Sphinx: A High-Performance Hybrid Index for Disaggregated Memory With Succinct Filter Cache

Jingxiang Li[§], Shengan Zheng[‡*], Bowen Zhang[§], Hankun Dong[§], Linpeng Huang[§*]
[§]Shanghai Jiao Tong University [‡]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University
[*]Corresponding authors: {shengan, lphuang}@sjtu.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132093

***Abstract*—Disaggregated memory (DM) architecture physically separates computing and memory resources into distinct pools interconnected via high-speed networks within data centers, with the aim of improving resource utilization compared to traditional architectures. Most existing range indexes for DM that support variable-length keys are based on adaptive radix trees. However, these indexes exhibit suboptimal performance on DM due to excessive network round trips during tree traversal and inefficient node-based caching mechanisms.**

To address these issues, we propose Sphinx, a novel hybrid index for DM. Sphinx introduces an Inner Node Hash Table to minimize the network round trips during index operations by replacing the sequential tree traversal with parallel hash reads. Sphinx incorporates a Succinct Filter Cache to further minimize network overhead while keeping the computing-side cache small and coherent. Experimental results show that Sphinx outperforms state-of-the-art counterparts by up to 7.3× in the YCSB benchmark.

I. Introduction

Disaggregated Memory (DM) architecture [1]–[5] has gained significant attention in both academia and industry in recent years. DM separates the memory and computation resources into distinct pools. This separation allows for independent scaling of these resources, which addresses the problem of low resource utilization in modern data centers. Since the memory side has minimal computing power, clients on the computing side typically access memory resources on the memory side via high-speed networks such as RDMA [6] and CXL [7], which enables clients to directly read from or write to remote memory without involving remote CPUs.

The Adaptive Radix Tree (ART) [8], [9] is a widely used index structure in many data center applications, such as key-value storage systems and transaction processing systems, due to its efficient support for variable-length keys. However, existing ARTs [9]–[12] exhibit suboptimal performance on DM due to excessive network round trips and the inefficient node-based caching mechanism. Specifically, executing index operations on tree-based indexes like ART requires traversing the hierarchical tree structure [13]. On DM, this traversal involves multiple network round trips between computing-side clients and the memory-side index. Since accessing remote memory remains significantly slower than accessing local memory, these excessive round trips result in high latency and become the primary performance bottleneck during index operations.

Existing ART for DM [10] attempts to reduce tree traversal round trips by caching certain inner nodes on the computing side. However, this node-based caching mechanism is inefficient on DM. Firstly, the computing side of DM has limited memory resources, allowing only a small subset of inner nodes to be cached. As a result, node-based caching fails to effectively reduce network round trips, leading to diminished performance. Furthermore, node-based caching introduces potential cache coherence issues, as remote inner nodes may be modified by other clients, causing inconsistencies between the computing-side cache and the memory-side index.

To address these issues, we propose Sphinx, a hybrid index designed to minimize network overhead during index operations while keeping the computing-side cache small and coherent. To reduce network round trips, Sphinx incorporates an *Inner Node Hash Table* to store the essential metadata of ART inner nodes. This allows clients to directly access inner nodes through reading hash entries in parallel, rather than traversing the tree sequentially. To achieve optimal performance with limited computing-side memory resources, we propose the *Succinct Filter Cache*. The succinct filter cache further reduces the network overhead by minimizing hash entries read to one in most cases. Instead of caching the full content of inner nodes, the succinct filter cache employs a succinct (space-efficient) data structure (e.g., cuckoo filter [14]) to track only the existence of inner nodes. As a result, the succinct filter cache significantly improves memory utilization and preserves coherence when remote inner nodes are modified by other clients.

