PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-in-Memory

Yiwei Zhao
Carnegie Mellon University
Pittsburgh, USA
yiweiz3@andrew.cmu.edu

Hongbo Kang
Tsinghua University
Beijing, China
khb20@mails.tsinghua.edu.cn

Ziyang Men
University of California, Riverside
Riverside, USA
zmen002@ucr.edu

Yan Gu
University of California, Riverside
Riverside, USA
ygu@cs.ucr.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, USA
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland
College Park, USA
laxman@umd.edu

Charles McGuffey
Reed College
Portland, USA
cmcguffey@reed.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, USA
gibbons@cs.cmu.edu

Abstract

Space-partitioning indexes are widely used for managing multi-dimensional data, but their throughput is often memory-bottlenecked. Processing-in-memory (PIM), an emerging architectural paradigm, mitigates memory bottlenecks by embedding processing cores directly within memory modules, allowing computation to be offloaded to these PIM cores.

In this paper, we present PIM-zd-tree, the first space-partitioning index specifically designed for real-world PIM systems. PIM-zd-tree employs a tunable multi-layer structure, with each layer adopting distinct data layouts, partitioning schemes, and caching strategies. Its design is theoretically grounded to achieve load balance, minimal memory-channel communication, and low space overhead. To bridge theory and practice, we incorporate implementation techniques such as practical chunking and lazy counters. Evaluation on a real-world PIM system shows that PIM-zd-tree's throughput is up to 4.25× and 99× higher than two state-of-the-art shared-memory baselines.

CCS Concepts: • **Theory of computation** → **Parallel algorithms**; **Distributed algorithms**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; *Parallel architectures.*

Keywords: space-partitioning index, processing-in-memory, near-data-processing, nearest neighbor search

ACM Reference Format:
Yiwei Zhao, Hongbo Kang, Ziyang Men, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2026. PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-in-Memory. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3774934.3786411

1 Introduction

Spatial indexes for managing multi-dimensional data points are fundamental, with broad applications in computational geometry [23, 46, 68, 73, 89], AI/ML [11, 34, 69, 80, 99], graphics [26, 39, 51, 59], radars and robotics [70, 71, 91, 94], and scientific simulations [21, 40, 45, 74]. Among these, some of the most well-known spatial indexes, such as kd-trees [8] and quad/octrees [75], are constructed by recursively partitioning the multidimensional space of data points—hence referred to as space-partitioning indexes. These structures support a variety of queries, including point searches, orthogonal range queries, and k-nearest neighbor (kNN) searches.

With the increasing amount of data in recent decades, space-partitioning indexes have become increasingly constrained by the high cost of memory access. Compared to on-chip computation, accessing off-chip memory is orders of magnitude slower and is often bottlenecked by the limited bandwidth attainable over off-chip memory channels (often referred to as the *memory wall* problem). *Processing-in-memory (PIM)*, a.k.a. *near-data-processing*, has recently gained attention as a compelling architectural paradigm to overcome the memory wall problem. By adding computational units (*PIM cores*) near or within memory modules, PIM enables computation to occur close to its data, in contrast to the traditional von Neumann architecture where any data must first be transferred to the CPU over off-chip memory channels. In bank-level in-memory processing (*BLIMP*) PIM designs, PIM cores are integrated directly into memory banks, enabling computation to be executed on *PIM modules* (PIM core and its local memory). This design improves both performance and energy efficiency by leveraging low-latency, low-energy on-chip memory accesses, while also exploiting memory bandwidth and computational resources that scale with the number of PIM modules. Consequently, off-chip communication can be substantially reduced.

Prior work on designing space-partitioning indexes for PIM (e.g., [20, 56, 88, 96]) has been limited. These works can be divided into two groups. The first group [20, 56, 88] are (*i*) evaluated only on simulators, not real-world PIM systems, and (*ii*) lack theoretical guarantees, which limits their ability to capture the fundamental characteristics of PIM systems. The second group [96] focuses exclusively on asymptotic theoretical analysis, with constant-factor costs and amortization overheads that will likely render such designs inefficient in practice (see §2.2 for discussion).

In this paper, the central question we aim to address is: How can a space-partitioning index be efficiently implemented on real-world PIM systems? At the same time, given the considerable diversity across existing PIM architectures, we also seek to answer the question: Can such an index be designed in a theoretically-grounded manner that captures the fundamental characteristics of PIM architectures, so that it may remain effective for future PIM systems?

To this end, we set out to implement on PIM systems a state-of-the-art shared-memory space-partitioning index with strong theoretical guarantees. There are two main variants: the kd-trees [8] that are usually based on object-median partitioning, and quad/octree that are based on spatial-median partitioning [75]. Their state-of-the-art shared-memory parallel designs are Pkd-trees [62, 63] and zd-trees [12, 61], respectively. Specifically, zd-trees are built by space-filling curves (Morton order curves in Fig. 1).

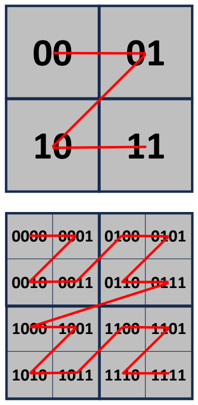


Figure 1. z-order

In this work, we consider adapting zd-trees on PIM, and present ***PIM-zd-tree***, the first space-partitioning index deployed (and shown to be efficient) on a *real-world* PIM system. We select the zd-tree for a few reasons. First, zd-trees are based on spatial-median partitioning, which is simpler in practice and requires no rebalancing; both are critical in achieving practical efficiency in PIM systems. Zd-trees are also deterministic, in that the structure is independent of the order of data point insertions (a.k.a. *history-independent*). This determinism simplifies programming and debugging both for developing and using the index.

