High-Performance and Cache-Efficiency Adaptive Radix Tree for Disaggregated Memory

Bowen Zhang, Shengan Zheng, Jingxiang Li, Shi Shu, Zhenlin Qi, Linpeng Huang, *Senior Member, IEEE*, and Hong Mei, *Fellow, IEEE*

***Abstract*—Disaggregated memory (DM) architectures enhance resource utilization and architectural scalability by separating computing and memory resources into distinct pools connected via high-speed interconnects, such as RDMA and CXL. Adaptive radix tree (ART) is a widely used range index known for its efficient support of variable-sized keys. However, traditional ART structures incur substantial network overhead in DM architectures during index traversal. In this paper, we propose Sphinx, a high-performance and cache-efficient ART for DM architectures. To minimize network roundtrips during index traversal, Sphinx employs *Flat Inner Node Table* to eliminate data dependencies among hierarchical ART nodes, enabling memory-level parallelism. Moreover, Sphinx introduces *Succinct Filter Cache*, which leverages a space-efficient cuckoo filter in the local cache of compute nodes to mitigate read amplification during index traversal. By further incorporating *Semi Lock-free ART Nodes*, Sphinx achieves high scalability with strong consistency guarantees. Experimental results show that Sphinx outperforms state-of-the-art counterparts by up to 7.3× in the YCSB benchmark.**

***Index Terms*—Disaggregated memory, RDMA, CXL, adaptive radix tree, index.**

I. Introduction

DISAGGREGATED memory (DM) architectures [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] have garnered significant attention from both academia and industry in recent years. In DM architectures, compute and memory resources are decoupled into separate pools, allowing each to scale independently. This separation enhances overall resource utilization and provides greater architectural flexibility. Typically, DM-based systems rely on high-speed interconnects, such as RDMA [2] and CXL [11], which enable clients on compute nodes to directly access data stored in memory nodes without involving remote computational resources.

Adaptive Radix Tree (ART) [12], [13], [14], [15], [16] is a widely used range index in many storage systems, such as key-value stores, databases, and file systems, due to its efficient support for variable-length keys. However, existing ART and its variants [13], [14], [15], [17] suffer from excessive network roundtrips in DM-based systems, significantly degrading performance. Specifically, executing index operations on tree-based structures like ART requires traversing a hierarchical tree structure, which involves multiple network roundtrips, resulting in considerable read amplification in DM architectures. Given that remote memory access remains considerably slower than local access, hierarchical traversal becomes a major performance bottleneck, leading to increased latency and reduced throughput for index operations.

The recent DM-based ART design [17] aims to reduce network roundtrips during tree traversal by caching a subset of inner nodes on compute nodes. However, due to the limited memory capacity inherent to compute nodes in DM architectures, only a small fraction of inner nodes can be retained locally. Consequently, under constrained cache budgets, node-based caching proves largely ineffective at alleviating network overhead [18], [19]. Furthermore, such caching introduces significant overhead in maintaining coherence [20]. When one client modifies a remote inner node, cached replicas on other clients may become stale, triggering costly cache invalidation and synchronization protocols to maintain consistency across compute nodes.

To address these challenges, we propose Sphinx, a DM-based variant of ART that minimizes the network overhead during index traversal while maintaining minimal cache usage in compute nodes. The system components of Sphinx are summarized in Table I. To reduce network roundtrips during index traversal, Sphinx introduces *Flat Inner Node Table*, a hash-based structure that breaks the data dependencies inherent in the hierarchical layout of traditional ART nodes. This design allows clients to retrieve the addresses of all inner nodes along the traversal path in parallel, effectively eliminating the need for sequential traversal. As a result, Sphinx significantly reduces the latency typically incurred during index traversal.

Moreover, Sphinx introduces *Succinct Filter Cache* to mitigate read amplification with minimal cache consumption during

Received 4 June 2025; revised 11 June 2026; accepted 14 August 2026. Date of publication 21 August 2026; date of current version 12 October 2026. This work was supported in part by the National Key Research and Development Program of China under Grant 2024YFB4505203; in part by the National Natural Science Foundation of China (NSFC) under Grant 62227809, Grant 62332012, and Grant 62302290; and in part by the Fundamental Research Funds for the Central Universities. Recommended for acceptance by C. Chen. *(Corresponding author: Shengan Zheng.)*

Bowen Zhang, Jingxiang Li, Shi Shu, Zhenlin Qi, Linpeng Huang, and Hong Mei are with the School of Computer Science, Shanghai Jiao Tong University, Shanghai 200240, China.

Shengan Zheng is with MoE Key Laboratory of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: shengan@sjtu.edu.cn).

Digital Object Identifier 10.1109/TC.2026.3726609

0018-9340 © 2026 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

TABLE I
THE OVERVIEW OF SYSTEM COMPONENTS

Component	Location	Access method
ART Nodes	Memory Node	RDMA
Flat Inner Node Table	Memory Node	RDMA
Succinct Filter Cache	Compute Node	Local Access
Clients	Compute Node	-

index traversal. Instead of caching the full content of inner nodes, Sphinx caches a compact cuckoo filter [21] on each compute node to track the set membership of inner nodes using 10-bit fingerprints of their prefixes (e.g., $fp(\epsilon)$, $fp(BL)$). The cuckoo filter enables efficient *approximate set membership* tests [22] to determine whether a specific inner node exists in the remote memory nodes. By caching the cuckoo filter in the compute nodes, the client can directly retrieve the address of the deepest inner node corresponding to the requested key from the remote *Flat Inner Node Table*, thereby avoiding multiple hash lookups. Due to the space-efficient nature of the cuckoo filter, *Succinct Filter Cache* significantly reduces read amplification while incurring only minimal cache overhead on compute nodes.

To further enhance scalability in DM-based systems, we propose *Semi Lock-free ART Nodes* for Sphinx. For regular node updates, Sphinx employs a lock-free concurrency control mechanism based on atomic hardware primitives, effectively avoiding the overhead associated with locking. In contrast, for the less frequent case of node type switching, Sphinx adopts a lightweight lock-based approach combined with a self-verification mechanism to ensure correctness and prevent potential concurrency anomalies with minimal overhead.

The contributions of this paper are summarized as follows:

- We conduct a comprehensive analysis of existing ARTs on DM architectures and demonstrate their performance limitations, which primarily stem from excessive network roundtrips caused by hierarchical tree traversal and space-inefficient node-based caching mechanisms.
- We propose Sphinx, a DM-optimized variant of ART that effectively reduces network overhead during index traversal while maintaining minimal cache usage on compute nodes. To reduce network roundtrips, Sphinx employs *Flat Inner Node Table* to enable parallel retrieval of inner node addresses. Furthermore, Sphinx incorporates *Succinct Filter Cache*, which leverages a space-efficient cuckoo filter to reduce read amplification.
- We design a semi lock-free concurrency control mechanism for Sphinx, which significantly enhances its scalability in DM environments.
- We conduct extensive experimental evaluations of Sphinx and compare it with state-of-the-art index structures in DM-based systems. Experimental results demonstrate that Sphinx achieves up to 7.3× performance improvement over existing approaches.

II. BACKGROUND AND MOTIVATION

A. Disaggregated Memory Architecture

Resource disaggregation [23], [24], [25], [26], [27], [28], [29], [30] is an emerging data center architecture that decouples hardware components traditionally integrated within monolithic servers, reorganizing them into multiple network-attached units, each forming an independent resource pool. This architecture facilitates independent scaling of resource pools, thereby enhancing resource utilization and improving architectural flexibility.

Disaggregated memory (DM) architecture [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] has gained considerable attention in recent years for its potential to fully realize the benefits of resource disaggregation. In DM architectures, computational resources (e.g., CPU cores) and memory resources (e.g., DRAM, Persistent Memory), which are traditionally co-located within monolithic servers, are physically separated into compute nodes and memory nodes. Compute nodes are equipped with substantial processing power but limited memory capacity, whereas memory nodes offer abundant memory resources but minimal computational capability.

To enable fast communication between compute and memory nodes, DM architectures typically rely on high-speed interconnects, such as RDMA [2] and CXL [11]. RDMA has been widely adopted in DM architectures due to its high bandwidth (e.g., 100∼400 Gbps) and low latency (e.g., ∼2μs) characteristics. More importantly, RDMA supports one-sided primitives, including `RDMA READ`, `RDMA WRITE`, and `RDMA ATOMIC`, allowing compute nodes to directly access remote memory without involving the CPUs on memory nodes. This capability is especially beneficial in DM architectures, where minimizing CPU overhead on memory nodes is critical due to their limited processing capacity.

