages the cached learned indexes to search the data K like Algorithm 2 and reads N subsequent data followed by K in the `RDMA READ` phase to deliver high-range query performance due to one RTT.

4.3 Insert

Figure 10 presents the pseudo-code of the one-sided insert operation on the compute nodes, which inserts a new item $< K, V >$ into the leaf predicted by the cached learned indexes in three phases, as shown in Section 3.4.

```
ALGORITHM 3: INSERT(K, V)
// First, remotely read the predicted leaves for the given key
remote_addresses = Search(K);
leaves = RdmaRead(remote_addresses);
// Second, lock and re-read L_insert and the synonym leaves
[L_insert, synonym_leaves] = LocalSearch(leaves, K);
if Contains(L_insert, K)==True
    return FAIL;
Lock(L_insert);
leaves = RdmaRead(L_insert, synonym_leaves);
L_insert = LocalSearch(leaves, K);
if isFull(L_insert)==True
    [alloc_num, slot_use] = RdmaFAA();
// Third, insert the data and write the modified data to the memory pool
InsertData(L_insert, K, V);
if CreateSynonym()==True
    UpdateLeafTable(alloc_num, slot_use);
RdmaWrite(L_insert);
Unlock(L_insert);
return SUCCESS;
```

Fig. 10. The pseudo-code of the insert operation.

First, the compute node remotely reads the predicted leaves via one-sided *Search*() function. The synonym leaves are not read in this phase, since the latest data is read in the second phase.

Second, the compute node locally searches the obtained leaves and determines which leaf to be inserted (represented as L_{insert}) by identifying the data range that covers the new key K. To avoid the inconsistency issue, the compute node locks L_{insert} by changing the lock bit to 1 via CAS and re-reads the data in L_{insert} and the synonym leaves in one doorbell batching for a small network penalty. When the fetched leaves have insufficient available slots to accommodate new data, the compute node fetches and increases the *alloc_num* of the leaf region and $slot_{use}$ of SLT by 1 via the `RDMA FAA` operation. In this case, the compute node constructs a new synonym leaf to insert new data, and the new leaf is inserted into SLT in the third phase.

Third, the compute node inserts the new item $< K, V >$ into the leaf according to the data order and writes the modified leaf into the memory nodes via the one-sided `RDMA WRITE` operation. At the same time, the compute node inserts the newly constructed synonym leaf into SLT and completes the insert operation by unlocking L_{insert} via CAS.

4.4 Update

Figure 11 presents the pseudo-code of the one-sided update operation on the compute nodes, which updates the item of $< K, V >$.

The compute node conducts the *Search*() operation to fetch the remote leaves to check if the request K exists. When the *Check*() function returns NULL, the compute node returns false to finish the update operation, since the target key does not exist in ROLEX. For the leaf (represented as L_{update}) containing the target key, the compute node changes the lock bit of L_{update} to 1 via CAS and re-reads the data in L_{update} to guarantee the data consistency. Finally, the compute node writes the new V into the memory nodes and unlocks L_{update}.

```
ALGORITHM 4: UPDATE(K, V)
// Remotely read the predicted leaves for the given key
remotea_ddresses = Search(K);
leaves = RdmaRead(remote_addresses);
// Check if the request K exists
if Check(K)==NULL
    return FAIL;
// Update V corresponds to the given K
L_update = LocalSearch(leaves, K);
Lock(L_update) and RdmaRead(L_update);
UpdateData(L_update, K, V);
RdmaWrite(L_update) and Unlock(L_update);
return SUCCESS;
```

Fig. 11. The pseudo-code of the update operation.

```
ALGORITHM 5: DELETE(K)
// Read remote leaves and check the existence of the given K
remote_addresses = Search(K);
leaves = RdmaRead(remote_addresses);
if Check(K)==NULL
    return FAIL;
// delete V corresponds to the given K
L_delete = LocalSearch(leaves, K);
Lock(L_delete) and RdmaRead(L_delete);
DeleteData(L_delete, K);
RdmaWrite(L_delete) and Unlock(L_delete);
return SUCCESS;
```

Fig. 12. The pseudo-code of the delete operation.

4.5 Delete

Figure 12 presents the pseudo-code of the one-sided delete operation on the compute nodes, which removes the data matching with the given K.

Like the previous operations, the compute node conducts the following operations in sequence: searches K from remote leaves, checks the existence of the given K, locks and re-reads the leaf (represented as L_{delete}) containing K, and deletes the data in L_{delete}. When L_{delete} becomes empty after removing K, the compute node processes the empty leaves in different ways according to the types of the leaves. The empty trained leaf is not removed from the leaf table until the next retraining to avoid the prediction error, while the non-retrained empty synonym leaf is removed by deleting the leaf pointer. At the same time, the leaf numbers of the removed leaves are recorded in the reallocating area for reuse.

4.6 Retrain

When constructing 2^7 synonym leaves, the compute node inserts the model pointer into *CirQ* to wait for asynchronous retraining. At the remote side, the memory pool assigns retraining threads to check the *CirQ* and concurrently retrain models in place, as shown in Figure 13.

```
ALGORITHM 6: RETRAIN()
// Check CirQ and concurrently retrain the models in place
while running
    if CheckCirQ()==NULL
        continue;
    RM = CirQ.head;
    newLT = ConstructLeafTable(RM.oldLT, RM.oldSLT);
    model = Retrain(RM, newLT);
    // replace the old model with the newly trained one
    Lock(RM);
    Update(model.SLT, RM.SLT);
    RM.pointer = model;
    Unlock(RM);
return SUCCESS;
```

Fig. 13. The pseudo-code of asynchronous retraining models.