Achieving an efficient PIM-based spatial index requires addressing two fundamental challenges:

(**Q1**) How can we achieve a good ***trade-off*** between PIM load balance, reduced off-chip communication, and low space consumption? Because PIM systems commonly operate in bulk-synchronous parallel (BSP) rounds [90], it is critical to avoid stragglers, which determine round completion time. Achieving such balance even under high workload skew, however, is particularly difficult. It often necessitates either (*i*) partitioning tasks and data at extremely fine granularity, which increases off-chip traffic, or (*ii*) replicating data across multiple PIM modules, which incurs additional space overhead and update costs [48, 50].

(**Q2**) How can we efficiently bridge theoretical designs and practice? Asymptotic analyses often overlook constant factors and amortization overheads, which can introduce significant inefficiencies in real-world deployments. Hence, we require implementation techniques that both preserve theoretical bounds and deliver high performance in practice.

To address (Q1), PIM-zd-tree divides the tree into three layers based on the properties of nodes, where each layer has its own strategy for data partitioning, placement and lightweight sharing (caching). These strategies reduce off-chip communication while guaranteeing low update and space overheads. Furthermore, PIM-zd-tree is designed to be user-tunable, allowing it to support different levels of skew tolerance and varying communication and space requirements by adjusting the layer division and data management strategy correspondingly. To address (Q2) and bridge the gap between theory and practice, PIM-zd-tree incorporates a set of implementation techniques (§6) that translate theoretical insights in §5 into practical efficiency (§7).

We implement PIM-zd-tree on UPMEM [72], a real-world BLIMP-based PIM system. PIM-zd-tree achieves up to 4.25× and 99× speedup over Pkd-tree [63] and zd-tree [12], two state-of-the-art non-PIM baselines and reduces memory-channel traffic by an average of 3.5× and 18.8×.

In summary, the main contributions of this paper are:

- We design PIM-zd-tree, a tunable spatial index for PIM that adapts to varying requirements in skew tolerance, communication, and space overheads, with strong theoretical grounding.
- We adopt implementation techniques that effectively translate theoretical efficiency into practical performance, leveraging fundamental characteristics of BLIMP.
- We present the first implementation and evaluation of a space-partitioning index on a real-world PIM system, demonstrating significant performance gains for emerging PIM systems over traditional systems.

Our code is available at https://github.com/cmuparlay/PIM-zd-tree.

2 Background

2.1 PIM Architecture and Computation Model

PIM Model. In this paper, we use the Processing-In-Memory (PIM) Model [47] for theoretical analysis. Experimental results from prior works [48, 50] show that the PIM Model is a good representation of a bank-level-in-memory-processing (BLIMP) system, which is commonly used in commercial real-world PIM systems like UPMEM [72] and Samsung PIMs [76].

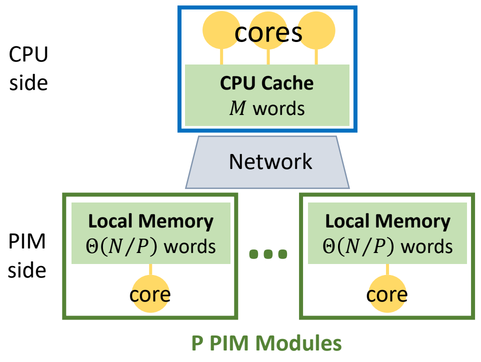


Figure 2. The Processing-in-Memory (PIM) Model [47].

The PIM Model, as shown in Fig. 2, consists of a host CPU and a PIM side of P PIM modules. The CPU features a standard multicore architecture with an L3 cache of size M words. Each *PIM module* integrates a small on-chip *local memory* or *PIM memory* of $\Theta(N/P)$ words (where N denotes the total space), and a general-purpose but relatively weak processor known as the *PIM core*. The host CPU can access its cache and all PIM memory. However, each PIM core can only access its own local memory. PIM modules cannot communicate directly and must exchange data via the CPU. Programs execute in bulk-synchronous parallel (BSP) rounds [90].

The PIM Model integrates both shared-memory and distributed metrics. For CPU computations, it quantifies the ***CPU work*** (the total number of instructions executed by the CPU) and ***CPU span*** (the critical path length) under a binary forking model [4, 13, 14]. For off-chip communication, it measures ***communication amount*** which is the sum total of words sent between the CPU and all PIM modules. For PIM programs, it measures the ***PIM time***, the *maximum* work on any PIM core within a round. Because PIM time is based on the maximum across all PIM modules, it is crucial to design algorithms that ensure good load balance across PIM modules, even under highly-skewed workloads.

A Real-World System: UPMEM. We evaluate our techniques on the latest PIM system from UPMEM [72] (recently acquired by Qualcomm). UPMEM's PIM DIMMs are plug-and-play DRAM DIMM replacements; thus, UPMEM can be configured with various ratios of traditional DRAM to PIM-equipped DRAM (the current maximum available configuration has 2560 PIM modules). The CPU has access to the traditional DRAM and all the PIM memory, but each PIM core only has access to its local memory. Each PIM module has up to 628 MB/s local DRAM bandwidth, so a machine with 2560 PIM modules can provide up to 1.6 TB/s aggregate bandwidth [37]. To move data *between* PIM modules, the CPU reads from the origin and writes to the target.

UPMEM's main memory (traditional DRAM) enables running programs that overflow the CPU's L3 cache, but these additional memory accesses bring another type of communication (not in the PIM Model): *CPU-DRAM communication.* Thus L3 cache efficiency is important for host programs.

