DART: A Lock-free Two-layer Hashed ART Index for Disaggregated Memory

BOWEN ZHANG, Shanghai Jiao Tong University, China
SHENGAN ZHENG*, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, China
SHI SHU, Shanghai Jiao Tong University, China
JINGXIANG LI, Shanghai Jiao Tong University, China
ZHENLIN QI, Shanghai Jiao Tong University, China
WEIQUAN HUANG, Shanghai Jiao Tong University, China
JIANGUO WANG, Purdue University, USA
LINPENG HUANG*, Shanghai Jiao Tong University, China
HONG MEI, Peking University, China

Disaggregated memory architecture decouples computing and memory resources into separate pools connected via high-speed interconnect technologies, offering substantial advantages in scalability and resource utilization. However, this architecture also poses unique challenges in designing effective index structures and concurrency protocols due to increased remote memory access overhead and its shared-everything nature.

In this paper, we present DART, a lock-free two-layer hashed Adaptive Radix Tree (ART) designed to minimize remote memory access while ensuring high concurrency and crash consistency in the disaggregated memory architecture. DART incorporates a hash-based *Express Skip Table* at its upper layer, which reduces the round trips of remote memory access during index traversal. In the base layer, DART employs an *Adaptive Hashed Layout* within ART nodes, confining remote memory accesses during in-node searches to small hash buckets. By further leveraging *Decoupled Metadata Organization*, DART achieves lock-free atomic updates, enabling high scalability and ensuring crash consistency. Our evaluation demonstrates that DART outperforms state-of-the-art counterparts by up to 5.8× in YCSB workloads.

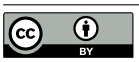
CCS Concepts: • **Information systems** → **Key-value stores**; **Data structures**; • **Computer systems organization** → **Distributed architectures**.

Additional Key Words and Phrases: Disaggregated Memory, Adaptive Radix Tree, Hash, Lock-free

ACM Reference Format:
Bowen Zhang, Shengan Zheng, Shi Shu, Jingxiang Li, Zhenlin Qi, Weiquan Huang, Jianguo Wang, Linpeng Huang, and Hong Mei. 2026. DART: A Lock-free Two-layer Hashed ART Index for Disaggregated Memory. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 22 (February 2026), 25 pages. https://doi.org/10.1145/3786636

*Corresponding author.

Authors' Contact Information: Bowen Zhang, Shanghai Jiao Tong University, Shanghai, China, bowenzhang@sjtu.edu.cn; Shengan Zheng, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China, shengan@sjtu.edu.cn; Shi Shu, Shanghai Jiao Tong University, Shanghai, China, shu-shi@sjtu.edu.cn; Jingxiang Li, Shanghai Jiao Tong University, Shanghai, China, lijingxiangkazexc@sjtu.edu.cn; Zhenlin Qi, Shanghai Jiao Tong University, Shanghai, China, qizhenlin@sjtu.edu.cn; Weiquan Huang, Shanghai Jiao Tong University, Shanghai, China, hwqqz678@sjtu.edu.cn; Jianguo Wang, Purdue University, West Lafayette, Indiana, USA, csjgwang@purdue.edu; Linpeng Huang, Shanghai Jiao Tong University, Shanghai, China, lphuang@sjtu.edu.cn; Hong Mei, Peking University, Beijing, China, meih@pku.edu.cn.



ACM 2836-6573/2026/2-ART22
https://doi.org/10.1145/3786636

1 Introduction

Disaggregated memory (DM) architectures [2, 11, 25, 27, 38, 41, 46, 54, 55, 59] have gained increasing popularity in recent years due to their superior scalability and efficient resource utilization. Unlike traditional distributed systems that tightly integrate computational and memory resources within monolithic servers, DM architectures separate these resources into a compute pool and a memory pool. Compute nodes are provisioned with high-performance CPU cores but limited local memory, while memory nodes provide substantial memory capacity without computational units. Compute nodes directly access data in the memory pool via high-speed interconnect technologies such as RDMA [7] and CXL [6]. This innovative architecture allows for independent scaling of compute and memory resources, thereby enhancing resource utilization and hardware elasticity [37, 39].

However, designing high-performance and highly scalable DM-based systems presents unique challenges. First, accessing remote memory via interconnects such as RDMA and CXL incurs significantly higher latency and reduced bandwidth compared to local memory access [13, 47, 48]. To achieve low latency and high throughput, DM-based systems must focus on minimizing both round trips and bandwidth consumption involved in remote memory access.

Second, DM-based systems face significant challenges in concurrency control as multiple client threads from compute nodes may concurrently access the shared memory pool. Effectively managing these concurrent accesses, particularly under skewed workloads, is crucial for achieving high scalability. Moreover, ensuring crash consistency during a client failure is essential for DM-based systems [49, 50, 56]. Without this, the failure of a single client could result in data remaining in a partially modified state, potentially exposing other concurrent clients to inconsistent data.

This paper focuses on index design over RDMA-based disaggregated memory. Adaptive Radix Tree (ART) [17] is a fundamental ordered index widely adopted in modern database systems, such as CedarDB [3], HyPer [12], and DuckDB [35], due to its effective handling of variable-sized keys and efficient memory utilization. However, our analysis reveals that conventional ART indexes suffer from excessive remote memory accesses and exhibit inefficient concurrency control when applied in disaggregated memory architectures.

Existing ART indexes [17, 25, 26] often result in excessive round trips and significant bandwidth consumption of remote memory accesses in DM-based architecture. Firstly, traversing the hierarchical levels of the ART structure necessitates multiple round trips, which severely impacts operation latency. SMART [25], the state-of-the-art DM-based ART, maintains an inner-node cache on compute nodes to reduce traversal overhead. However, the performance benefits are inherently limited by the constrained cache capacity typically available in compute nodes [24]. Secondly, during in-node searches at each level, current designs fail to strike a balance between the required round trips and bandwidth usage. A typical ART implementation [17] often requires accessing auxiliary metadata before traversing the nodes to narrow the search range, incurring additional round trips of remote memory access. To avoid this overhead, SMART performs in-node search by fetching the entire node in a single round trip. However, this design introduces significant read amplification.

Furthermore, current ART indexes lack efficient mechanisms to ensure both concurrent and crash consistency in disaggregated memory systems. Firstly, previous works [26, 42] rely on lock-based writes, which not only increase the round trips of remote memory access during locking and unlocking but also diminish scalability under high contentions, particularly in skewed and write-intensive workloads. Although SMART [25] employs a lock handover mechanism to alleviate performance degradation, it does not fundamentally eliminate lock-related overhead [40]. Secondly, in the event of client crashes, existing DM-based ART cannot guarantee data consistency in memory nodes, as its write operations are not failure-atomic. As a result, a client failure may leave the data within memory nodes in a partially updated and inconsistent state. While prior persistent

memory-based ART indexes [16, 26] ensure failure atomicity for write operations, they typically require extra round trips for logging or updating metadata, exacerbating remote memory access overhead when applied to DM-based systems.

In this paper, we propose DART, a lock-free two-layer hashed Adaptive Radix Tree (ART) designed to minimize remote memory accesses while effectively ensuring concurrency and crash consistency in DM-based architectures. DART adopts a two-layer hashed representation for the radix tree, leveraging the advantages of hash indexes for efficient point queries while preserving the radix tree's strengths in supporting variable-sized keys and range queries. Furthermore, DART employs lock-free concurrency control to achieve high concurrency while guaranteeing crash consistency.

We introduce an *Express Skip Table* as the upper layer of DART, leveraging a DM-friendly hash table to provide express paths to lower-level inner nodes of the radix tree during hierarchical tree traversals. This design effectively reduces the traversal depth, thereby minimizing the round trips of remote memory access during index operations. In the base layer, DART incorporates an *Adaptive Hashed Layout* within the inner nodes of the radix tree. By replacing traditional radix tree nodes with adaptive hash tables, DART reduces remote memory access during in-node searches to a small hash bucket.

Moreover, DART utilizes *Decoupled Metadata Organization* to facilitate lock-free concurrency control, achieving both high concurrency and crash consistency. By decoupling metadata from the data of each inner node and embedding it into their parent node pointers, DART enables failure-atomic updates using hardware atomic primitives. This allows DART to perform lock-free index operations, ensuring that each write operation is completed without exposing any intermediate state. Consequently, concurrent threads are prevented from reading inconsistent data, and the failure of a single client does not interfere with the regular operations of others.

We implement DART in an RDMA-based disaggregated memory cluster and conduct a comprehensive evaluation against state-of-the-art DM-based index structures. Under YCSB workloads, DART achieves up to 3.2× speedup over the state-of-the-art DM-based ART and up to 5.8× over the state-of-the-art B+Trees. To gain deeper insight into the sources of DART's performance advantages, we measure the network round trips and bandwidth consumption of each index under various workloads. Furthermore, we perform an ablation study on DART's design to quantify the impact of each design component on the overall performance improvement. The code is open-sourced at https://github.com/SJTU-DDST/DART.

Contributions. We conduct an in-depth analysis of the limitations of existing ART indexes in DM architecture. To address this, we propose DART, a novel two-layer hashed ART with lock-free concurrency control for DM-based systems. The core innovations of DART include:

- **Express Skip Table:** DART incorporates a DM-friendly hash table in the upper layer to provide express paths during tree traversal, minimizing the round trips of remote memory access.
- **Adaptive Hashed Nodes:** In the base layer, DART employs a hash-based layout within each tree node, mitigating read amplification during in-node search.
- **Decoupled Metadata Organization:** DART decouples the metadata from node data to achieve lock-free atomic updates, ensuring both high concurrency and crash consistency.

The remainder of this paper is organized as follows. Section 2 introduces the background of disaggregated memory architecture and adaptive radix tree, and presents the motivation behind our design. Section 3 describes the design of DART, highlighting three core components: *Express Skip Table*, *Adaptive Hashed Layout*, and *Decoupled Metadata Organization*. Section 4 discusses implementation details, including lock-free index operations and memory allocation mechanisms. Section 5 presents a detailed experimental evaluation, comparing DART with state-of-the-art DM-based indexes.