Meanwhile, the emerging CXL protocol introduces an alternative approach, allowing compute nodes to access remote memory via memory semantics (e.g., `LOAD` and `STORE`) with even lower latency (∼400ns). However, as CXL 3.0-compatible hardware is not yet widely available, our experimental testbed is implemented on an RDMA-based DM platform. Nevertheless, the design proposed in this paper remains compatible with CXL-based DM architectures, demonstrating adaptability to future advancements in memory disaggregation technologies.

B. Adaptive Radix Tree

The Adaptive Radix Tree (ART) [12] is a space-efficient variant of the radix tree, designed to compactly represent variable-length keys. As shown in Fig. 1, each inner node in ART encodes a common prefix shared by the keys beneath it, enabling efficient prefix-based index traversal. To optimize memory usage, ART employs adaptive node types with varying capacities (e.g., 4, 16, 48, and 256), allowing nodes to dynamically grow or shrink based on the number of children. In addition, ART integrates a path compression technique to reduce tree height, thereby mitigating the traversal depth during index operations. Specifically, it merges consecutive inner nodes with a single child into a single consolidated node (e.g., the inner node labeled LYR in Fig. 1) and prunes redundant inner nodes along paths that lead to a single leaf (e.g., the path from LYR to $LYRICS$ in Fig. 1).

However, existing ARTs [13], [14], [17] struggle to deliver high performance in DM architectures due to the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

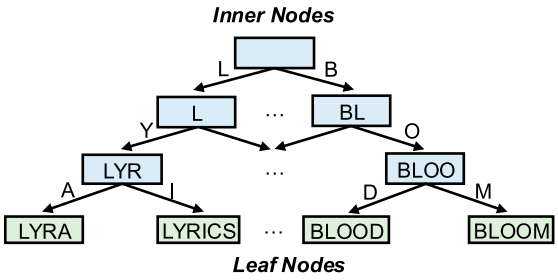


Fig. 1. Adaptive radix tree.

substantial network overhead introduced during index traversals. The inherently hierarchical structure of ART limits memory-level parallelism, as each node access requires first resolving the memory address stored in its parent. In DM architectures, the index structure resides on memory nodes, which offer abundant memory capacity but are physically decoupled from the clients executing on compute nodes. As a result, clients must perform a large number of network roundtrips to traverse the ART structure. This intensive network communication exacerbates access latency and accelerates bandwidth saturation, ultimately degrading overall system performance.

State-of-the-art DM-based ARTs, such as SMART [17], attempt to mitigate this problem by caching frequently accessed inner nodes on compute nodes. During index traversal, the client first attempts to retrieve the target node from the local cache. If the node is not available locally, it is then fetched from remote memory. However, this node-based caching mechanism is inefficient under memory-constrained compute nodes. First, it requires a substantial amount of memory to achieve high performance, which is impractical given the limited capacity of compute nodes. Our evaluation reveals that even with a cache size equivalent to 17.6% of the dataset, SMART still demonstrates suboptimal performance (see details in Sec. V-B). Second, node-based caching introduces cache coherence challenges, as remote ART nodes may be concurrently modified by clients on different compute nodes. Although SMART mitigates this issue by preallocating large memory regions for each inner node and using a reverse checking mechanism, it still incurs considerable cache management overhead [31] and results in significant memory waste (see details in Sec. V-D).

C. Approximate Set Membership

Approximate set membership tests are employed to determine whether a given item is likely present in a large set with a small probability of false positives or definitively absent [21]. Many databases, caches, routers, and storage systems leverage probabilistic data structures to efficiently handle *approximate set membership* tests, thereby reducing space consumption and enhancing performance.

The most widely used data structure for *approximate set membership* tests is the bloom filter [22], which has been studied extensively due to its memory efficiency. The bloom filter comprises k independent hash functions and a bit array, where all bits are initially set to 0. To insert a key, the bloom filter applies its k hash functions, mapping it to k positions in the bit array and setting the corresponding bits to 1. The lookup operation computes the k hash values and retrieves the corresponding bits in the bit array. If all k bits are set to 1, the query returns true, indicating that the key is possibly in the set. Otherwise, it returns false, meaning the key is definitely not present.

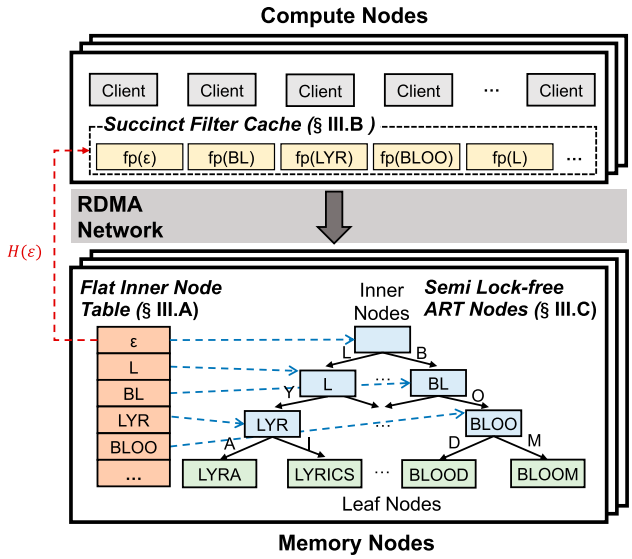


Fig. 2. Overview of Sphinx.

However, standard bloom filters do not support deletions. Many variants of bloom filters, such as counting bloom filter [32], blocked bloom filter [33], and quotient filter [34], extend the bloom filter to enable delete operations. Nevertheless, these approaches often involve trade-offs, either introducing substantial space overhead or significantly degrading performance.

Cuckoo filter [21] is a compact variant of the cuckoo hash table that supports both insertions and deletions while maintaining space efficiency comparable to that of bloom filters. Instead of storing full keys, the cuckoo filter inserts only a fingerprint, a short bit string derived from the item using a hash function, into the hash table. By densely packing fingerprints, the cuckoo filter achieves high space utilization. To perform a set membership query, the filter simply checks whether the fingerprint of the queried item exists in the table. If an identical fingerprint is found, the query returns true.

III. Design

In this section, we present Sphinx, a high-performance and cache-efficient ART for DM architectures that minimizes network overhead during index traversal with low cache space consumption on compute nodes. As shown in Fig. 2, Sphinx is deployed in a DM-based cluster consisting of compute nodes and memory nodes interconnected through an RDMA network. In Sphinx, ART nodes are evenly distributed across memory nodes using consistent hashing. Clients on compute nodes employ one-sided RDMA primitives to efficiently traverse the index structure stored in memory nodes. Table II lists the symbols used in the subsequent sections for clarity.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

TABLE II
LIST OF SYMBOLS

Symbol	Description
K	Query key in index operations
V	Value associated with key K
K_1, K_2	Lower and upper bound keys in a range scan
L	Length of a key (in bytes)
P	Prefix of a key
ϵ	Empty prefix (root prefix of ART)
$H_1(\cdot), H_2(\cdot)$	Two candidate hash functions used in the cuckoo filter
$fp(\cdot)$	10-bit fingerprint stored in the cuckoo filter
$H(\cdot)$	Hash function used in the *Flat Inner Node Table*
$fp_2(\cdot)$	12-bit fingerprint stored in the *Flat Inner Node Table*

As illustrated in Fig. 2, Sphinx integrates three complementary techniques within a hierarchical architecture. At the foundation, Sphinx is an ART index that utilizes *Semi Lock-free ART Nodes* to ensure high scalability and strong consistency under concurrent access. Built upon this foundation, the *Flat Inner Node Table* reorganizes inner nodes to enable parallel traversal, thereby reducing network roundtrips. On top of this structural optimization, the *Succinct Filter Cache* further filters out unnecessary inner node accesses, reducing bandwidth consumption. Together, these three techniques synergistically enable a high-performance ART index in DM architectures that minimizes network roundtrips and bandwidth consumption during index traversal, while achieving strong consistency and high scalability under concurrent access. Since any limitation in network roundtrips, bandwidth consumption, or concurrency control can become a system bottleneck, removing any of these design components would fundamentally compromise overall performance.

To minimize network roundtrips during index traversal, Sphinx introduces *Flat Inner Node Table*, which employs a hash-based structure to index ART inner nodes stored in memory nodes. The *Flat Inner Node Table* eliminates data dependencies inherent in the hierarchical structure of ART nodes, enabling clients to retrieve remote inner nodes in parallel, rather than relying on sequential tree traversal. This design significantly reduces the number of network roundtrips required during index traversal, thereby minimizing the latency of index operations.

Moreover, Sphinx incorporates a *Succinct Filter Cache*, a lightweight caching mechanism deployed on each compute node to filter out unnecessary inner node reads, thereby reducing the consumption of network bandwidth. The *Succinct Filter Cache* leverages a compact cuckoo filter to identify the deepest inner node corresponding to a given query key through a series of local *approximate set membership* tests. This mechanism prevents Sphinx from accessing excessive hash entries from the remote *Flat Inner Node Table*, thereby preserving network bandwidth.

