RADAR: A Skew-Resistant and Hotness-Aware Ordered Index Design for Processing-in-Memory Systems

Yifan Hua, *Student Member, IEEE*, Shengan Zheng, Weihan Kong, Cong Zhou, Kaixin Huang, Ruoyan Ma, and Linpeng Huang, *Senior Member, IEEE*

***Abstract*—Pointer chasing becomes the performance bottleneck for today's in-memory indexes due to the memory wall. Emerging processing-in-memory (PIM) technologies are promising to mitigate this bottleneck, by enabling low-latency memory access and aggregated memory bandwidth scaling with the number of PIM modules. Prior PIM-based indexes adopt a fixed granularity to partition the key space and maintain static heights of skiplist nodes among PIM modules to accelerate index operations on skiplist, neglecting the changes in skewness and hotness of data access patterns during runtime. In this article, we present RADAR, an innovative PIM-friendly skiplist that dynamically partitions the key space among PIM modules to adapt to varying skewness. An offline learning-based model is employed to catch hotness changes to adjust the heights of skiplist nodes. In multiple datasets, RADAR achieves up to 198.2x performance improvement and consumes 47.4% less memory than state-of-the-art designs on real PIM hardware.**

***Index Terms*—Processing-in-memory, ordered index, pointer chasing, load balance, index partition.**

I. Introduction

POINTER chasing [11], [43] in today's data-intensive applications exhibits irregular and unpredictable memory access patterns, leading to a poor cache hit rate and excessive memory accesses. The increasing performance gap between processor speeds and memory access speeds (i.e., the memory wall [14], [42]) renders pointer chasing the dominant performance bottleneck in these applications.

Manuscript received 13 March 2024; revised 9 June 2024; accepted 2 July 2024. Date of publication 9 July 2024; date of current version 25 July 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500303, in part by the National Natural Science Foundation of China (NSFC) under Grant 62227809 and Grant 62302290, in part by the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0102, and in part by the Natural Science Foundation of Shanghai under Grant 22ZR1435400. Recommended for acceptance by D. Li. *(Corresponding authors: Shengan Zheng; Linpeng Huang.)*

Yifan Hua, Weihan Kong, Cong Zhou, Kaixin Huang, Ruoyan Ma, and Linpeng Huang are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: huahuahuahua@sjtu.edu.cn; weihankong@sjtu.edu.cn; cong258258@sjtu.edu.cn; kaixinhuang@sjtu.edu.cn; maruoyan@sjtu.edu.cn; lphuang@sjtu.edu.cn).

Shengan Zheng is with the School of Electronic Information and Electrical Engineering, MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: shengan@sjtu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3424853

By embedding processors in memory, processing-in-memory (PIM) technologies [13], [27], [28], [29], [30], [31], [32] enable computations to be executed closer to memory, which is promising to break the memory wall. In PIM systems, the host side dispatches batches of operations to PIM modules and collects requested data from the PIM side [4], [5], [6], [7], [15], exploiting the high parallelism between PIM modules and accelerating pointer chasing.

Skiplist [9], [10], [41] is a widely used in-memory ordered index in memory systems [17], [19], [20], [35], [36], [37], [38], [39], [40], supporting both point (Get, $Predecessor$, $Insert$, $Delete$, $Update$) and range ($Scan$) operations through pointer chasing. Recent works focus on improving the throughput of skiplist by utilizing the PIM system, and they adopt two structures to store key-value pairs: range partition [6], [7], [15] and node distribution [4], [5]. Range partition [6], [7], [15] statically divides the entire key space into multiple disjoint coarse-grained ranges, maintained by each PIM module. They achieve high parallelism under uniformly random workloads,[1] but suffer from load imbalance (i.e., batches of operations may concentrate on keys in a range and are unevenly dispatched to PIM modules) under skewed workloads.[2] Node distribution [4], [5] lowers the load imbalance risk under skewed workloads by randomly distributing skiplist nodes among PIM modules and pulling frequently requested nodes from the PIM side to the host side. However, this structure results in poor node locality where adjacent nodes linked by a pointer are stored in different PIM modules, contributing to a large quantity of overheads for pointer chasing between PIM modules.

Both range partition and node distribution designs experience three limitations. First, they employ a fixed coarse or fine granularity to partition the key space among PIM modules under workloads with any skewness. Coarse granularity suffers from load imbalance under skewed workloads, while fine granularity incurs unnecessary overheads for skew-resistance and has poor performance under uniformly random workloads. Second, they maintain static heights of skiplist nodes regardless of the node

[1] A uniformly random workload refers to a workload where data and queries are with uniformly random keys. For example, workloads in the Zipfian distribution [21] with α values equal to 0 are uniformly random workloads.

[2] A skewed workload refers to a workload where data and queries are with non-uniform keys. For example, in the Zipfian distribution [21], the larger the α value, the more skewed the workloads are.

hotness changes. More pointer chasing operations are required to index hot keys stored in lower layers. Third, the parallelism between the host side and the PIM side is not fully exploited. When one side runs, the other side is idle most of the time.

This paper proposes RADAR, a skew-Resistant AnD hotness-Aware oRdered index for PIM systems. RADAR adopts a hash-based subrange distributed structure to dynamically partition the key space into multiple disjoint variable-length subranges, evenly distributed among PIM modules by hashing the start key of each subrange. By dynamically adjusting the granularity of each subrange between the range granularity in range partition and the node granularity in node distribution under varying skewness, RADAR benefits from both enhanced node locality and high parallelism exploitation. To cope with load imbalance stemming from a significant number of index operations concentrating on some subranges (i.e., skewed subranges), RADAR employs an adaptive subrange redistribution mechanism. RADAR splits skewed subranges into multiple finer-grained subranges and redistributes them to idle PIM modules, or pulls key-value pairs in the finest-grained skewed subranges from the PIM side to the host side. To better exploit the parallelism between the host side and the PIM side and be aware of the node hotness changes in each subrange, a machine learning (ML) model is trained on the host side to adjust the heights of skiplist nodes on the PIM side during the idle time of each side. Concisely, this paper makes the following contributions:

- We propose a brand new hash-based subrange distributed structure for ordered indexes in PIM systems to exploit the high parallelism between PIM modules and enhanced node locality in each PIM module.
- We present an adaptive subrange redistribution mechanism for RADAR to dynamically partition the key space among PIM modules under varying skewness.
- We design a learning-based node hotness classification model for RADAR to adaptively adjust the heights of skiplist nodes during runtime. To the best of our knowledge, RADAR is the first PIM-friendly index design that successfully incorporates an ML model for hotness prediction.
- RADAR outperforms state-of-the-art designs in multiple datasets on real PIM hardware, with up to 198.2x higher throughput and 47.4% less memory consumption.

The remainder of this paper is organized as follows. Section II provides the background of the PIM system and prior PIM-friendly ordered indexes, and our motivations. Section III presents the design overview of RADAR. We describe the adaptive index partition and hotness-aware node height adjustment in Sections IV and V, respectively. The experimental results are presented in Section VI. Section VII discusses related work, and Section VIII concludes the paper.

II. Background and Motivation

This section provides background on pointer chasing, the structure of the PIM system, and previous PIM-friendly ordered index works. We analyze the limitations of previous works and summarize our motivations to design a new PIM-friendly ordered index.

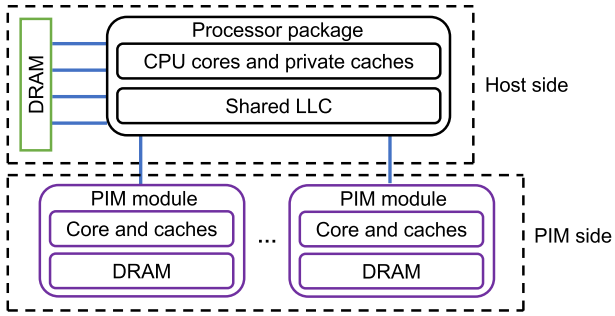


Fig. 1. The PIM architecture.

A. Pointer Chasing and PIM System

Pointer chasing is a fundamental operation in many data structures [9], [11], [15], [22], [23], [45], [47] such as tree, linked list, skiplist, and graph. Since nodes linked by pointers may be stored in memory far away from each other, pointer chasing has a poor cache hit rate and requires a significant number of memory accesses to look for the next hop. Consequently, the pointer chasing overhead becomes the performance bottleneck [5], [7], [14], [15], [48], [49] in these data structures. In a conventional machine, the cost of a pointer chasing operation involves the CPU processing overhead (3∼5 CPU cycles for loading instruction and pointer value, calculating the address, and dereferencing pointer), the cache miss overhead (50∼100 CPU cycles for L1, L2, and L3 caches), and the memory access overhead (100∼200 CPU cycles).

The emergence of Process-in-memory (PIM) technologies [2], [8], [27], [28], [29], [30] introduce a new paradigm that can enhance the efficiency of pointer chasing by allowing it to be executed within the memory devices. As shown in Fig. 1, the PIM system consists of two parts: the host side and the PIM side. The host side involves multiple CPU cores, caches, and DRAM. The PIM side includes multiple PIM modules. Each PIM module is composed of a data processor unit (DPU) with weaker computing capability than CPU, and capacity-limited caches and DRAM. PIM cores are closer to memory and access memory faster than CPU cores. The host side can send code to the PIM modules, launch the code, and detect when the code completes. It can also send data to and receive data from the PIM side. The two components of the PIM architecture, the host side and the PIM side, prefer different kinds of workloads [5]. The distributed PIM side prefers uniformly random workloads and suffers from the load imbalance under skewed workloads. In contrast, the host side prefers skewed workloads since the spatial and temporal locality characteristics in these workloads lead to better CPU cache efficiency.

Currently, UPMEM [18] (2.1 GHz CPUs on the host side and 350 MHz DPUs on the PIM side) is the only commercially-available PIM product. Compared with a conventional machine equipped with DDR4-2666, UPMEM has higher data processing parallelism and lower memory access latency since a large number of DPUs are embedded in memory. However, the existing UPMEM system has high overheads for host side calling PIM modules, host-PIM communication, and inter-PIM

communication. The memory bandwidth of a conventional 8 GB DDR4-2666 DIMM is 21 GB/s. The latency of a CPU core accessing the conventional DDR4-2666 memory for a 64-bit memory word is roughly 90 ns. In the existing UPMEM system, the memory bandwidth is 630 MB/s in a PIM module at a DPU frequency of 350 MHz. The aggregate intra-PIM bandwidth is linear with the number of PIM modules. An 8 GB UPMEM DIMM has 128 DPUs and its aggregate intra-PIM bandwidth is up to 79 GB/s. The latency of a PIM core accessing the PIM's memory for a 64-bit memory word in existing UPMEM is 14 ns. Therefore, the DPUs in UPMEM access memory faster than CPUs since they are closer to memory. In addition, if the system achieves load balance among all PIM modules, UPMEM can provide higher aggregate intra-PIM memory bandwidth than conventional DDR4-2666. Despite the DPUs in UPMEM accessing memory faster than CPUs, the latency for the host side calling DPUs in UPMEM is 0.5 ms, which is much higher than the memory access latency. The existing UPMEM system only offers 25 GB/s host-PIM communication bandwidth, which is smaller than the aggregate intra-PIM bandwidth. The existing UPMEM system does not support direct communication between PIM modules due to the scarce on-chip routing resources. The communication between PIM modules relies on host-PIM communication to exchange data, which involves loading data from one PIM module to the host's cache and then storing it to the destination PIM module. Considering the poor host-PIM communication ability, inter-PIM communication can easily become the performance bottleneck. The existing UPMEM system only offers 25 GB/s total host-PIM and inter-PIM communication bandwidth.