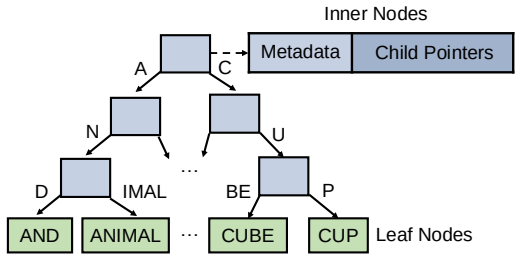


Fig. 1. Adaptive radix tree

2 Background and Motivation

2.1 Disaggregated Memory Architecture

Disaggregated memory (DM) architecture [11, 15, 27, 28, 36–39, 41, 44–47] is a new concept of separating monolithic servers into computing and memory pools. The compute pool comprises nodes equipped with powerful CPU cores but limited memory capacity, while the memory pool houses substantial amounts of volatile or persistent memory. This novel architecture facilitates the independent scaling of diverse resource pools, significantly improving resource utilization and hardware elasticity.

For DM architectures, clients in the compute pool access data stored in the memory pool through high-speed interconnects, such as RDMA [7] and CXL [10]. This paper primarily focuses on RDMA-based DM architectures, which are widely adopted in prior works [23–25, 41, 43–47, 59]. RDMA provides one-sided operations that allow compute nodes to directly read or write remote memory via `RDMA READ` and `RDMA WRITE`, bypassing the remote CPU and kernel. Moreover, RDMA supports one-sided atomic primitives (e.g., `RDMA CAS`, and `RDMA FAA`) that can atomically update an 8-byte word, enabling efficient concurrency control in DM-based systems.

However, DM architecture also introduces several new challenges. First, remote memory access via RDMA is substantially more expensive than local memory access. The latency of one-sided RDMA operations (~2us) is an order of magnitude higher than that of local memory access. Additionally, the bandwidth of RDMA (100~200 Gbps) is also lower than that of local memory. Consequently, DM-based systems must effectively minimize both round trips and bandwidth consumption for remote memory access to achieve low latency and high throughput.

Second, DM architecture enables numerous clients in compute nodes to concurrently access a shared memory pool, introducing new challenges for concurrency protocols. DM-based systems must be highly scalable to effectively manage concurrent accesses from numerous clients. Furthermore, it is crucial for clients to ensure crash consistency when accessing the shared memory pool [49, 50, 56]. If a client crashes while modifying shared memory, the data in memory nodes may be left in an inconsistent state. Without guaranteed crash consistency, the failure of a single client can result in other clients in the compute pool accessing inconsistent data.

2.2 Adaptive Radix Tree

The radix tree is a widely used prefix-based ordered index structure that efficiently supports variable-sized keys. As shown in Figure 1, each inner node in a radix tree represents a sequence of bits or characters, while the path to a leaf node indicates the key of that leaf. Among its variants, the Adaptive Radix Tree (ART) [17, 18] stands out for its space efficiency and high performance. ART dynamically selects different inner node sizes based on the number of child nodes, effectively

Table 1. The impact of the lock on the performance of ART.

	Throughput (Mops/s)		Latency (us)	
	w/ lock	w/o lock	w/ lock	w/o lock
zipfian	1.4	5.2	16.8	13.0
uniform	4.7	5.2	16.8	12.9

mitigating the excessive space consumption in traditional radix trees. Furthermore, ART leverages path compression to eliminate inner nodes with only a single child, thereby reducing tree height.

Despite these advantages, current ART designs [16, 17, 25, 26] fall short of addressing the unique challenges posed by disaggregated memory architectures. Both the index structure and concurrency protocols of ART require a significant redesign to achieve low latency and high scalability while ensuring crash consistency.

Index Structure. Existing ART structures do not account for the significant overhead associated with remote memory accesses, leading to suboptimal performance in disaggregated memory architectures. First, the hierarchical nature of ART structures requires multiple round trips of remote memory access for index traversal. The traversal depth grows proportionally with key length, and dependencies between nodes at different levels further limit memory-level parallelism. As a result, clients must sequentially traverse the tree from root to leaf, with each level requiring one or more round trips over RDMA. Second, current ART node designs face challenges in reducing both round trips and bandwidth consumption of remote memory access during in-node searches. Many prior approaches [16, 17, 26] store auxiliary metadata in the node header to accelerate child pointer lookups. However, this results in multiple round trips of remote memory access since the client must first retrieve the metadata before using it to locate the target child pointer. Alternatively, a naive solution is to read the entire node in a single round trip, but this causes substantial bandwidth consumption due to large read amplification. Therefore, when we directly adapt the original ART to the DM architecture, we observe that executing a single 128-byte key-value search requires 5 network round trips and 6.5 KB of memory access on the *randint* dataset, and 14.4 round trips and 7.7 KB on the *email* dataset.

Concurrency Control. The conventional concurrency protocols for ART predominantly rely on lock-based writes, which are inefficient in disaggregated memory architectures. To evaluate the impact of lock-based concurrency control, we measure the write throughput under high concurrency (168 threads) and the single-thread write latency of ART in DM-based systems, both with and without locking. As shown in Table 1, lock-based concurrency control increases write latency by 30% because locking and unlocking introduce additional network round trips on the critical path. Moreover, it significantly degrades write throughput by 3.71× under skewed workloads (zipfian access distribution), as frequent lock failures and retries under high contention lead to wasted network resources.

Moreover, for tolerating the crashes of concurrent clients in compute nodes, ensuring the failure atomicity of index operations is essential in disaggregated memory systems [49, 56]. Without such guarantees, a client failure may leave the index structure in memory nodes in a partially updated and inconsistent state, potentially disrupting the correct execution of concurrent clients. Most existing ART variants, including SMART [25], do not provide failure-atomicity for write operations, as index updates typically involve modifying multiple memory words, whereas current hardware only supports atomic operations on a single 8-byte word. As a result, data consistency on memory nodes cannot be guaranteed in the event of client failures. Recent efforts in persistent memory-based indexing [16, 52, 53] have explored failure-atomic writes to achieve crash consistency. However,

these approaches remain inefficient in disaggregated memory systems. Most of them [21, 26, 58] first update the index structure in a non-visible manner, followed by an atomic update of 8-byte metadata (e.g., bitmap) to make the changes visible. Others [14, 22, 31] utilize a log-based strategy to enable correct recovery after a crash. However, both approaches necessitate multiple round trips of remote memory access, thereby increasing write latency.

2.3 SMART

SMART [25] is the state-of-the-art DM-based ART index that partially addresses the challenges discussed in Section 2.2. However, its design still suffers from several fundamental limitations.

To mitigate remote memory accesses, SMART [25] caches frequently accessed inner nodes at the compute nodes. However, the limited memory available at compute nodes [20, 24, 48] severely constrains the achievable performance improvements from such caching mechanisms. As shown in Figure 7, during search operations (YCSB-C), SMART still incurs an average of 3.9 network round trips and 3.7 KB of memory access on the *randint* dataset, and 7.9 round trips and 2.0 KB on the *email* dataset. Moreover, SMART employs an unscalable FIFO-based cache policy, which incurs substantial contention during cache admission/eviction, consuming significant CPU cycles and thereby limiting the overall scalability of the system [23].

To enhance concurrent scalability, SMART adopts a hybrid concurrency control mechanism. It enables lock-free operations on inner nodes by embedding part of the node metadata into their parent pointers. However, to maintain concurrent consistency, SMART must use fixed-size inner nodes, which results in very large space consumption (see Table 2). For leaf nodes, SMART employs lock-based operations and adopts a lock handover mechanism [43] within the same compute node to reduce locking overhead under skewed workloads. Nevertheless, this approach cannot fundamentally eliminate lock overhead due to the inherent exclusion of locks [40]. Furthermore, handing over locks within the compute node may cause unreleased locks in the event of client crashes, interfering with normal executions on other compute nodes [49]. In addition, SMART does not ensure the failure-atomicity of index operations, rendering it more vulnerable to client crashes.

3 DART Design

Based on the analysis in Section 2, we establish two key design goals for the disaggregated ART index: *(1) Minimize both round trips and bandwidth consumption of remote memory accesses. (2) Maximize scalability while maintaining crash consistency during concurrent access.*

To achieve these goals, we introduce DART, a lock-free two-layer hashed Adaptive Radix Tree (ART) for RDMA-based disaggregated memory architectures. DART incorporates hash structures into the radix tree to minimize remote memory access overhead in DM-based systems. Hash indexes provide $O(1)$ complexity for index operations, significantly decreasing round trips and bandwidth consumption compared to tree-based structures. On the other hand, the radix tree offers distinct advantages over hash indexes, particularly in efficiently supporting variable-sized keys and range queries. By adopting a two-layer hashed representation for ART, DART effectively harnesses the strengths of both indexing methods.

Figure 2 depicts the overall architecture of DART. The upper layer comprises a hash-based *Express Skip Table* (Section 3.1), which provides an express path to a lower-level inner node during the traversal of the hierarchical radix tree. This design effectively reduces the traversal depth, significantly minimizing the round trips required for remote memory access. The base layer consists of *Adaptive Hashed Nodes* (Section 3.2), which replace traditional radix tree nodes with hash tables, leading to a substantial reduction in bandwidth consumption during in-node searches.