To further enhance concurrent scalability, Sphinx employs *Semi Lock-free ART Nodes* to minimize concurrency control overhead. Sphinx employs lock-free concurrency control for routine node updates and selectively uses lock-based mechanisms for the less frequent cases of node type switching. This hybrid approach enhances the efficiency of concurrent access while effectively preventing consistency anomalies.

A. Flat Inner Node Table

Sphinx introduces *Flat Inner Node Table* to eliminate data dependencies among hierarchical ART nodes, thereby enabling memory-level parallelism during index traversal. This design allows clients on compute nodes to retrieve the addresses of multiple ART inner nodes stored on memory nodes in parallel, reducing the number of required network roundtrips.

As illustrated in Fig. 2, the *Flat Inner Node Table* employs a hash table to index ART inner nodes. In this structure, each key corresponds to the full prefix of an inner node, while the associated value stores the memory address of that node along with a small amount of essential metadata. During index operations, rather than sequentially traversing the tree to fetch inner nodes one by one, the client can query the hash table to retrieve the addresses of all relevant inner nodes along the traversal path in parallel. Based on these results, the client can then directly access the deepest inner node corresponding to the requested key, requiring only a single network roundtrip.

To illustrate how the *Flat Inner Node Table* reduces network roundtrips during index traversal, we present the workflow of Sphinx using the example shown in Fig. 2. Given a query key (e.g., *LYRICS*), the client performs parallel searches in the hash table for all potential inner node keys that might exist along the traversal path (*LYRICS*, *LYRIC*, *LYRI*, *LYR*, *LY*, and *L*). Querying all possible prefixes is necessary because the path compression mechanism of ART prevents the client from directly inferring the exact prefixes of inner nodes on the traversal path. Based on the results of these hash lookups, the client retrieves the addresses of all inner nodes along the traversal path (*LYR*, *L*) from the successfully matched hash entries. It then directly fetches the deepest inner node (*LYR*) from the memory nodes, which serves as the parent of the target leaf node containing the requested key.

The *Flat Inner Node Table* enables Sphinx to locate the parent node of the target leaf node within a single network roundtrip, effectively eliminating the need for hierarchical index traversal. Sphinx adopts RACE hashing [35] as the index structure for the *Flat Inner Node Table*, allowing each hash query to be completed with a single network roundtrip. Moreover, Sphinx can issue all prefix hash queries concurrently within one network roundtrip through doorbell batching mechanism [36], a NIC-supported mechanism widely adopted in prior work [2], [17], [19], [35]. Doorbell batching enables clients to issue multiple outstanding RDMA requests in parallel. Specifically, the client posts multiple RDMA requests to the NIC and then rings a single doorbell, allowing the NIC to fetch multiple requests in a batch via DMA.

Additionally, RACE hashing resolves conflicts among multiple prefixes using several collision resolution techniques, including bucket associativity and overflow colocations [35]. Specifically, as shown in Fig. 4, conflicting prefixes are stored in an associative hash bucket with multiple slots. Consecutive

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

hash buckets share an overflow bucket, which stores prefixes that cannot be accommodated in their original buckets.

The *Flat Inner Node Table* imposes only minimal memory overhead on the memory nodes. Specifically, its memory footprint is significantly smaller than that of the inner nodes themselves. Each hash entry stores a 12-bit key fingerprint, a 48-bit memory address, and a 4-bit node type identifier, all of which are compactly encoded within a single 8-byte slot, as illustrated in Fig. 4. In contrast, a single ART inner node consumes between 64 and 2056 bytes of memory. As demonstrated by our experiments in Section V-D, the *Flat Inner Node Table* introduces only a 3.3∼4.9% increase in overall memory usage on the memory nodes.

B. Succinct Filter Cache

Although *Flat Inner Node Table* effectively reduces network roundtrips during index traversal, it still consumes substantial network bandwidth and introduces excessive network requests. For a key of length L, the client must perform L hash queries to locate its deepest inner node, leading to a rapid saturation of both network bandwidth and the IOPS limit of the RDMA NIC.

To address the aforementioned issue, we propose *Succinct Filter Cache*, which reduces the number of network requests required to locate the deepest inner node corresponding to a given key in the *Flat Inner Node Table* from L to just one. Our key insight is that this process can be transformed into L *Approximate Set Membership* tests (see details in Section II-C), thereby eliminating the need for directly querying the *Flat Inner Node Table*. To achieve this, Sphinx maintains a cuckoo filter [21] on each compute node, which tracks the set membership of all inner nodes stored in the *Flat Inner Node Table*. This allows the client to efficiently check whether a specific inner node exists in the *Flat Inner Node Table* by performing a lightweight *approximate set membership* test. By leveraging this approach, the client can locally determine the prefix of the deepest inner node associated with any given key. It then issues a single network request to retrieve the corresponding hash entry from the *Flat Inner Node Table*, significantly reducing network overhead.

As shown in Fig. 2, in compute nodes, Sphinx leverages a cuckoo filter to store fingerprints of inner node prefixes (e.g., $fp(\epsilon)$, $fp(BL)$) in a compact hash table. The detailed structure of the *Succinct Filter Cache* is illustrated in Fig. 3. To determine whether a given prefix P exists in the *Succinct Filter Cache*, the client performs an *approximate set membership* test. Specifically, the client computes two hash values (H_1 and H_2) for the prefix and checks whether the corresponding hash fingerprint (fp) is present in either candidate buckets indicated by these hash values. If the fingerprint is found, it is highly likely that an inner node with prefix P exists in the *Flat Inner Node Table*, with only a small probability of false positives.

During index traversal, given a query key (e.g., *LYRICS*), the client first performs *approximate set membership* tests on all of its prefixes (*LYRICS*, *LYRIC*, ..., *L*) using the *Succinct Filter Cache*. Based on the results, the client identifies the prefix of the deepest inner node (*LYR*) along the traversal path for the query key. Once this prefix is determined, the client retrieves the address of the corresponding inner node from the remote *Flat Inner Node Table* with a single network request, thereby significantly reducing the number of hash table lookups.

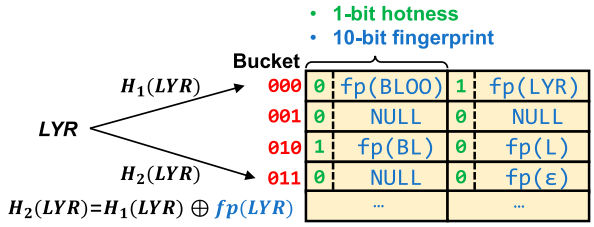


Fig. 3. Succinct filter cache (*fp(x)* indicates the fingerprint of prefix x).

The *Succinct Filter Cache* requires only a small amount of cache space on compute nodes and introduces minimal overhead for maintaining cache coherence. Its underlying cuckoo filter is highly space-efficient, achieving a low false positive rate (1%) for *approximate set membership* tests by storing just a 10-bit fingerprint per inner node. As a result, the *Succinct Filter Cache* is significantly more space-efficient than prior node-based caching mechanisms [17], which may consume up to 2056 bytes to cache a single inner node. Moreover, the *Succinct Filter Cache* inherently preserves coherence, even when inner nodes in remote memory are modified by other clients. Such modifications typically affect only the node type, node contents, or partial prefixes, while the full prefix remains unchanged. Since the *Succinct Filter Cache* tracks only the set membership of full prefixes, these updates do not compromise its correctness or consistency.