The memory pool sorts and retrains the data covered by the pending retraining model (represented as *RM*). Instead of moving all data into a new continuous area, the memory pool keeps all data in place and constructs a new LT to re-sort the leaves in the old LT and SLT. Since all data move within the assigned leaves following the proposed data movement constraints, the compute nodes concurrently access and modify the leaves with consistency guarantees, as shown in Section 3.5. After obtaining a new model trained from the new LT, the memory pool locks *RM* and inserts the synonym leaves constructed during the retraining phase into the new SLT to guarantee that the new model correctly identifies all leaves. Finally, the memory pool successfully retrains the model by replacing the model pointer with the newly retrained one and unlocking *RM*.

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

We run all experiments on a cluster with three compute nodes and three memory nodes, and each server node is equipped with two 26-core Intel(R) Xeon(R) Gold 6320R CPUs @2.10Ghz, 188GB DRAM, and one 100Gb Mellanox ConnectX-5 IB RNIC. The RNIC in each machine is connected with a 100Gbps IB switch. We limit the computing resources utilization (i.e., one CPU core in our testbed) for the memory node, which is reasonable because of the limited computing capability in the typical memory pools [17, 46]. All compute nodes run with 24 threads by default.

Workloads. We use YCSB [50] with both uniform and Zipfian request distributions to evaluate the performance, which contains six default workloads, including (A) update heavy (50% updates), (B) read mostly (95% read), (C) read only, (D) read latest (5% insert), (E) short ranges (95% range request), and (F) read-modify-write (50% modifications). Apart from these workloads, we also evaluate the performance under write-intensive requests with two real-world and two synthetic datasets [24]. Among them, *Weblogs* and *DocID* respectively contain 200 and 16 million key-value pairs with different data distributions. The two synthetic datasets contain 100 million items and respectively meet the normal and lognormal data distributions. We configure all workloads with 8B keys and pointers (i.e., referring to variable-length values) like existing schemes [24, 47] for comprehensive evaluations.

Counterparts for comparisons. We compare ROLEX with four state-of-the-art distributed KV stores. Specifically, FG [55] and Sherman [46] design RDMA-enabled B-link trees for the

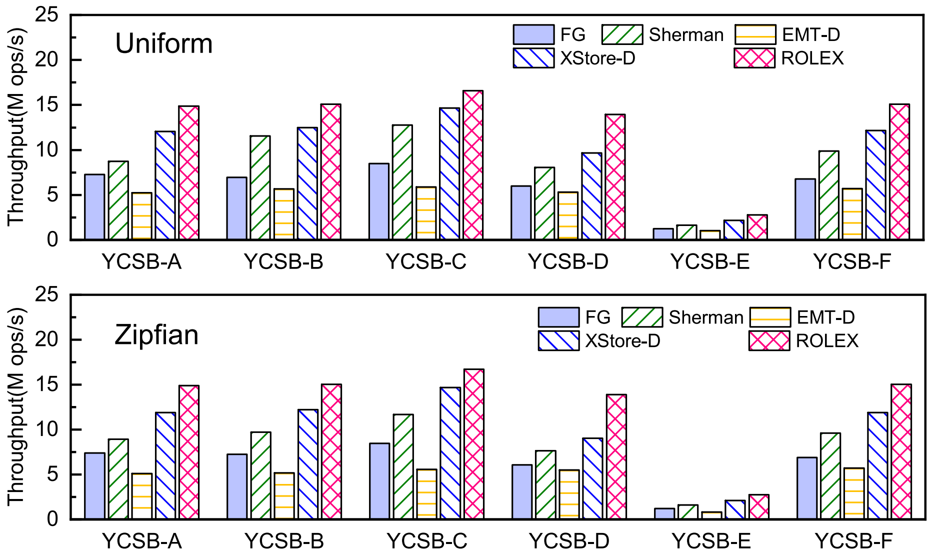


Fig. 14. The throughputs on various YCSB workloads.

disaggregated memory systems. We directly run the source codes of Sherman. Since FG is not open source, we implement FG from scratch faithfully following the original design principles, as well as caching the top-level nodes on compute nodes for better performance. We also adopt the similar ideas of EMT-D [23] and XStore-D [47] on the disaggregated systems, i.e., using the limited computing resources of memory nodes to show the performance of RPC-based schemes. EMT-D transfers all requests to memory nodes via eRPC (RDMA-based RPC), while XStore-D accesses read-only workloads via compute nodes and relies on memory nodes to process write-intensive requests. We configure our implemented ROLEX with 16 slots in each leaf, as well as setting 16 as the maximum model error to train PLR models for efficient system performance. We further leverage one CPU core on the memory node and disable the garbage collection and durability functions for all counterparts to facilitate fair comparisons.