Although the architecture of a PIM system is similar to that of a distributed system, there are some differences between the two architectures. Compared with a distributed system, a PIM system has a smaller data size and simpler architecture. 1) In file systems, clients can communicate with other servers and clients through the network. A client can figure out the object's location in the distributed system independently. However, in PIM systems, applications cannot directly access PIM modules. PIM modules can only process application tasks through the host side. Applications send batches of index operations to the host side. Then the host side dispatches these index operations to PIM modules. After retrieving requested data from all PIM modules, these data are transmitted to the host side and returned to applications. 2) Due to the scarce on-chip routing resources in PIM modules, existing PIM systems do not support direct communication between PIM modules. The communication between PIM modules relies on host-PIM communication to exchange data. Considering the poor host-PIM communication ability, inter-PIM communication can easily become the performance bottleneck. Moreover, the basic system call overhead for the host side calling PIM modules is expensive, leading to the poor communication performance between PIM modules as well. 3) The computing capabilities of PIM cores are much weaker than those of CPU cores. The memory and cache capacities in a PIM module are much smaller than those on the host side. Therefore, compared to a device in distributed systems, a PIM module has a smaller data size and processes simpler tasks. 4) Since existing DRAM-based PIM modules are composed of volatile memory, there is no need to consider data recovery by storing multiple data replications in different PIM modules as well as failure domain selection. 5) A distributed system may be composed of heterogeneous devices such as HDD and SSD with different capacities, exhibiting different performance. Therefore, these devices have different weights in the distributed system for data distribution. However, in PIM systems, all the PIM modules are with the same hardware and software configurations. There is no distinction between PIM modules. 6) In a distributed system, a device may be added into the system or removed from the system. The weights of devices may be adjusted due to the added device and data may be migrated from the removed device to other devices. However, in a PIM system, the number of PIM modules is fixed during runtime.

PIM provides an opportunity to accelerate pointer chasing due to the fast memory access and high parallelism between PIM modules. Since the DPU in a PIM module is embedded closer to memory, the memory access overhead of a pointer chasing operation in a PIM system is lower than that in a conventional machine. Furthermore, a DPU has fewer caches than a CPU due to the limited on-chip resources. Therefore, the cache miss overhead of a pointer chasing operation in a PIM system is lower than that in a conventional machine. In a PIM module in UPMEM, the cost of a pointer chasing operation involves the DPU processing overhead (3∼5 DPU cycles for loading instruction and pointer value, calculating the address, and dereferencing pointer), the cache miss overhead (1 DPU cycle for L1 cache), and the memory access overhead (4 DPU cycles).

B. PIM-Friendly Ordered Indexes

Many prior works [4], [5], [6], [7], [15] employ PIM to enhance the throughput of skiplist, an ordered index widely used in memory systems [17], [19], [20], [35], [37], [38], [39], [40]. To benefit from the parallelism of the PIM system and amortize the communication cost between PIM modules (PIM-PIM) and between the host side and the PIM side (host-PIM), index operations are packed into batches and dispatched to PIM modules. For processing a batch of index operations on the skiplist, the host side dispatches the index operations to PIM modules. After receiving these index operation requests, each PIM module executes the requests in its local vault and sends the result back to the host side. Prior works can be divided into two categories based on the methods for partitioning the key space and constructing ordered indexes: range partition [6], [7], [15] and node distribution [4], [5].

Range partition: As Fig. 2(a) illustrates, range partition designs [6], [7], [15] statically partition the entire key space into disjoint coarse-grained ranges with the same length, maintained by each PIM module. Each PIM module builds an independent skiplist based on their key-value pairs. The search path from top to bottom for each index operation is integral in one PIM module, without communication overhead between PIM modules. Since the system performance is determined by the busiest PIM module, for workloads that request for uniformly random

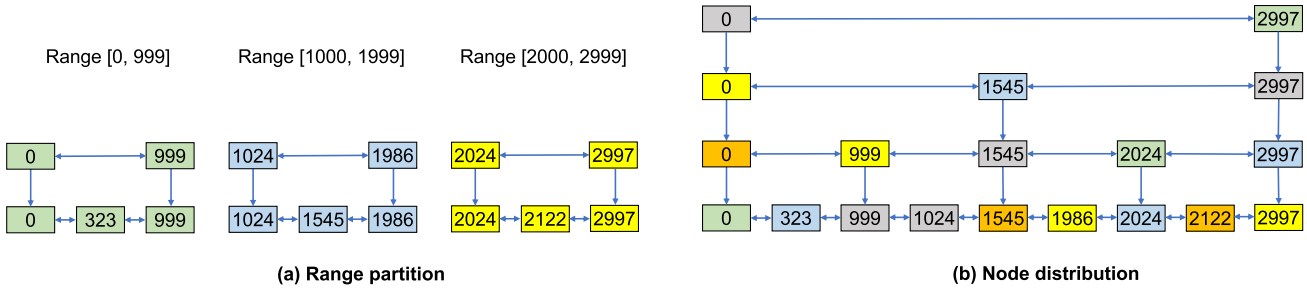


Fig. 2. Range partition and node distribution. In (a) and (b), nodes in different PIM modules are indicated by different colors. Values are stored in the bottom layer of the skiplist.

keys, each PIM module handles a similar number of requests and range partition designs achieve high parallelism. However, for skewed workloads that concentrate on keys in a small subset of the partitions, some PIM modules are overwhelmed while the rest PIM modules are idle, resulting in load imbalance across PIM modules.

Node distribution: Node distribution designs [4], [5], [47] aim to mitigate the load imbalance issue in range partition, as shown in Fig. 2(b). Since the coarse granularity approach to partition the key space in range partition designs suffers from load imbalance, node distribution designs adopt a fine granularity approach to partition the key space among PIM modules. Skiplist nodes are randomly distributed across PIM modules. This approach prevents the execution of pointer chasing from top to bottom within only a subset of PIM modules under skewed workloads. Frequently requested nodes are pulled from the PIM side to the host side for better load balance. The search path from top to bottom for each index operation is sliced into many segments in different PIM modules. At the end of each segment, index operations are redispatched to different PIM modules to the next segment, requiring PIM-PIM communication. Index operations may share a common node (i.e., contention node) on their search paths and be routed to the same PIM module, resulting in load imbalance. Pulling contention nodes from the PIM side to the host side alleviates the load imbalance, but requires multiple host-PIM communication rounds. Since the skiplist is built based on all key-value pairs rather than key-value pairs in each PIM module, it has a greater height compared to skiplists in range partition designs, contributing to more pointer chasing operations to traverse the skiplist from top to bottom.

C. Motivation

As described before, PIM provides an opportunity to accelerate pointer chasing operations due to lower memory access and cache miss overheads. However, existing range partition and node distribution designs experience limitations in the PIM system. Range partition designs suffer from load imbalance under skewed workloads since they statically partition the key space among PIM modules. Node distribution designs show robust resistance to skewed workloads, but bring unnecessary overheads for skew-resistance and perform much worse than range partition designs under uniformly random workloads. Limitations of state-of-the-art PIM-friendly indexes are summarized in Table I and detailed as follows. N in this section denotes the total number of key-value pairs in the PIM system.

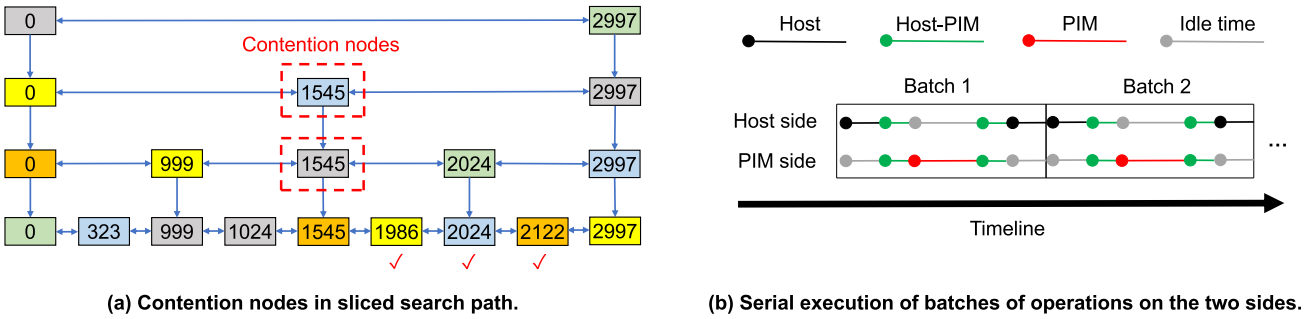
First, current skew-resistant ordered index structures bring unnecessary overheads: a considerable quantity of pointer chasing operations, contention nodes on the search paths, host-PIM and PIM-PIM communication cost, due to the following reasons:

- The height of the skiplist is not lowered by distributing skiplist nodes among PIM modules, resulting in excessive pointer chasing operations to traverse the high skiplist from top to bottom. For example, in Fig. 2(b), although key-value pairs in the bottom layer are evenly distributed among PIM modules and each PIM module maintains less than N key-value pairs, the skiplist height is still $O(\log N)$ and higher than that in range partition designs in Fig. 2(a).
- Skiplist nodes cannot perceive the hotness changes during runtime to dynamically adjust their heights. More pointer chasing operations are required to index hot keys stored in lower layers.
- The sliced search path from top to bottom for each index operation brings more contention nodes and communication cost. At the end of each search path slice, index operations are redispatched to all PIM modules since the next hop for each operation may reside in a distinct PIM module. Common shared nodes on the search paths of these index operations become contention nodes. For example, in Fig. 3(a), although the requested keys are uniformly distributed across PIM modules (three queries for 1,986, 2,024, and 2,122 in the bottom layer), common shared nodes (blue 1,545 and grey 1,545) on the search paths still become contention nodes, resulting in load imbalance. Pulling contention nodes to the host side and redispatching index operations to PIM modules require host-PIM and PIM-PIM communication on the critical path respectively, giving rise to high communication cost. *Thus, the overhead for skew-resistance should be reduced by lowering the skiplist height, dynamically adjusting heights of skiplist nodes with hotness changes, and minimizing the number of search path slices in different PIM modules:*

Second, in both range partition and node distribution designs, the granularity to partition the key space is static under any skewness during runtime. Each PIM module maintains a large

TABLE I
LIMITATIONS OF STATE-OF-THE-ART PIM-FRIENDLY INDEXES

	Skew-resistance	Low skiplist height	Hotness-awareness	Integral search path	Low host-PIM/PIM-PIM communication cost	Adaptive partition granularity	Host-PIM pipelining
Range partition	×	✓	×	✓	✓	×	×
Node distribution	✓	×	×	×	×	×	×
RADAR	✓	✓	✓	✓	✓	✓	✓



(a) Contention nodes in sliced search path. **(b) Serial execution of batches of operations on the two sides.**

Fig. 3. Limitations of state-of-the-art PIM-friendly indexes. In (a), nodes in different PIM modules are indicated by different colors. In (b), lines in different colors represent different works executed on the two sides.

and consecutive key range in range partition designs while a significant number of discontinuous fine-grained keys in node distribution designs. Coarse partition granularity exhibits good performance under uniformly random workloads, but suffers from load imbalance and lacks parallelism among PIM modules under skewed workloads. Fine partition granularity sacrifices node locality in each PIM module to lower the load imbalance risk, resulting in excessive PIM-PIM communication cost and degraded performance under uniformly random workloads. *As a result, the granularity to partition the key space should be dynamically adjusted to adapt to different skewness during runtime.*

Third, both range partition and node distribution do not fully exploit the parallelism between the host and PIM sides. The execution time for indexing consists of three non-overlapping components: host-only time, PIM-only time, and host-PIM communication time. Host-PIM communication requires both host and PIM sides, but the other two components only utilize one side. Since each batch of index operations may consist of both read and write operations, the system should guarantee the order of read-write/write-write operations in two adjacent batches, which are serially executed as Fig. 3(b) shows. When one side runs, the other side is idle most of the time. *Therefore, the parallelism between the host side and the PIM side should be fully exploited by pipelining the two sides.*

III. DESIGN OVERVIEW

RADAR is a high performance ordered index designed for PIM systems. We first describe its API and target workloads. Then, we introduce its design goals and explain how RADAR achieves these goals. Finally, we give the overview of RADAR's structure. Notations in the following sections are given in Table II.