Due to the probabilistic nature of the cuckoo filter, the *Succinct Filter Cache* may occasionally produce false positives, incorrectly indicating that a prefix exists in the remote *Flat Inner Node Table* when it does not. Although such cases are rare (with a false positive rate below 1%), Sphinx still incorporates a double-check mechanism and an optimistic search strategy to ensure correctness and maintain high performance in the presence of these anomalies. As illustrated in Fig. 4, both the hash entry and the corresponding inner node include metadata that allows the client to detect mismatches without performing unnecessary tree traversals. Specifically, each hash entry contains a 12-bit hash fingerprint ($fp2$), while the header of the inner node stores a 43-bit full-prefix hash. The double-check mechanism ensures that Sphinx only proceeds with a traversal from a potentially incorrect inner node in the extremely rare case where both the $fp2$ and the full-prefix hash collide. Since such collisions are exceedingly rare, overall system performance remains unaffected. More importantly, the correctness of index operations is fully preserved. In these rare cases, the client performs an optimistic search starting from the incorrect inner node, eventually reaching a leaf node. The client then compares the full key in the leaf node with the requested key to determine whether a false positive has occurred. Our experiments show that when a false-positive event occurs, the additional verification step increases the search latency by 37.5% and 42.4% on the `u64` and `email` datasets, respectively. However, since

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

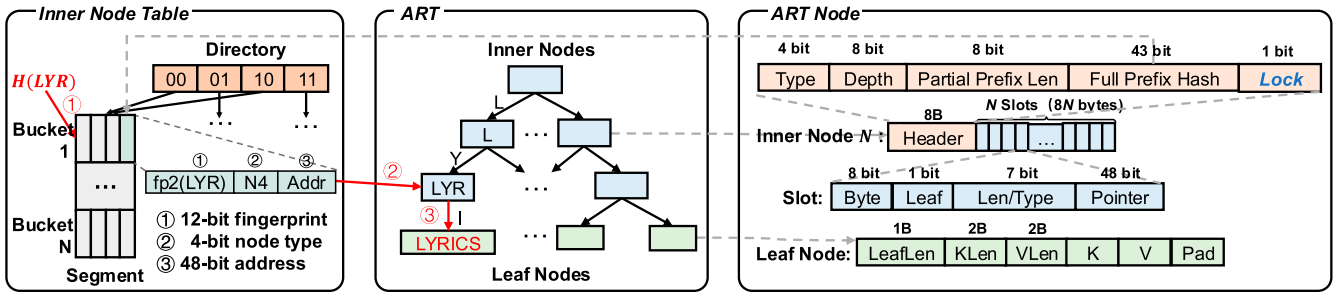


Fig. 4. Detailed data structures of Sphinx.

the false-positive rate of the cuckoo filter is below 1%, the amortized performance impact on Sphinx is very small (less than 0.5%).

To handle scenarios where the dataset is so large that even tracking the set membership of all inner nodes on compute nodes becomes infeasible, we propose a lightweight hot-prefix identification mechanism within the *Succinct Filter Cache*, as illustrated in Fig. 3. This design is inspired by the second-chance page replacement policy, which approximates the LRU algorithm using just a single hotness bit per entry. The hotness bit is set to 1 upon access, indicating that the entry was recently used. During insertions, if both candidate buckets are full, the client randomly selects an entry with a hotness bit of 0 for eviction. If all entries in both buckets have their hotness bits set to 1, the client resorts to the cuckoo hashing mechanism to relocate existing entries and make room for the new one. During relocation, the hotness bits of all moved entries are reset to 0, marking them as eligible for future replacement.

C. Semi Lock-Free ART Node

Fig. 4 illustrates the detailed structure of ART nodes. To enhance the concurrent scalability in DM-based systems, Sphinx employs lock-free concurrency control for normal node updates while utilizing lock-based mechanisms only for infrequent node type switches.

Lock-free Node Update. Sphinx achieves lock-free write operations on inner nodes based on atomic hardware primitives. To efficiently handle potential inconsistencies in the *Flat Inner Node Table* and *Succinct Filter Cache* caused by inner node updates, we further propose a client-active inconsistency repair mechanism.

Fig. 4 illustrates the inner node structure of Sphinx, which is similar to that used in prior work [17]. Each child pointer is encapsulated in an 8-byte slot that stores both the pointer address and relevant metadata (e.g., next key byte, node type), enabling clients to atomically insert new nodes using atomic hardware primitives. Specifically, to insert a node (either an inner or a leaf node), Sphinx first prepares the new node in an out-of-place manner and then atomically installs its metadata and address into an empty slot of the parent node using a single `RDMA CAS`. This process can be completed in a lock-free manner since no intermediate inconsistent state is exposed to concurrent operations.

However, inserting new inner nodes may introduce temporary inconsistencies in the inner node table and filter cache. This occurs when the key-value pair being inserted only partially matches the prefix of an existing inner node, triggering a node split. In such cases, Sphinx inserts a new inner node that points to both the original inner node and the new key-value pair. Because Sphinx cannot atomically update the ART structure, the inner node table, and the filter cache in a single step, the latter two may temporarily become stale after an inner node insertion.

To address this issue, Sphinx introduces a client-active inconsistency repair mechanism to maintain the consistency of the inner node table and the filter cache. We observe that the inner node table and filter cache do not require synchronous updates after each inner node insertion. While temporarily outdated entries may lead the client to begin traversal from a suboptimal inner node, causing slight performance degradation, they do not affect correctness. Therefore, Sphinx asynchronously updates the inner node hash table via a single `RDMA CAS` following the lock-free protocol proposed in RACE hashing [35]. Additionally, during index traversal, if a client encounters an inner node that is missing from the filter cache, it actively updates the cache. In this way, even though the inner node table and filter cache are not synchronously maintained, they eventually converge to a consistent state without compromising the correctness of index operations.

Lock-based Node Type Switch. One of the key characteristics of an ART-based index is its ability to dynamically select different inner node types based on the number of child pointers. Node type switching involves copying data from the old inner node to a new one, an operation that cannot be performed atomically. In a lock-free setting, concurrent write operations may modify the old node during the copying process, leading to issues such as *lost updates*. Existing solutions [17] address this by preallocating sufficiently large space for each inner node, enabling in-place type switching to ensure consistency. However, this approach significantly increases memory consumption.

Therefore, Sphinx proposes a lock-based node type switch mechanism combined with a self-verifying optimistic read strategy to efficiently handle this problem. As shown in Fig. 4, Sphinx maintains a lock field in the header of each inner node. A client initiating a node type switch sets this lock field using an `RDMA CAS`. Concurrent operations on inner nodes must verify the lock status both before and after their execution to

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

ensure that no type switch occurs during the operation, thereby avoiding concurrency anomalies. When reading an inner node, Sphinx retrieves both the node header and child pointers in a single operation, allowing it to verify the lock status with minimal additional overhead. To further reduce the cost of post-operation verification, Sphinx employs the doorbell batching mechanism to send two RDMA operations in parallel: the first operation accesses the inner node, and the second subsequently verifies the lock status. This design ensures that lock verification does not incur additional network roundtrips during index operations.

IV. Index Operations

In this section, we present the detailed implementation of the index operations in Sphinx, including search, insert, update, delete, and scan.

Search (K). We use `Search(LYRICS)` as an example to illustrate the end-to-end workflow of search operations. As shown in Fig. 4, the client first locally identifies the longest prefix P (i.e., *LYR*) of key *LYRICS* that exists in the *Succinct Filter Cache* by conducting *approximate set membership* tests on all possible prefixes of *LYRICS* (i.e., *LYRICS*, *LYRIC*, . . ., *L*). It then retrieves the address of the inner node with prefix P (i.e., *LYR*) from the remote *Flat Inner Node Table* in a single network roundtrip (①). Specifically, Sphinx adopts RACE hashing [35] as the index of the *Flat Inner Node Table*. The hash structure includes a directory that points to the addresses of multiple hash segments. To avoid additional network roundtrips when resolving segment addresses via the directory, each compute node maintains a local directory cache, which consumes only 2∼5% of the space required by the *Succinct Filter Cache* and fits within the constrained memory resources of compute nodes. Once the relevant segment is identified, the client accesses the corresponding hash bucket and retrieves the hash entries that match the 12-bit fingerprint ($fp2$) of prefix P.

In most cases, the hash query to the *Flat Inner Node Table* returns a single matching entry, from which the client obtains the memory address of the corresponding inner node, typically the deepest inner node associated with the key (i.e., *LYR*). As shown in Fig. 4, the client then issues an `RDMA READ` to fetch this inner node (②) and begins tree traversal from that node, eventually reaching the leaf node (③) that contains the value corresponding to K (*LYRICS*). By leveraging the *Succinct Filter Cache*, the client can identify the deepest matching inner node in advance, thereby reducing the traversal to typically a single network roundtrip. However, the client may fetch a non-deepest inner node due to temporary staleness in the *Succinct Filter Cache*. As discussed in Section III-C, when this occurs, the client updates the filter cache by inserting any missing prefixes to ensure the cache remains fresh and accurate.

In rare circumstances, a hash query to the *Flat Inner Node Table* may either fail to return a matching entry or yield multiple candidates. The former typically results from a false positive in the *Succinct Filter Cache*. In such cases, the client falls back to retrieving the inner node with the second-longest matching prefix from the *Flat Inner Node Table*. The latter case arises when the limited precision of the 12-bit fingerprint causes an ambiguous match. To resolve this, the client fetches both candidate inner nodes from remote memory and performs parallel traversals from each, ensuring correctness.

Insert (K, V). The client begins by locating the deepest matching inner node following the search procedure and traverses the tree from that point until it reaches a leaf node, verifying that the prefix of the inner node matches the key K. It then creates a new leaf node containing the inserted key-value pair and writes it to remote memory using an `RDMA WRITE`. To make the insertion visible, the client atomically installs a pointer to the new leaf node into the empty slot of its parent node via an `RDMA CAS`.