5.2 Overall Performance in YCSB

Figure 14 shows the throughputs on various YCSB workloads with both Uniform and Zipfian distributions. In general, ROLEX achieves competitive performance with XStore-D on static workloads while achieving higher throughput on dynamic workloads due to not relying on remote CPUs.

Static workload (YCSB C). On the static workloads, XStore-D and ROLEX efficiently read remote data via one `RDMA READ` according to the prediction results of the learned models, which achieve higher performance than FG and Sherman due to fewer RTTs caused by the local cache. EMT-D achieves the lowest throughput, since the memory nodes have insufficient computing resources to process the data requests. ROLEX achieves higher performance than XStore-D due to the high model accuracy. Specifically, ROLEX leverages the OptimalPLR algorithm [49] to train models according to the data distributions, which guarantees that all model errors are smaller than the predefined threshold. However, XStore-D leverages the recurve model index scheme [24] for training and fails to adaptively train models according to the data distribution. Some model errors are large when failing to train sufficient models, causing a large prediction range and lower performance than ROLEX in the read-only workloads.

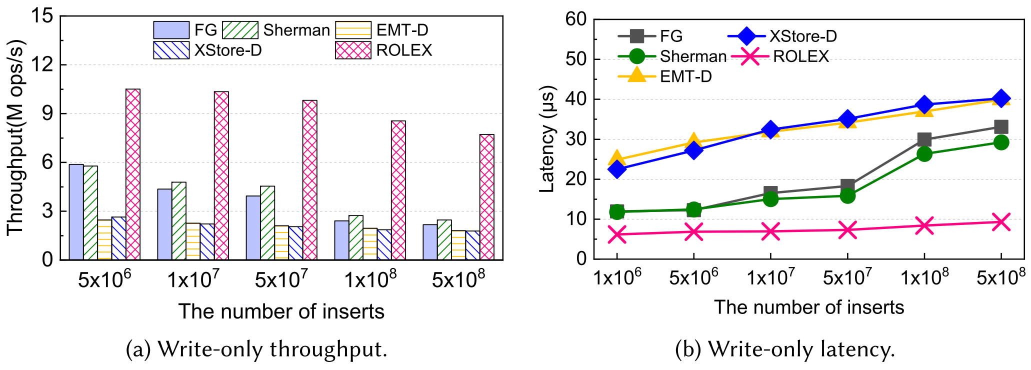


Fig. 15. The performance on write-only workloads.

Read-write workloads (YCSB A, B, D, F). For data modifications, both XStore-D and EMT-D transfer data requests to the remote side and achieve low throughput, due to the limited CPU cores on memory nodes. The performance of FG and Sherman is limited by the local cache due to the large memory footprint of inner nodes. ROLEX achieves higher performance than other schemes due to exploiting the learned local cache with the efficient one-sided `RDMA WRITE`. Specifically, ROLEX outperforms FG, Sherman, EMT-D, and XStore-D by up to 2.1×, 1.7×, 2.8×, and 1.3×, respectively, on workload A, since ROLEX directly updates the remote data without involving remote CPUs. For workload D, 5% insertions are mixed with 95% searches, and ROLEX improves the throughput by about 1.5× over other schemes. The reason is that the caches of other schemes become invalid during insertion, while ROLEX leverages the stale cache to write data in synonym leaves. We obtain similar observations on workloads B and F.

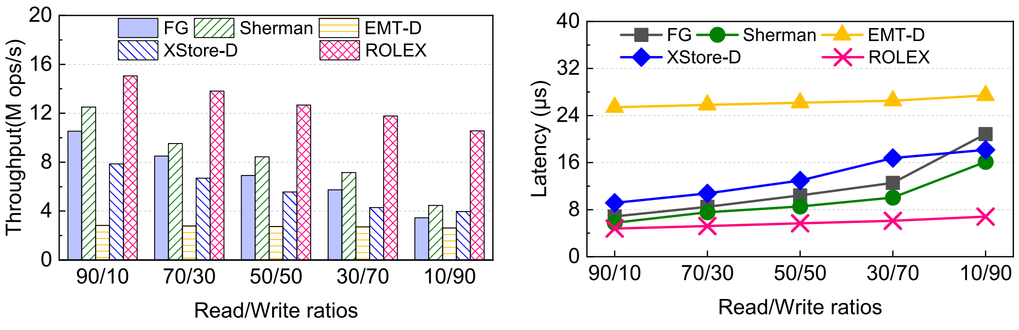
Range-query workload (YCSB E). Workload E contains 95% range query and 5% insert requests. We observe that ROLEX improves the performance by 67% over other schemes, since all data are kept sorted in the synonym leaves during insertion and the range queried data are fetched in a doorbell batching by `RDMA READ`.

5.3 Performance in Various Scenarios

Apart from YCSB, we have similar observations on other representative workloads, including Weblogs, DocID, Normal, and Lognormal. Figures 15 and 16 show the performance of different schemes in various scenarios.