Applications on PIM. Though the idea of PIM dates back to the 1970s [86], it has regained attention recently, due to the development in 3D-stack memory fabrication [44] and the release of real-world PIM products [42, 72, 76]. Hundreds of academic works have been published (see the references of [6, 66]). PIM systems have been widely used in accelerating applications of databases [9, 22, 48, 54, 58], machine learning [15, 16, 41, 95, 97], graph processing [18, 55, 81, 87], genome analysis [19, 28, 31, 64] and security [3, 30, 33, 36].

2.2 Prior Work: PIM-Friendly Indexes

Space-Partitioning Indexes on PIM. Most prior works [20, 56, 88] are evaluated only on simulators, not real-world PIM systems, and lack theoretical foundations. The only work with theoretical guarantees [96] relies on periodic reconstruction of imbalanced subtrees as a core approach. However, this approach is fundamentally impractical on real systems, as its additional round complexity incurs substantial latency from *mux switch overheads* [54] in current PIM architectures.

Range-Partitioning Indexes. Early PIM-based indexes [24, 25, 60] adopt range-partitioning approaches, where the key space is divided into disjoint ranges, each stored on a PIM module. While such designs are effective in reducing communication, they are highly *vulnerable* to workload skew.

Skew-Resistant Indexes. More recent skew-resistant indexes [47–50, 96] employ finer-grained data placement and replication strategies to mitigate skew. However, most of these designs are purely theoretical [47, 49, 96] and lack practical validation. PIM-tree [48, 50] represents an implementation effort, but (*i*) its design cannot be directly extended to spatial indexes due to fundamental structural differences, and (*ii*) it sacrifices performance in non-skewed workloads in order to guarantee good performance even under skew.

2.3 zd-Tree

The zd-Tree [12, 61], the primary data structure in this work, is a space-partitioning index of n multi-dimensional points. In a nutshell, it is a kd-tree whose splitting rule uses z-order (see Fig.1). The tree is built by letting the root represent the entire bounding box of the dataset, and splitting the points into child nodes at level i based on whether the bit at place i of the z-order key is 0 or 1. Both internal nodes and leaf nodes store information about their bounding box. Internal nodes also store their children, while leaf nodes store the points they contain. The number of points in a leaf is bounded by a constant, and every point is included in exactly one leaf.

From one perspective, the zd-tree is similar to an oct-tree (in 3 dimensions), except that every three levels of a zd-tree corresponds to one level of an oct-tree. From another perspective, the zd-tree can be viewed as a radix tree (or trie) whose stored keys are the z-ordered integer of the points. We adopt an implementation of a *compressed* radix tree, where we (*i*) omit all empty leaves, and (*ii*) merge each node that

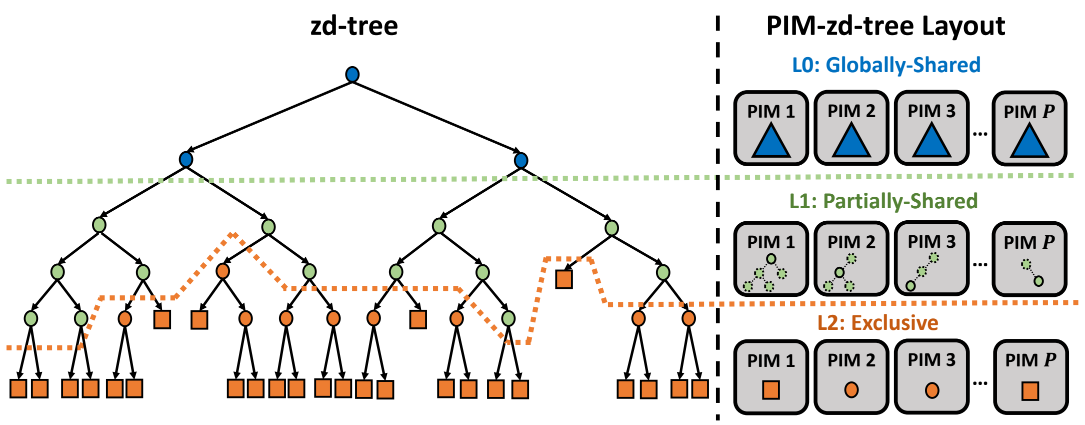


Figure 3. PIM-zd-tree with its three-layer structure and corresponding data layout across PIM modules. Squares denote leaves, while circles represent internal nodes. Solid-frame nodes indicate master storage, whereas dashed-frame nodes represent cached data used in data sharing.

has only one child with that child (i.e., we compress all paths consisting of nodes with only one child). Note that after this compression, all internal nodes have exactly two children, and there are $2n + O(1)$ nodes in total in the zd-tree.

The zd-tree supports correct operations on arbitrary multi-dimensional datasets and has demonstrated practical efficiency [12, 61] on numerous real-world datasets. It can also achieve the theoretical cost bounds detailed in Lemma 2.1.

Lemma 2.1 (zd-Tree Properties [12]). *Given a zd-tree storing n points in a set P with bounded ratio (Defn. 1, §5) and bounded expansion constant γ (Defn. 2, §5), it can be proved that: (i) The height of the tree is $O(\log n)$. (ii) The tree can be built using $O(n)$ work and $O(n^\epsilon)$ span, for a constant $\epsilon < 1$. (iii) Finding the k-nearest neighbors of a point $p \in P$ requires expected $O(k \log k)$ work. (iv) Inserting k nodes into the tree requires $O(k \log(1 + n/k))$ work and $O(k^\epsilon + polylog(n))$ span.*

3 PIM-zd-tree

In this section, we introduce PIM-zd-tree—a batch-dynamic zd-tree data structure designed for Processing-In-Memory (PIM). PIM-zd-tree maintains a binary zd-tree and this section introduces the main techniques for data partitioning and replication used in our design.