The contributions of this paper are summarized as follows:

- We thoroughly analyze existing ARTs on DM, demonstrating that they provide inferior performance due to excessive network round trips introduced by traversing the hierarchical tree structure and the unsuitable node-based caching mechanism.
- We propose Sphinx, a hybrid range index supporting variable-length keys on DM. By introducing the *Inner Node Hash Table*, Sphinx minimizes network round trips by replacing the sequential tree traversal with reading hash entries in parallel.
- We propose the *Succinct Filter Cache*, which further reduces the network overhead of Sphinx and enables it to achieve optimal performance with limited computing-side memory resources.
- We conduct extensive evaluations of Sphinx and compare it with the state-of-the-art range indexes for variable-length keys on DM. Experimental results show that

Sphinx outperforms existing systems in terms of both throughput and latency.

II. Background And Motivation

A. Disaggregated Memory Architecture

Disaggregated memory (DM) architecture physically separates computing resources (e.g., CPU cores) and memory resources (e.g., DRAM) into distinct pools within data center, which aims to improve the resource utilization in traditional data center architectures [5], [15], [16]. In a DM setup, Compute Nodes (CNs) contain extensive computing resources but have limited memory. Conversely, Memory Nodes (MNs) are equipped with substantial memory resources but only wimpy computing power. CNs and MNs are interconnected via high-speed networks such as RDMA [6] and CXL [7].

RDMA is a promising solution for disaggregated memory architecture in modern data centers, as it not only offers high bandwidth (e.g., 200 Gbps) and low latency (e.g., 2 μs) [17]–[19] but also provides a set of one-sided operations (`RDMA Read`/`Write`/`CAS` (`Compare And Swap`) /`FAA` (`Fetch And Add`)), allowing clients in CNs to directly access remote memory without requiring CPU intervention on the MN side. This capability is particularly beneficial for DM, where minimizing CPU overhead on MNs with limited computing resources is critical.

B. Adaptive Radix Tree

The Adaptive Radix Tree (ART) [8] is a space optimized variant of the radix tree [20]. As shown in the memory nodes of Fig. 1, each inner node in ART represents a unique prefix of a key stored in the tree. This characteristic allows it to efficiently support variable-length keys. To improve space utilization, ART employs inner nodes with different capacities (e.g., 4, 16, 48 and 256) to store their children, allowing them to grow dynamically as new keys are inserted into the tree. To reduce tree height and enhance the performance of index operations, ART incorporates a path compression mechanism to merge multiple inner nodes with only one child into a single inner node (e.g., the inner node marked as *LYR* in Fig. 1) and eliminate inner nodes along the path to a single leaf node (e.g., the path from *LYR* to *LYRICS* in Fig. 1).

Existing ARTs [9], [10], [12] exhibits suboptimal performance on DM due to excessive network round trips. ARTs follow a hierarchical structure [13], in which a node can only be accessed after its address is retrieved from its parent node. In DM architectures, the index resides on the MN side with sufficient memory, which is separated from the clients on the CN side. This separation necessitates excessive network round trips during tree traversal, resulting in high latency and early saturation of network resources.

Existing ART for DM [10] attempts to mitigate this issue by caching certain inner nodes. During index operations, the client first traverses the tree formed by the cached inner nodes locally, then proceed to traverse the remote tree from the deepest node reached in the local traversal. However,

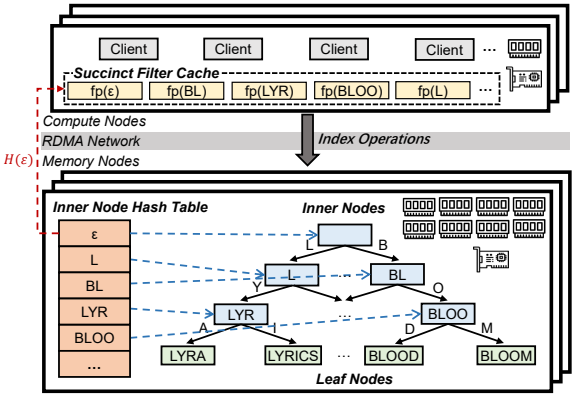


Fig. 1: Overview of Sphinx.

this node-based caching mechanism is inefficient in memory-limited CNs. First, node based caching requires a significant amount of memory to achieve optimal performance, which is impractical for memory-limited CNs. For example, as shown in Sec. V-B, SMART [10], the state-of-the-art ART on DM, still deliver suboptimal performance even with a cache size set to 17.6% of the dataset size. Second, node-based caching can lead to cache coherence issues, as remote ART nodes may be updated by other clients. Although SMART mitigates this issue by preallocating a space of Node-256 for each inner node and employing an associated reverse checking mechanism, it incurs significant memory overhead on the MN side. The overhead ranges from 2.1–3.0× in our experiments, as detailed in Sec. V-D.