Throughput with intensive writes. Figure 15(a) shows the throughput of inserting different numbers of data. As we constantly insert data, ROLEX achieves significant performance improvements over other schemes. Specifically, ROLEX improves the insert throughput by up to 2.1×, 1.8×, 4.5×, and 4.3× over FG, Sherman, EMT-D, and XStore-D, respectively. The main reason is that the local cache is fully exploited by ROLEX with one-sided RDMA operations, while the footprints of inner nodes in tree-based schemes overflow the cache and the remote CPUs limit the write performance of RPC-based schemes. Moreover, we evaluate the latencies of the insert operations for different schemes, and the results are shown in Figure 15(b). We observe that ROLEX incurs low latency since the stale cache identifies the leaf to be inserted according to the prediction results of the learned models. For the monotonically increasing keys, ROLEX shows low performance when multiple compute nodes contend for the same leaf lock, which is alleviated by sharing the leaf lock among multiple threads of the same compute node.

Performance with hybrid read-writes. Figures 16(a) and 16(b) respectively show the throughput and latency under various read/write ratios. The performance of EMT-D doesn't decrease



(a) Hybrid read/write throughput. (b) Hybrid read/write latency.

Fig. 16. The performance with various read/write scenarios.

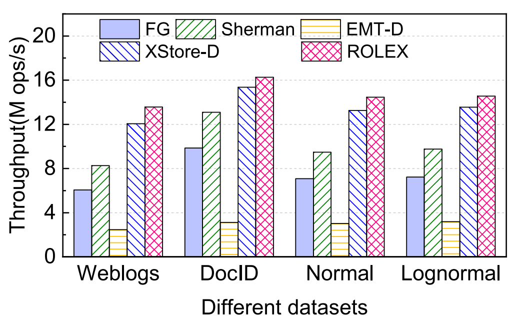
much with the increasing write ratios, since the remote memory nodes suffer from the bottleneck of insufficient computing resources and achieve low performance even under intensive read requests. XStore-D achieves high performance on read-heavy workloads, while significantly decreasing the performance as the write ratio increases, because XStore-D reads data with one-sided RDMA while transferring most data requests to the remote side as the number of write requests increases. ROLEX, FG, and Sherman achieve higher performance than other schemes due to not being limited by the remote CPUs. ROLEX improves the throughput by 2.2× and 1.7× over FG and Sherman, since the improvements mainly come from the efficient learned local cache. FG and Sherman have to spend multiple RTTs on retrieving the remote data when the inner nodes overflow the limited local cache.

The latency of ROLEX is lower than that of RPC-based schemes in the disaggregated memory systems, since the latency of accessing remote data comes from the network roundtrip and the index structure traversal. ROLEX traverses the cached learned indexes via the compute nodes, while RPC-based systems traverse the index structures via the memory nodes. In the disaggregated memory systems, the compute nodes have sufficient computing resources to support high concurrent access, while, however, the memory nodes have limited computing resources and fail to meet the requirements for processing intensive index requests.

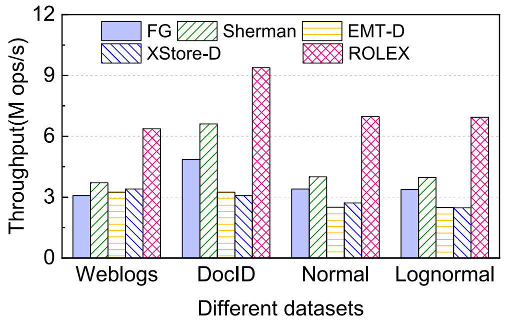
Performance with various data distributions. The data distributions impact the model accuracy of the learned indexes, which decrease the performance when the learned models deliver low accuracy. Figure 17 shows the throughput on various workloads with different data distributions, including Weblogs, DocID, Normal, and Lognormal. We observe that ROLEX achieves higher read performance than XStore-D. The main reason is that the improved OptimalPLR algorithm trains independent linear regression models with high accuracy according to the data distributions.

Performance on skewed workloads. The access patterns of requests may become skewed, e.g., reading/writing data in a certain range, rather than following the same data distributions as training. To show the effects of the skewed workloads, we define the *Hotspot Ratio* as the accessed range divided by the range of the trained data, where a small hotspot ratio represents a large skewness since a small data range is accessed. Figure 18(a) shows the read throughputs of different schemes on the skewed workloads. We observe that ROLEX delivers higher performance than other schemes even with a small hotspot ratio, since ROLEX efficiently leverages the learned models to access the remote data via the one-sided operations. However, the tree-based schemes deliver low performance due to incurring multiple network roundtrips.

Figure 18(b) shows the write throughputs of different schemes on the skewed workloads. We observe that ROLEX delivers low performance with the small hotspot ratio, since different compute

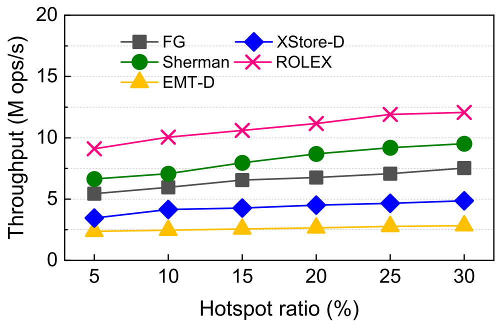


(a) Read throughput.

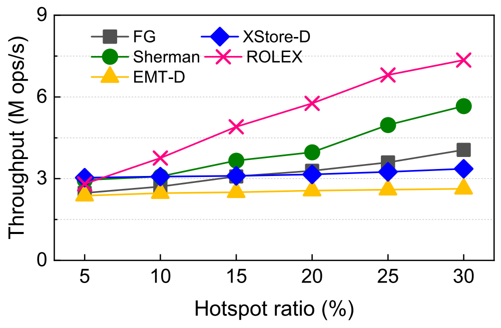


(b) Write throughput.