TABLE II
NOTATIONS IN THIS PAPER

Notation	Definition
N	The number of key-value pairs in the system.
P	The number of PIM modules in the system.
S	The number of index operations in a batch.

API: RADAR supports a wide range of index operations, including point operations (*Get(key)*, *Predecessor(key)*, *Insert(key, value)*, *Delete(key)*, *Update(key, value_prev, value_curr)*) and range operation (*Scan(Lkey, Rkey, if_include_Lkey, if_include_Rkey)*).

Target workloads: RADAR targets both uniform workloads and non-uniform workloads with varying skew. RADAR adaptively adjusts the index partition strategy under varying skewness during runtime.

Goals and proposed techniques: As shown in Table III, RADAR aims to address the limitations in state-of-the-art PIM-friendly ordered indexes, by adopting the following techniques:

- The hash-based subrange distributed structure partitions the entire key space into multiple disjoint subranges with different granularities. RADAR distributes key-value pairs within these disjoint subranges to PIM modules by a hash function. Instead of partitioning the index built based on all key-value pairs in previous skew-resistant designs, RADAR builds an individual index in each PIM module and distributes key subranges to PIM modules using hash tables, lowering the skiplist height from $O(logN)$ to $O(log\frac{N}{P})$. To reduce the host-PIM and PIM-PIM communication cost in state-of-the-art works, the search path to traverse the skiplist from top to bottom in RADAR is integral in each PIM module. With the skewness changes of workloads, the granularities of these subranges are adjustable rather than fixed in previous range partition

TABLE III
DESIGN GOALS AND PROPOSED TECHNIQUES IN RADAR

Techniques \ Goals	Skew-resistance	Low skiplist height	Hotness-awareness	Integral search path	Low host-PIM/PIM-PIM communication cost	Adaptive partition granularity	Host-PIM pipelining
Hash-based subrange distribution (Section 4.1)		✓		✓	✓	✓	
Adaptive subrange redistribution (Section 4.2)	✓				✓	✓	
Learning-based node hotness classification (Section 5)			✓				✓

and node distribution designs, combining the advantages of coarse granularity (range partition) designs and fine granularity (node distribution) designs.

- The adaptive subrange redistribution mechanism evaluates the skewness and adaptively adjusts the granularity of subranges during runtime. To be skew-resistant under high skewness, instead of employing a fixed node chunking granularity in prior works, RADAR splits coarse-grained skewed subranges into multiple fine-grained subranges and redistributes key-value pairs within these fine-grained subranges to other idle PIM modules. To reduce memory consumption for index node replication in previous node distribution designs, RADAR only stores one copy of each index node in the PIM system. Besides, key-value pairs within the finest-grained skewed subranges are pulled to the host side to benefit from CPU cache locality. The redistribution is accomplished in one PIM-PIM communication round and one host-PIM communication round to lower the communication cost, contributing to lower system call overhead for calling PIM modules compared with state-of-the-art skew-resistant works.
- To address the node hotness-unawareness in previous works, RADAR employs a learning-based time series model to predict the hotness of subranges through features of historical index operations. By raising the heights of skiplist nodes in hot subranges and lowering the heights of skiplist nodes in cold subranges, RADAR reduces the number of pointer chasing operations to retrieve requested values compared with state-of-the-art designs. RADAR pipelines the host side and the PIM side to run the node hotness classification model, leveraging resources on the two sides during their idle time.

Overview of RADAR's structure: Fig. 4 presents the overview of RADAR. Applications send batches of index operations to RADAR and request for data through indexing keys in RADAR. By querying some hash tables on the host side, the preprocess layer determines which PIM module to send each index operation to and evaluates the skewness across PIM modules. Then, RADAR dispatches a batch of index operations to their target PIM modules to retrieve requested data and decides whether to redistribute key-value pairs among PIM modules based on the workload skewness. In parallel, features of index operations in each batch are fed into a node hotness classification model for training and making hotness predictions for skiplist nodes in PIM modules. The model then raises the heights of hot nodes and lowers the heights of cold nodes in skiplist to reduce the number of pointer chasing operations. Note that the model running

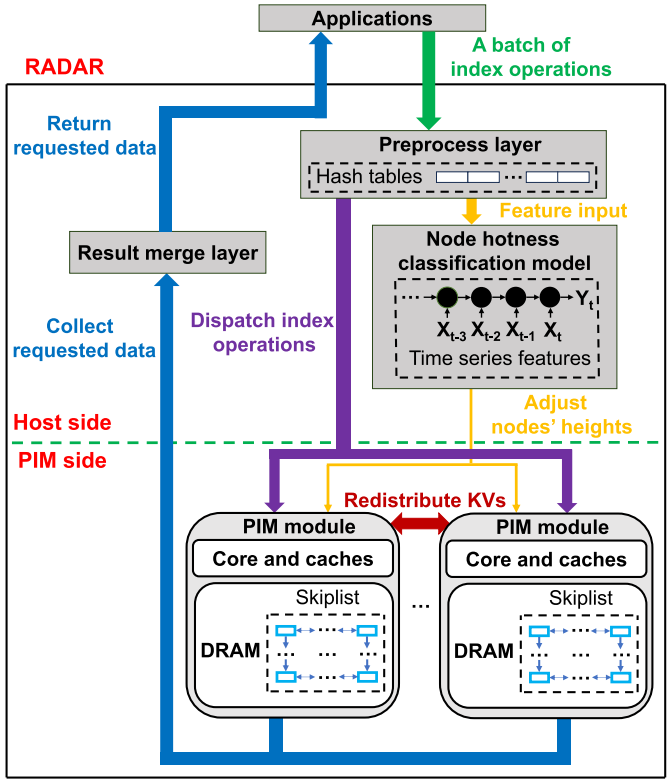


Fig. 4. Overview of RADAR.

is offline and asynchronous with processing index operations. After retrieving requested data from PIM modules, these data are transmitted to the host side, merged by the merge layer, and returned to applications.

IV. ADAPTIVE INDEX PARTITION

In this section, we describe how RADAR distributes key-value pairs, constructs the skiplist, and enhances load balance across PIM modules under different skewness. We first illustrate the metadata and the subrange distributed structure on the host and PIM sides in Section IV-A. Then we present the adaptive subrange redistribution mechanism under different skewness in Section IV-B.

A. Hash-Based Subrange Distribution

To distribute skiplist nodes across PIM modules for skew-resistance while enhancing node locality in each PIM module, the key space is partitioned into multiple disjoint subranges.

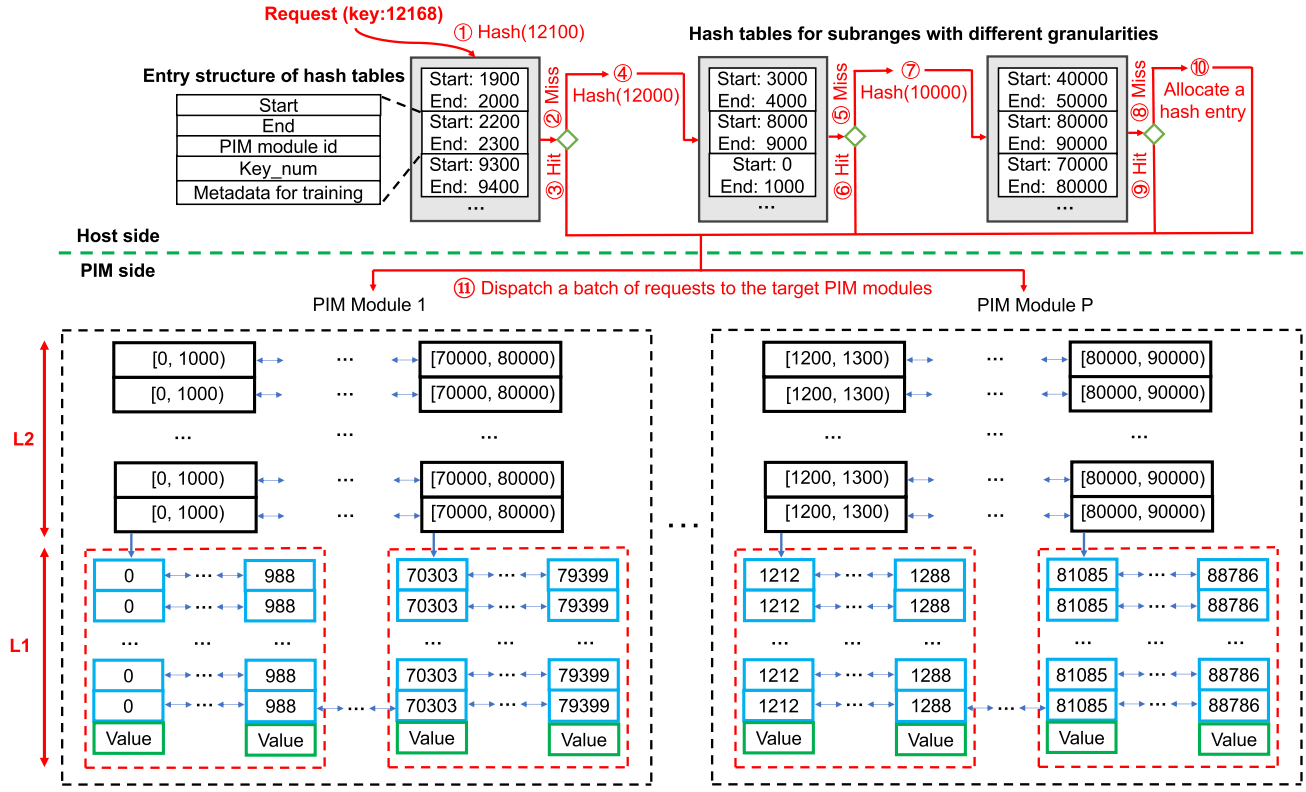


Fig. 5. Hash-based subrange distribution. A black box in L2 represents a subrange node. In each PIM module, all subrange nodes in L2 compose a skiplist. A blue box in L1 represents a key node and a green box in L1 represents the value of the key. Key nodes and their values within a red dotted box in L1 compose an individual skiplist for a subrange.