Moreover, DART employs *Decoupled Metadata Organization* (Section 3.3) to facilitate lock-free concurrency control. By decoupling the metadata from each tree node and embedding it into

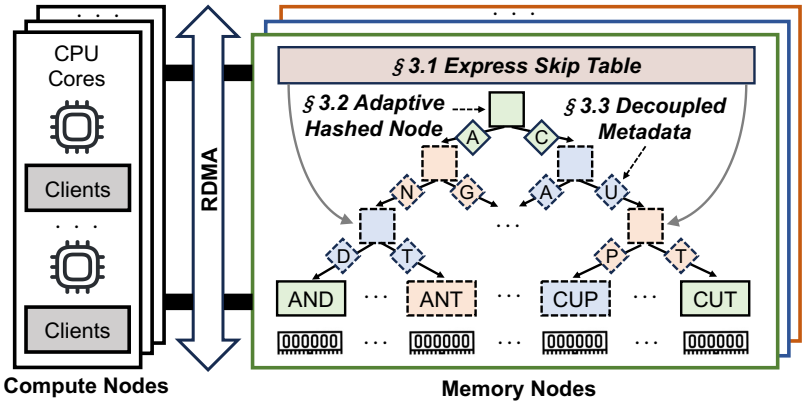


Fig. 2. The overall architecture of DART

the traversal path, DART achieves lock-free atomic writes, ensuring high scalability and crash consistency.

3.1 Express Skip Table

In this section, we introduce the hash-based *Express Skip Table* at the upper layer of DART, designed to minimize the round trips involved in remote memory accesses. The core idea is to provide an express path during index traversal through hash mapping, allowing clients to bypass the initial levels of the radix tree and directly skip to a lower-level inner node (*express node*), thereby reducing the levels required to traverse the hierarchical radix tree structure.

However, constructing an efficient *Express Skip Table* for the radix tree presents significant challenges. It requires careful selection of suitable inner nodes as express nodes that not only reduce the round trips of remote memory access but also minimize memory usage and update frequency. Moreover, due to the path compression in radix trees, the inner nodes along the traversal path of each key are uncertain, making it challenging to determine the express node for a given key. Additionally, the skip table must incorporate an optimized index structure to minimize remote memory access for retrieving express nodes.

Our proposed hash-based *Express Skip Table* effectively addresses these challenges. As shown in Figure 3, DART constructs a DM-friendly hash table to store the metadata of carefully selected express nodes from the radix tree. Each express node employs **the concatenated key prefix along its path as the key** in the hash table, leveraging the unique property of the radix tree, where each inner node is uniquely identified by this key prefix. By leveraging the flat structure of the hash table, DART enables express nodes to be located with just a single round trip.

To efficiently identify the express nodes for any given key, DART selects express nodes based on a predefined prefix length list. All inner nodes with prefix lengths from this list are chosen as express nodes. For any given key, its *express prefix length* is determined as the largest value from the list that is smaller than the length of the key. During the index traversal, DART begins by locating the express node of the requested key from the *Express Skip Table* using the key's *express prefix* (the key prefix with express prefix length). Once the express node is identified, DART proceeds to traverse the radix tree starting from this node, bypassing all upper-level inner nodes. To maximize round-trip savings, DART initially constructs a generalized prefix length list, inspired by the skip list [34]. This list is then dynamically adjusted at runtime based on the access patterns of workloads.

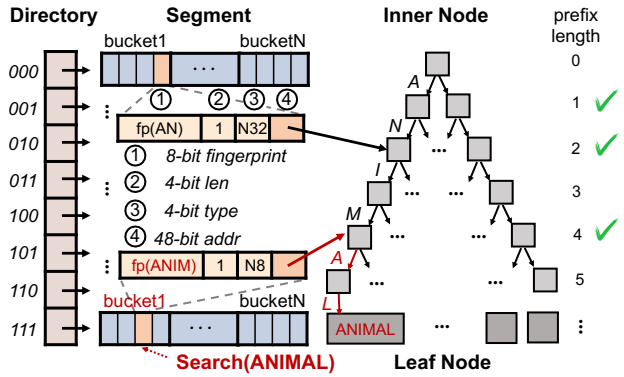


Fig. 3. Express skip table

In the following, we will detail the *Express Skip Table* by outlining its structure and the approaches for selecting express nodes.

Hash Structure. The *Express Skip Table* employs an extendible hash structure, as depicted in Figure 3. Similar to RACE Hashing [59], the hash table consists of a directory, where each entry points to a segment containing multiple hash buckets. To obtain the metadata of express nodes, the client uses the highest few bits of key hash to locate the segment's address in the directory, and then utilizes the lowest few bits of key hash to identify the corresponding hash bucket within the segment. Given the relatively small directory size (typically less than 1MB), it can be cached locally at the compute node, requiring only a single remote memory access to retrieve a bucket from the segment during searches.

As shown in Figure 3, each hash bucket contains multiple 8-byte slots to store the metadata of express nodes in the ART tree, including 8-bit `fingerprint`, 4-bit `len`, 4-bit `type`, and 48-bit `address`. DART optimistically leverages 1-byte fingerprints to represent variable-length keys of express nodes, instead of introducing an indirection layer to store full keys as in prior works [52, 59]. This design eliminates additional round trips to access indirect key-value layers during searches. In most cases, the 1-byte fingerprint uniquely identifies express nodes within a single 64-byte bucket. On the rare occasions where multiple keys in the same bucket share a fingerprint, DART traverses from multiple express nodes in parallel, ensuring the target key-value pair can be located within one of their subtrees. The `len`, `type`, and `address` fields represent the compressed prefix length, node type, and remote memory address of the express node, respectively. Using a 48-bit address representation is a common practice [25, 59], as it covers an address space of up to 256 TB.

Express Node Selection. For the selection of express nodes, we first propose a generalized approach inspired by the skip list. As shown in Figure 3, we define a prefix length list, consisting of 2^0, 2^1, 2^2, ..., 2^n. This means that all inner nodes with a prefix length of 2^k are selected as express nodes. DART stores the metadata of selected express nodes in the hash-based *Express Skip Table*. For a key of length L, the client can uniquely determine its express prefix length as the largest 2^k that is smaller than L. During the search, the client first retrieves the metadata of the express node in the skip table based on the express prefix of the key. Once the express node is identified, the index traversal initiates a downward search from this express node, significantly reducing the traversal depth by more than half. This approach skips 2^k levels from the root to the express node, leaving fewer than 2^k levels to traverse from the express node to the target key-value pair.

To build the *Express Skip Table*, DART employs a depth-first algorithm to scan the entire radix tree and selects all inner nodes with a prefix length of 2^k as express nodes for insertion into the hash table. Due to path compression, some key-value pairs may lack an inner node at an exact 2^k prefix length. In such instances, the inner node compressing the 2^k-th byte is designated as the express node. When inserting this node into the table, the 2^k prefix is retained as the key, while the value records the difference between its actual prefix length and 2^k in the *length* field (Figure 3). This ensures that a key can consistently locate its corresponding express node despite path compression, while the recorded length guarantees accurate traversal from that node.

The *Express Skip Table* supports dynamic updates throughout the index expansion process, including when the index grows from an initially empty state. During insertions, updates to the *Express Skip Table* occur infrequently and never compromise traversal correctness. Specifically, updates are triggered in the following two cases: *(1) Node split at the 2^k level.* When an inner node that compresses multiple bytes of the key prefix splits and produces a new node at the 2^k level, a corresponding hash entry is inserted into the *Express Skip Table* for the newly created express node. *(2) Node type switch at the 2^k level.* When an express node at the 2^k level undergoes a type switch to adjust its capacity, the *Express Skip Table* first freezes the corresponding hash entry (see details in Section 4), and then atomically updates it upon completion of the switch.

DART tolerates temporary inconsistencies in the *Express Skip Table* during updates. If a node split or a node type switch occurs but the *Express Skip Table* has not yet been updated, the search process safely falls back to traversing the index structure through the express node at the 2^{k-1} level, ensuring the correctness of the traversal. The *Express Skip Table* also exhibits low memory usage. The express nodes partition the radix tree into multiple subtrees, each rooted at an inner node with a 2^k prefix length. This allows numerous key-value pairs with the same express prefix to share a common express node, significantly reducing the number of express nodes to maintain.

Runtime Adjustment. To further reduce the round trips of remote memory access, DART periodically adjusts the prefix length list based on the access patterns of workloads during runtime. We formulate the configuration of the prefix length list as an optimization problem, aiming to maximize round-trip savings within the constraints of a predefined number of express nodes. The benefit of designating each inner node as an express node is quantified by multiplying the access frequency of key-value pairs in its subtree by the node's distance from the root. To efficiently monitor the access frequency with minimal overhead during the runtime, DART updates the frequency field of the key-value entry in every R-th request, similar to the previous work [4].

DART employs a lightweight greedy algorithm to solve this optimization problem. First, a DFS algorithm is used to traverse the radix tree, calculating the round-trip savings for each node as a potential express node. This allows us to easily estimate the benefit of selecting inner nodes with a prefix length of i as express nodes. Due to the interdependencies among nodes, selecting one node as an express node can impact the benefits of others being selected. Therefore, it is impractical to determine an optimal prefix length list in a single DFS. Instead, we select only the prefix length that offers the highest benefit during the initial DFS. A second DFS traversal is then performed to update the benefits for other nodes. This process repeats until the available number of express nodes is exhausted or the benefit of adding another level as express nodes becomes negligible.

3.2 Adaptive Hashed Layout

In this section, we present *Adaptive Hashed Layout* for DART, incorporating the hash structure into the radix tree to harness the advantages of both indexes. The core idea involves employing a hash-based layout within each inner node of the radix tree to minimize remote memory access

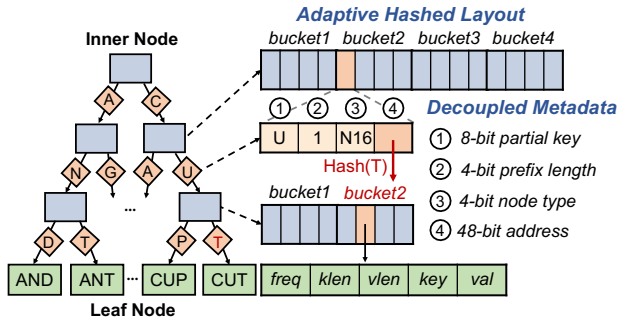


Fig. 4. Adaptive hashed layout and decoupled metadata organization

overhead during in-node searches, while maintaining the prefix-based hierarchical structure of the radix tree to support variable-sized keys and range queries.