A straightforward design to place a zd-tree on P PIM modules is to partition the tree into P disjoint subtrees with equal sizes, and place each tree on a different PIM module. However, such design is highly sensitive to workload skew—in the worst case, all operations in a batch target the tree on one PIM module and leave all the others idle.

To address this challenge, our PIM-zd-tree initially distributes each tree node across PIM modules using a hash-based randomization strategy, ensuring that even adversarial operations cannot consistently target the same node. We refer to these distributed nodes as *master nodes*. However, relying solely on master nodes does not reduce off-chip communication: during searches, every tree edge incurs a remote access because parent and child nodes are typically placed on different PIM modules. As a result, the communication cost remains comparable to that of shared-memory systems, undermining the motivation for adopting PIM.

In this section, we introduce our main design to *reduce communication* over this naïve master node design, without violating load balance or incurring a large space cost.

3.1 Overall Structure

The PIM-zd-tree divides the data structure into three layers. As shown in Fig. 3, from top (root) to bottom (leaf nodes), the tree is divided into (*i*) Level 0 (L0): globally-shared nodes; (*ii*) Level 1 (L1): partially-shared nodes; and (*iii*) Level 2 (L2): exclusive nodes. In each layer, a different strategy is adopted for data partitioning and caching.

Our key observation in a tree data structure is that the internal tree nodes that lie in the upper part of a tree are more frequently accessed in (top-down) searches and less frequently modified in dynamic updates compared to the nodes in lower levels. Meanwhile, the number of such nodes (the size of the upper part) is relatively small compared with the lower levels. Thus, sharing a consistent copy of the upper part nodes across different hardware modules (CPU and/or PIM modules) would be beneficial without incurring unacceptable overheads in space and update costs. Intuitively, the higher the position of a node is inside the tree, the more times it should be replicated on different modules.

Positional Descriptor of a Tree Node. We first introduce how to divide each node into its corresponding layer based on its positional information. In this paper, we use the notion of ***subtree size***—the total number of multi-dimensional data points contained in all the descendant leaf nodes of a node, denoted as $T(N_i)$ for internal node N_i—to represent the positional information of an internal node. Unlike B-trees, zd-trees are not strictly balanced. Thus, the *height* representation of each node (as in the PIM-tree [48]) might be an imprecise indicator of the position of this node inside the zd-tree. For instance, even for a dataset with a bounded expansion constant (Defn. 2), a node with $\log n$ depth from the root can either be a leaf node or a node with $\Theta(\sqrt{n})$ descendants. In contrast, the subtree size is a more accurate descriptor on the position of a node, where nodes with larger subtree sizes lie in the higher parts in the tree.

The PIM-zd-tree uses two tunable thresholds, θ_{L0} and θ_{L1} ($\theta_{L0} \geq \theta_{L1} > 0$), to control the tree structure. For all nodes N_i with $T(N_i) \geq \theta_{L0}$, they are categorized as L0 nodes. Any node with $T(N_i) < \theta_{L1}$ is categorized as an L2 node. The rest are categorized as L1 nodes.

Globally-Shared Nodes (L0). For the nodes in the uppermost part of the tree, their information and the tree structure will be shared globally across *all* the PIM modules. Given such storage, the concept of master nodes is unnecessary. The size of L0 can be tuned by the designer or the user.

When the L0 nodes all fit in the CPU cache, L0 will be maintained completely in the cache. Because the query and update workload starts from the host CPU side, keeping the L0 structure in CPU cache is equivalent to sharing the structure over all PIM modules.

On the other hand, when the size of L0 exceeds the CPU cache, its structure will be replicated over all PIM modules. To keep overhead of space and updates low, the size of L0 should be bounded by selecting an appropriate θ_{L0} value.

Partially-Shared Nodes (L1). Each tree node in L1 will have their master node stored on a random PIM module. The tree structure of other nodes in L1 will be shared and attached to the master storage as an auxiliary structure to reduce communication in tree traversal. However, due to the large number of L1 nodes, a full global data sharing of the entire L1 structure over all PIM modules will be too costly in terms of update cost and space overhead.

In PIM-zd-tree, for each of the L1 nodes, a copy of all its ancestors and descendants (and the corresponding tree structure) in L1 will be attached to the master storage and stored on the same PIM module. This information suffices to cover all traversal paths of a search query that is passing through this L1 node, and thus the subsequent search query could be executed locally through these cached tree structures.

Exclusive Nodes (L2). For internal and/or leaf nodes with $T(N_i) < \theta_{L1}$, due to their large total number, high frequency in updates and unlikeliness in facilitating search queries, a PIM-zd-tree does not make replicas of these nodes and only stores their master copies. Each leaf node is allowed to hold a maximum of B data points (defined later in §3.2).

Promotion and Demotion. PIM-zd-tree assumes that structural changes to the tree occur only through dynamic updates (insertions and deletions). Such updates affect the subtree sizes of all internal nodes along the path from the root to the modified leaf node. Roughly, if the updated size of an internal node causes its transition between categories (L0/L1/L2), the node is *promoted* or *demoted* accordingly, and its associated caching for data sharing is adjusted to reflect the change.