These subranges are evenly distributed among PIM modules by applying a hash function to the start key of each subrange, preventing skewed subranges from being stored in one PIM module. The granularity of each subrange is dynamically adjustable with changes in the skewness of requested data during runtime, combining the advantages of both fine granularity and coarse granularity. Each PIM module maintains an individual skiplist built based on keys within its subranges, ensuring the integrity of the search path for each index operation within a PIM module and lowering the skiplist height from $O(logN)$ to $O(log\frac{N}{P})$. Fig. 5 illustrates the structure of hash-based subrange distribution in RADAR.

Metadata and workflow on the host side: On the host side, to better leverage the advanced computing resources and avoid expensive pointer chasing, RADAR only performs a limited number of hash operations to dispatch index operations to target PIM modules. The overhead of employing simple traditional hashing is smaller than that of employing intelligent algorithms. For data migration between PIM modules, modifying traditional hash tables is less time-consuming than retraining an intelligent model. Given that dispatching index operations to PIM modules is a critical step on the host side as shown in Fig. 4, traditional hashing is more suitable than intelligent algorithms in PIM systems. Hash tables are built for subranges with different granularities. For each index operation, RADAR queries the hash tables to locate the target subrange containing the requested key. The entry structure of hash tables includes the start and the end of the subrange, the id of the PIM module storing all keys within the subrange, the number of keys in the subrange, and some metadata for training (detailed in Section V-A). The subrange granularities are dynamically adjusted under different skewness (detailed in Section IV-B). For each index operation, RADAR queries hash tables for subranges from fine to coarse.

To illustrate, Fig. 5 shows an example where the granularities of subranges are 100, 1,000, and 10,000. Three hash tables contain the metadata of subranges with the three granularities, respectively. For a requested key 12168, RADAR queries the three hash tables from fine to coarse. For a hash table hit (③, ⑥, and ⑨), RADAR finishes the hash table query. For a hash table miss (② and ⑤), RADAR continues to query the next hash table for coarser subranges. RADAR at most performs hash operations on 12100 (①), 12000 (④), and 10000 (⑦) in their respective hash tables to locate the target hash entry and determine the target PIM module to dispatch the request to (⑪). If a new inserted key results in misses across all hash tables (⑧), RADAR allocates a hash entry for the finest-grained subrange containing the key (⑩) to reduce the number of misses in subsequent hash table queries.

Skiplist built on the PIM side: On the PIM side, to better leverage the fast memory access and avoid expensive computing, RADAR executes pointer chasing in the skiplist. Each PIM module maintains multiple subranges of the key space with different granularities, and builds a skiplist based on these keys. The skiplist in each PIM module is divided into the upper part

and the lower part. The upper part (L2) consists of subrange nodes and the lower part (L1) contains key nodes within the corresponding subrange. All subrange nodes in L2 compose a skiplist and key nodes within each subrange in L1 compose a skiplist. For processing a request to traverse the skiplist in a PIM module from upper to lower, RADAR first locates the subrange containing the requested key in L2 and then locates the requested key in L1. In each PIM module, RADAR employs the array structure to store nodes with the same subrange or key in different layers. If keys are found in the upper layer, their values can be retrieved without pointer chasing to lower layers. The search path for each index operation traversing the skiplist from top to bottom is integral in a PIM module, without PIM-PIM communication cost. For better range scan performance, in each PIM module, all leaf nodes in L1 are linked. Note that if the number of keys in a subrange is 0, to reduce the memory consumption and the pointer chasing cost, the subrange does not appear on either the host side or the PIM side.

RADAR adopts several methods in the subsequent sections to detect hot subranges and keys, and reduce the number of pointer chasing operations in L2 and L1 for obtaining the required key-value pairs. Skewed subranges are split into finer-grained subranges during runtime and key nodes in L1 are redistributed to other PIM modules, contributing to fewer pointer chasing operations when searching for hot keys in L1 (detailed in Section IV-B). Furthermore, the heights of subrange nodes in L2 are adjustable. Raising the heights of hot subrange nodes and lowering the heights of cold subrange nodes can reduce the number of pointer chasing operations when traversing L2 (detailed in Section V).

B. Adaptive Subrange Redistribution

The adaptive subrange redistribution mechanism dynamically adopts different subrange redistribution strategies under varying skewness. Under low skewness, each PIM module processes similar number of index operations without load imbalance risk. However, under high skewness, requested keys may concentrate on some subranges (i.e., skewed subranges), bringing load imbalance among busy PIM modules (each busy PIM module processes more than $\frac{S}{P}$ index operations) and idle PIM modules (each idle PIM module processes less than $\frac{S}{P}$ index operations). To mitigate the load imbalance among PIM modules, each coarse-grained skewed subrange in busy PIM modules is split into P finer-grained disjoint subranges, and key-value pairs within these finer-grained subranges are redistributed to idle PIM modules. Besides, key-value pairs within the finest-grained skewed subranges in busy PIM modules are pulled to the host side since these subranges cannot be further split into finer granularity.

Dynamic subrange granularities: Since the skewness is changeable over time, the subrange granularities are dynamically adjusted. In the initial state, all the subranges are with the same granularity. Compared with initially splitting the key space in large granularity, initially splitting the key space in small granularity results in high metadata overhead. Considering a system involving millions of key-value pairs, the system requires recording the location of each small subrange in PIM modules, incurring high metadata overhead. In contrast, initially splitting the key space in large granularity, only a small percentage of skewed subranges are split into fine-grained subranges and redistributed to PIM modules. Therefore, employing a large granularity to initially split the key space is more suitable. The key space is partitioned into P disjoint ranges distributed to P PIM modules. For index operations requesting for uniformly random keys within unskewed subranges, each PIM module processes similar number of index operations with high parallelism. RADAR does not split these unskewed subranges into finer-grained subranges. For index operations concentrating on keys within a small portion of skewed subranges, each of these skewed subranges is split into P finer-grained subranges until it cannot be further split. If there are no subranges remaining in a hash table (e.g., all subranges in the hash table have been split into finer-grained subranges), the hash table is deleted from the host side.

Subrange redistribution decision: The subrange redistribution decision is made based on the benefit of load balance and the cost to achieve the load balance under varying skewness. Given that the performance bottleneck is determined by the busiest PIM module, RADAR adopts a greedy strategy to make subrange redistribution decision for PIM modules from busy to idle.

The benefit ($Benefit$) of load balance, which is achieved by redistributing a skewed subrange from the busiest PIM module to idle PIM modules, is the load reduction of the busiest PIM module among these load-redistributed PIM modules.

$$Benefit = N_{curr\_busiest} - N_{new\_busiest}, \tag{1}$$

where $N_{curr\_busiest}$ and $N_{new\_busiest}$ is the number of index operations processed by the busiest PIM module before and after redistributing the skewed subrange, respectively.

The subrange redistribution process for a skewed subrange containing n_k key-value pairs includes two parts: retrieving and deleting n_k key-value pairs in the skewed subrange, and redistributing n_k key-value pairs to idle PIM modules. The retrieving and deleting process is essentially the same as a range operation to traverse n_k key-value pairs in the busiest PIM module. The redistributing process transmits n_k key-value pairs between PIM modules, then inserts a subrange node to L2 and rebuilds a skiplist in L1 based on the redistributed key-value pairs in each idle PIM module simultaneously. Note that the key-value pairs inserted in L1 are consecutive and in order, and the rebuild process can be accomplished in PIM modules' cache with negligible cost compared to pointer chasing in memory. To minimize the PIM-PIM communication cost of redistributing, RADAR redistributes all nodes in one communication round between PIM modules until no skewed subrange is identified. Thus, the cost of the subrange redistribution process includes the cost of a range operation ($Cost_{range}$), a point operation ($Cost_{point}$), as well as an additional PIM-PIM communication ($Cost_{comm}$).

Since the main cost for each index operation is pointer chasing in memory, RADAR uses the number of pointer chasing operations as a metric to evaluate the three costs of the subrange redistribution process for a skewed subrange. For processing

each batch of index operations, RADAR calculates the average number of pointer chasing operations in a point (Avg_{point}) and a range (Avg_{range}) operation. RADAR assumes that the number of pointer chasing operations in $Cost_{point}$ is the same as that in Avg_{point} in the recent batch. $Cost_{range}$ requires $log\frac{N}{P} + n_k$ pointer chasing operations in expectation. n_k key-value pairs are transmitted between PIM modules in $Cost_{comm}$, which is converted to the number of pointer chasing operations in memory.

$$Cost_{point} = Avg_{point} \tag{2}$$

$$Cost_{range} = log\frac{N}{P} + n_k \tag{3}$$

$$Cost_{comm} = \frac{n_k \times KV\_size \times T_{comm}}{T_{pim\_mem}}, \tag{4}$$

where KV_size is the average size of a key-value pair in bytes, T_{comm} is the time to transmit a byte between PIM modules, and T_{pim_mem} is the time for a memory access in a PIM module. The values of KV_size, T_{comm}, and T_{pim_mem} are constants in the PIM system.

We use the the following formula to evaluate the skewness of a batch of index operations and decide whether to split or pull (SP) a skewed subrange in busy PIM modules, by converting the benefit of load balance and the subrange redistribution cost into the number of index operations in the batch.

$$SP = Benefit - \frac{Cost_{point} + Cost_{range} + Cost_{comm}}{\beta Avg_{point} + (1-\beta)Avg_{range}}, \tag{5}$$

where β is the percentage of point operations in the batch and its value can be obtained in the preprocess. RADAR calculates SP for skewed subranges in PIM modules from busy to idle: if $SP \leq 0$, RADAR gains no benefit from the subrange redistribution and does not spilt or pull the skewed subrange; if $SP > 0$, RADAR splits the skewed subrange to idle PIM modules or pulls it to the host side. In this way, under low skewness, there is no subrange redistribution overhead since $SP \leq 0$. Under high skewness, the number of redistributed nodes is far smaller than the batch size since requests concentrate on a small portion of nodes. Only these skewed nodes across busy PIM modules are concurrently redistributed to idle PIM modules. RADAR does not merge fine-grained subranges into coarse-grained subranges. Since only a small percentage of skewed subranges rather than all subranges are split into fine-grained subranges, the metadata overhead is not very high. Merging multiple fine-grained subranges into a coarse-grained subrange requires migrating key-value pairs within these subranges from several PIM modules to one PIM module. The data migration overhead is more significant than the benefit of reducing the metadata overhead due to the expensive inter-PIM communication. Experiments in Section VI demonstrate that the subrange redistribution overhead is substantially lower than the load imbalance overhead and request redispatching overhead in state-of-the-art designs under varying skewness.

The adaptive subrange redistribution design is not only effective for load balance but also reduces the number of keys in skewed subranges, resulting in fewer pointer chasing operations when searching for hot keys in L1. The number of communication rounds for processing each batch of index operations is upper-bounded by two, one (PIM-PIM communication) for redistributing key-value pairs within skewed subranges to idle PIM modules and the other (host-PIM communication) for retrieving required values of the index operations and pulling nodes within the finest-grained skewed subranges.

V. Hotness-Aware Node Height Adjustment

In this section, we first present the learning-based node hotness classification model for detecting hot subranges and adjusting the heights of subrange nodes in Section V-A. Then we describe how RADAR incorporates the hotness classification model in PIM systems in Section V-B.