Fig. 17. The performance under various data distributions.



(a) Read throughput.



(b) Write throughput.

Fig. 18. The performance on skewed workloads.

nodes compete for the same data leaf. However, ROLEX increases the performance when the hotspot ratio becomes large, since the improved OptimalPLR algorithm trains independent linear regression models according to the data distributions, which alleviates the contentions for the data leaves among different compute nodes. Therefore, the compute nodes conduct efficient one-sided RDMA operations to access and modify the remote data.

For the extremely skewed workloads, the SLTs are completely filled and wait for the retraining phase. In this case, ROLEX splits the model by copying model parameters and constructing new SLTs to contain the skewed data, which is our future work.

5.4 Scalability Performance

Figure 19 shows the throughput of various schemes with different numbers of cores on the compute nodes. We observe that the performance of EMT-D doesn't increase when configuring more cores on compute nodes, since the bottleneck of EMT-D is the remote CPUs of memory nodes, rather than the compute nodes. The throughputs of other schemes increase with the number of cores on compute nodes, as shown in Figure 19(a), because FG, Sherman, XStore-D, and ROLEX don't rely on the remote CPUs to process the read requests. However, the write performance of XStore-D fails to scale out with the number of cores on compute nodes, as shown in Figure 19(b), since XStore-D quickly runs out the available computing resources of the memory nodes. The read and

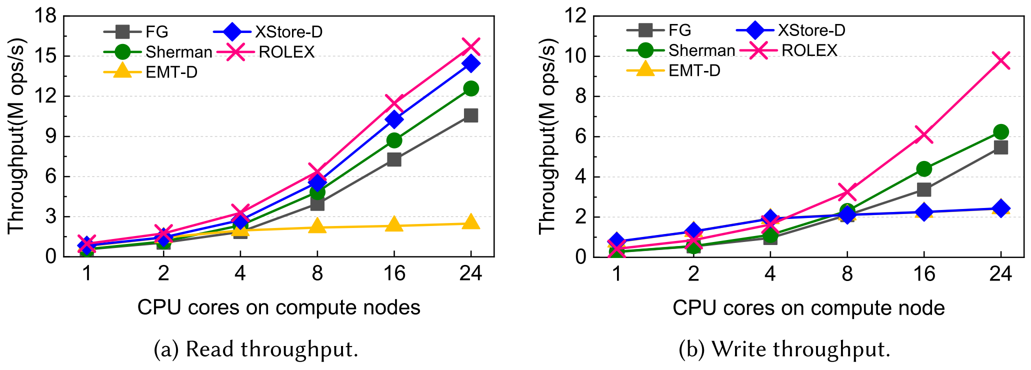


Fig. 19. Scalability with various CPUs on compute nodes.

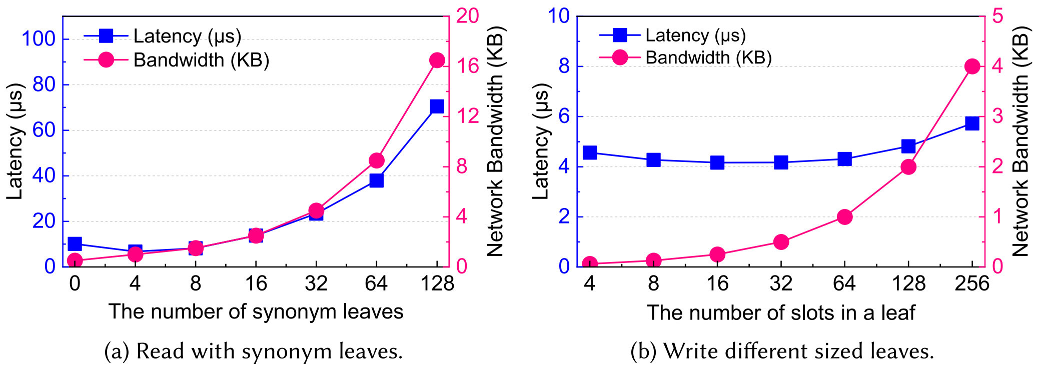


Fig. 20. In-depth analysis. *We evaluate the latency and network bandwidth consumption when reading/writing different numbers of data.*

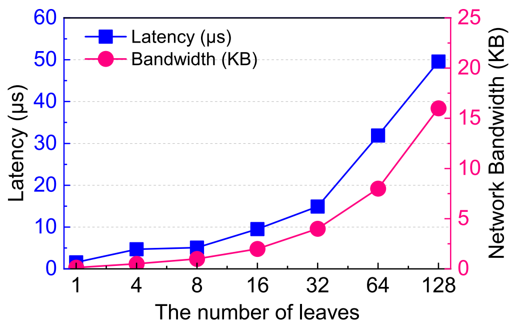
write performance of ROLEX increases with the increasing number of cores on compute nodes, since different threads don't block each other.