III. Design

Based on the above analysis, we propose Sphinx, a hybrid index for variable-length keys on DM, designed to minimize network overhead in index operations while keeping the CN-side cache small and coherent.

The architecture of Sphinx is illustrated in Fig. 1. The ART Nodes of Sphinx are evenly distributed across MNs by consistent hashing [21]. To minimize network round trips during index operations, each MN in Sphinx employs an *Inner Node Hash Table* to store the addresses and lightweight metadata of ART inner nodes allocated to that MN. This reduces network round trips during index operations by replacing the sequential tree traversal with reading hash entries in parallel. To further reduce the network overhead, Sphinx introduces a *Succinct Filter Cache* that efficiently filter out unnecessary hash entry reads, thereby reducing network bandwidth consumption. The succinct filter cache adopts a succinct data structure (e.g., cuckoo filter [14]) to track the existence of inner nodes, rather than storing their full content. This approach not only minimizes memory consumption on the CN side, but also eliminates cache coherence issues that could arise from directly caching the inner nodes.

A. Inner Node Hash Table

To reduce network round trips, Sphinx introduces an MN-side *Inner Node Hash Table* for the ART, designed to break

the serial dependencies between ART nodes. As illustrated in Fig. 1, we construct a hash table for the inner nodes of the ART, where the key is the full prefix of each inner node, while the value stores the node's address and its associated metadata. During index operations, instead of sequentially fetching each inner node through tree traversal, the client is able to read all the hash entries of these inner nodes in parallel. Based on these hash entries, the client then retrieves the deepest inner node with another network round trip.

With the inner node hash table, locating the value corresponding to a key can be optimized in a parallel manner. Specifically, for a given key K (e.g., *LYRICS* in Fig. 1), the client retrieves all hash entries corresponding to the prefixes of K (*LYRICS*, *LYRIC*, ..., L) in parallel to find the deepest inner node corresponding to K. This approach stems from the path compression mechanism [8] used in ART, which prevent the client from directly determining the existence of each inner node, necessitating a read of all possible hash entries (can be further optimized to only read one hash entry in most cases, as described in Sec. III-B). We adopt RACE hashing [22] as the index of the inner node hash table, which allows each hash entry to be retrieved within a single network round trip. By further utilizing the doorbell batching mechanism [23], which allows sending multiple RDMA operations to the NIC in a single batch, reading all these hash entries can be performed in a single round trip.

To further retrieve the value of K with these hash entries, the client then identifies the valid hash entry corresponding to the longest prefix (*LYR*), which represents the deepest inner node. Next, the client retrieves this deepest inner node from MNs using the metadata and address stored in the hash entry. Finally, the client retrieves the target leaf node through this deepest inner node and extracts the value from the leaf node.

The inner node hash table incurs only a slight memory consumption on the MN side. Specifically, it is significantly smaller than the inner nodes themselves, as each hash entry only stores the address of the inner node and associated metadata (e.g., a short fingerprint and node type as shown in Fig. 3), fitting within just 8 bytes. In comparison, an ART inner node consumes 40–2056 bytes of memory. Our experiments in Sec. V-D show that the inner node hash table incurs only 3.3–4.9% additional memory usage on the MN side.

B. Succinct Filter Cache

Although reading inner nodes directly from the inner node hash table reduces network round trips, it does not alleviate bandwidth consumption. Specifically, for a key of length L, the client must read $\Theta(L)$ hash entries to locate the deepest inner node, offering no advantage over tree traversal. Additionally, similar to tree traversal, reading multiple hash entries generates a significant load on network, as the NIC must process a substantial number ($\Theta(L)$) of network packets for each index operation. This high load accelerates the saturation of network resources, ultimately diminishing throughput of the system.