A. Learning-Based Node Hotness Classification

The node hotness classification model aims to reduce the number of pointer chasing operations by raising the heights of hot subrange nodes and lowering the heights of cold subrange nodes in L2. For processing a request to traverse the skiplist from upper to lower in a PIM module, RADAR first locates the subrange containing the requested key in L2 and then locates the requested key in L1. For requesting keys within hot subranges, raising the height of hot subrange nodes contributes to less pointer chasing when locating the subrange node in L2. Therefore, assigning the hot subrange node a higher height reduces the number of pointer chasing operations. Subrange nodes are classified into H_{L2}-class (H_{L2} is the height of L2 in PIM modules) based on their hotness. RADAR trains a lightweight time series model for node hotness classification.

Model selection: The node hotness classification model uses features of historically requested subranges to make an H_{L2}-class hotness prediction for each currently requested subrange. The prediction result indicates which level of L2 the subrange node should be placed in. Then the model adjusts the heights of those subrange nodes whose heights deviate most from the predicted results. A supervised time series model is suitable to extract features from historical index operations and classify node hotness. The machine-learning time series model extracts features from historical index operations and has a high prediction accuracy for multi-class node hotness classification. Furthermore, since there is a large amount of idle time on the host and PIM sides in state-of-the-art designs as shown in Fig. 3(b), RADAR carefully leverages these idle time for model training and prediction without incurring extra overhead compared with using a simple static mapping method. Therefore, instead of using a simple static mapping method, RADAR employs a machine-learning model to solve this problem due to the high node hotness classification accuracy without incurring extra overhead. RADAR incorporates the widely used gated recurrent unit (GRU) [26], a lightweight supervised machine learning model capable of processing time series data and learning the labeled input features, as shown in Fig. 6. To ensure that the time for training and making predictions in one epoch on the host side is overlapped by the index processing time on the PIM side, the data sets for training and making predictions are sampled

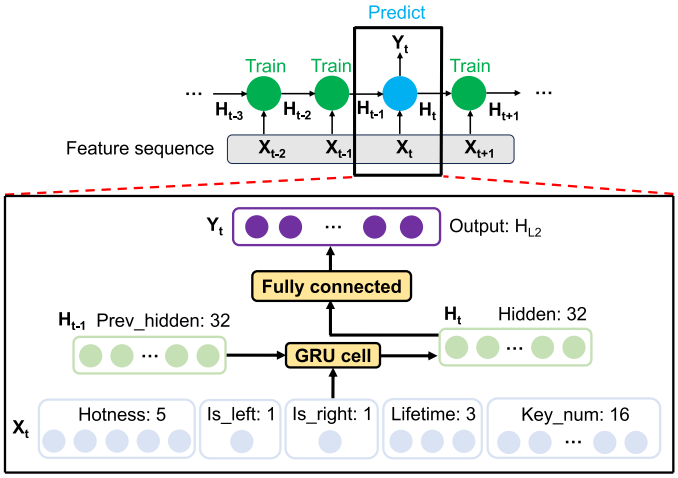


Fig. 6. The node hotness classification model.

from each batch of index operations. A curvilinear model is trained to fit hotness thresholds for data labeling by sampling and sorting recently requested nodes.

Model structure and feature extraction: In each window for processing multiple batches of index operations, the GRU model is trained for one epoch for each batch and makes a prediction at the end of the window. For training and predicting hot subranges, RADAR chooses the hotness of subranges as feature input. In addition, since the data access patterns of a large number of workloads exhibit spatial locality (data near the requested data tends to be requested later) and temporal locality (requested data tends to be requested again later) [50], [51], [52], [53], [54], [55], [56], the spatial locality and temporal locality features are helpful for training and predicting hot subranges. Since a subrange containing a large number of keys is more likely to be requested than a subrange containing a small number of keys, the number of keys in a subrange is used as feature input as well. During our design iterations, the following features of each subrange are selected as the model input: the hotness of subranges evaluated by counters (Hotness); the spatial locality of workload evaluated by whether the adjacent subranges are requested (Is_left and Is_right); the temporal locality of workload evaluated by the number of window gaps between two accesses to the subrange (Lifetime), which is equivalent to using the global window number as a virtual clock; the number of keys in a subrange (Key_num). For efficient processing, RADAR breaks numerical inputs into hexadecimal digits and concatenates them as the model inputs. The number of digits used for each feature is chosen so that most cases can be handled without overflow. The model is trained with the cross entropy loss function [44] and the Adam optimizer [46] using the training data sampled from each batch. The GRU model has a single-layer hidden state with 32 neurons. On the output side, the hidden state of the last GRU cell is pushed through a fully connected layer to produce H_{L2} output values, determining which layer in L2 should a subrange node be placed in. Finally, $argmax$ is applied to get the prediction result. If the number of nodes in the predicted layer reaches the upper limit, RADAR sets the maximum output value to zero and applies $argmax$ again to get the result with sub-maximum confidence.

Adaptive data labeling: Since RADAR uses supervised machine learning to predict the hotness of subranges, with changes in the hotness of requested data during the workload's runtime, thresholds between adjacent hotness classes are adaptively set for data labeling. On the output side, the model is designed to make a H_{L2}-class hotness prediction for each requested subrange based on their historical hotness. Ideally, subrange nodes are placed in each layer of L2 from top to bottom according to their hotness. However, sorting all subrange nodes by hotness in each window consumes a large amount of time. To avoid the time-consuming sorting, RADAR samples and sorts some requested subrange nodes in each epoch in the current window, and trains a lightweight curvilinear regression model [24], [25] to fit the threshold function in the next window. Note that the number of sampled nodes in each epoch is adjustable according to different batch sizes. If the value of H_{L2} changes with the insertion, deletion, splitting, and pulling of subrange nodes, the curvilinear regression model fits new thresholds by using current thresholds.

B. Host-PIM Pipelining Mechanism

As described in Section II-C, RADAR has potential to leverage resources on the host side or the PIM side when the other side is busy. RADAR runs the node hotness classification model to figure out hot and cold subranges and adaptively adjusts the heights of subrange nodes, by pipelining the host side and the PIM side as Fig. 7 illustrates. Apart from the host, host-PIM, and PIM time for processing batches of index operations, RADAR makes full use of the idle time of both the host side and the PIM side to train model, make hotness predictions, adjust the hotness thresholds for adaptive data labeling, and adjust the heights of subrange nodes in PIM modules. Since the PIM side consumes more time than the host side for processing index operations, and to better leverage the advanced computing and memory resources on the host side, in each window, the model trains and predicts on the host side when the PIM side is busy. To entirely overlap the training and prediction time on the host side with the index processing time on the PIM side while providing sufficient nodes' features for model training in a window, the frequency of height adjustment is dependent on the index processing time on the PIM side and the model training efficiency on the host side. The model is trained for one epoch for each batch of index operations. At the end of each window, the model predicts by using features of the last batch of index operations as input, and adaptively adjusts thresholds between adjacent hotness classes. Finally, the PIM side raises or lowers the heights of those subrange nodes whose current heights deviate most from the predicted heights. Compared with previous works [4], [5], [6], [7], [15] that do not make use of the idle time on the host and PIM sides in PIM systems, RADAR leverages these idle time to implement the height adjustment mechanism without incurring extra overhead.

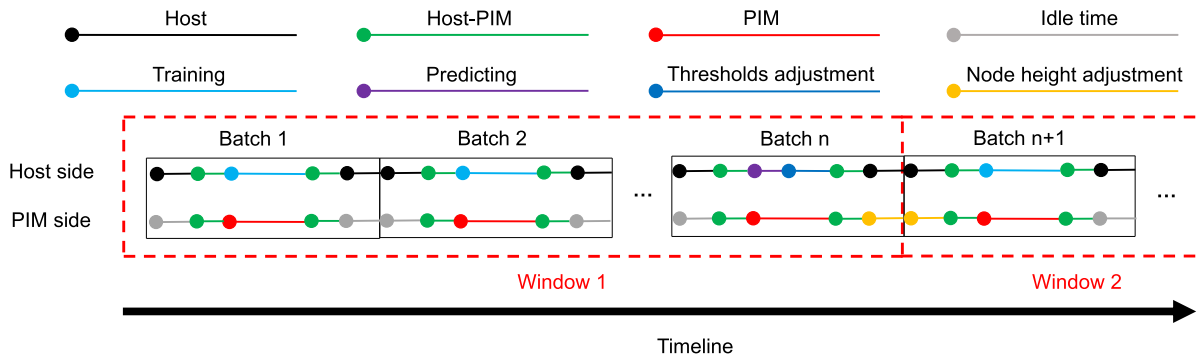


Fig. 7. Host-PIM pipelining. Training, predicting, hotness thresholds adjustment, and node height adjustment of the model are pipelined with index operations processing.

VI. Evaluation

In this section, we evaluate the performance of RADAR against state-of-the-art PIM-friendly and traditional ordered index designs in multiple datasets.

A. Experimental Setup

Testbed: Our testbed machine for PIM-friendly indexes is a dual-socket server with two Intel(R) Xeon(R) Silver 4216 CPUs, each CPU equipped with 16 cores at 2.10 GHz and 22 MB cache. Each socket has six memory channels: four DDR4-2666 DIMMs are installed on two channels, while eight UPMEM DIMMs are on the other four channels. Each of the sixteen UPMEM DIMMs has two ranks, each rank has eight chips, and each chip has eight PIM modules. There are 2048 PIM modules in total. For fair comparisons, our testbed machine for traditional indexes has the same CPU and the memory channel count. DDR4-2666 DIMMs are installed on all the memory channels.

Compared Systems: We compare RADAR against three state-of-the-art PIM-friendly designs: Range-partition [6], Jump-push [4], and PIM-tree [5]; three traditional designs: Bronson BST [12], (a,b)-tree [1], and Unrolled Skiplist [10]. We borrow Kang's [5] implementations of Range-partition, Jump-push, and PIM-tree. We directly use the open-sourced code of Bronson BST [12], (a,b)-tree [1], and Unrolled Skiplist [10].

Workload Setup: In all experiments, the same as the prior design [5], we first insert 500 million key-value pairs in each index, then evaluate the index by running 100 million index operations. The size for each batch of index operations is 2 million. Each scan operation retrieves 32 $\sim$ 1024 key elements in expectation. The sizes of both keys and values are set to 8 bytes.

B. Design Space Exploration

For parameter settings in RADAR, the number of sampled subrange nodes in each batch for training and the number of training epochs in each window influences the system performance. We explore the model prediction accuracy and the throughput performance of all index operations supported by RADAR, with different numbers of sampled nodes and training epochs. The number of sampled nodes in a batch varies from 4 thousand to 32 thousand, and the number of training epochs in a window ranges from 2 to 6.

Fig. 8 illustrates the prediction accuracy of the node hotness classification model. Fixing the number of sampled nodes, the model prediction accuracy gradually rises with more training epochs and tends to saturate after 4 epochs, enabling the model to learn the recent features of index operations. For the same number of training epochs in a window, sampling more nodes in a batch for training gives rise to better feature extraction and higher prediction accuracy. However, higher prediction accuracy does not necessarily lead to better system performance.

Fig. 9 shows the throughput performance of RADAR with different parameter configurations normalized to RADAR without the node hotness classification model. For sampling 4 thousand nodes and 8 thousand nodes, the time for training on the host side is entirely hidden by the indexing time (42 milliseconds) on the PIM side. Therefore, higher prediction accuracy contributes to better throughput performance since fewer pointer chasing operations are required for searching hot skiplist nodes. For sampling 16 thousand nodes and 32 thousand nodes, the time for training on the host side is much longer than that for indexing on the PIM side, generating idle time on the PIM side and degrading the system performance.