If the disaggregated memory system is not assumed, in our evaluation, EMT-D and XStore-D achieve higher performance than other designs when configuring the memory nodes with more than 20 CPU cores, since 20 CPU cores in memory nodes meet the requirements of processing various index operations. However, it is worth noting that our article mainly focuses on the disaggregated memory systems, which generally configure limited computing resources (i.e., much lower than 20 CPU cores) on the memory nodes.

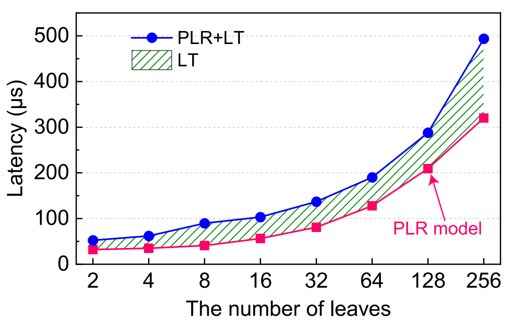
5.5 In-depth Analysis

We conduct three optimizations in ROLEX, including operation decoupling, one-sided indexing, and asynchronous retraining, which efficiently support the system to obtain high performance. We evaluate the efficiency of different optimizations in Figures 20 and 21.

Operation decoupling. An important insight of ROLEX is that we decouple the insertion and retraining operations to enable the compute nodes to directly insert data into the memory pool, which leverages the stale models to identify the new data. As shown in Figures 20 and 21, although retraining incurs long latency, ROLEX achieves low latency to read and write remote data, since the



(a) Write multiple leaves.



(b) The latency of training leaves.

Fig. 21. In-depth analysis. *We evaluate the latency and network bandwidth consumption when inserting multiple leaves, as well as the latency when training leaves.*

operation decoupling moves the retraining phase out of the critical path and enables the compute nodes to insert data without waiting for the retraining.

One-sided indexing. The compute nodes access remote data via one-sided indexing, which incurs low latency and bandwidth consumption when operating on a small range of data, since one-sided indexing efficiently exploits the benefits of RDMA doorbell batching. We observe that ROLEX achieves high performance when respectively setting ϵ and δ to [8, 256] and [8, 128], which achieves an efficient tradeoff between the accessing efficiency and the retraining frequency. Specifically, ϵ and δ respectively represent the maximum prediction error and the leaf size. As shown in Figures 20(a) and 21(a), a large ϵ provides a large prediction range, which consumes much network bandwidth and latency to identify the requested data. ROLEX achieves high performance when reading/writing 8 to 256 data, where the number of data is calculated by multiplying the size and the number of the leaves. Moreover, the small δ provides small-size leaves, which frequently triggers retraining since the leaves have insufficient slots to accommodate new data. However, as shown in Figure 20(b), too large δ consumes much network bandwidth for modifying remote data, since ROLEX reads/writes data in the granularity of a leaf.

Asynchronous retraining. ROLEX asynchronously retrains the models to construct new models and leaf tables, which increases the model accuracy to read and write few leaves. As shown in Figure 21, the operations upon a small number of leaves significantly reduce the latency and network bandwidth consumption. Figure 21(b) shows the retraining latency using a single CPU core. We observe that training models and constructing leaf tables on 128 leaves consume about 300μs. Unlike conventional learned indexes [10, 14, 40], ROLEX doesn't need to move or re-sort any data during retraining, since all data are kept sorted during data modifications.

Co-routine optimization. The compute nodes use co-routines to conduct the index operations of ROLEX to improve the throughput of handling requests. To demonstrate the efficiency of co-routines, we initially load 1 million data on the memory nodes and train models on the data leaves, and then respectively conduct 1 million search, update, insert, and delete index operations with different numbers of co-routines on the compute nodes. Figure 22 shows the results of different index operations when we constantly increase the number of co-routines from 1 to 16. From these results, we observe that the throughput increases 2.3× on average when using multiple co-routines, since multiple co-routines enable the compute nodes to concurrently process the data requests. We also observe that the compute nodes achieve the highest performance when configuring eight co-routines, and the performance gradually stops increasing when configuring more co-routines. The

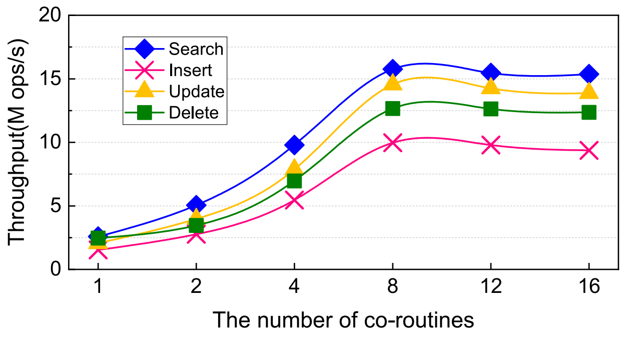
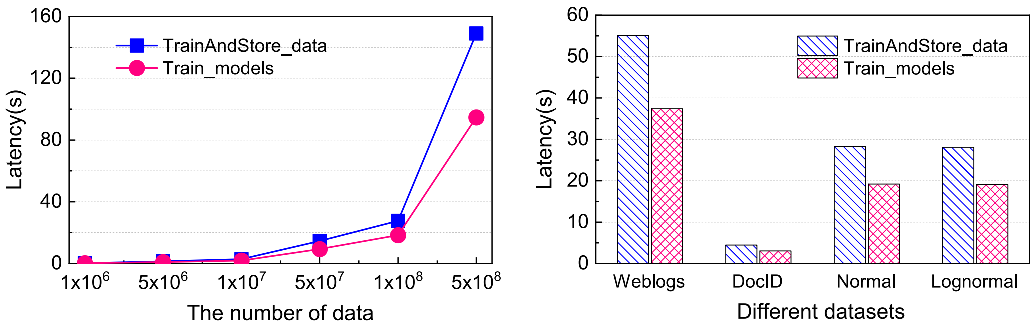


Fig. 22. The throughputs with different numbers of co-routines.