As shown in Figure 4, DART organizes the hierarchical radix tree nodes based on key prefixes. Each inner node maintains an array of child pointers with a maximum fanout of 256, corresponding to all possible 1-byte partial keys. Leaf nodes store the complete key-value entries, where the key is composed of the concatenated partial keys gathered from inner nodes along the traversal path.

Different from the traditional radix tree, DART employs an *Adaptive Hashed Layout* for each inner node to reduce remote memory access during in-node searches while optimizing memory usage. The adaptive hashed nodes organize child pointers according to the hash value of their partial keys, thereby limiting the remote memory access to a small bucket when traversing each inner node. As shown in Figure 4, each inner node is divided into multiple small hash buckets, each containing pointers to next-level nodes that share the same hash value for their partial keys. During the traversal of each inner node level, the client retrieves only the specific hash bucket determined by the hash value of the next partial key from the remote memory. After retrieving the targeted hash bucket, the client performs a linear search within it to locate the child pointer corresponding to the requested key.

In this way, remote memory access during the traversal of an inner node is limited to a small hash bucket instead of the entire node, resulting in significant reductions in bandwidth consumption. The bucket size configuration is adjustable to align with the optimal memory access granularity of different systems, considering their underlying physical memory type (e.g., DRAM, PM) and interconnect technologies (e.g., RDMA, CXL). By default, we set the bucket size to 128-byte, which achieves a good balance between performance and memory utilization in our testbed.

Moreover, *Adaptive Hashed Layout* in DART allows for the construction of variable-sized inner nodes by adjusting the number of hash buckets. Unlike the fixed-size inner nodes employed by previous systems like SMART [25], which often suffer from inefficient memory usage due to varying numbers of child pointers across nodes, DART employs six types of inner nodes with a 1-byte fanout: N8, N16, N32, N64, N128, and N256. Specifically, N16, N32, N64, and N128 incorporate 1, 2, 4, 8 hash buckets of 128 bytes, respectively. For nodes with significantly fewer than 16 child pointers, DART constructs a more compact node, referred to as N8, which employs a 64-byte hash bucket. Given that the inner node with 1-byte fanout can accommodate up to 256 child pointers, N256 is implemented as a direct-mapped 256-pointer array, following the original ART design [17].

Additionally, DART achieves memory utilization comparable to that of the original ART, as the *adaptive hashed layout* introduces both benefits and drawbacks for memory efficiency. On the positive side, DART provides a wider range of node size options, which enhances space utilization.

On the negative side, fragmentation across hash buckets causes some memory waste. Our evaluation shows that DART consumes 30% and 7% less memory than the original ART on the *randint* and *email* datasets, respectively, indicating that the positive and negative effects are nearly balanced.

3.3 Decoupled Metadata Organization

Although the two-level hashed ART significantly reduces remote memory access during index traversal, it still faces two key challenges in metadata management. First, during the in-node search, the client must retrieve its metadata (e.g., *node type* and *prefix length*). Typically, this metadata is stored in the header of the inner node, as illustrated in Figure 1, which incurs an additional round trip for remote memory access. Second, achieving lock-free concurrency control and crash consistency during write operations presents substantial challenges. Updates to inner nodes often require modifications to both data and metadata located in separate memory words, while existing hardware only supports atomic primitives (e.g., `RDMA CAS` and `RDMA FAA`) for a single word.

We employ *Decoupled Metadata Organization* to address the aforementioned challenges. As shown in Figure 4, DART decouples the storage of metadata from the data of each inner node and embeds it within the child pointer in its parent node, forming an 8-byte compound child pointer. Through *Decoupled Metadata Organization*, each inner node in DART consists solely of several hash buckets, eliminating the need for metadata in the node header. Each hash bucket maintains an array of compound child pointers, each embedding both the metadata and the memory address of the corresponding child node. The detailed layout of the compound child pointer is structured as follows: ① partial key (8 bits), the 1-byte key prefix represented by the child node. ② prefix length (4 bits), representing the compressed path length of the child node (DART uses the *optimistic path compression* proposed in ART [17]); ③ node type (4 bits), which employs 3-bit to represent seven different node types (N8, N16, N32, N64, N128, N256, Leaf) and reserves the final bit as a *frozen bit* to indicate an ongoing node type switch (see details in Section 4.2); ④ address (48 bits), the memory address of the child node. Following prior work [25], we use the first 8 bits to identify the memory node and the remaining 40 bits to represent the address offset within that node. This design supports up to 256 memory nodes, each providing up to 1 TB of addressable memory. Each leaf node stores a single key-value entry, which includes its access frequency (`freq`), the key length (`klen`), the value length (`vlen`), as well as the key and value themselves.

For read operations, the client can obtain both the address and metadata of the child node from the compound pointer when searching for a child node within an inner node. If the partial key of the child node matches the requested key, the client directly reads the target hash bucket of the child node based on the hash value of the next partial key, eliminating the need for additional remote memory accesses to metadata.

Moreover, by leveraging the *Decoupled Metadata Organization*, DART achieves both lock-free concurrency control and crash consistency by atomically updating an 8-byte compound pointer. When updating an inner node, DART first prepares a new child node (whether a leaf node or an inner node) and then employs atomic hardware primitives (e.g., `RDMA CAS`) to atomically modify the corresponding compound pointer. All write operations in DART, including insert, update, delete, node split, and node type switch, can be implemented in such an out-of-place update manner (see details in Section 4). Concurrent updates to the same child pointer are serialized by the atomic hardware primitives, thereby ensuring lock-free concurrent consistency. Moreover, the effects of write operations adhere to all-or-nothing semantics during a client crash. If the 8-byte compound pointer has been atomically written to memory during the crash, the write operation is considered successful. Otherwise, the process of creating new nodes remains invisible to other clients. This ensures that the index structure does not expose any inconsistent intermediate states, thereby preventing any impact on the normal executions of other clients.

Algorithm 1. DART search algorithm.

```
1  def search(key):
2      express_prefix = get_from_prefix_length_list(key)
3      node_meta, pos = express_skip_table.find(express_prefix)
4      while node_meta.type != LEAF:
5          partial_key = key.at(pos)
6          bkt_idx = hash(partial_key)
7          bucket = RDMA_READ (node_meta.addr + bkt_idx * bkt_size, bucket_size)
8          pos = pos + node_meta.len
9          node_meta = bucket.find(partial_key)
10         if !node_meta:
11             return None
12     leaf = RDMA_READ(node_meta.addr, get_size(node_meta))
13     return leaf.value
```

4 Lock-free Index Operations

In this section, we first present the detailed implementation of DART's lock-free index operations, including search, insert, update, and delete, as well as its memory allocator. DART achieves lock-free concurrency control by utilizing `RDMA CAS` to atomically install 8-byte compound pointers for new nodes into the radix tree. The lock-free atomic writes ensure that the index structure remains consistent during concurrent access and client crashes, guaranteeing both concurrent and crash consistency.

4.1 Search

We start by detailing the search process, as it forms the foundation for all other index operations. As shown in Algorithm 1, the client initiates the search by retrieving the metadata for the express node corresponding to the requested key from the *Express Skip Table* (line 2-3). It then traverses from this express node towards the leaf node, bypassing the upper levels of the radix tree. Specifically, the client identifies the next partial key following the express prefix (line 5) and calculates its hash value (line 6). Utilizing this hash value along with the node type and base address of the express node, the client determines the memory address of the target hash bucket that contains the next partial key. The client then retrieves the corresponding hash bucket from remote memory (line 7) and conducts a linear search to identify the child pointer aligning with the next partial key (line 9). This recursive process continues until the leaf node with the target key-value entry is located. Lock-free search operations always read consistent data, as all write operations described below guarantee atomic updates.

For range queries, DART also initiates index traversal from an express node, determined by the largest common prefix between the *begin_key* and *end_key*. During traversal at each level, DART retrieves the child nodes within the target range in parallel. The recursive traversal continues until all leaf nodes within the search range are found. Of note, although DART guarantees each key-value entry returned in a range query is consistent, it does not ensure the range query is atomic with concurrent write operations, like previous DM-based ordered indexes [24, 25, 43, 57].

4.2 Insert

Normal Insertion. The client traverses the inner nodes as search operations until it cannot find the child node corresponding to the prefix of the requested key. The index traversal executes concurrently with the creation of a new leaf node containing the requested key-value entries to

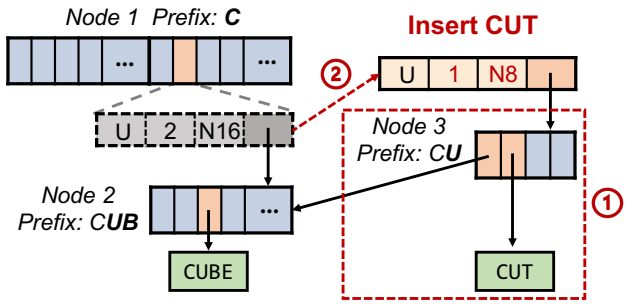


Fig. 5. The node split in DART

hide the latency of remote memory access. Subsequently, the client uses an `RDMA CAS` to atomically install the compound pointer corresponding to the new leaf node into the first empty slot of the target hash bucket in the last-level inner node. The insertion is committed if the `RDMA CAS` succeeds. Otherwise, the client attempts to install the compound child pointer in the next empty slot and repeats the process until successful.

Node Split. The path compression leads to a special case during insertion, where the prefix of the requested key only partially matches the prefix of the found inner node. As illustrated in Figure 5, the partial key represented by *Node 2* is *CUB* due to path compression. However, the inserted key *CUT* only matches part of the prefix of *Node 2* (*CU*). In such a situation, a node split is required.