For the rest of experiments in this paper, we present our results of RADAR by sampling 8 thousand nodes in each batch of index operations and training 4 epochs in each window. With this configuration, RADAR achieves 71.8% node classification accuracy and 1.18x normalized throughput improvement.

C. Microbenchmarks

To show the system performance under different skewness, we generate workloads in Zipfian distribution [21] with α values ranging from 0 (uniformly random) to 1.2. Fig. 10 shows the throughput performance of state-of-the-art PIM-friendly and traditional ordered indexes.

Effectiveness of our proposed optimizations: We break down our proposed optimizations to analyze their effectiveness.

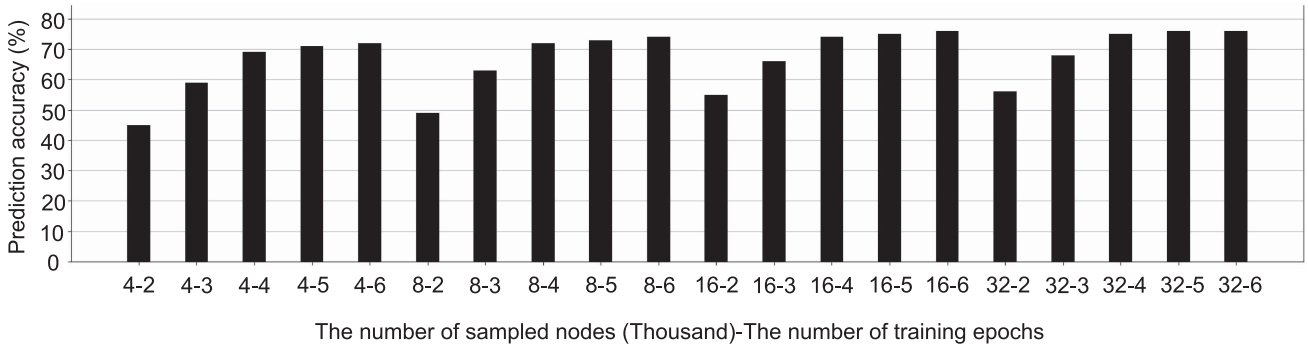


Fig. 8. Prediction accuracy with different numbers of sampled nodes and training epochs.

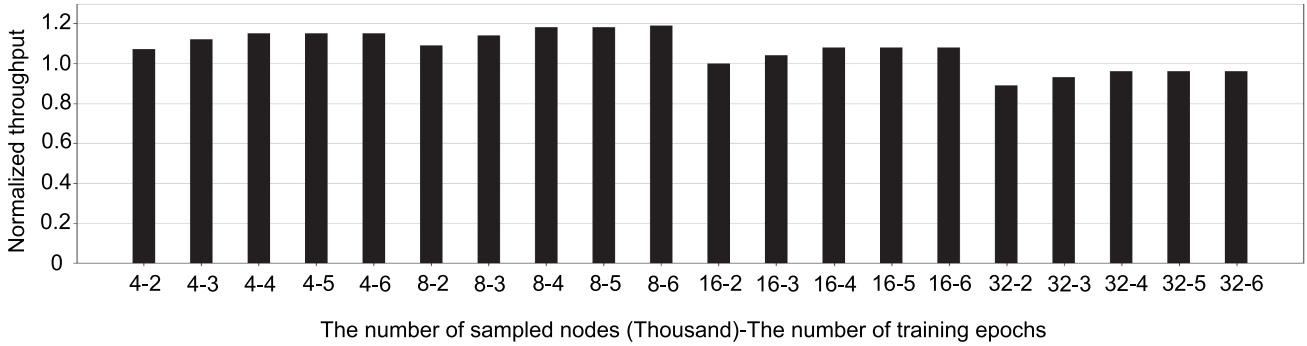


Fig. 9. Normalized throughput performance with different numbers of sampled nodes and training epochs.

RADAR (basic) only employs the hash-based subrange distributed structure. Based on RADAR (basic), the adaptive subrange redistribution is adopted by RADAR (no GRU), and the GRU model is further added to RADAR. The performance of RADAR (basic) drops with the workload becoming skewed, but still outperforms Range-partition. This is because although some subranges become skewed due to the lack of node redistribution, the granularity of each subrange is far finer than that in Range-partition. The hash-based subrange distribution prevents these skewed subranges from being stored in one PIM module. RADAR (no GRU) shows robust resistance to skewed workloads due to the adaptive subrange redistribution for skewed subranges. The host-PIM pipelining and GRU model improve the performance by 12% (for $\alpha = 0$) $\sim$ 22% (for $\alpha = 1.2$) for point operations and by 5% (for $\alpha = 0$) $\sim$ 9% (for $\alpha = 1.2$) for range operations. The reason why point operations achieve more benefits than range operations is that our learning-based node hotness classification model reduces the number of pointer chasing operations to locate subranges in L2 rather than those to traverse the ordered linked list in L1 for range operations. Besides, the learning-based node hotness classification design is more effective for skewed workloads than uniformly random workloads since raising the heights of skewed subrange nodes could reduce more pointer chasing operations.

Performance comparisons: RADAR outperforms all state-of-the-art designs for all operations under different skewness, other than Range-partition for scan operation under uniformly random workloads. Under uniformly random workloads, RADAR performs 19.8% worse than Range-partition for scan operation due to the overhead of querying hash tables on the host side, while performing better for other operations attributed to the node hotness classification. As α increases, the performance of Range-partition drops sharply (up to 168.1x worse than RADAR) since the key space is statically partitioned in coarse granularity among PIM modules. RADAR achieves up to 6.8x and 15.8x higher throughput than traditional ordered indexes for query and update operations respectively, due to the parallelism exploitation in the PIM system. The worse performance of the two tree indexes for update operations is attributed to the cost of maintaining tree balance. As α increases, traditional indexes perform better while PIM-friendly indexes perform worse since skewed workloads enable the host side to benefit from cache locality but cause load imbalance among PIM modules. Node distribution designs show robust resistance to skewed workloads but perform worse than RADAR (up to 198.2x for Jump-push and 7.3x for PIM-tree), owing to their large quantity of pointer chasing operations, contention nodes on the search paths, host-PIM and PIM-PIM communication cost, idle time on both host side and PIM side, and static granularity to partition the key space under any skewness. Jump-push samples and records previous search paths, bringing a considerable number of inter-PIM-module pointer chasing operations. PIM-tree exhibits worse performance for insert, delete, and

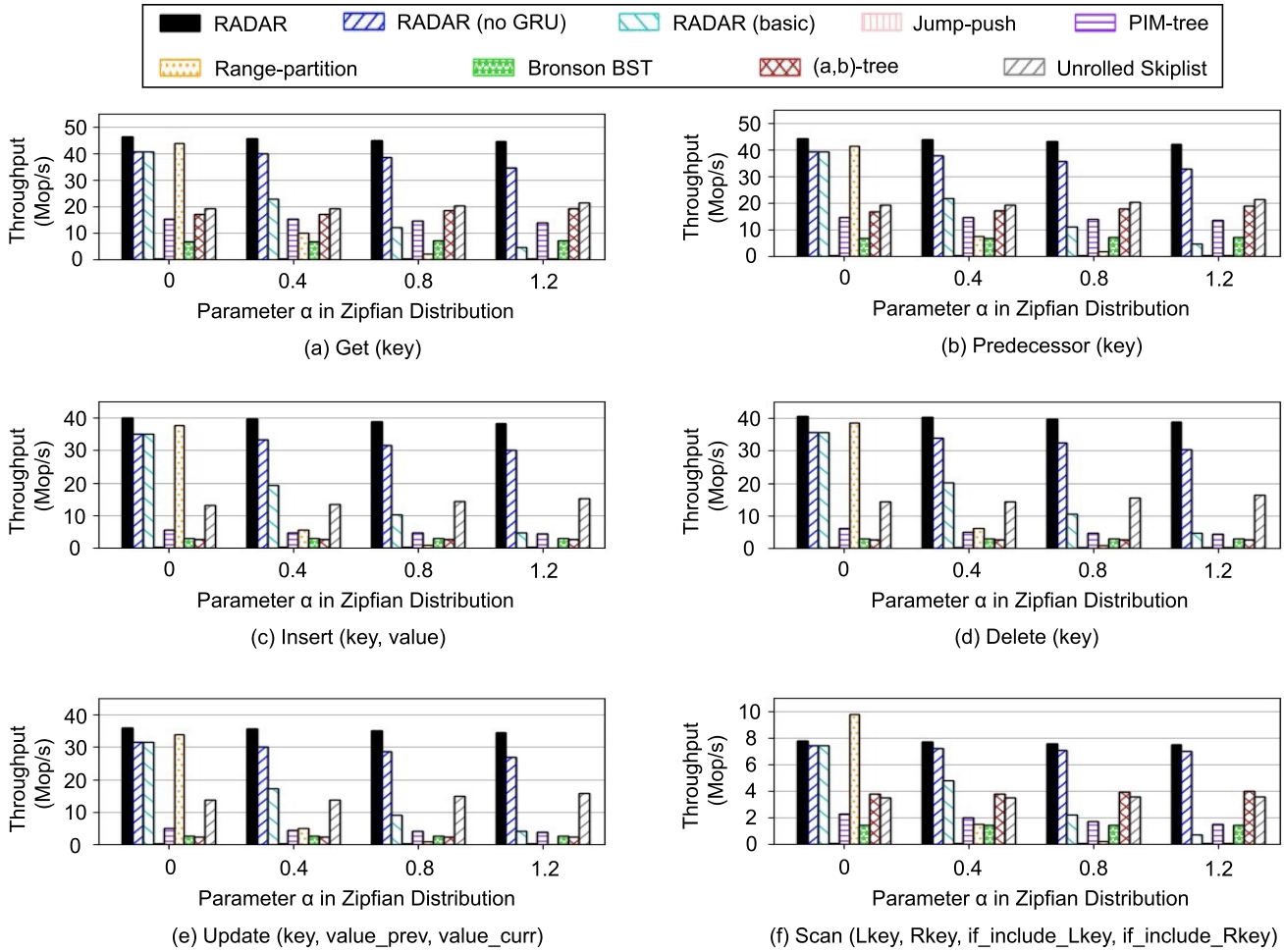


Fig. 10. Throughput of index operations.

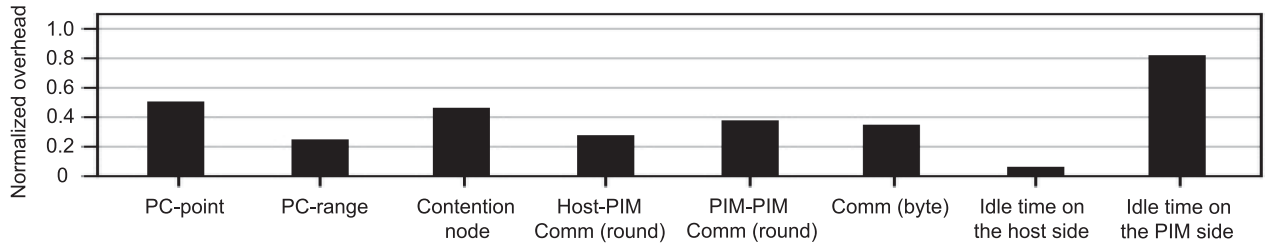


Fig. 11. Overheads mentioned in Section II-C of RADAR, normalized to PIM-tree.

update than query operations due to the cost of maintaining tree balance.