(a) Different numbers of data. (b) Different datasets.

Fig. 23. The training overhead. *ROLEX conducts the training operation offline to avoid the long latency and asynchronously retrains a small number of data for the high model accuracy.*

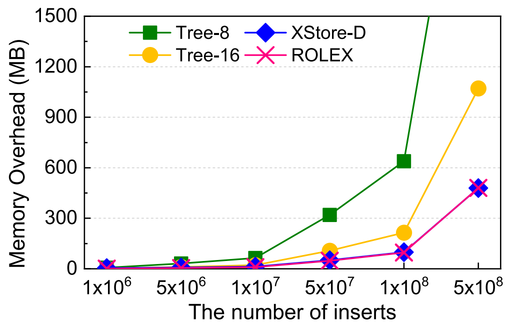
main reason is that the CPU on the compute node runs out before being able to process the massive data requests and becomes the bottleneck of the entire system.

5.6 Overhead Analysis

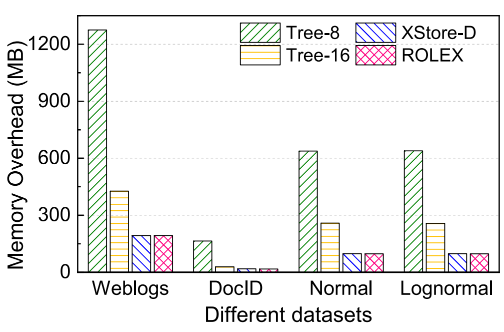
In this section, we evaluate and analyze the overheads of ROLEX, including the training latency and the memory consumption.

Training latency. Figure 23 shows the latency of training different numbers of data. We observe that the training latency proportionally increases with the number of training data, since the compute overhead comes from the training algorithm with $O(N)$ complexity, where N represents the number of training data. On average, ROLEX spends 0.28μs on training one data to obtain the trained models and store the data in the leaves. Unlike the training phase, ROLEX asynchronously retrains the models in the *CirQ* without moving the data in the leaves, which consumes about 0.19μs on retraining one data on average.

Figure 24 shows the memory footprints of the metadata in different schemes, where the metadata refer to the data that are required for caching. For example, the metadata consist of the inner nodes for the tree-based schemes, while consisting of trained models and leaf tables for XStore-D and ROLEX. We observe that the memory overheads in tree-based structures rapidly increase with the increasing data, because many levels of inner nodes are constructed for indexing. Moreover, the metadata overheads significantly increase when using small inner nodes due to requiring more



(a) Different numbers of data.



(b) Different datasets.

Fig. 24. The memory footprints of the metadata. *Tree-# represents that an inner node contains # keys.*

Table 1. The Metadata Analysis for ROLEX

Number of Data	$5 * 10^6$	$1 * 10^7$	$5 * 10^7$	$1 * 10^8$	$5 * 10^8$
Number of Models	5,153	10,283	51,111	101,936	526,236
Size of Models (MB)	0.0798	0.157	0.779	1.555	8.03
Size of LT (MB)	4.768	9.537	47.683	95.367	476.837

levels. Unlike tree-based structures, XStore-D and ROLEX leverage the linear regression models for indexing, and each model only contains two parameters and is much smaller than the inner nodes. As shown in Table 1, the memory overhead of ROLEX mainly comes from the LTs, which account for 98% of the total memory consumption. These models can be fully cached by the compute nodes, while the LTs can be fetched as needed when the limited cache fails to maintain all LTs.

6 RELATED WORK

The disaggregated memory systems. The promising disaggregated memory systems [29, 35, 36, 41, 45, 55] break a monolithic server into independent components to enhance the hardware scalability, which achieves high resource utilization by scaling out different hardware components [16, 52]. Different components communicate with each other via efficient RDMA techniques [4, 5, 19, 39]. Existing academic studies attempt to bring the disaggregated memory systems into practice via hardware designs [29, 30]. Clio [17] proposes a hardware-software co-designed disaggregated memory system to equip each memory node with dedicated computing resources. LegoOS [36] proposes an OS model to manage disaggregated systems. Remote regions [1], LITE [43], and Semeru [45] are used to efficiently manage the remote memory resources. AIFM [34] designs a simple API for applications to use the remote memory. With the widely used NVM [31, 37, 51], Clover [42] remotely manages the persistent memory with low costs. FORD [53] enables the disaggregated memory systems to efficiently support transactions.