To address the above issue, we propose the *Succinct Filter Cache*, a lightweight per-CN cache based on cuckoo filter [14].

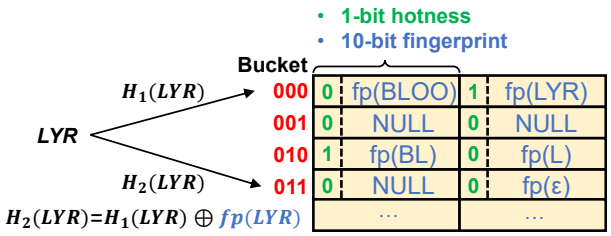


Fig. 2: Succinct Filter Cache.

The succinct filter cache reduces the number of hash entry reads per index operation from $\Theta(L)$ to one in most cases, while maintaining minimal space consumption and preserving cache coherence. The succinct filter cache tracks the existence of inner node prefixes in the remote index using a cuckoo filter, a succinct data structure. This allows the client to locally determine the prefix of the deepest inner node for the key with minimal CN-side memory usage. Consequently, the client only needs to fetch a single hash entry corresponding to the deepest inner node, significantly reducing network overhead associated with reading multiple hash entries. Additionally, the succinct filter cache maintains coherence when remote inner nodes are modified, as it does not store their content.

As illustrated in Fig. 2, the succinct filter cache follows the structure of cuckoo filter. To perform an existence check (i.e., to determine whether a given prefix exists in the succinct filter cache), the client computes the hash values (H_1 and H_2) of the prefix and checks if the corresponding hash fingerprint (fp) is present in the candidate buckets determined by the hash values. During index operations, for a given key K (e.g., *LYRICS*), the client first performs existence checks for all prefixes of K (*LYRICS*, *LYRIC*, ..., L) in the succinct filter cache and identify the longest prefix P that exists in the succinct filter cache (*LYR*). The client then only fetches the hash entry of P from remote inner node hash table. Finally, the client retrieves the deepest inner node and further retrieves the leaf node to extract the value corresponding to K.

In most cases, as described above, an index operation can be performed in three network round trips: reading the hash entry, reading the inner node, and retrieving the leaf node. Due to the probabilistic nature of cuckoo filter, in false-positive cases (<1%), the succinct filter cache may incorrectly indicate that a prefix exists in the remote index when it does not. Nevertheless, both the performance and correctness of index operations remains unaffected by these rare cases. As illustrated in Fig. 3, both the hash entry and the inner node contain metadata that allows the client to avoid traversing the tree starting from an unmatched inner node. For example, the hash entry includes a 12-bit hash fingerprint (fp_2), while the inner node header provides a 42-bit full prefix hash. In the rare scenario where both the fp_2 and the full prefix hash collide, Sphinx may traverse the tree starting from an unmatched inner node U. However, the correctness is still guaranteed, as the client will eventually reach a leaf node to retrieve the corresponding value. At this point, the client computes the common prefix between the key used in the index operation

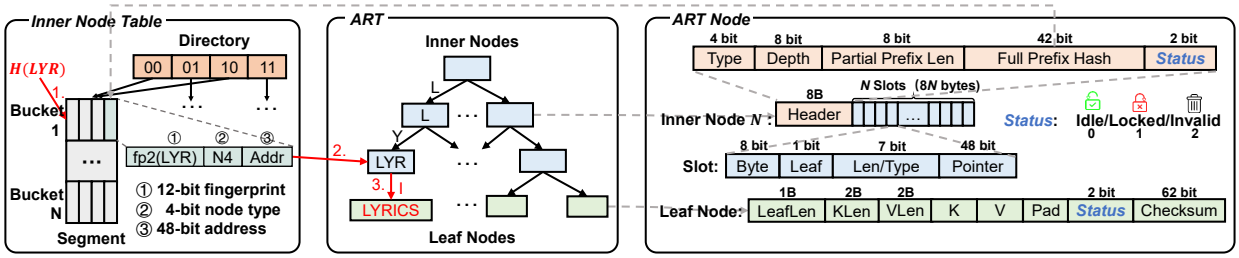


Fig. 3: Detailed data structures of Sphinx

and the key stored in the leaf node. If the common prefix is shorter than the prefix associated with U, the client identifies the false positive and retries the index operation with a shorter prefix from the succinct filter cache. Since such cases are extremely rare ($<$0.01%), the overall performance remains unaffected.