DART guarantees the atomic completion of node splits by installing an 8-byte compound child pointer. The detailed process is illustrated in Figure 5. ①: DART allocates a new inner node (*Node 3*) that represents the common prefix (*CU*) shared between the inserted key-value pair (*CUT*) and the prefix of the original inner node (*CUB*). This new node hosts two child pointers: one pointing to the original inner node (*Node 2*) and the other to the leaf node containing the new key-value pair (*CUT*). ②: DART then atomically updates the compound child pointer in the parent node (*Node 1*) to indicate this new inner node (*Node 3*) using an `RDMA CAS`. As step ① remains invisible to concurrent clients, the node split is completed atomically in step ②, thereby ensuring both lock-free concurrent consistency and crash consistency.

Node Type Switch. The node type switch occurs when an inner node either cannot accommodate additional child pointers or has an excess of empty slots. In this situation, DART first freezes the target node by atomically updating the compound child pointer in its parent node, changing the node type to *Frozen*. This prevents concurrent operations from accessing the node, thereby avoiding concurrency anomalies. Then, DART copies the data from the old node to a new node. Finally, it atomically updates both the *node type* and the *memory address* of the compound child pointer in the parent node using `RDMA CAS`.

DART introduces a *grace period* during the node type switch to prevent anomalies in a corner case where a concurrent operation observes a node as unfrozen but the node becomes frozen before the access. Specifically, once a node type switch marks a node as frozen, it waits for one grace period (set to p95 latency of regular index operations) before executing the switch. Each concurrent index operation reads the frozen state when it first accesses the node. If the operation completes within the grace period, no node type switch can occur during its execution, thereby eliminating the need for state validation. For the few long-running operations that exceed the grace period (~5% cases), a post-operation state validation is required. If the node is found to be frozen, the operation waits for the type switch to complete and retries on the new node to ensure correctness.

The length of the grace period is determined at runtime by sampling operation latencies and setting it to the p95 latency. In this way, the grace period exceeds the execution time of 95% of operations, ensuring that 95% of operations do not need to validate the frozen state. Moreover, once the compute node detects that the current p95 latency deviates significantly from the previously established grace period, the system updates the grace period to match the current p95 latency, without requiring any manual tuning by the user.

Skip Table Updates. Node splits and node type switches may require updating the *Express Skip Table*. Our skip table directly adopts the lock-free concurrency protocols from the RACE hashing [59], which atomically update an 8-byte slot for all write operations. Moreover, as discussed in Section 3.1, the *Express Skip Table* employs a fallback mechanism that allows it to temporarily lag behind updates to the ART structure without compromising correctness.

4.3 Update/Delete

The client first performs a search operation to locate the compound child pointer to the leaf node corresponding to the requested key-value entry. For update operations, the client creates a new leaf node containing the new key-value entries. This procedure can also overlap with index traversal to hide remote memory access latency. Then, the client employs an `RDMA CAS` to atomically update the compound child pointer, replacing the old leaf node with the new one. For delete operations, the client directly uses an `RDMA CAS` to atomically set the found compound child pointer to `NULL`.

4.4 Memory Allocator

To minimize the network overhead during memory management, DART adopts a two-level memory allocator, similar to FUSSE [38]. Each memory node partitions its memory space into coarse-grained memory blocks (e.g., 16 MB). Clients allocate these blocks from memory nodes in a round-robin fashion and subsequently manage them independently using fine-grained slab allocators [1].

To safely reclaim deleted nodes, DART extends the epoch-based reclamation mechanism [9] to support disaggregated memory. The primary goal of this mechanism is to reclaim memory only after all threads that may access the deleted nodes have completed their operations. To achieve this, each compute node maintains a global epoch, while each client records a local epoch, which is synchronized with the global epoch after each index operation. Additionally, each client maintains a per-epoch deleted list to temporarily store deleted nodes awaiting reclamation. A background thread periodically (e.g., every second) checks whether all local epochs of clients are consistent with the global epoch. If so, all index operations initiated in the previous epoch have been completed. At that point, the global epoch is incremented across all compute nodes. The overhead of global epoch synchronization is negligible, as it involves synchronizing only an 8-byte global epoch at a low frequency (once per second). When a node is deleted, DART appends it to the deleted list associated with the current epoch. Any node deleted in epoch e is safely reclaimed in epoch $e + 2$, ensuring that all index operations with potential access to the deleted node have completed.

4.5 Consistency Guarantee

DART achieves atomic updates for all write operations by using `RDMA CAS` to atomically install 8-byte pointers for new nodes into the radix tree, thereby ensuring both concurrent consistency and crash consistency. In the following, we describe how DART handles write-write conflicts, write-read conflicts, and client crashes.

Write-Write Conflicts. DART categorizes write operations into two types. *Type 1:* Operations that atomically update an existing compound pointer in an inner node, including updates, deletions, node splits, and node type switches. *Type 2:* Operations that insert a new compound pointer into an empty slot via atomic primitives, corresponding to normal insertions.

Concurrent *Type 1* operations targeting the same slot are naturally serialized by the `RDMA CAS`. For *Type 2* operations, concurrent insertions of the same key contend on the same empty slot, and only one will succeed, thereby preventing duplicate keys. Notably, *Type 1* and *Type 2* operations do not interfere with each other, as they access disjoint slots within the node.

Write-Read Conflicts. DART ensures that read operations always observe consistent and latest data, as all writes are performed atomically and follow an all-or-nothing semantics. To prevent concurrent access to reclaimed nodes, DART incorporates an epoch-based reclamation mechanism (see details in Section 4.4), ensuring deleted nodes are reclaimed only after all potentially accessing threads have finished. Furthermore, DART combines a *grace period* with *post-operation validation* (see details in Section 4.2) to avoid reading stale nodes during the node type switch.

Moreover, DART ensures that the DFS traversal in runtime adjustment is thread-safe under concurrent writes. This is because our lock-free concurrency control prevents DFS from observing any inconsistent nodes. Additionally, DFS does not require an atomic snapshot of the entire index, allowing nodes to be added or removed during traversal. This is because the purpose of DFS is to calculate the benefit of selecting nodes at each level as express nodes. Since each level typically contains a large number of nodes, occasional inserts and deletes have a negligible impact on the overall benefit calculation for that level.

Client Crashes. DART guarantees data consistency in memory nodes after client crashes by ensuring that all write operations are either fully applied or not visible at all. If a client crashes before the `RDMA CAS` is committed, the update remains invisible and does not affect the ART structure. Conversely, a successful `RDMA CAS` ensures that the write is committed and immediately visible to other threads, maintaining data consistency despite client failures.

Moreover, when a client crashes, the compute node loses the information of its fine-grained slab allocator. To prevent memory leaks caused by client crashes, the compute node can reconstruct the state of its local allocator upon restart by performing the following steps: (1) Retrieve the addresses of coarse-grained blocks allocated to the compute node from allocators in memory nodes. (2) Traverse the DART structure from the root to identify all tree nodes belonging to the compute node. (3) Combine the above information to rebuild the state of the compute node's local fine-grained slab allocator. Moreover, the reconstruction process can be performed in the background, as index operations on the critical path can request new memory from the memory nodes even before the local allocator state is fully restored.

5 Evaluation

5.1 Experimental Setup

Testbed. We conduct our evaluation on an RDMA-based disaggregated memory cluster comprising three compute nodes and three memory nodes. Each compute node is equipped with two 2.6GHz Intel Xeon Gold 6348 CPUs (each with 28 cores) and 1GB DRAM, while each memory node is provisioned with 128GB of DRAM. Compute and memory nodes are interconnected via 100Gbps Mellanox ConnectX-6 InfiniBand NICs.

Workloads. We evaluate DART using YCSB workloads [5], including A (50% search, 50% update), B (95% search, 5% update), C (100% search), D (latest-read, 95% search, 5% insert), E (95% scan, 5% insert), LOAD (100% insert). By default, a zipfian key access distribution with a skewness parameter of 0.99 is employed for the YCSB evaluation. Our evaluations are conducted on two datasets: *randint* and *email*. The *randint* dataset consists of fixed-sized 8-byte keys drawn from a uniform distribution over the range of $[0, 2^{63} - 1]$. The *email* [8] contains variable-sized keys ranging from 2 to 32 bytes, with an average size of 18 bytes. Following the preprocessing steps (i.e., swap username and domain fields of email address) from prior works [25], the *email* dataset exhibits a skewed key distribution, with many email addresses sharing a common prefix. By default, each dataset comprises 60 million

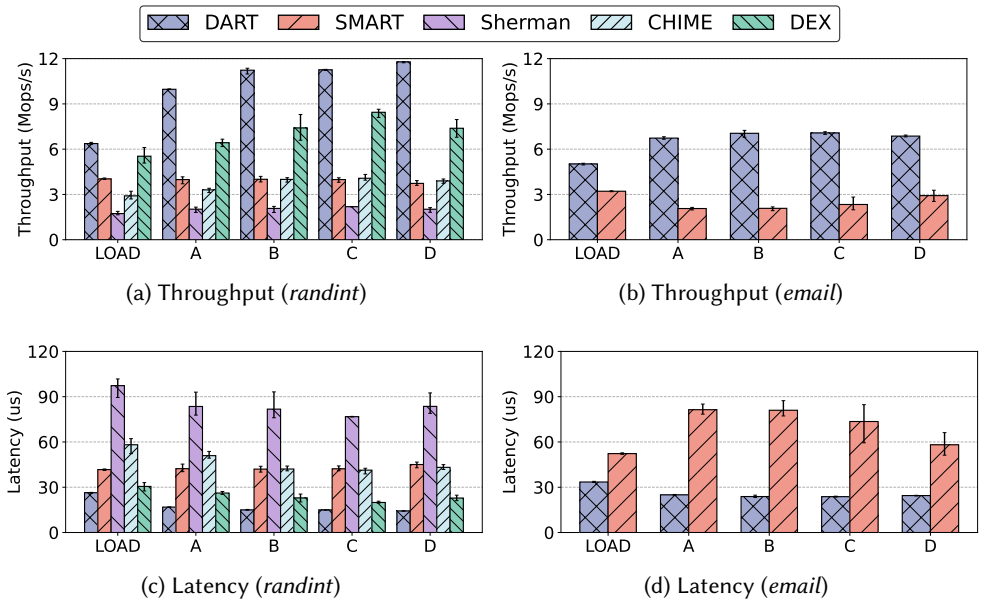


Fig. 6. The average throughput and latency in YCSB workloads.

key-value entries, each with a 64-byte value. For YCSB A–E, each index is first bulk-loaded with 60 million key-value entries, followed by 60 million workload operations.