Overheads for skew resistance: Overheads mentioned in Section II-C of RADAR are normalized to PIM-tree (the latest skew-resistant PIM-friendly design) in Fig. 11. The average number of pointer chasing operations for each point (PC-point) and range (PC-range) operation is reduced by 50.7% and 75.4%, respectively. The pointer chasing reduction is due to the lower skiplist height, the ability to retrieve values of upper-layer keys without pointer chasing to lower layers, the adaptive heights of subrange nodes, and the enhanced node locality in PIM modules. The number of contention nodes pulled from the PIM side to the host side is 46.2% of that in PIM-tree, stemming from the integral search path guaranteed in one PIM module. The host-PIM (Host-PIM comm (round)) and PIM-PIM (PIM-PIM comm (round)) communication rounds are reduced by 72.9% and 62.5%, respectively. PIM-tree requires multiple host-PIM communication rounds to pull contention nodes, while RADAR only requires one round to retrieve values and pull nodes in skewed subranges. As for PIM-PIM comm (round), the sliced search path in PIM-tree requires batches of operations to be redispatched across PIM modules regardless of the skewness of workloads. RADAR requires zero rounds for uniformly random workloads and one round for skewed workloads to split

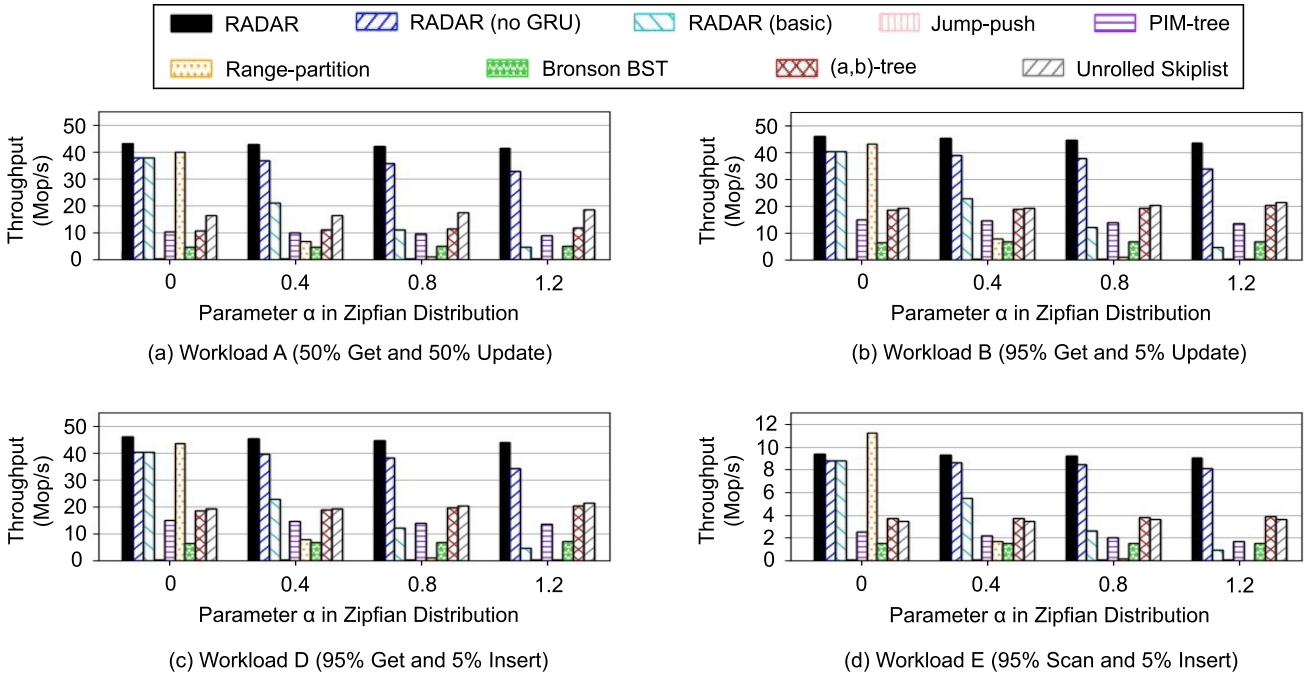


Fig. 12. Throughput of the YCSB workloads.

hot subranges. Since the communication between PIM modules in UPMEM must go through the host, we use the data transmitted between the host side and the PIM side in bytes (Comm (byte)) to measure the communication overhead, which is 65.3% lower than that of PIM-tree due to the reduction in redispatching index operations and pulling contention nodes. The idle time on the host side and the PIM side is reduced by 94.2% and 18.7% respectively, thanks to the host-PIM pipelining mechanism.

To sum up, under low skewness, RADAR and range partition designs significantly outperform node distribution designs due to the enhanced node locality in each PIM module and lower communication and pointer chasing overhead. Under high skewness, RADAR achieves the best performance while range partition designs perform the worst. The overhead for achieving skew-resistance in RADAR is much lower than that in state-of-the-art node distribution designs.

D. YCSB Benchmark

We evaluate all ordered indexes using five YCSB [3] workloads:

Workload A: Read-write-balance (50% Get and 50% Update);
Workload B: Read-intensive (95% Get and 5% Update);
Workload C: Read-only (100% Get);
Workload D: Read-latest (95% Get and 5% Insert);
Workload E: Short-ranges (95% Scan and 5% Insert).

We generate zipfian-skewed index operations and their throughput performance is shown in Fig. 12 (The performance of YCSB C is the same as that in Fig. 10(a)). RADAR achieves up to 193.7x higher throughput than other state-of-the-art designs. The results again confirm the fragility of range partition designs and the robustness of RADAR under skewed workloads, as well as more performance benefits achieved by RADAR compared with node distribution and traditional index designs.

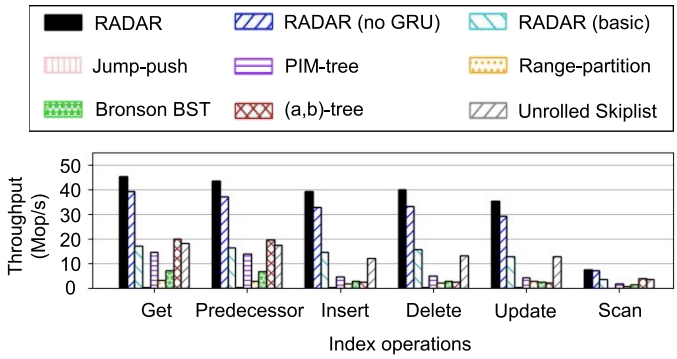


Fig. 13. Throughput of the wikipedia workload.

E. Workload of Real-World Skewness

We evaluate all indexes under a workload with real-world skewness using the publicly available wikipedia dataset [16]. The same as previous research [5], we extract words from each document, then use (word, document id) pairs as keys and random 8-byte integers as values to preserve the skewness of english words. Fig. 13 gives the performance of all ordered indexes under wikipedia workload. RADAR achieves the best performance (up to 182.6x and 12.9x than PIM-friendly and traditional ordered indexes, respectively) under the workload of real-world skewness.

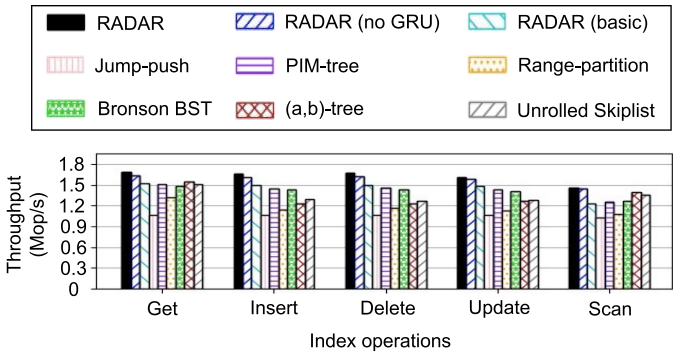


Fig. 14. Throughput of the wikipedia workload for end-to-end evaluation.

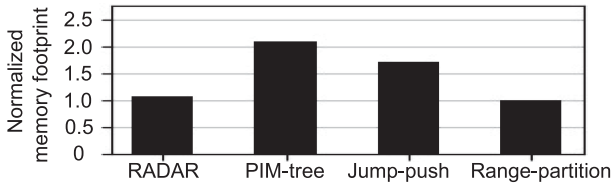


Fig. 15. Normalized memory footprint for PIM-friendly indexes.

F. End-to-End Evaluation

Redis [17] is a popular in-memory key-value store using a hash table as its index. We replace its internal index with our evaluated indexes for end-to-end evaluation. RADAR achieves 1.12x $\sim$ 1.58x higher throughput than other indexes under the wikipedia workload in Fig. 14. The performance gap among indexes becomes smaller due to the high server-client communication overhead.

G. Memory Footprint

Fig. 15 illustrates the memory footprint of state-of-the-art PIM-friendly indexes normalized to that of Range-partition, for running all benchmarks in Section VI. Range-partition consumes the least amount of memory only for storing skiplists in PIM modules. RADAR consumes 7% more memory than Range-partition due to the hash tables on the host side for varied-granularity subranges, as well as 34.3% and 47.4% less memory than Jump-push and PIM-tree respectively since both of them replicate upper-layer skiplist nodes in multiple PIM modules to achieve load balance under skewed workloads.

VII. Related Work

Processing-in-memory: To deal with the well-known memory wall [14], [42] problem, many researchers have proposed processing-in-memory (PIM) technologies [27], [28], [29], [30], [31], [32], [33], [34] over the past decades. By embedding processors in memory, PIM enables computations to be executed closer to memory. In recent few years, PIM products have entered the commercialization phase. UPMEM [18] is a commercially-available DDR4-PIM product. Programmable RISC cores are placed near every DRAM memory bank, resulting in a remarkable increase in total bandwidth. In addition, Samsung has proposed HBM-PIM products [33], designed to efficiently process memory-bound basic linear algebra subprograms (BLAS) that do not benefit from on-chip cache, such as scalar-vector, vector-vector, and matrix-vector operations. SK-Hynix has also developed their PIM product named AiM [34] based on GDDR6 memory, exhibiting significant potential in accelerating LSTM models.

PIM-friendly skiplist: The emergence of processing-in-memory technologies provide an opportunity to accelerate index operations in memory systems and databases. In general, prior works can be divided into two categories based on the methods for partitioning the key space and constructing ordered indexes: range partition designs [6], [7], [15] and node distribution designs [4], [5].

Range partition designs statically partition the entire key space into multiple coarse-grained key ranges with the same length. Each PIM module maintains a key range and constructs an individual skiplist. Liu et al. [15] propose a PIM-based concurrent skiplist supported both point index operations and range index operations. For processing an index operation for a key on the skiplist, the CPU on the host side first compares the key with the start key of each key range, to determine which PIM module the key belongs to. Then the index operation request is sent to that PIM module. After the PIM module retrieving the request, it executes the operation in the local vault and sends the result back to the host side. The process requires one communication round between the host side and the PIM side, without communication between PIM modules. The authors do not evaluate their proposed index on a real PIM hardware or a simulated PIM system. Choe et al. [6] implement Liu's [15] design using a full-system architecture simulator. They use benchmarks requesting for uniformly random keys to evaluate the PIM-based concurrent skiplist. The experimental results confirm that the range partition design achieves higher parallelism than traditional skiplist under uniformly random workloads. Based on Liu's [15] design, HybriDS [7] splits the skiplist into a host-managed portion consisting of upper-layer nodes and an PIM-managed portion consisting of the remaining lower-layer nodes. Since index operations access upper-layer nodes more frequently than lower-layer nodes, HybriDS better exploits cache locality on the host side. Although these range partition designs exhibit high parallelism under uniformly random workloads, they suffer from load imbalance under skewed workloads.