To address scenarios where the dataset size is so large that even tracking the existence of all inner nodes in CNs becomes infeasible, as illustrated in Fig. 2, we further propose a lightweight mechanism to identify hot prefixes in the succinct filter cache. This mechanism is inspired by the second chance page replacement policy [24], which approximates the LRU algorithm using only one additional *hotness* bit per entry. Specifically, when a new prefix is inserted, its *hotness* bit is initialized to 0, indicating it has not been recently used. The *hotness* bit is set to 1 when the entry is accessed, marking it as recently used. During insert operations, if both candidate buckets are full, the client randomly selects an entry with the *hotness* bit set to 0 for replacement. If all entries in both candidate buckets have their *hotness* bits set to 1, the client adopts the cuckoo hashing mechanism [25] to relocate existing entries and make space for the new one. During relocation, the *hotness* bits of all relocated entries are reset to 0, indicating they are eligible for future eviction.

The succinct filter cache is particularly well-suited for DM. First, as demonstrated in prior work [14], using a 10-bit fingerprint per item is sufficient to achieve a low false positive rate ($<$1%) for existence checks, making it significantly more space-efficient than the node-based caching mechanism, which require 40–2056 bytes of memory to cache a single inner node. Second, the succinct filter cache inherently maintains coherence even when inner nodes in the remote memory are modified by other clients. These modifications impact only types, slots, and partial prefixes of inner nodes, leaving their full prefixes unchanged. Consequently, they have no effect on the coherence of the succinct filter cache, which exclusively tracks the existence of full prefixes.

C. Concurrency Control

Sphinx employs a lightweight concurrency control mechanism that enables lock-free read operations and node-grained locks for write operations. To further reduce network overhead in update operations, Sphinx adopts a checksum-based in-place update scheme for leaf nodes.

For concurrency control, as illustrated in Fig. 3, most components (e.g., hash entries, headers and slots) of Sphinx fit within 8 bytes, enabling clients to modify them atomically using the `RDMA CAS` operation. In the inner node hash table, both read and write operations are executed without acquiring locks, as a write operation only affects an 8-byte hash entry and can be performed atomically. For the ART, Sphinx resolves write-write conflicts using a node-grained lock (status) and stores lightweight metadata (e.g., the type and depth of the inner node) in the header of inner nodes to synchronize lock-free reads with a write.

With the consistency maintained in both the ART and the inner node hash table, the correctness of Sphinx is ensured in most cases. The remaining correctness issue arises when a client reads an outdated hash entry while another client is performing a *node type switch*. In ART, a node type switch occurs when a client attempts to insert a new key into a full inner node. In such cases, a new inner node with more slots is allocated, and the original node is replaced by the new one. However, after the original node is removed from the ART, a client may still access it through the inner node hash table before the change propagates to the hash table, resulting in a temporary inconsistency. To address this issue, as shown in Fig. 3, the client sets the status field of the old inner node to *Invalid* following a node type switch. Readers retrieving an inner node from MNs check the status field before use. If the status field is marked *Invalid*, the reader retries the index operation.

For the update operation, if the new value fits within the original leaf node, the client performs an in-place update to reduce network overhead. As shown in Fig. 3, the client begins the in-place update by locking the leaf node, setting its status field to *Locked* using one `RDMA CAS` operation. Sequentially, the client writes the new value, updates the status field to *Idle* and computes the new checksum of the leaf node locally. Finally, the client writes the entire leaf node back to corresponding MN using a single `RDMA Write` operation. To avoid reading partially modified data, all clients verify the checksum of a leaf node before using it. This in-place update scheme eliminates an RDMA operation during an update operation by combining the lock release with the value write, thereby reducing network overhead.