Learned indexes for storage systems. The *learned indexes* [24] leverage calculations to predict positions for the given keys. Prior designs focus on various scenarios to enable the learned indexes to be widely used, including dynamically adapting to new data distributions [10, 14, 15], concurrent systems [40], and LSM-based [9] and network-attached [47] KV stores. Motivated by the learned indexes, some studies leverage machine learning models to construct learned systems; e.g., DeepDB [18] proposes a learned DBMS, Tsunami [11] constructs learned multi-dimensional indexes, and LISA [27] learns spatial data for high performance. Hai et al. [25] comprehensively evaluate the performance of tree-based and learned indexes on disks, which provide valuable guidelines to design efficient disk-based indexes. DILI [28] constructs a novel distribution-driven learned index tree for in-memory one-dimensional search keys.

Network-attached key-value stores. Due to the salient features of RDMA [4, 35, 39, 52], constructing RDMA-enabled in-memory key-value stores [23, 33, 47, 55] becomes efficient for distributed storage systems. Existing studies rely on two-sided RDMA verbs to process the data requests [6, 21, 23]. However, such server-centralized designs suffer from the CPU bottleneck when processing intensive requests [22, 47, 48] due to the poor computing capability of memory nodes. Unlike them, one-sided RDMA enables compute nodes to directly access the remote data without involving remote CPUs [13, 42, 56]. For the ordered KV stores, Cell [33], FG [55], and Sherman [46] cache top-level nodes to reduce the number of RTTs based on B-link trees [26]. XStore [47] proposes a learned cache to further reduce the network penalty, which incurs one RTT to access the remote data. Moreover, FUSEE [38] efficiently disaggregates the metadata management to achieve both high performance and resource efficiency. Unlike them, we design ROLEX for the disaggregated memory systems to efficiently process various requests via one-sided RDMA operations.

7 CONCLUSION

This article proposes ROLEX, a scalable RDMA-oriented ordered key-value store using learned indexes for the disaggregated memory systems. ROLEX decouples the insertion and retraining operations, which enables the compute nodes to directly modify the remote data without retraining models. Other compute nodes identify the newly modified data via the stale models with consistency guarantees. ROLEX asynchronously retrains modes to improve the model accuracy. Our evaluation results demonstrate that ROLEX achieves high performance on both static and dynamic workloads in the context of the disaggregated memory systems. We have released the open-source codes for public use in GitHub.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their valuable comments.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: A simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC'18)*. 775–787.

[2] Amazon. 2021. Amazon Elastic Block Store. https://aws.amazon.com/ebs/?nc1=h_ls

[3] Amazon. 2021. Amazon s3. https://aws.amazon.com/s3/

[4] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.* 9, 7 (2016), 528–539.

[5] Amanda Carbonari and Ivan Beschasnikh. 2017. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets'17)*. 164–170.

[6] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys'19)*. 19:1–19:14.

[7] Douglas Comer. 1979. The ubiquitous B-tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[8] Intel Corporation. 2021. Intel Rack Scale Design Architecture. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

[9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 155–171.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*. 969–984.

[11] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.
[12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.
[13] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 54–70.
[14] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.
[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. 1189–1206.
[16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 249–264.
[17] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.
[18] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.
[19] HP. 2021. The Machine. https://www.hpl.hp.com/research/systems-research/themachine/
[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*. 187–200.
[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference (SIGCOMM'14)*. 295–306.
[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 185–201.
[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 1–16.
[24] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. 489–504.
[25] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable learned indexes meet disk-resident DBMS - From evaluations to design choices. *Proc. ACM Manag. Data* 1, 2 (2023), 139:1–139:22.
[26] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (1981), 650–670.
[27] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD'20)*. 2119–2133.
[28] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A distribution-driven learned index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.
[29] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA'09)*. 267–278.
[30] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture (HPCA'12)*. 189–200.
[31] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC'17)*. 773–785.
[32] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the 7th EuroSys Conference 2012 (EuroSys'12)*. 183–196.
[33] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-tree store. In *2016 USENIX Annual Technical Conference (ATC'16)*. 451–464.
[34] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 315–332.
[35] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.* 40, 1 (2017), 27–37.

[36] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 69–87.

[37] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. 323–337.

[38] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A fully memory-disaggregated key-value store. In *21st USENIX Conference on File and Storage Technologies (FAST'23)*. 81–98.

[39] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki-Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 255–270.

[40] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A scalable learned index for multicore data storage. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*. 308–320.

[41] Muhammad Tirmazi, Adam Barker, Nan Deng, Md. E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next generation. In *15th EuroSys Conference 2020 (EuroSys'20)*. 30:1–30:14.

[42] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (ATC'20)*. 33–48.

[43] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 306–324.

[44] Jérôme Vienne, Jitong Chen, Md. Wasi-ur-Rahman, Nusrat S. Islam, Hari Subramoni, and Dhabaleswar K. Panda. 2012. Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems. In *IEEE 20th Annual Symposium on High-performance Interconnects (HOTI'12)*. 48–55.

[45] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 261–280.

[46] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed B+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*.

[47] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 117–135.

[48] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 87–104.

[49] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *VLDB J.* 23, 6 (2014), 915–937.

[50] Yahoo. 2019. Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB

[51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. 221–234.

[52] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2017. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.* 10, 6 (2017), 685–696.

[53] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST'22)*. 51–68.

[54] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. 523–536.

[55] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. 741–758.

[56] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (ATC'21)*. 15–29.

Received 8 May 2023; accepted 14 August 2023