An inner node split is triggered when the inserted key-value pair only partially matches the prefix of an existing inner node along the traversal path. In this case, the client creates a new inner node that points to both the existing inner node and the new leaf node. The client then uses an `RDMA CAS` to atomically install this new inner node into the parent of the original node. Subsequently, it asynchronously updates the *Flat Inner Node Table* and the *Succinct Filter Cache* to reflect the new inner node.

An inner node type switch occurs when the current inner node no longer has enough empty slots to accommodate new insertions. In this situation, the client sets the lock field of the node using an `RDMA CAS`, notifying other concurrent clients that a type switch is in progress. The client then relocates all child pointers to a newly allocated inner node of a larger type. Finally, it atomically installs the new node into the parent of the original inner node using another `RDMA CAS`.

Update (K, V). For update operations, the client first locates the target leaf node of the requested key by following the standard search procedure. It then prepares a new leaf node with the updated content and atomically updates the corresponding child pointer in the parent node using an `RDMA CAS`.

Delete (K). Similar to update operations, delete operations require locating the parent node of the target leaf node that contains the key K. The client then atomically sets the corresponding child pointer in the parent node to `NULL` using an `RDMA CAS`.

Scan (K_1, K_2). To perform a range scan between keys K_1 and K_2, the client first computes their longest common prefix P. It then retrieves the deepest inner node corresponding to P using the *Flat Inner Node Table* and *Succinct Filter Cache*. Starting from this node, the client traverses the tree and collects all leaf nodes with keys in the range between K_1 and K_2, aggregating the associated key-value pairs into the result set.

Client-side Crash Handling. In a disaggregated index, client crashes may lead to partial updates, deadlocks, and memory leaks. Sphinx primarily follows existing works [37], [38], [39], [40] to address these issues. (1) *Partial updates.* To preserve the correctness and eventual consistency of the index structure in the presence of client crashes, Sphinx must guarantee all-or-nothing transactional semantics: an interrupted write operation should either become fully visible or leave the index structure on memory nodes unchanged. Sphinx achieves this guarantee by preventing partial updates and ensuring atomicity for all modifications. Specifically, every write operation, including

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

both regular node updates and node type switches, follows an out-of-place update procedure. A client first allocates and initializes a new node and then atomically installs it into the index structure using an `RDMA_CAS` operation. If the `CAS` succeeds, the new node becomes atomically visible in the index. Otherwise, the newly prepared node remains unreachable, leaving the index structure unchanged. Consequently, other clients can observe either the old version or the new version of a node, but never a partially updated state. Therefore, Sphinx preserves correctness and eventual consistency even in the presence of client crashes. (2) *Deadlocks.* Sphinx follows prior works [37], [38], [39] to adopt a timeout-based mechanism to address deadlocks. In Sphinx, locking is required only during node type switches. A client crash during this process may leave a lock held indefinitely. To avoid this, Sphinx assigns each lock an expiration time, which is large enough for a node type switch. If another client observes that a lock has been held longer than the expiration threshold and the associated pointer remains unchanged, it treats the lock as expired and proactively releases it. (3) *Memory leaks.* To handle memory leakage caused by client crashes, Sphinx adopts the FUSSE design [40]. A crashed client may leave allocated nodes unlinked from the ART, resulting in unreachable memory blocks. FUSSE mitigates this problem by performing a full traversal of the ART index upon crash detection, identifying all memory blocks that remain reachable, and reconstructing the allocator state accordingly.

V. Evaluation

A. Experimental Setup

Testbed. We emulate an RDMA-based DM cluster using three dual-socket servers, each equipped with 128GB of DRAM, two 2.6GHz Intel Xeon Gold 6348 CPUs (28 cores per socket), and a dual-port Mellanox ConnectX-6 NIC (2×100 Gbps). Following prior work [2], [17], we co-locate one compute node and one memory node on each machine due to limited hardware availability. The compute node utilizes the CPU, while the memory node uses the DRAM. To faithfully emulate a disaggregated setup, we enforce that all memory accesses from compute nodes to memory nodes occur via one-sided RDMA, even when both reside on the same physical server.

Workloads. We evaluate Sphinx using the YCSB benchmark, covering six standard workloads: A (50% read, 50% update), B (95% read, 5% update), C (100% read), D (95% latest read, 5% update), E (95% scan, 5% insert), and LOAD (100% insert). By default, workloads follow a zipfian key distribution with a skewness factor of 0.99.

We use two datasets in our evaluation: *u64* and *email*. The *u64* dataset consists of uniformly distributed 8-byte integer keys. The *email* dataset is derived from real-world email addresses [41], with key lengths ranging from 2 to 32 bytes and an average length of 18.93 bytes. Unless otherwise specified, each key-value pair contains a 64-byte value. In Section V-E, we vary the value size from 64 to 1024 bytes to study performance sensitivity to value size.

Compared Systems. Since Sphinx is built upon ART, we compare it with SMART [17], which is the state-of-the-art DM-based ART variant. We also evaluate SMART+C, which is configured with a 10× larger cache on compute nodes, to show the impact of a large cache capacity. Furthermore, we include the original ART [12] as a baseline, which we reimplement from scratch to support DM, in order to quantify the performance gains brought by our designs. To evaluate Sphinx not only against ART-based designs but also against alternative indexes, in Section V-F, we further compare it with B$^+$Tree and hash-based indexes, which are widely adopted in practical systems. Specifically, we include Sherman [2] and CHIME [19] as representative DM-based B$^+$Tree designs, and RACE [35] as a representative DM-based hash index.

Cache Policy. Sphinx leverages *Succinct Filter Cache* while SMART, Sherman, and CHIME cache frequently accessed tree nodes on compute nodes. Given the limited DRAM capacity on compute nodes, we set the default cache size to 20 MB (200 MB for SMART+C). In Section V-E, we further explore index performance across a wide range of cache sizes.

B. Overall Performance

In this section, we evaluate the overall performance of Sphinx with the YCSB benchmark. Fig. 5 displays the throughput of the systems under the YCSB benchmark with the *u64* and *email* datasets.

Base Operations (YCSB A-D, LOAD). As shown in Fig. 5, Sphinx achieves 1.2∼3.6× higher throughput on the *u64* dataset and 1.9∼7.3× higher throughput on the *email* dataset than SMART, SMART+C, and ART. This improvement stems primarily from our *Flat Inner Node Table* and *Succinct Filter Cache* designs, which enable a single remote hash query to locate the deepest inner node of the target key. This design significantly reduces both network roundtrips and bandwidth consumption during index traversal. In contrast, SMART and ART rely on sequential traversal of the radix tree, resulting in substantial network overhead.

Sphinx demonstrates even greater performance gains on datasets with larger key sizes (e.g., *email*), as its traversal overhead remains constant regardless of tree height. This is because Sphinx maintains a constant traversal overhead across different datasets while the traversal overhead of other ART indexes is proportional to the tree height. Additionally, Sphinx outperforms SMART+C while using only 10% of its cache space, highlighting the efficiency of *Succinct Filter Cache* in delivering high performance under tight memory constraints.

Range Query (YCSB E). For range queries, the performance bottleneck shifts from index traversal to reading leaf nodes. As a result, SMART and SMART+C achieve performance comparable to Sphinx, despite their higher traversal overhead. Across both the u64 and email datasets, Sphinx, SMART, and SMART+C achieve 2.3∼3.1× higher throughput than ART, primarily due to the use of the doorbell batching mechanism [36], which effectively hides the latency of remote leaf node accesses.

C. Scalability

In this section, we evaluate the scalability of different ART-based indexes using both write-intensive workloads (YCSB-A) and read-intensive workloads (YCSB-C) under the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

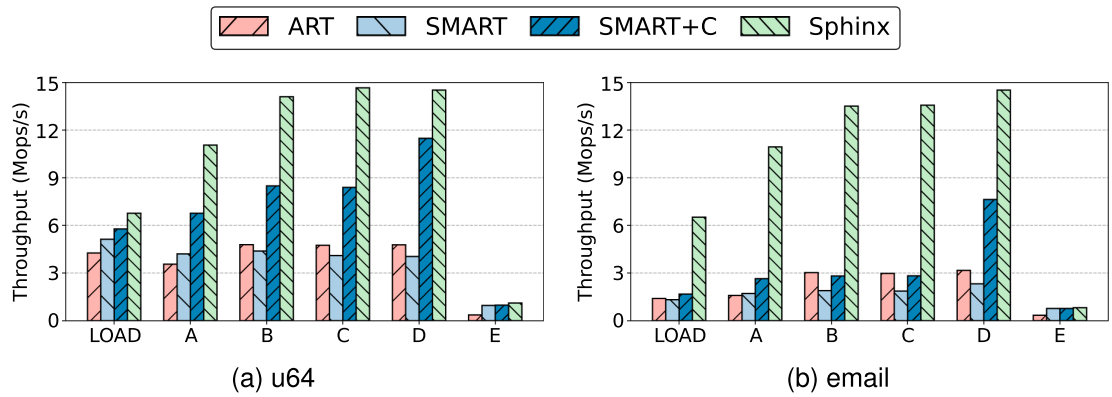


Fig. 5. The throughput under the YCSB benchmark.