IV. Index Operations

In this section, we present the detailed implementation of the index operations in Sphinx. By leveraging the inner node hash table and succinct filter cache, Sphinx achieves high performance in index operations, primarily by replacing the

sequential tree traversal with a single hash entry read. This substantially reduces network overhead in terms of both round trips and bandwidth consumption.

Search (K). The client first calculates the hash values of all prefixes of K and identifies the longest prefix P exists in the succinct filter cache. As shown in Fig. 3, the client then fetch the corresponding hash bucket from the inner node hash table in one round trip. To avoid additional network round trips in determining the address of buckets through the directory, each CN maintains a local directory cache, which is small enough to fit within limited CN-side memory (typically 2–5% of the succinct filter cache size). The client then iterates through the hash bucket and fetch all the inner nodes whose hash entry has the same 12-bit fingerprint (fp_2) as P. Typically, there is only one candidate hash entry, as discussed in prior works [26]. Finally, the client performs tree traversals starting from the fetched inner nodes, ultimately reaching the leaf nodes, where the value corresponds to K is extracted. In most cases, the tree traversal requires only one additional network round trip, as the succinct filter cache enables the client to determine the deepest inner node of K. In rare cases, the client may fetch an inner node that is not the deepest due to the structural modifications made by other clients to the remote ART. In such case, the client updates the succinct filter cache for any prefixes not present in the cache to maintain its freshness.

Insert (K, V). The client begins by locating the deepest inner node following the search procedure, and then traverses the tree starting from the node until it reaches a leaf node, verifying that the inner node prefix matches K. Subsequently, the client writes the new leaf node with the new key-value pair using an `RDMA write`. If the operation triggers a node type switch or a node split, the client also writes a new inner node with a slot pointing to the newly created leaf node. To ensure consistency, the client acquires node-grained locks on the affected inner node by setting its status field to $Locked$ with an `RDMA CAS`. This lock acquisition is piggybacked onto the write using doorbell batching [23] to reduce network round trips. Once the lock is acquired, The client makes the insertion visible by atomically install the pointer to the new node in the parent node's slot with an `RDMA CAS`, followed by a piggybacked lock release using another `RDMA CAS`.

After the operation is completed, the client updates the succinct filter cache if the operation triggers a node split, where a new inner node with a new prefix is added to the index. The client updates only its local succinct filter cache, while the synchronization of caches on other CNs is deferred until their clients encounter the corresponding inner node during tree traversal, as described in the search procedure. Similarly, the inner node hash table is updated after a node type switch or a node split to reflect changes applied to the ART. This update can be performed atomically using an `RDMA CAS`, as the client modifies only one 8-byte hash entry.

Update (K, V). The client first retrieves corresponding leaf node following the search procedure. The structure of the leaf node is shown in Fig. 3. Similar to prior works [10], [22], the size of the leaf node is aligned to 64 B, and the unit of the $LeafLen$ field is 64 B. If the leaf node has sufficient space to store V (i.e., the required size is within $64 \times LeafLen$ B), the client performs an in-place update as described in Sec. III-C. Otherwise, the client performs an out-of-place update similar to the insert operation.

Delete (K). The client retrieves the leaf node following the search process and sets its status to $Invalid$ using an `RDMA CAS` operation. The client then updates corresponding slot of its parent node to NULL similar to the insert operation.

Scan (K_1, K_2). The client traverses the tree starting from the root node, retrieves all leaf nodes with keys between K_1 and K_2, and adds extracted key-value pairs to the result set.

V. Evaluation

A. Experimental Setup

Testbed. We emulate an RDMA-based DM cluster with three dual-socket machines, each equipped with 128GB of DRAM, two 2.6GHz Intel Xeon Gold 6348 CPUs (each with 28 cores), and one 2×100 Gbps dual-port Mellanox ConnectX-6 NIC. To save machine resources, we run both a CN and an MN on each machine, as in prior works [10], [27]. We manually limit the index cache memory usage on CNs and the CPU core usage on MNs to simulate a DM cluster.