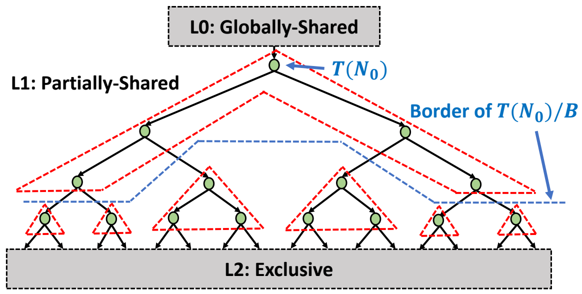


Figure 4. Example of dividing an L1 subtree into meta-nodes. Each region enclosed by a red dashed line corresponds to a meta-node. The L1 root has subtree size $T(N_0)$, while the blue dashed line marks the boundary below which all nodes satisfy $T(N_i) < T(N_0)/B$. Recursive division would be applied if any L1 nodes further satisfy $T(N_i) < T(N_0)/B^2$, but this step is omitted for clarity.

3.2 Chunking for Imbalanced Trees

Chunking (or blocking) is a widely adopted technique in locality-aware data structures, such as transforming binary search trees into B-trees. However, traditional chunking methods that rely on adjusting fanout (as in B-trees) are incompatible with imbalanced trees like zd-trees, since the fanout approach fundamentally assumes that tree height and level are well-defined and meaningful.

PIM-zd-tree employs a chunking strategy for its L1 and L2 layers that is based entirely on subtree sizes, guided by a user-defined chunking factor B. For a highest node N_i in L1 or L2, all descendants N_j satisfying $T(N_j) \geq T(N_i)/B$ are grouped into the same chunk as N_i. Each chunk, which naturally forms an imbalanced tree, is referred to as a ***meta-node***. This chunking process is then applied recursively to the highest unchunked nodes until every node is contained within some meta-node. An example is shown in Fig. 4.

All nodes within the same meta-node are placed on a single PIM module, and any internal caching for data sharing among them is now eliminated. The original zd-tree is thus restructured into a higher-level tree composed of meta-nodes. For nodes in L1, remote caching to enable data sharing across different meta-nodes is preserved, but maintained at the granularity of meta-nodes rather than individual nodes.

3.3 Push-Pull Search

We use *push-pull search* [48–50, 96, 98] as a core technique in PIM-zd-tree to achieve load balance when accessing L1 and L2 nodes under skewed workloads. Unlike distributed systems, PIM architectures feature a powerful host CPU, enabling the use of shared-memory techniques. Push-pull search exploits this capability by flexibly coordinating computation between the host CPU and the PIM modules, in contrast to prior distributed algorithms that rely solely on offloading computation.

Push-pull search uses contention information within a batch to decide whether computation in a PIM-zd-tree is performed on the CPU side or on PIM modules. To illustrate, consider a top-down Search query from root to leaf. In a PIM execution round with batch size S, suppose m queries need to access an L1 meta-node M_i to determine which descendant subtree to follow. If m is below a load-imbalance threshold (defined later), all m queries are *pushed* to the PIM module storing M_i, where the search proceeds using the local caching of its descendant subtrees until it reaches the L1-L2 border. Conversely, if m exceeds the threshold, then M_i will be *pulled* from the PIM side into the CPU, where a parallel search is performed. Notably, only the master storage of the meta-node is fetched; caching of its descendant meta-nodes is excluded to prevent communication imbalance. After the CPU search, the m queries are partitioned according to the destination descendants from their meta-node traversal, and the push-pull decision is recursively applied level by level at the granularity of meta-nodes until the leaves are reached.

3.4 Lazy Counters

As noted, subtree size is a key component in the design of the PIM-zd-tree. However, maintaining an accurate and consistent version of this metric across all nodes is challenging. A straightforward approach would involve storing precise counters on every node and its replicas, while ensuring consistency during dynamic updates. But the overall strategy of the PIM-zd-tree is to *replicate higher-level nodes more extensively*. As a result, changes in subtree counters propagate toward the upper levels of the tree, making it prohibitively expensive to maintain strict consistency during updates.

A key observation is that a provably small degree of approximation is acceptable in these counters. Prior work has explored the design of randomized counters [65, 83–85, 93, 96]. However, employing randomization in practice on PIM systems can lead to irregular and unpredictable execution flows, in addition to the overhead of random number generation or hash computations on lightweight PIM cores.

To resolve this challenge, PIM-zd-tree adopts ***lazy counters***, which in each node maintains a slightly out-of-date global snapshot of the subtree sizes. This snapshot is within a degree of approximation to ensure algorithm correctness, is replicated across caching, and is infrequently updated.

Specifically, all nodes record changes in their subtree sizes during dynamic updates. Changes are propagated to parent nodes and global snapshots are synchronized across all replicas on other PIM modules only when the change exceeds a threshold Δ, $-T(N_i)/2 < \Delta < T(N_i)$. With suitable choices of Δs in each layer as detailed in Table 1, lazy counters achieve sufficient accuracy as formalized in Lemma 3.1.

Lemma 3.1 (Lazy Counter Value)**.** *In PIM-zd-tree, the value of a global snapshot counter $SC(N_i)$ on any node N_i always satisfies $T(N_i)/2 \le SC(N_i) \le 2T(N_i)$.*

Proof Sketch. Due to space limitations, we present only a proof sketch for the insertion-only case, where $T(N_i)/2 \le SC(N_i) \le T(N_i)$. The cases with deletions and $T(N_i) \le SC(N_i) \le 2T(N_i)$ are symmetric.

In the insertion-only setting, it is immediate that $SC(N_i) \le T(N_i)$, so we show $T(N_i)/2 \le SC(N_i)$. Any L0 node has $T(N_i) \ge \theta_{L0}$, and since the unfinished updates are fewer than θ_{L0}, it follows that $T(N_i)/2 \le SC(N_i)$. Similarly, any L1 global snapshot deviates from the true subtree size by at most a factor of 0.5. □

Table 1. Lazy counter configurations in each level.