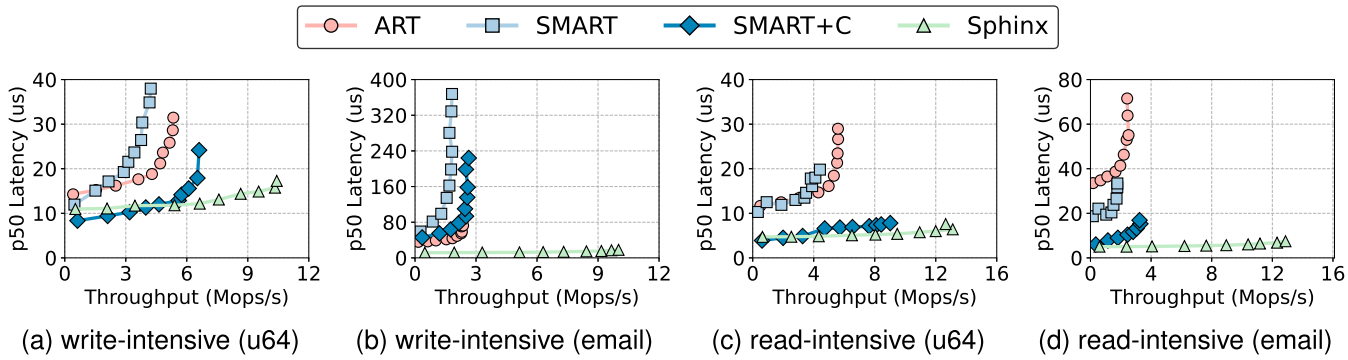


Fig. 6. The scalability with increasing client number in the write-intensive and read-intensive workloads.

u64 and *email* datasets. Following prior works [2], [17], [18], we adopt coroutines to hide latency caused by polling the completion queue, treating each coroutine as an independent client.

Fig. 6 presents the throughput and latency curves for each index as the number of clients increases from 3 to 192, evenly distributed across three compute nodes. Sphinx exhibits strong scalability with increasing clients, achieving up to 5.5× higher throughput and 20.3× lower latency on write-intensive workloads, and up to 7.3× higher throughput and 9.6× lower latency on read-intensive workloads. Sphinx's scalability stems from two core designs. First, the proposed *Flat Inner Node Table* and *Succinct Filter Cache* significantly reduce network roundtrips and bandwidth consumption during index traversal, preventing early saturation of network resources under concurrent access. Second, Sphinx introduces a semi lock-free concurrency control mechanism that greatly improves the efficiency of concurrency control, thereby enhancing the scalability under high concurrency.

Fig. 7 shows the throughput of each index as we increase the number of memory nodes from 1 to 3. Sphinx achieves 1.3∼3.6× and 3.2∼7.3× higher throughput than the compared systems on the *u64* and *email* datasets, respectively, as it minimizes both network roundtrips and bandwidth consumption during index operations. Fig. 7 also demonstrates that all evaluated indexes scale well with the increasing number of memory nodes. This behavior can be attributed to the fact that, given a sufficient number of clients, adding more memory nodes provides additional network resources to serve client requests, thereby improving overall throughput.

D. Space Consumption

In this section, we evaluate the memory footprint of ART-based indexes on memory nodes across different datasets. Each system is populated with 60 million key-value pairs, with each entry sized at 128 bytes, and the resulting memory usage is recorded.

As shown in Table III, Sphinx incurs 26∼28% more space consumption on memory nodes than the original ART to maintain its *Flat Inner Node Table*, equivalent to only 4∼6% of the total dataset sizes. In contrast, SMART consumes 16.3∼18.1× more memory than the original ART, due to its preallocation of a full Node-256 for each inner node to avoid concurrency anomalies. These results demonstrate that Sphinx achieves strong consistency guarantees and substantial performance benefits with significantly lower memory overhead on memory nodes.

E. Performance Under Various Configurations

In this section, we evaluate index performance under varying system configurations by adjusting cache sizes on compute nodes and value sizes in the datasets.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

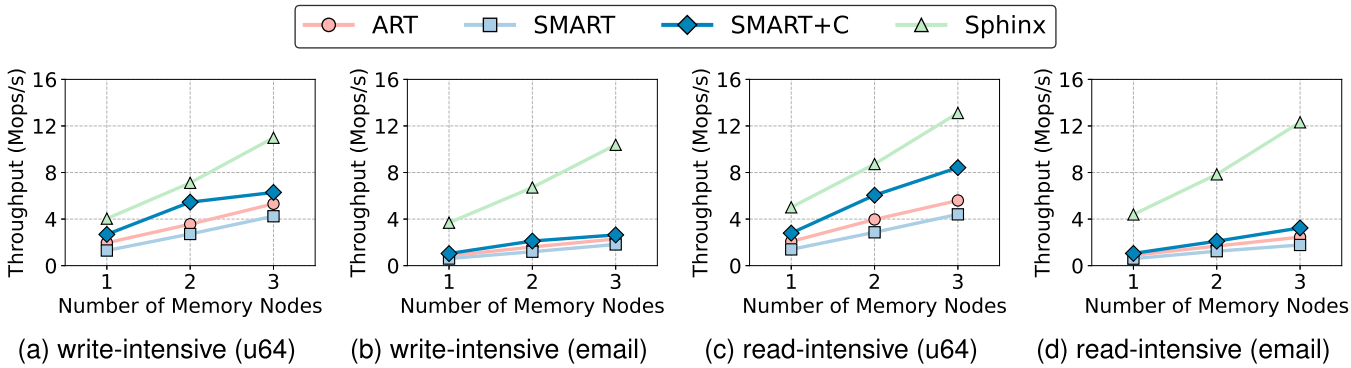


Fig. 7. The scalability with increasing memory nodes in the write-intensive and read-intensive workloads.

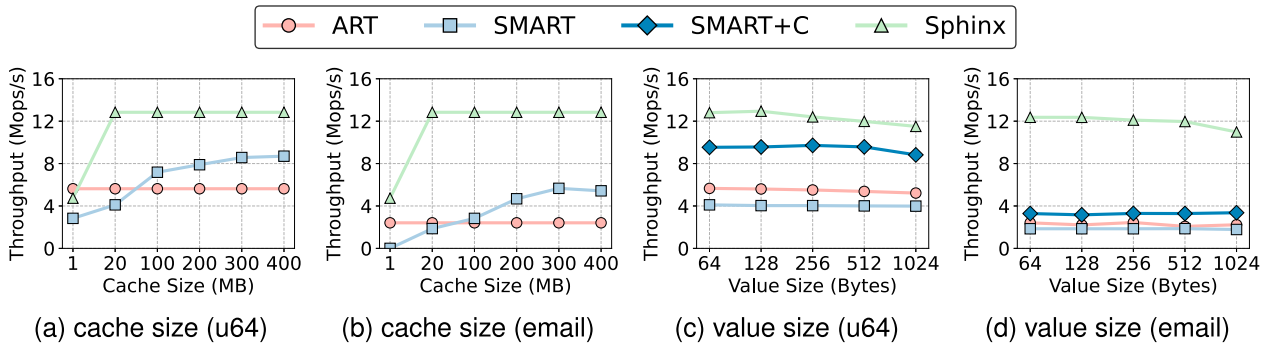


Fig. 8. The index performance with varying cache sizes and value sizes.

TABLE III
THE MEMORY USAGE ON MEMORY NODES UNDER DIFFERENT DATASETS

	Key-value	ART	SMART	Sphinx	
				ART	Hash
u64	7.15 GB	1.09 GB	17.86 GB	1.09 GB	0.28 GB
email	7.15 GB	1.52 GB	27.51 GB	1.52 GB	0.43 GB

Cache Sizes. Fig. 8(a) and 8(b) show index throughput as the cache size increases on compute nodes. We observe that Sphinx outperforms other ART-based indexes, even with a significantly smaller cache, owing to its use of *Succinct Filter Cache*. In contrast, SMART adopts a node-based caching policy that requires substantially more cache to be effective. As a result, even with large cache capacities, the performance of SMART remains substantially lower than that of Sphinx, especially on datasets with large key sizes (e.g., *email*).