Workloads. We evaluate Sphinx using the YCSB benchmark [28]. Specifically, we use six types of 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 an additional LOAD workload (100% insert). By default, the workloads follow a zipfian key distribution with a skewness factor of 0.99. The size of the value for each key-value pair is set to 64 bytes.

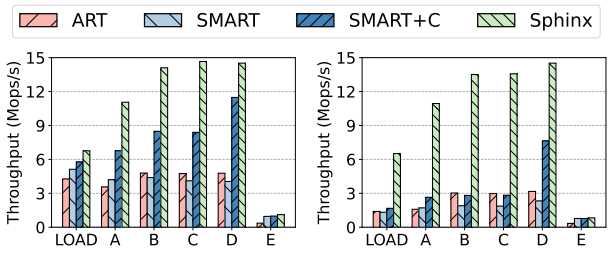
Datasets. Two datasets are used for evaluation: *u64* and *email*. The *u64* dataset consists of 8-byte fixed length integers generated from a uniform distribution, while the *email* dataset comprises publicly available email addresses [29] with sizes ranging from 2 to 32 bytes (an average size of 18.93 bytes).

Comparisons. We compare Sphinx with *SMART* [10], the state-of-the-art range index supporting variable-length keys on DM. Additionally, We port the original *ART* to DM and use it as a baseline. Similar to Sphinx and SMART, *ART* also adopts one-sided RDMA operations to access and modify the index and data stored in remote memory. The CN-side cache size of SMART and Sphinx is set to 20 MB, which is 4.2% of the u64 dataset size and 1.8% of the email dataset size. To further demonstrate the high performance of Sphinx with limited cache size, we also compare Sphinx with *SMART+C*, which has a CN-side cache size of 200 MB (41.7% and 17.6% of the u64 and email datasets, respectively).

B. Overall Performance

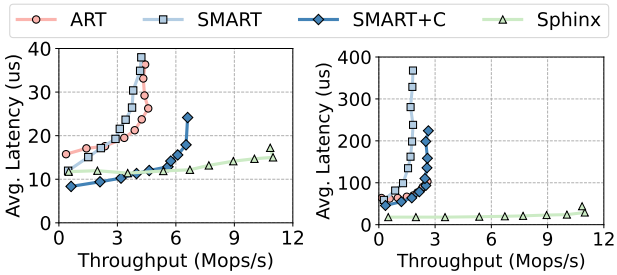
In this section, we evaluate the overall performance of Sphinx with the YCSB benchmark. Fig. 4 displays the throughput of the systems under the YCSB benchmark using the *u64* and *email* datasets.

Base Operations (YCSB A-D, LOAD). As shown in Fig. 4, Sphinx achieves 1.2–3.6× higher throughput on the u64



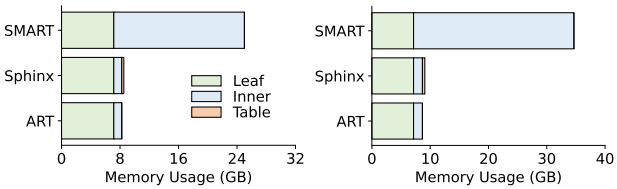
(a) u64 (b) email

Fig. 4: The throughput under the YCSB benchmark.



(a) u64 (b) email

Fig. 5: The scalability in the write-intensive YCSB-A workload.



(a) u64 (b) email

Fig. 6: The memory usage across different datasets

dataset and 1.9–7.3× higher throughput on the email dataset than SMART, SMART+C and ART. By combining the *Inner Node Hash Table* and *Succinct Filter Cache*, Sphinx effectively minimizes network round trips and bandwidth consumption during index operations by replacing the sequential tree traversal with a single hash entry read. This is especially beneficial for the email dataset, where larger key sizes and deeper tree structure exacerbate traversal costs. In contrast, SMART and ART suffer from excessive network round trips during index operations, leading to lower throughput. Furthermore, Sphinx outperforms SMART+C with only 10% of its CN-side cache, demonstrating the efficiency of the *Succinct Filter Cache* in achieving high performance with limited CN-side memory space. Finally, Sphinx maintains consistent performance across both u64 and email datasets, indicating its suitability for variable-length keys.