Compared Systems. The primary comparison system of our evaluation is SMART [25], the state-of-the-art ART for DM-based systems. To demonstrate the advantages of DART over other tree-based index structures, we also compare it with state-of-the-art B+Trees, including Sherman [43], CHIME [24], and DEX [23]. We evaluate B+Trees only on the *randint* dataset as their open-source codes do not directly support datasets with variable-sized keys (e.g., *email*). Additionally, we adapt the original ART [17] to the DM architecture to perform factor analysis of DART's design components (Section 5.3). Among these indexes, DEX is a disaggregated index built on a shared-nothing (sharding-based) architecture, which distributes tree nodes across memory nodes according to range partitioning. In contrast, all other indexes adopt a shared-everything architecture, where tree nodes are distributed across memory nodes in a round-robin manner.

Due to the nature of disaggregated memory, where compute nodes have limited memory capacity, we follow prior studies [23–25, 43] to evaluate the performance of DM-based indexes under small cache sizes ranging from 1 MB to 400 MB. By default, DART caches only the hash directory of the *Express Skip Table* in RACE [59] (~1MB). When the cache space is sufficient, DART further caches the metadata of hot express nodes in compute nodes. For other compared systems, we configure them to cache hot tree nodes using the same cache capacity as DART.

5.2 Overall Performance

In this section, we present the overall performance of each index under YCSB workloads. Figure 6 shows the average throughput and latency with 168 client threads in *randint* and *email* datasets, along with the performance variance across three test runs. To provide further insight into the experimental results, Figure 7 illustrates the number of round trips and the bandwidth consumption for remote memory access during the evaluation.

As illustrated in Figure 6a and 6b, DART delivers 1.3~5.8× higher throughput on the *randint* dataset and 2.4~3.4× higher throughput on the *email* compared to other indexes. Figure 6c and 6d

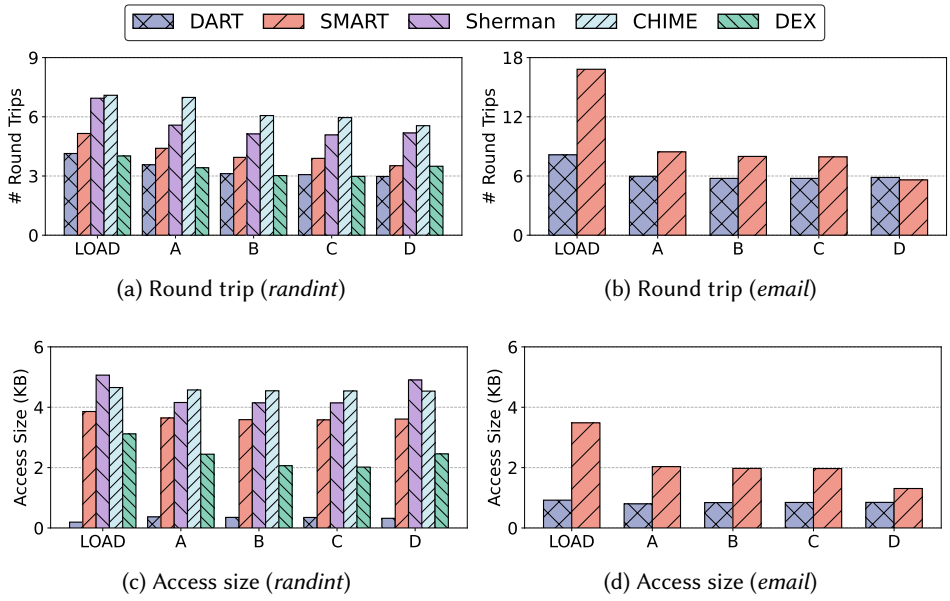


Fig. 7. The average number of round trips and sizes of remote memory access.

further show that DART reduces latency by up to 5.9× compared to its counterparts. In YCSB workloads, the main performance bottleneck for both read and write operations arises from the remote memory accesses required during index traversal. DART demonstrates substantial performance improvements over other systems by effectively minimizing both the round trips and the bandwidth consumption of remote memory accesses. As illustrated in Figure 7, DART achieves the lowest round trips and bandwidth consumption compared to other systems in most cases. We attributed this to our three key innovations: *Express Skip Table*, *Adaptive Hashed Layout*, and *Decoupled Metadata Organization*. The *Express Skip Table* allows DART to bypass the initial levels of the radix tree, enabling traversal to start directly from lower-level nodes. This design significantly reduces both round trips and bandwidth usage during index traversal, especially for datasets with long keys (e.g., *email*). The *Adaptive Hashed Layout* further confines remote memory access during in-node searches to a small hash bucket, thereby effectively minimizing bandwidth consumption. This is particularly advantageous for datasets with randomly distributed keys (e.g., *randint*), which typically generate large inner nodes in the upper layer of the radix tree. Additionally, *Decoupled Metadata Organization* enables lock-free atomic updates, improving concurrent scalability and reducing round trips required for locking.

In contrast, SMART incurs significantly higher round trips and bandwidth consumption for remote memory access. This is primarily due to its need to traverse multiple levels of the hierarchical tree structure during index operations. Additionally, SMART suffers from significant read amplification during in-node searches, as it needs to read the entire node at each level of traversal. Given the limited memory capacity of compute nodes in disaggregated memory architectures, caching only a small number of inner nodes is insufficient to effectively reduce traversal overhead within the tree-based structure [20, 48]. As for disaggregated B+Trees, such as Sherman, CHIME, and DEX, they also suffer from significant remote memory access overhead when traversing hierarchical tree-based structures.

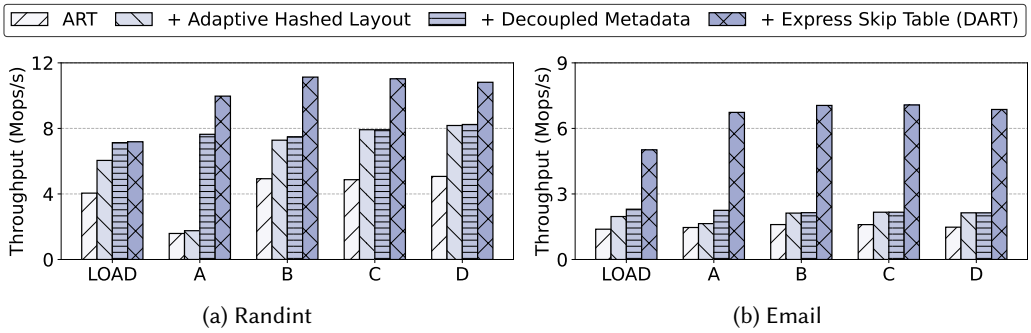


Fig. 8. Ablation analysis of DART's design (each configuration cumulatively includes all prior components)

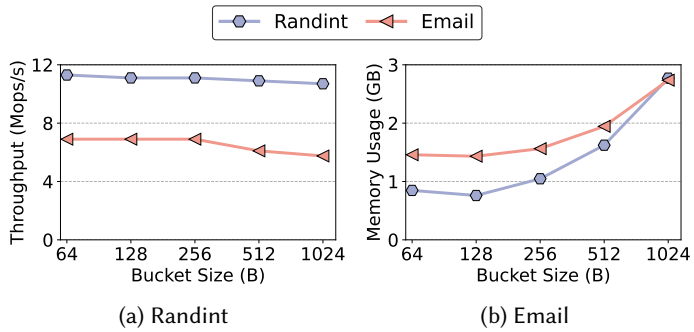


Fig. 9. The throughput and memory usage of DART under different bucket sizes

5.3 Factor Analysis for DART Design

In this section, we analyze the performance impact of each design component in DART. Figure 8 highlights the contributions of our three key innovations: *Adaptive Hashed Nodes*, *Decoupled Metadata Organization*, and *Express Skip Table*.

Adaptive Hashed Node. The *Adaptive Hashed Node* enhances the performance of DART by up to 1.7× as it significantly reduces the bandwidth consumption during in-node search. In a naive ART, the entire node must be read from remote memory because the position of the child pointer cannot be determined beforehand. This results in substantial read amplification, especially on datasets with many large nodes (e.g., *randint*). DART addresses this by employing *Adaptive Hashed Layout*, restricting remote memory access to a small bucket during in-node searches, thereby reducing bandwidth consumption by 6.4~6.7× on the *randint* dataset and 2.1~2.3× on the *email* dataset. As shown in Figure 9, selecting a bucket size of 128 bytes for *Adaptive Hashed Layout* achieves a good balance between throughput and memory utilization. Larger bucket sizes increase read amplification and reduce the variety of inner node types, thereby lowering throughput and memory efficiency. Conversely, smaller bucket sizes (e.g., 64 bytes) do not improve performance, since the throughput of small-size RDMA read operations is comparable to that of 128-byte reads. Moreover, smaller bucket sizes increase memory fragmentation across buckets, slightly reducing memory utilization.