Node distribution designs randomly distribute skiplist nodes to PIM modules, preventing the execution of pointer chasing from top to bottom within only a subset of PIM modules under skewed workloads. Jump-push [4] is designed based on a key observation: once the search paths of keys l and r share a lower part node v, searching any key $u \in [l, r]$ will also reach node v. Thus, the search for u can directly start from the lowest common ancestor of these two paths. Jump-push adopts a multi-round sample search to record search paths. In each round, it doubles the sample size and uses the search paths recorded in previous rounds to decide start nodes of sample queries in this round.

However, the sampling incurs a significant number of inter-PIM-module pointer chasing operations to search for the lower part. Moreover, Jump-push requires recording entire search paths, burdening CPUs on the host side. PIM-tree [5] is designed based on Jump-push. For better load balance, upper-level nodes are replicated in multiple PIM modules. The data replication consumes more memory in PIM modules and brings synchronization overhead among PIM modules. A portion of nodes are chunked to enhance node locality. Frequently-requested skiplist nodes are pulled to the host side for better cache locality exploitation. Pulling these nodes requires additional communication overhead between the host side and the PIM side.

VIII. Conclusion

This paper presents RADAR, an ordered index that employs a novel hash-based subrange distributed structure in PIM systems to benefit from both load balance and enhanced node locality. The subrange granularity is dynamically adjustable to adapt to varying skewness during runtime. An offline machine learning model is applied to adjust the heights of skiplist nodes with hotness changes. In our evaluation, RADAR achieves up to 198.2x performance improvement and consumes 47.4% less memory than state-of-the-art designs in multiple datasets.

Acknowledgment

The authors would like to thank the anonymous editor and reviewers for their valuable feedback and insightful suggestions.

References

[1] T. Brown, "Techniques for constructing efficient lock-free data structures," 2017, *arXiv:1712.05406*.
[2] A. Fatima, S. Liu, K. Seemakhupt, R. Ausavarungnirun, and S. Khan, "vPIM: Efficient virtual address translation for scalable processing-in-memory architectures," in *Proc. IEEE 60th ACM Des. Autom. Conf.*, 2023, pp. 1–6.
[3] B. F. Cooper et al., "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[4] H. Kang et al., "The processing-in-memory model," in *Proc. 33rd ACM Symp. Parallelism Algorithms Architectures*, 2021, pp. 295–306.
[5] H. Kang et al., "PIM-tree: A skew-resistant index for processing-in-memory," in *Proc. VLDB Endowment*, vol. 16, no. 4, pp. 946–958, 2022.
[6] J. Choe et al., "Concurrent data structures with near-data-processing: An architecture-aware implementation," in *Proc. 31st ACM Symp. Parallelism Algorithms Architectures*, 2019, pp. 297–308.
[7] J. Choe et al., "HybriDS: Cache-conscious concurrent data structures for near-memory processing architectures," in *Proc. 34th ACM Symp. Parallelism Algorithms Architectures*, 2022, pp. 321–332.
[8] J. Park, S. Lee, and J. Lee, "NTT-PIM: Row-centric architecture and mapping for efficient number-theoretic transform on PIM," in *Proc. 60th ACM/IEEE Des. Automat. Conf.*, 2023, pp. 1–6.
[9] J. Zhang et al., "S3: A scalable in-memory skip-list index for key-value store," in *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2183–2194, 2019.
[10] K. Platz, N. Mittal, and S. Venkatesan, "Concurrent unrolled skiplist," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1579–1589.
[11] K. Yang et al., "Random walks on huge graphs at cache efficiency," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 311–326.
[12] N. G. Bronson et al., "A practical concurrent binary search tree," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 257–268, 2010.
[13] O. Mutlu et al., "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, Singapore: Springer Nature, 2022, pp. 171–243.
[14] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.
[15] Z. Liu et al., "Concurrent data structures for near-memory computing," in *Proc. 29th ACM Symp. Parallelism Algorithms Architectures*, 2017, pp. 235–245.
[16] Wikimedia Foundation, "Wikepedia: Database download," 2016. Accessed: Feb. 15, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Database_download
[17] Redis, "Redis," 2009. Accessed: Feb. 18, 2024. [Online]. Available: https://redis.io
[18] UPMEM, "UPMEM technology," 2023. Accessed Feb. 21, 2024. [Online]. Available: https://www.upmem.com/technology
[19] M. N. Vora, "Hadoop-HBase for large-scale data," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, 2011, pp. 601–605.
[20] A. Wahid and K. Kashyap, "Cassandra–A distributed database system: An overview," in *Proc. Emerg. Technol. Data Mining Inf. Secur.*, 2019, pp. 519–526.
[21] G. K. Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Dublin, Ireland: Ravenio Books, 2016.
[22] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 402–415.
[23] J. Hu, Z. Wei, J. Chen, and D. Feng, "RWORT: A read and write optimized radix tree for persistent memory," in *Proc. IEEE 41st Int. Conf. Comput. Des.*, 2023, pp. 194–197.
[24] G. W. Snedecor, "Curvilinear regression," in *Statistical Methods: Applied to Experiments in Agriculture and Biology*. Ames, IA, USA: Iowa State College Press, 1938, pp. 308–335.
[25] M. Ezekiel and K. A. Fox, "Methods of correlation and regression analysis: Linear and curvilinear," *J. of the Amer. Statal Assoc.*, vol. 123, no. 3, pp. 307–308, 1959.
[26] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (GRU) neural networks," in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst.*, 2017, pp. 1597–1600.
[27] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 105–117.
[28] B. Asgari, R. Hadidi, J. Cao, D. E. Shim, S.-K. Lim, and H. Kim, "FAFNIR: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architecture*, 2021, pp. 908–920.
[29] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.
[30] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 283–295.
[31] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 113–124.
[32] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and efficient neural network acceleration with 3D memory," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 751–764.
[33] Y.-C. Kwon et al., "25.4 A 20 nm 6Gb function-in-memory DRAM, based on HBM2 with a 1.2T flops programmable computing unit using bank-level parallelism, for machine learning applications," in *Proc. IEEE Int. Solid-State Circuits Conf.*, San Francisco, CA, USA, 2021, pp. 350–352.
[34] Y. Kwon et al., "System architecture and software stack for GDDR6-AiM," in *Proc. IEEE Hot Chips 34 Symp.*, 2022, pp. 1–25.
[35] J. Chen et al., "The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database," in *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1401–1412, 2016.
[36] C. Huang et al., "RS-store: A skiplist-based key-value store with remote direct memory access," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, Cham: Springer International Publishing, 2020, pp. 314–323.
[37] S. Dong et al., "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–32, 2021.
[38] J. Yeon et al., "JellyFish: A fast skip list with MVCC," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 134–148.
[39] W. Kim et al., "ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 161–177.

[40] O. Balmau et al., "Unlocking memory in persistent key-value stores," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 80–94.
[41] I. Dick, A. Fekete, and V. Gramoli, "A skip list for multicore," *Concurrency Comput. Pract. Exp.*, vol. 29, no. 4, 2017, Art. no. e3876.
[42] S. A. McKee, "Reflections on the memory wall," in *Proc. 1st Conf. Comput. Front.*, 2004, Art. no. 162.
[43] G. Weisz et al., "A study of pointer-chasing performance on shared-memory processor-FPGA systems," in *Proc. ACM/SIGDA Int. Symp. Field- Program. Gate Arrays*, 2016, pp. 264–273.
[44] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 8792–8802.
[45] S. Assadi, G. Kol, R. R. Saxena, and H. Yu, "Multi-pass graph streaming lower bounds for cycle counting, max-cut, matching size, and other problems," in *Proc. IEEE 61st Annu. Symp. Found. Comput. Sci.*, 2020, pp. 354–364.
[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[47] H. Kang et al., "PIM-trie: A Skew-resistant Trie for Processing-in-Memory," in *Proc. 35th ACM Symp. Parallelism Algorithms Architectures*, 2023, pp. 1–14.
[48] X. Q. Li, G. M. Tan, and N. H. Sun, "PIM-Align: A processing-in-memory architecture for FM-Index search algorithm," *J. Comput. Sci. Technol.*, vol. 36, pp. 56–70, 2021.
[49] S. Hosseinzadeh, A. Parvaresh, and D. Fey, "Optimization of OLAP in-memory database management systems with processing-in-memory architecture," in *Proc. Int. Conf. Archit. Comput. Syst.*, Cham, Switzerland: Springer Nature, 2023, pp. 264–278.
[50] M. Snir and J. Yu, "On the theory of spatial and temporal locality," Comput. Sci. Dept., Univ. of Illinois at Urbana-Champaign, Tec. Rep. DCS-R-2005-2564, 2005.
[51] J. B. Camiña, R. Monroy, L. A. Trejo, and M. A. Medina-Pérez, "Temporal and spatial locality: An abstraction for masquerade detection," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 9, pp. 2036–2051, Sep. 2016.
[52] Y. Sui, G. Wang, L. Zhang, and M.-H. Yang, "Exploiting spatial-temporal locality of tracking via structured dictionary learning," *IEEE Trans. Image Process.*, vol. 27, no. 3, pp. 1282–1296, Mar. 2018.
[53] Y. Hua, S. Zheng, J. Yin, W. Chen, and L. Huang, "Bumblebee: A MemCache design for die-stacked and off-chip heterogeneous memory systems," in *Proc. 60th ACM/IEEE Des. Autom. Conf.*, 2023, pp. 1–6.
[54] F. Montesano, R. Marotta, and F. Quaglia, "Spatial/temporal locality-based load-sharing in speculative discrete event simulation on multi-core machines," in *Proc. ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2022, pp. 81–92.
[55] B. S. Gill and D. S. Modha, "WOW: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches," in *Proc. Proc. 4th Conf. USENIX Conf. File Storage Technol.*, 2005, Art. no. 10.
[56] A. A. Prihozhy and O. N. Karasik, "Inference of shortest path algorithms with spatial and temporal locality for Big Data processing," in *Proc. Big Data and Adv. Analytics: Proc. of VIII Int. Conf.*, 2022, pp. 56–66.

Yifan Hua (Student Member, IEEE) is currently working toward the PhD degree with Shanghai Jiao Tong University, China. His research interests include non-volatile memory systems, processing-in-memory systems, and hybrid memory management.

Shengan Zheng received the BS and PhD degrees from Shanghai Jiao Tong University, in 2014 and 2019, respectively. He is currently an assistant professor with Shanghai Jiao Tong University. His research interests include memory systems, storage systems, and distributed systems.

Weihan Kong is currently working toward the PhD degree with Shanghai Jiao Tong University. His research interests include hybrid memory systems and processing-in-memory systems.

Cong Zhou is currently working toward the PhD degree with Shanghai Jiao Tong University, China. His research interests include in-memory computing and distributed memory systems.

Kaixin Huang received the PhD degree from Shanghai Jiao Tong University, in 2021. His research interests include non-volatile memory management and distributed systems.

Ruoyan Ma is currently working toward the master's degree with Shanghai Jiao Tong University. His research interests include machine learning system and near data processing.

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