Layer	$\Delta_{\min}$	$\Delta_{\max}$
L0	$-\theta_{L0}/2$	θ_{L0}
L1	$-0.5\min\left\{\theta_{L1}, \log_B \frac{\theta_{L0}}{\theta_{L1}}\right\}$	$\min\left\{\theta_{L1}, \log_B \frac{\theta_{L0}}{\theta_{L1}}\right\}$
L2	0	0

4 Operations

We describe here our algorithms for implementing PIM-zd-tree and various queries over them. The theoretical analysis on the cost of these algorithms will be provided in §5.

4.1 Top-Down Search

Top-down Search queries (Alg. 1) try to locate the leaf node where a given data point lies in. This can be used as a preprocessing for dynamic updates and kNN queries. The search answers a batch Q of queries by traversing L0, L1, and L2, using push-pull search (for L1 and L2) and local caching.

Algorithm 1. Search (Q: batch of query points)

```
1. [L0] Traverse L0 to search $Q$ by (1) searching in CPU cache; or
   (2) dividing $Q$ into $P$ groups, each searched on a PIM module.
2. [L1 Pull] While the number of queries that will be sent to each
   PIM module for L1 is imbalanced (i.e., the busiest module gets
   more than 3× the average load), do:
   a. Pull all meta-nodes with more than $K = B\log_B \frac{\theta_{L0}}{\theta_{L1}}$ queries
      back to the CPU.
   b. Search through the meta-nodes on the CPU.
3. [L1 Push] Push load-balanced queries in $Q$ to the PIM modules
   holding their L1 nodes, and traverse L1 using local caching.
4. [L2 Push-Pull] For each level in L2, perform one push-pull
   round: Pull the L2 nodes with more than $K = B$ queries to the
   CPU and search in the CPU cache; otherwise, search on PIM.
   Perform such rounds until reaching the bottom level and re-
   trieve the corresponding leaf nodes.
```

4.2 Dynamic Updates

Insert adds new data points to PIM-zd-tree, while Delete removes existing data points from the structure. Here we only provide Insert in Alg. 2 due to space constraints.

Step 3c proceeds as follows. Since the search trace is recorded on the CPU, the update occurs in two rounds. In the first round, the CPU allocates space in the cache to reserve memory for all remote-node modifications and fixes all local parent-child links. In the second round, once these cache reservations are in place, the system updates all remote parent-child links to reflect the structural changes.

For Step 3d, promoting L2 nodes to L1 is straightforward: the CPU modifies the corresponding cache entries of their L1 ancestors, as in Step 3c. Promoting L1 nodes to L0, however, involves additional coordination. Each PIM module maintains a buffer of L1 nodes scheduled for promotion. In the first round, PIM modules send the promotion candidates to the CPU and garbage-collect the local cache entries of these nodes. In the second round, the CPU broadcasts each promoted node to the fully replicated L0 across all PIM modules, and the related remote parent-child links are constructed.

Algorithm 2. INSERT(Q: batch of inserted points)

```
1. SEARCH (Q) and record the search traces on CPU. (A *search trace* is the on-PIM addresses of the starting and ending nodes along each top-down search path within every replica.)
2. In one communication round, for each key to insert:
    a. Create a new leaf node if SEARCH ends at an empty child.
    b. If SEARCH ends in an existing leaf, return the address if the leaf node is not full after insertion. Otherwise, return the points in the original leaf node to be split, and create a new leaf node.
    c. Create new nodes if a compressed tree edge is split.
    d. the CPU will deduplicate if multiple keys to insert conflict in creating the same new nodes.
3. In two communication rounds:
    a. Insert data points to the leaf nodes.
    b. Link all parent–child pointers for newly-created nodes.
    c. Modify the shared data caching. (Two rounds.)
    d. Promote/demote internal nodes. (Two rounds.)
    e. Update the lazy counters.
```

4.3 k Nearest Neighbors

A k nearest neighbor (kNN) query requires to return the exact k nearest points in PIM-zd-tree to a given point with a pre-defined distance metric (e.g., ℓ_1-norm or ℓ_2-norm).

Algorithm 3. KNN (Q: batch of query points; k: INT)

```
1. SEARCH (Q) and record the search traces.
2. For each $q \in Q$, find on the search trace the lowest node $N_{q,1}$ whose lazy counter records $SC(N_{q,1}) \geq k$. Use push-pull search to traverse its descendants to find $k$ nearest candidates.
3. For each $q \in Q$, find on the search trace the lowest node $N_{q,2}$ which entirely contains the smallest sphere $\mathbb{S}_q$ centered at $q$ that contains all the $k$ candidates from $N_{q,1}$.
4. For each $q \in Q$, use push-pull search to traverse the descendants of $N_{q,2}$ who intersect with $\mathbb{S}_q$. Return all points from the descendants of $N_{q,2}$ that lie inside $\mathbb{S}_q$.
5. Filter the returned points on the CPU, outputting the final $k$NN for each $q$.
```

4.4 Orthogonal Range Query

An orthogonal range query, or box query, specifies one (or a batch of) axis-aligned rectangular boxes. There are two categories based on their output: a BoxCount query returns the number of points in PIM-zd-tree that fall within the box, while a BoxFetch query retrieves all such points. The execution procedure closely follows that of Search, where push-pull search is applied level by level. The key difference is that box queries must also track all nodes intersecting the query box. Pseudocode is omitted due to space constraints.

5 Theoretical Analysis

In this section, we analyze PIM-zd-tree on the PIM Model and show that it achieves good performance[1] regardless of workload skew. Our analysis in this section adopts two common assumptions used in prior work: bounded ratio (Defn. 1) and bounded expansion constant (Defn. 2). However, this does not mean that PIM-zd-tree is only practically efficient under such dataset distributions. In §7, we will show that the design choices we made in PIM-zd-tree are practically efficient in various real-world datasets, regardless of whether these datasets satisfy the two assumptions or not.