Decoupled Metadata Organization. DART utilizes the *Decoupled Metadata Organization* to achieve lock-free concurrency control, thereby boosting the performance of DART by up to 4.8× in

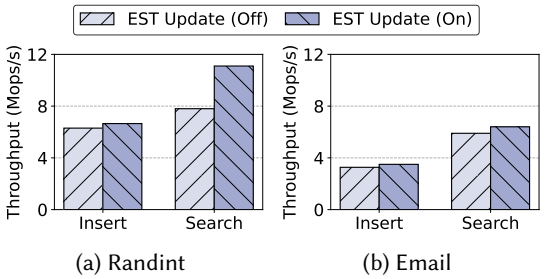


Fig. 10. Performance with EST (Express Skip Table) update off/on

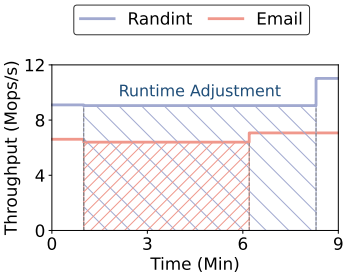


Fig. 11. Performance impact of runtime adjustment.

write-intensive and skewed workloads (e.g., YCSB-A). Traditional lock-based writes do not scale well in disaggregated memory systems due to additional round trips for locking and unlocking. In high-contention workloads, frequent lock failures and retries further degrade performance. With *Decoupled Metadata Organization*, DART can complete each write operation using a single atomic primitive, eliminating the need for locks and achieving superior performance in disaggregated memory systems.

Express Skip Table. DART achieves up to a 3.4× performance improvement with the *Express Skip Table* by drastically reducing the round trips of remote memory access during index traversal, particularly on datasets with long keys (e.g., *email*). Unlike traditional radix trees, which traverse from the root node, DART directly skips to an express node, bypassing the initial levels of the tree. The *Express Skip Table* saves around 2 round trips for the *randint* dataset and an average of 8.7 round trips for the *email* dataset during index traversal, leading to significant performance gains.

Moreover, *Express Skip Table* supports dynamic updates as the index grows. We measure the performance of insert and concurrent search, with the *EST (Express Skip Table) update* enabled and disabled, to evaluate its overhead and benefits. As shown in Figure 10, dynamically updating the EST improves concurrent search performance by up to 42%, as newly added express nodes reduce the number of layers that must be traversed to locate recently inserted key-value entries. Moreover, the performance improvement brought by EST update is more pronounced on sparse datasets (e.g., *randint*) than on dense datasets (e.g., *email*). This is because, in dense datasets, newly inserted key-value entries are more likely to fall under existing express nodes, resulting in fewer necessary updates to the express skip table. Figure 10 also shows that enabling EST update does not degrade insertion performance, indicating that updating the EST incurs negligible overhead. This is due to two factors. First, the update frequency of the express skip table is relatively low, adding less than 2% network round trips. Second, newly inserted express nodes also reduce the traversal cost of subsequent insert operations.

To further evaluate the overhead and benefits of runtime adjustment of *Express Skip Table*, we measure the index performance before, during, and after runtime adjustment. As shown in Figure 11, the overhead of runtime adjustment is minimal as it only degrades the index performance by less than 3% during the runtime adjustment. This is because we employ only a single background thread to perform a DFS traversal of the index and construct a new express skip table, which introduces minimal interference to foreground index operations. Moreover, runtime adjustment is triggered only when the dataset distribution changes, which occurs infrequently in practice. Figure 11 also shows that the runtime adjustment improves index performance by up to 21%. This is because the runtime adjustment dynamically determines a better prefix length list for different datasets, thereby reducing additional network round trips during traversal.

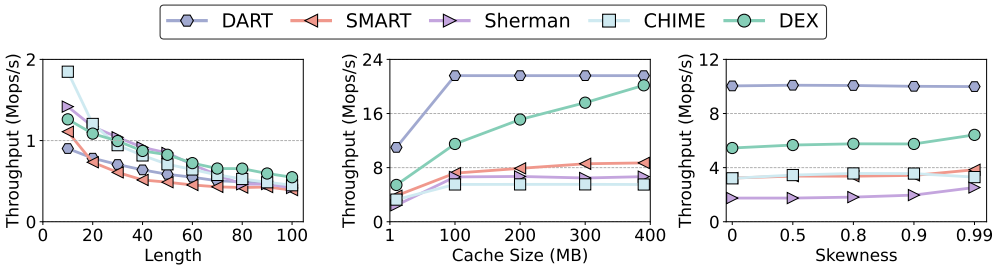


Fig. 12. The performance of range query.

Fig. 13. Sensitivity to cache sizes.

Fig. 14. Sensitivity to data skewness.

5.4 Range Query

In this section, we evaluate the performance of range queries across different scan lengths under YCSB E workloads. As illustrated in Figure 12, the performance of DART is comparable to SMART, as the primary bottleneck in range queries arises from reading leaf nodes rather than from index traversal. Additionally, B+Tree-based indexes achieve up to 2.0× higher scan throughput than ART-based indexes when the scan length is short, as they store key-value entries contiguously in the leaf nodes.

5.5 Sensitivity Analysis

Cache Sizes. To evaluate the impact of cache sizes, we present the index search performance under varying cache sizes. As shown in Figure 13, DART outperforms DEX under small cache sizes by up to 2.0×, whereas DEX achieves comparable performance to DART when the cache size becomes large. This is because DEX adopts a shared-nothing (sharding-based) architecture, whereas DART and the other compared systems employ a shared-everything design. The shared-nothing architecture is highly cache-efficient, as it minimizes cache coherence overhead, which explains DEX's advantage under large cache sizes. However, a major limitation of the shared-nothing design (e.g., DEX) is its poor support for cross-partition transactions in upper-layer applications, whereas indexes based on the shared-everything architecture (such as DART and SMART) do not suffer from this limitation. Figure 13 also shows that the performance of SMART is significantly lower than DART even under large cache sizes due to the following two reasons: (1) The index of SMART is relatively large, as shown in Table 2, and the limited cache size within compute nodes in a disaggregated memory architecture cannot hold all index nodes. As a result, the performance improvement from caching is limited. (2) SMART employs an unscalable FIFO-based cache policy, which incurs substantial contention during cache admission/eviction, consuming significant CPU cycles and thereby limiting the overall scalability of the system. This drawback has also been highlighted in the DEX paper [23].

Skewness. To evaluate the impact of skewness, we measure the index performance in YCSB-A workloads under varying skewness. As shown in Figure 14, DART achieves 1.6~5.7× higher performance compared with other indexes under different skewness levels. Moreover, the performance of DART is insensitive to skewness. We explain this result from two perspectives, *access locality* and *concurrency contention*, as both can be affected by skewness. (1) DART's design exhibits minimal dependence on access locality. By default, DART caches only a small hash directory of the express skip table on compute nodes. Each index traversal accesses this directory, which remains cache-resident and is thus unaffected by access locality. All subsequent accesses in DART are performed via RDMA, rendering its overall performance insensitive to locality effects. (2) The *decoupled metadata organization* enables DART to employ scalable lock-free concurrency control,

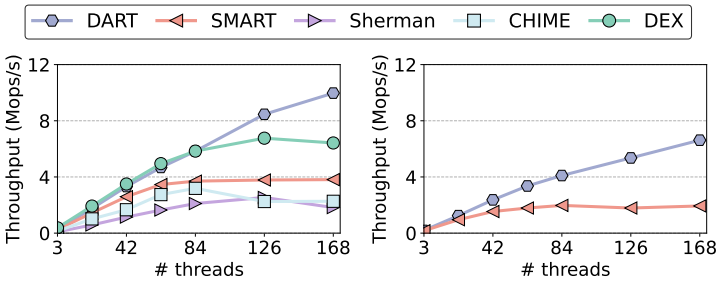


Fig. 15. The concurrent scalability with increasing number of client threads

Table 2. The memory usage (GB) across different datasets

		DART	SMART	Sherman	CHIME	DEX
Key-value		7.15				
Index	*randint*	0.86	17.86	1.63	2.04	2.09
	email	1.44	27.51	/	/	/

which effectively avoids lock contention under skewed workloads, thereby preventing performance degradation caused by contention.

Figure 14 also shows that other compared indexes experience slight performance degradation as skewness decreases. The primary reason is that lower skewness reduces access locality, thereby lowering the cache hit rate and degrading performance. Note that concurrency contention (under high skewness) has a relatively minor impact on performance in these indexes, as they handle part of the locking locally within the compute nodes, thereby reducing the network overhead of lock operations. However, this approach lacks fault tolerance to client crashes, as we discussed in Section 2.3.

5.6 Scalability

This section evaluates the concurrent scalability of DART by varying the number of client threads from 3 to 168 threads on compute nodes under the YCSB-A workload. As depicted in Figure 15, DART demonstrates strong scalability as the number of client threads increases, whereas other compared systems hit performance bottlenecks when the number of client threads exceeds 84 or 126. DART's superior scalability stems from two key factors. First, DART significantly reduces both round trips and bandwidth consumption through its *Express Skip Table* and *Adaptive Hashed Node*. Consequently, network bandwidth is not saturated, even when utilizing all available CPU cores in the compute nodes. Second, DART employs *Decoupled Metadata Organization* to achieve lock-free concurrency control, ensuring high scalability.

5.7 Memory Consumption

In this section, we compare the memory usage of DART with its counterparts across different datasets. As shown in Table 2, DART demonstrates the lowest memory consumption across both datasets. In contrast, SMART uses substantially more memory compared to DART. This is because the adaptive nodes in SMART only reduce the memory access size during index traversal, but do not save space consumption. Each inner node in SMART occupies around 2KB (equivalent to the size of an N256 node in DART), regardless of how few child nodes it contains. As a result, the index

size of SMART can reach 2.5 ~ 3.8× larger than the key-value entries. Moreover, DART consumes 1.9~2.4× less memory for index storage compared to B+Tree-based indexes like Sherman, CHIME, and DEX, due to the utilization of adaptive node sizes.