Range Query (YCSB E). For 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 [23], which hides the latency associated with reading nodes. The performance of Sphinx, SMART and SMART+C is comparable, as the main bottleneck of the workload lies in reading the leaf nodes rather than in tree traversal.

C. Scalability

In this section, we evaluate the scalability of the systems using the YCSB-A workload (50% read, 50% update, skewed) with both the *u64* and *email* datasets. Following prior works [10], [27], [30], we adopt coroutines to hide latency associated with polling the completion queue across all systems, with each coroutine treated as a worker. Fig. 5 presents the throughput-latency curve of the systems under the YCSB-A workload with various numbers of workers in CNs (6–192 workers, evenly distributed across 3 CNs). Sphinx scales well as the number of workers increases in CNs, achieving up to 2.6× higher throughput and 2.2× lower average latency on the u64 dataset, and up to 6.1× higher throughput and 11.7× lower average latency on the email dataset. This performance gain is primarily attributed to the inner node hash table and succinct filter cache, which minimize network overhead in terms of both round trips and bandwidth consumption, thereby preventing premature saturation of network resources.

D. Space Consumption

In this section, we evaluate the MN-side space consumption of the systems across different datasets. We insert 60 million key-value pairs into each system and measure the resulting MN-side memory usage. As shown in Fig. 6, the inner node hash table incurs only a slight increase in memory consumption compared to the original ART, with a 3.3% increase for the u64 dataset and a 4.9% increase for the email dataset. In contrast, SMART consumes 2.1–3.0× more memory than the original ART, as it preallocates a space of Node-256 for each inner node to prevent potential cache coherence problems.

VI. Conclusion

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

Acknowledgment

We sincerely thank the anonymous reviewers for their constructive comments and suggestions. This work is supported by National Key Research and Development Program of China (No. 2022YFB4500303), National Natural Science Foundation of China (NSFC) (No. 62332012, 62227809, 62302290), the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project (No. 2021SHZDZX0102), and Natural Science Foundation of Shanghai (No. 22ZR1435400).

REFERENCES

[1] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

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

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

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

[5] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "Mind: In-network memory management for disaggregated data centers," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 488–504.

[6] InfiniBand Trade Association, "InfiniBand Trade Association," 2024, accessed: 2024-11-19. [Online]. Available: https://www.infinibandta.org/

[7] Compute Express Link, "About CXL™," https://www.computeexpresslink.org/about-cxl, 2024, accessed: 2024-11-06.

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

[9] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The art of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–8.

[10] X. Luo, P. Zuo, J. Shen, J. Gu, X. Wang, M. R. Lyu, and Y. Zhou, "SMART: A High-Performance adaptive radix tree for disaggregated memory," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 553–571.

[11] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "ROART: range-query optimized persistent ART," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 1–16.

[12] 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 *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 257–270.

[13] A. Zeitak and A. Morrison, "Cuckoo trie: Exploiting memory-level parallelism for efficient dram indexing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 147–162.

[14] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[15] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *Proceedings of the international symposium on quality of service*, 2019, pp. 1–10.

[16] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[17] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[18] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[19] NVIDIA, "Connectx-6 ethernet smartnic," https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024, accessed: 2024-11-04.

[20] D. R. Morrison, "Patricia – practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[22] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, "One-sided RDMA-Conscious extendible hashing for disaggregated memory," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 15–29.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 437–450.

[24] F. J. Corbato, *A paging experiment with the multics system*. Massachusetts Institute of Technology, 1968.

[25] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[26] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 371–384. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[27] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b+ tree index on disaggregated memory," in *Proceedings of the 2022 international conference on management of data*, 2022, pp. 1033–1048.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[29] Fonxat, "300 million email database," https://archive.org/details/300MillionEmailDatabase, 2018, accessed: 2024-11-05.

[30] X. Wei, R. Chen, and H. Chen, "Fast RDMA-based ordered Key-Value store using remote learned cache," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 117–135. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/wei