For readers less interested in the mathematical details, we suggest focusing on the main conclusions of Theorems 5.1 and 5.3 to 5.5; or directly on Table 2, which summarizes the configurations implemented on real-world machines.

Definition 1 (Bounded Ratio [5, 12, 17])**.** *Given a point set P of size n, let d_{max} denote the maximum distance between any two points in the set, and let d_{min} denote the minimum distance. Then P has* ***bounded ratio*** *if*

$$\frac{d_{max}}{d_{min}} = poly(n).$$

Definition 2 (Bounded Expansion Constant [1, 2, 10, 12, 27, 52, 53, 79])**.** *Given a point set P contained in a bounded Euclidean space X, P has* ***expansion contant*** *γ if for $\forall x \in X$ and $\forall r > 0$, if $|box(x, r)| = k > 1$ then*

$$|box(x, 2r)| \leq \gamma k.$$

The expansion constant is referred to as ***bounded*** *if $\gamma = O(1)$.*

We also define (α, β)-skew in Defn. 3 to asymptotically characterize the skew of the distribution in a batch.

[1]All analytical bounds in this section would include a cost of D for each D-dimensional point accessed in a leaf. We omit this for clarity.

Definition 3 (Skew). *A batch of S queries with keys in the range $[U_l, U_r]$ is defined to have (α, β)-skew iff for every integer $\forall i \in [1, \beta]$, the number of keys falling in the subinterval $\left[U_l + \frac{i-1}{\beta}(U_r - U_l), U_l + \frac{i}{\beta}(U_r - U_l)\right]$ is at most $\frac{S}{\alpha}$.*

In other words, when the key range is divided into β equal-sized subranges, each subrange has at most a $1/\alpha$ fraction of the query keys.

5.1 Space Consumption

Theorem 5.1 (Space). *A PIM-zd-tree containing n data points takes $O\left(n + \frac{nP}{\theta_{L0}} + \frac{n}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}}\right)$ space.*

Proof Sketch. Due to bounded expansion constant, L0 has $O(n/\theta_{L0})$ nodes. L1 has $O(n/\theta_{L1})$ nodes, each of which is replicated $O\left(\log_{\max\{\gamma-1, \frac{1}{\gamma-1}\}} \frac{\theta_{L0}}{\theta_{L1}} / \log B\right) = O\left(\log_B \frac{\theta_{L0}}{\theta_{L1}}\right)$ times (the maximum height of an L1 path). □

5.2 Top-Down Search

The costs for Search is presented in Theorem 5.3. By properly tuning θ_{L0}, θ_{L1} and B, the PIM-zd-tree can be adapted to varying degrees of (α, β)-skew, ensuring a desired cost bound. Load balance is proved using Lemma 5.2.

Lemma 5.2 (Balls into Bins [77]). *Uniformly randomly placing weighted balls with total weight $W = \sum w_i$ and $w_i < W/(P \log P)$ into P bins yields $O(W/P)$ weight per bin whp*[2].

Theorem 5.3 (Search). *A batch of S Search queries can be executed in worst-case $O(\log_B \theta_{L0})$ communication rounds, and takes a total of $O(S \log_B \theta_{L1})$ communication amount and $O(S \log n)$ PIM work. The PIM execution is load-balanced whp if the batch size is $S = \Omega(P \log P \cdot B \log_B \theta_{L0})$ or if the batch has $\left(P \log P \log_B \theta_{L0}, \frac{n}{\theta_{L0}}\right)$-skew. CPU takes $O(S \log_B \theta_{L1})$ expected work and $O(\log S \log_B \theta_{L1} + \log_B \theta_{L0})$ span whp.*

Proof Sketch. The PIM work is $O(S \log_B n)$ to search the S queries through the tree. The worst-case communication round is the total height of L1 and L2 (due to pulling at every level), which is $O(\log_B \theta_{L0})$. Communication amount for each Search query, due to the amortization of push-pull search, is $O(1)$ for L0 and L1, and is height $O(\log_B \theta_{L1})$ for L2. When either the batch size or the skew condition holds, the balance in PIM computation and communication can be proved using Lemma 5.2. The proof for CPU execution uses parallel work-efficient semi-sort [35] and radix sort [43]. □

5.3 Dynamic Updates

In this section, we present the overall cost of dynamic updates. We restrict our analysis here to the insertion-only case due to space constraints.

[2]We use $O(f(n))$ **with high probability** (***whp***) (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

Theorem 5.4 (Insert). *A batch of S Insert can be executed in worst-case $O(\log_B \theta_{L0})$ communication rounds, and takes a total of $O\left(\frac{SP}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} + S \log_B \theta_{L1}\right)$ communication amount and $O\left(\frac{SP}{\theta_{L0}} \log \frac{n}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} \log \frac{\theta_{L0}}{\theta_{L1}} + S \log n\right)$ PIM work. The PIM execution is load-balanced whp if batch size $S = \Omega\left(P \log P \cdot \left(B \log_B \theta_{L0} + \log \frac{\theta_{L0}}{\theta_{L1}}\right)\right)$ or if the batch has $\left(P \log P \log_B \theta_{L0}, \frac{n}{\theta_{L0}}\right)$-skew and $S = \Omega(P \log P \log \frac{\theta_{L0}}{\theta_{L1}})$. CPU execution takes $O\left(\frac{SP}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} + S \log_B \theta_{L1}\right)$ expected work and $O(sort(S) + \log S \log_B \theta_{L1} + \log_B \theta_{L0})$ span whp.*