6 Related Work

Disaggregated Index. Disaggregated indexes are fundamental building blocks in disaggregated memory systems, and they have gained considerable attention in recent years. SepHash [28] and RACE [59] are one-sided RDMA-friendly hashing indexes, optimizing point queries with low network round trips. Sherman [43] is a disaggregated B+Tree that combines RDMA hardware and software optimizations to enhance index performance. CHIME [24] and Deft [40] further reduce in-node search overhead by incorporating hash-based or learned-based node structures. Despite this, the hierarchical nature of the B+Tree requires multiple network round trips to traverse the index. ROLEX [20] proposes a learned index that reduces the number of network round trips during traversal by predicting the location of key-value entries using learned models. However, ROLEX's performance degrades rapidly under dynamic workloads involving insertions and deletions, as these operations disrupt the previously learned data distribution. SMART [25] is a disaggregated radix tree that supports variable-sized keys and range queries. However, like Sherman, SMART must traverse its hierarchical structure, reading entire nodes at each level, leading to substantial read amplification. Although SMART employs client-side caching to mitigate traversal overhead, its performance gains are severely limited by the constrained cache capacity of compute nodes and the high overhead of cache maintenance [24, 40]. Sphinx [19] reduces the required cache size by employing a succinct filter. However, it still suffers from high read amplification during in-node searches and relies on inefficient lock-based concurrency control. DEX [23] and dLSM [44, 47] adopt a shared-nothing architecture that performs a logical partitioning of keys across different compute nodes to minimize cache coherence overhead. However, this approach has a fundamental limitation that makes it difficult to support upper-layer transactions that require access to cross-partition keys.

Adaptive Radix Tree. ART [17] is a space-efficient radix tree that adaptively changes its node size according to the number of child nodes. However, the original ART design faces significant remote memory access overhead and limited concurrent scalability while also lacking support for crash consistency in DM-based systems. In recent years, several persistent memory (PM) variants of ART, such as WOART [16], ROART [26], and ERT [42], have been developed to guarantee crash consistency. These variants typically first update the node data in a non-visible manner, followed by an atomic update of metadata (e.g., bitmap) to make the changes visible, leading to multiple round trips of remote memory access in DM-based systems. For mitigating memory access overhead, ERT [42] employs extendible hash structures to increase the fanout of the radix tree, thereby reducing the traversal depth. However, this approach introduces high memory access overhead during in-node searches and suffers from low space efficiency when handling skewed datasets. Heart [30] adopts unified, PM-friendly node structures to reduce memory access during in-node searches. However, it still requires hierarchical traversal and relies on version-based optimistic concurrency control, leading to high round trips of remote memory access in DM-based systems. Cuckoo Trie [51] is an efficient DRAM-based index that exploits memory-level parallelism in trie-based indexes. However, when applied to DM-based systems, it suffers from significant read amplification due to the need to access a large number of tree nodes.

7 Conclusion

In this paper, we present DART, the two-layer hashed ART index with lock-free concurrency protocols for DM architecture. DART introduces a hash-based *Express Skip Table* in the upper layer

and *Adaptive Hashed Nodes* in the base layer to minimize remote memory access overhead during index operations. Moreover, DART employs *Decoupled Metadata Organization* to achieve atomic updates, enabling lock-free concurrency control and ensuring crash consistency. Our evaluation demonstrates that DART outperforms the state-of-the-art counterparts by up to 5.8× in YCSB workloads. Besides, this paper currently focuses on RDMA-based DM architectures, and we plan to extend our study to CXL-based DM architectures in future work. We expect DART to be applicable to cache-non-coherent CXL (1.0 and 2.0), and it would be interesting to investigate its performance under cache-coherent CXL (3.0). Another interesting direction is to integrate DART into a real disaggregated database system such as Neon [29] and OpenAurora [32, 33].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and insightful suggestions. This work is supported by National Key Research and Development Program of China (Grant No. 2024YFB4505203), National Natural Science Foundation of China (NSFC) (Grant No. 62332012, 62227809, 62302290), and the Fundamental Research Funds for the Central Universities. Jianguo Wang acknowledges the support of the National Science Foundation under Grant Number 2337806.

References

[1] Jeff Bonwick et al. 1994. The slab allocator: An object-caching kernel memory allocator.. In *USENIX summer*, Vol. 16.
[2] Miao Cai, Junru Shen, and Baoliu Ye. 2024. Ethane: An Asymmetric File System for Disaggregated Persistent Memory. In *2024 USENIX Annual Technical Conference (USENIX ATC)*. 191–207.
[3] CederDB. 2024. https://cedardb.com.
[4] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value store. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 239–252.
[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. 143–154.
[6] Debendra Das Sharma Danny Moore. 2022. CXL 3.0: Enabling composable systems with expanded fabric capabilities. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL _3.0-Webinar_FINAL.pdf.
[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 401–414.
[8] Fonxat. 2018. 300 million email database. https://archive.org/details/ 300MillionEmailDatabase.
[9] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report.
[10] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC)*. 287–294.
[11] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. 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 (ASPLOS)*. 417–433.
[12] HyPer. 2013. https://hyper-db.de.
[13] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. CXL-ANNS: Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC)*. 585–600.
[14] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC)*. 773–787.
[15] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory (Extended Version). *arXiv preprint arXiv:2209.08743* (2022).
[16] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*. 257–270.
[17] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *International Conference on Data Engineering (ICDE)*. 38–49.
[18] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–8.

[19] Jingxiang Li, Shengan Zheng, Bowen Zhang, Hankun Dong, and Linpeng Huang. 2025. Sphinx: A High-Performance Hybrid Index for Disaggregated Memory With Succinct Filter Cache. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7.

[20] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[21] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment (PVLDB)* 13, 7 (2020), 1078–1090.

[22] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment (PVLDB)* 13, 8 (2020), 1147–1161.

[23] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory. *Proceedings of the VLDB Endowment (PVLDB)* 17, 10 (2024), 2603–2616.

[24] Xuchuan Luo, Jiacheng Shen, Pengfei Zuo, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2024. CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*. 110–126.

[25] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 553–571.

[26] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[27] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 757–773.

[28] Xinhao Min, Kai Lu, Pengyu Liu, Jiguang Wan, Changsheng Xie, Daohui Wang, Ting Yao, and Huatao Wu. 2024. SepHash: A Write-Optimized Hash Index On Disaggregated Memory via Separate Segment Structure. *Proceedings of the VLDB Endowment (PVLDB)* 17, 5 (2024), 1091–1104.

[29] Neon. 2025. https://github.com/neondatabase/neon.

[30] Liangxu Nie, Shengan Zheng, Bowen Zhang, Jinyan Xu, and Linpeng Huang. 2023. Heart: a Scalable, High-performance ART for Persistent Memory. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 487–490.

[31] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 371–386.

[32] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1–26.

[33] Xi Pang and Jianguo Wang. 2026. Reducing Tail Latency in Storage-Disaggregated Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

[34] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[35] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data (SIGMOD)*. 1981–1984.

[36] Chaoyi Ruan, Yingqiang Zhang, Chao Bi, Xiaosong Ma, Hao Chen, Feifei Li, Xinjun Yang, Cheng Li, Ashraf Aboulnaga, and Yinlong Xu. 2023. Persistent memory disaggregation for cloud-native relational databases. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 3*. 498–512.

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

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

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

[40] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2025. Deft: A scalable tree index for disaggregated memory. In *Proceedings of the Twentieth European Conference on Computer Systems (Eurosys)*. 886–901.

[41] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 37–44.

[42] Ke Wang, Guanqun Yang, Yiwei Li, Huanchen Zhang, and Mingyu Gao. 2023. When Tree Meets Hash: Reducing Random Reads for Index Structures on Persistent Memories. In *Proceedings of the ACM on Management of Data (SIGMOD)*. 1–26.

[43] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 international conference on management of data (SIGMOD)*. 1033–1048.

[44] Ruihong Wang, Chuqing Gao, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2024. Optimizing LSM-based Indexes for Disaggregated Memory. *VLDB Journal* 33, 6 (2024), 1813–1836.

[45] Ruihong Wang, Jianguo Wang, and Walid G. Aref. 2025. Cache Coherence Over Disaggregated Memory. *Proceedings of the VLDB Endowment (PVLDB)* 18, 9 (2025), 2978–2991.

[46] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proceedings of the VLDB Endowment (PVLDB)* 16, 1 (2022), 15–22.

[47] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *International Conference on Data Engineering (ICDE)*. 2835–2849.

[48] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Xstore: Fast rdma-based ordered key-value store using remote learned cache. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–32.

[49] Xingda Wei, Haotian Wang, Tianxia Wang, Rong Chen, Jinyu Gu, Pengfei Zuo, and Haibo Chen. 2023. Transactional Indexes on (RDMA or CXL-based) Disaggregated Memory with Repairable Transaction. *arXiv preprint arXiv:2308.02501* (2023).

[50] Yi Xu, Suyash Mahar, Ziheng Liu, Mingyao Shen, and Steven Swanson. 2024. Position: CXL Shared Memory Programming: Barely Distributed and Almost Persistent. *arXiv preprint arXiv:2405.19626* (2024).

[51] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting memory-level parallelism for efficient DRAM indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 147–162.

[52] Bowen Zhang, Shengan Zheng, Liangxu Nie, Zhenlin Qi, Linpeng Huang, and Hong Mei. 2024. Exploiting Persistent CPU Cache for Scalable Persistent Hash Index. In *International Conference on Data Engineering (ICDE)*. 3851–3864.

[53] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment (PVLDB)* 15, 6 (2022), 1187–1200.

[54] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling {Multi-Versioning} for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 801–819.

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

[56] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 658–674.

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

[58] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 461–476.

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

Received July 2025; revised October 2025; accepted November 2025