Value Sizes. Fig. 8(c) and 8(d) report index performance across a range of value sizes for both *u64* and *email* datasets. We observe that value size has little impact on overall performance, as the dominant bottleneck lies in network overhead incurred during index traversal, rather than in key-value access.

F. Comparisons With Other DM-Based Indexes

In this section, we compare the performance of Sphinx with other DM-based indexes, including a hash index (i.e., RACE), and two B$^+$Tree-based indexes (i.e., Sherman and CHIME). As shown in Fig. 9, Sphinx consistently outperforms all evaluated indexes. This advantage stems from its *Flat Inner Node Table* and *Succinct Filter Cache*, which significantly reduce network overhead during index traversal.

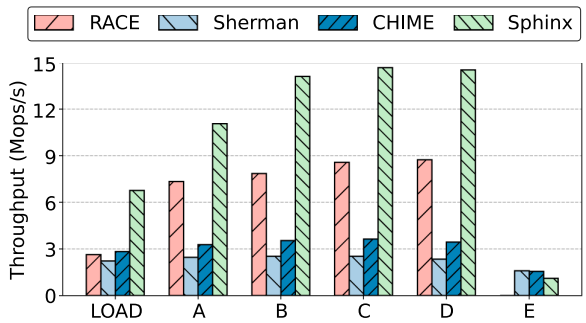


Fig. 9. The performance comparisons with other state-of-the-art DM-based indexes.

Sherman and CHIME suffer from high traversal costs in DM-based systems due to their reliance on sequential traversal through hierarchical B$^+$Tree structures, similar to the traditional ART-based indexes. In contrast, RACE hashing achieves better performance than Sherman and CHIME by adopting a flat layout that eliminates hierarchical traversal. In theory, RACE could deliver performance comparable to Sphinx, as both minimize traversal-induced network overhead. However, RACE's current implementation does not leverage coroutines to

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

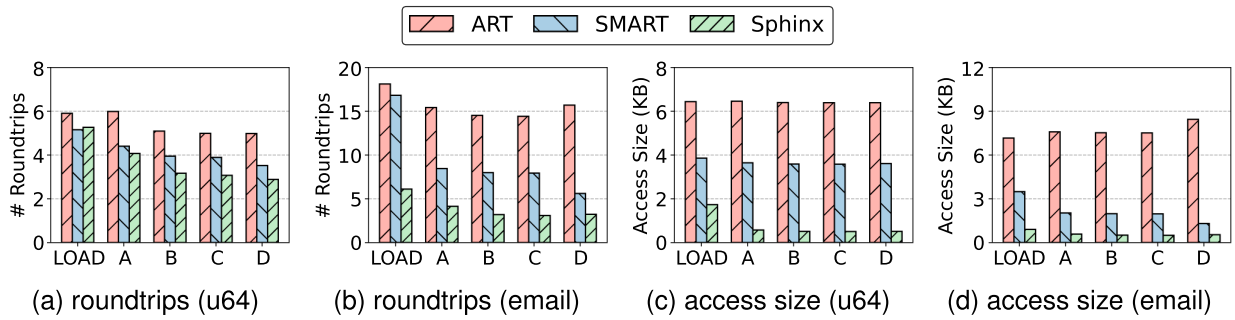


Fig. 10. The average network overhead (roundtrip number and access size) per operation in YCSB workloads.

TABLE IV
AVERAGE NETWORK OVERHEAD (ROUNDTRIP NUMBER AND ACCESS SIZE) PER INDEX OPERATION AND THE FRACTION OF THESE OVERHEADS ATTRIBUTED TO LEAF-NODE ACCESSES UNDER THE YCSB-E WORKLOAD

		ART	SMART	Sphinx
u64	# roundtrips	101.5 (96.1%)	15.4 (87.0%)	14.6 (85.5%)
	size (KB)	24.6 (74.1%)	16.6 (78.5%)	15.4 (85.0%)
email	# roundtrips	149.3 (90.0%)	24.7 (72.1%)	19.9 (89.3%)
	size (KB)	26.9 (72.0%)	15.8 (87.5%)	14.5 (95.2%)

fully saturate available network bandwidth, resulting in lower observed throughput compared to Sphinx. Additionally, unlike Sphinx, RACE does not support range queries due to its unordered storage layout.

G. In-Depth Analysis

In this section, we conduct an in-depth analysis for the network overhead of ART-based indexes under the YCSB workloads to better explain the performance improvement of Sphinx observed in Section V-B.

Range Query (YCSB E). Table IV presents the average network overhead per index operation, including the number of roundtrips and the access size, as well as the fraction of these overheads attributed to leaf-node accesses under the YCSB-E workload. For both Sphinx and SMART, the majority of roundtrips and bandwidth consumption (i.e., access size) during range queries (72.1%∼95.2%) is attributed to leaf-node accesses. This behavior stems from the fact that, although the inner-node traversal overhead in range queries is comparable to that in point queries, the leaf-node access overhead increases substantially because multiple leaf nodes must be retrieved to cover the key-value entries within the queried range. Since Sphinx primarily reduces the network overhead of inner-node traversal, while the leaf-node access overhead is largely determined by the scan length of each range query, its performance advantage over SMART becomes less pronounced for range queries compared to point queries.

Table IV further shows that both Sphinx and SMART incur significantly fewer roundtrips than ART due to the use of doorbell batching. Specifically, Sphinx and SMART retrieve multiple leaf nodes (eight by default) within a single network roundtrip, whereas ART fetches leaf nodes individually. As a result, Sphinx and SMART achieve substantially higher throughput than ART under the YCSB-E workload (see Fig. 5).

Base Operations (YCSB A-D, LOAD). Fig. 10 presents the average roundtrip number and access size per operation under YCSB workloads. By incorporating *Flat Inner Node Table* and *Succinct Filter Cache* designs, Sphinx achieves up to 1.7× and 4.9× lower network roundtrips under the *u64* and *email* datasets, respectively, compared to SMART and ART. Moreover, Sphinx reduces bandwidth consumption by 2.2∼12.6× and 2.4∼15.7× under the *u64* and *email* datasets, due to smaller remote memory access sizes. The network overhead reduction in *email* datasets is more significant than that in *u64* datasets, as the larger key sizes in *email* datasets lead to higher traversal depth for SMART and ART, while Sphinx maintains a constant traversal overhead regardless of the key size.

VI. RELATED WORK

Disaggregated Index. Driven by the numerous benefits of memory disaggregation, such as improved resource utilization and enhanced scalability, researchers have proposed a variety of index structures specifically designed for DM architectures. RACE [35], Sephash [42], and RCuckoo [43] are one-sided RDMA-conscious extendible hashing indexes that enable fast point queries, but lack support for range queries. FG [20] represents the first B$^+$Tree variant tailored for DM architectures, supporting both point and range queries via one-sided RDMA primitives. Building upon FG, Sherman [2] introduces hierarchical on-chip locks and a two-level versioning mechanism to further accelerate concurrent accesses. CHIME [19] adopts hopscotch-hashing-based leaf nodes to mitigate read amplification during lookups. SMART [17], a disaggregated radix tree, employs a hybrid concurrency control mechanism to reduce locking overhead. However, these tree-based disaggregated indexes still suffer from excessive network roundtrips due to their hierarchical traversal patterns. To address this, ROLEX [44] proposes a learned index that leverages learned models to predict the location of key-value entries, thereby reducing the network overhead during index traversal. Nonetheless, ROLEX introduces considerable read amplification under dynamic workloads and does not support variable-sized keys.

Adaptive Radix Tree. ART [12] is a widely adopted range index in storage systems, primarily due to its efficient support for variable-sized keys and its space-efficient design.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

However, in DM architectures, ART suffers from excessive network roundtrips during index traversal. SMART [17], the state-of-the-art DM-based ART, introduces a node-based caching mechanism that stores frequently accessed inner nodes in the local cache of compute nodes. Nevertheless, the limited cache capacity on compute nodes allows only a small fraction of inner nodes to be cached. While this mitigates traversal overhead to some extent, it does not fundamentally eliminate the problem. ERT [16], a persistent memory-based radix tree, reduces traversal depth by increasing the tree's fanout through an extendible hash structure. However, under DM architectures, it still incurs substantial network overhead when accessing each tree level. Cuckoo trie [45], a prefix-based tree designed for local DRAM, utilizes a hashed trie representation to exploit memory-level parallelism during lookup operations. However, it suffers from significant read amplification, as it pessimistically accesses all possible inner nodes during each search to ensure correctness.

VII. Conclusion

In this paper, we propose Sphinx, a novel hybrid index supporting variable-length keys on DM. Sphinx introduces *Flat Inner Node Table* to break the serial dependency in ART, offering an opportunity to mitigate the excessive network roundtrips during tree traversal with parallel hash entry reads. Sphinx further employs *Succinct Filter Cache* to significantly reduce network bandwidth usage during index operations while keeping the cache in compute nodes succinct and coherent. Our evaluations demonstrate that Sphinx outperforms existing counterparts by up to 7.3× in the YCSB benchmark.

References

[1] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 33–48.

[2] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed B+ tree index on disaggregated memory," in *Proc. Int. Conf. Manage. Data*, 2022, pp. 1033–1048.

[3] M. Zhang, Y. Hua, P. Zuo, and L. Liu, "FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory," in *Proc. 20th USENIX Conf. File Storage Technol. (FAST)*, 2022, pp. 51–68.

[4] J. Shen et al., "Ditto: An elastic and adaptive memory-disaggregated caching system," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 675–691.

[5] H. A. Maruf et al., "TPP: Transparent page placement for CXL-enabled tiered-memory," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2023, pp. 742–755.

[6] M. Zhang, Y. Hua, and Z. Yang, "Motor: Enabling multi-versioning for distributed transactions on disaggregated memory," in *Proc. 18th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2024, pp. 801–819.

[7] R. Wang, J. Wang, and W. G. Aref, "Cache coherence over disaggregated memory," *Proc. VLDB Endow.*, vol. 18, no. 9, pp. 2978–2991, 2025.

[8] X. Wang, Y. Li, K. Wu, W. Zhu, Y. Li, and Y. Xu, "FineMem: Breaking the allocation overhead vs. memory waste dilemma in fine-grained disaggregated memory management," in *Proc. 19th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2025, pp. 57–74.

[9] H. Lu et al., "Fast distributed transactions for RDMA-based disaggregated memory," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2025, pp. 943–958.

[10] R. Wang, J. Wang, P. Kadam, M. T. Özsu, and W. G. Aref, "DLSM: An LSM-based index for memory disaggregation," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2835–2849.

[11] Compute Express Link, "About CXL™," 2024. Accessed: Jun. 1, 2026. [Online]. Available: https://www.computeexpresslink.org/about-cxl

[12] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 38–49.

[13] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The art of practical synchronization," in *Proc. 12th Int. Workshop Data Manage. New Hardw.*, 2016, pp. 1–8.

[14] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 257–270.

[15] S. Ma et al., "ROART: Range-query optimized persistent art," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 1–16.

[16] K. Wang, G. Yang, Y. Li, H. Zhang, and M. Gao, "When tree meets hash: Reducing random reads for index structures on persistent memories," *Proc. ACM Manage. Data*, vol. 1, no. 1, pp. 1–26, 2023.

[17] X. Luo et al., "SMART: A high-performance adaptive radix tree for disaggregated memory," in *17th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2023, pp. 553–571.

[18] X. Wei, R. Chen, and H. Chen, "Fast RDMA-based ordered Key-Value store using remote learned cache," in *Proc. 14th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, USENIX Assoc., Nov. 2020, pp. 117–135. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/wei

[19] X. Luo, J. Shen, P. Zuo, X. Wang, M. R. Lyu, and Y. Zhou, "Chime: A cache-efficient and high-performance hybrid index on disaggregated memory," in *Proc. ACM SIGOPS 30th Symp. Oper. Syst. Princ.*, 2024, pp. 110–126.

[20] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast RDMA-capable networks," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 741–758.

[21] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Experiments Technol.*, 2014, pp. 75–88.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2018, pp. 69–87.

[24] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022, pp. 417–433.

[25] G. Li, W. Tian, J. Zhang, R. Grosman, Z. Liu, and S. Li, "GaussDB: A cloud-native multi-primary database with compute-memory-storage disaggregation," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 3786–3798, 2024.

[26] X. Yang et al., "PolarDB-MP: A multi-primary cloud-native database via disaggregated shared memory," in *Proc. Companion Int. Conf. Manage. Data*, 2024, pp. 295–308.

[27] X. Luo, R. Alagappan, and A. Ganesan, "SplitFT: Fault tolerance for disaggregated datacenters via remote memory logging," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 590–607.

[28] X. Pang and J. Wang, "Understanding the performance implications of the design principles in storage-disaggregated databases," *Proc. ACM Manage. Data*, vol. 2, no. 3, pp. 1–26, 2024.

[29] Q. Yu, C. Guo, J. Zhuang, V. Thakkar, J. Wang, and Z. Cao, "CaaS-LSM: Compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure," *Proc. ACM Manage. Data*, vol. 2, no. 3, pp. 1–28, 2024.

[30] R. Qin et al., "Mooncake: Trading more storage for less computation—A KVCache-centric architecture for serving LLM chatbot," in *Proc. 23rd USENIX Conf. File Storage Technol. (FAST)*, 2025, pp. 155–170.

[31] B. Lu, K. Huang, C.-J. M. Liang, T. Wang, and E. Lo, "DEX: Scalable range indexing on disaggregated memory," *Proc. VLDB Endow.*, vol. 17, no. 10, pp. 2603–2616, 2024.

[32] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[33] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-and space-efficient bloom filters," in *Proc. Int. Workshop Experimental Efficient Algorithms*, New York, NY, USA: Springer, 2007, pp. 108–121.

[34] M. A. Bender et al., "Don't thrash: How to cache your hash on flash," in *Proc. 3rd Workshop Hot Topics Storage File Syst. (HotStorage)*, 2011, pp. 1–5.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.

[35] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, "One-sided RDMA-conscious extendible hashing for disaggregated memory," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2021, pp. 15–29.
[36] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2016, pp. 437–450.
[37] J. Wang, Q. Wang, Y. Zhang, and J. Shu, "Deft: A scalable tree index for disaggregated memory," in *Proc. 20th Eur. Conf. Comput. Syst.*, 2025, pp. 886–901.
[38] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed lock management with RDMA: Decentralization without starvation," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1571–1586.
[39] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proc. 25th Symp. Oper. Syst. Princ.*, 2015, pp. 87–104.
[40] J. Shen et al., "FUSEE: A fully memory-disaggregated key-value store," in *Proc. 21st USENIX Conf. File Storage Technol. (FAST)*, 2023, pp. 81–98.
[41] Fonxat, "300 million email database," 2018. Accessed: Jun. 1, 2026. [Online]. Available: https://archive.org/details/300MillionEmailDatabase
[42] X. Min et al., "SepHash: A write-optimized hash index on disaggregated memory via separate segment structure," *Proc. VLDB Endow.*, vol. 17, no. 5, pp. 1091–1104, 2024.
[43] S. Grant and A. C. Snoeren, "Cuckoo for clients: Disaggregated cuckoo hashing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2025, pp. 959–972.
[44] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "Rolex: A scalable RDMA-oriented learned key-value store for disaggregated memory systems," in *Proc. 21st USENIX Conf. File Storage Technol. (FAST)*, 2023, pp. 99–114.
[45] A. Zeitak and A. Morrison, "Cuckoo trie: Exploiting memory-level parallelism for efficient DRAM indexing," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 147–162.



Bowen Zhang received the B.S. degree in 2021 from Shanghai Jiao Tong University, where he is currently working toward the Ph.D. degree. His research interests include disaggregated memory systems, persistent memory systems, and key-value stores.



Shengan Zheng received the B.S. and Ph.D. degrees from Shanghai Jiao Tong University in 2014 and 2019, respectively. He is currently an Associate Professor with Shanghai Jiao Tong University. His research interests include persistent memory-based storage systems, file systems, and distributed systems.



Jingxiang Li received the B.S. degree in 2023 from Shanghai Jiao Tong University, where he is currently working toward the M.S. degree. His research interests include disaggregated memory systems and key-value stores.



Shi Shu received the B.S. degree in 2024 from the Northwestern Polytechnical University. He is currently working toward the Ph.D. degree with Shanghai Jiao Tong University. His research interest includes disaggregated memory systems for large language models.



Zhenlin Qi received the B.S. degree in 2021 from Shanghai Jiao Tong University, where he is currently working toward the Ph.D. degree. His research interests include file systems, memory management systems, and distributed system designs.



Linpeng Huang (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University, in 1989 and 1992, respectively. He is currently a Professor in computer science with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service-oriented computing.



Hong Mei (Fellow, IEEE) received the Ph.D. degree from Shanghai Jiao Tong University, China, in 1992. He is a Professor of computer science with Shanghai Jiao Tong University and Peking University, China. His research interests include software engineering and system software. He is a fellow of TWAS, a member of Chinese Academy of Sciences, and a Foreign Member of the Academia Europaea.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on November 03,2026 at 12:40:59 UTC from IEEE Xplore. Restrictions apply.