Proof Sketch. For the proof of computation work, we refer to the proof of shared-memory zd-trees [12]. For the communication analysis, we combine Theorem 5.3 and additional costs in updating the data sharing structures. The update costs of data sharing structures are in the same frequency as lazy counters: updating P L0 copies every $\Theta(\theta_{L0})$ updates in expectation, and $\Theta(\log_B \frac{\theta_{L0}}{\theta_{L1}})$ L1 copies every $\Theta(\theta_{L1})$ updates in expectation. The $sort(S)$ term in the CPU span denotes the best-known parallel span for sorting S items [13, 29]. This cost arises in Insert, where points within a batch must be sorted when inserted into the same leaf and when potentially constructing a subtree. In contrast, Search does not require this span overhead, as it only relies on key equality rather than strict ordering, and thus requires only a semi-sort. □

5.4 k Nearest Neighbors

Theorem 5.5 (kNN). *Finding the k nearest neighbors of a data point requires expected $O(k + \log_B \theta_{L1})$ communication amount, worst-case $O(\log_B \theta_{L0})$ communication rounds, expected $O(k + \log n)$ PIM work, expected $O(k \log k + \log_B \theta_{L1})$ CPU work and expected $O(k)$ CPU cache footprint.*

Proof Sketch. For zd-trees, k nodes in expectation will be touched in a kNN, which makes up the PIM work, together with an $O(\log n)$ top-down search cost [12]. Since Alg. 3 uses push-pull search in all upwards and downwards searching, the communication amount and CPU work in expectation is equivalent to the height of levels without data-sharing (i.e., $\Theta(\log_B \theta_{L1})$) plus the output size k. The CPU has another $O(k \log k)$ work due to a priority queue. □

6 Implementation

To demonstrate the practical efficiency of our methods, we implement two configurations of PIM-zd-tree that represent the two extremes of the design frontier. The first is a ***throughput-optimized*** version, which prioritizes communication and computation efficiency. The second is a ***skew-resistant*** version, capable of tolerating arbitrary adversarial skew when $S = \Omega(P \log^2 P)$. Both the throughput-oriented and skew-resistant versions are special cases of the tunable design described in §3.1 and §3.2. Their key configurations and the operation costs are summarized in Table 2.

We implement our methods on UPMEM [37, 72]. In this section, we describe the practical techniques adopted to achieve high performance. Most of these techniques build on fundamental characteristics of BLIMP, and we expect them to apply to a wide range of architectures beyond UPMEM.

Table 2. Configurations of implementations, space consumption, and communication amount per operation.

Method	Throughput-Optimized	Skew-Resistant
θ_{L0}	n/P	$\Theta(P)$
θ_{L1}	1	$\Theta(\log_B P)$
B	θ_{L0}	$\Theta(1) = 16$
Allowed Skew	$(P \log P, \frac{n}{P})$	Arbitrary
Required S	$\Omega(P \log P)$	$\Omega(P \log^2 P)$
Space	$O(n)$	$O(n)$
Search	$O(1)$	$O(\log_B \log_B P)$
Updates	$O(1)$	$O(\log_B \log_B P)$
kNN	$O(k)$	$O(k + \log_B \log_B P)$

Practical Chunking. We provide an adaptive node structure design for L1 with two capacity modes, which is inspired by the adaptive radix tree (ART) [57]. It implements the imbalanced-tree-shaped chunking described in §3.2. The key observation is that dense internal nodes are highly likely to appear in the inner levels of the tree, where the chunk structure tends to be well balanced. In contrast, sparse internal nodes are more likely to occur near the leaf levels, where the chunk structure may become imbalanced.

- **Sparse Mode**: If an L1 chunk contains fewer than $B/4$ nodes, we use two arrays of length $B/4$—one for keys and one for pointers. Keys are stored in sorted order, and each pointer is aligned with its corresponding key.
- **Dense Mode**: If the chunk contains at least $B/4$ nodes, we instead use an array of B pointers to represent the root node of the chunk. A descendant can be located with a single lookup using the key byte as an index.

Our subtree-size based chunking differs from conventional chunking, which uses a fixed-height fanout. Such conventional chunking can lead to imbalanced tree shapes, requiring $L/\log B$ jumps across chunks for key length L and fanout B. The intuition behind our approach is that chunk shapes are biased toward the longer branch in the original tree. As a result, jumps can be reduced from $L/\log B$ down to L/B.

Fast z-Order Computation. Computing the z-order of the keys is a critical step in the performance of PIM-zd-tree. Most prior academic works adopt the direct bit-wise interleaving method (e.g., [12, 61]), which has a complexity of $O(\text{bits})$.

In contrast, our implementation employs a faster Z-order computation [7, 82] based on the recursive construction of gaps over the original coordinates, reducing the complexity to $O(\log(\text{bits}))$. We further optimize for commonly encountered low-dimensional settings and extend the implementation to support higher-dimensional cases that are not covered by existing implementations. As an example, we illustrate below how 3D data points are transformed into 64-bit keys:

```
function Split_By_Three(uint64 x) { // x in [0, 2^21]
    x = (x | (x << 32)) & 0x001f00000000ffff;
    x = (x | (x << 16)) & 0x001f0000ff0000ff;
    x = (x | (x << 8))  & 0x100f00f00f00f00f;
    x = (x | (x << 4))  & 0x10c30c30c30c30c3;
    x = (x | (x << 2))  & 0x1249249249249249;
    return x;
}
function Z_Order_Key_3d(uint64 x, y, z) {
    x = Split_By_Three(x);
    y = Split_By_Three(y);
    z = Split_By_Three(z);
    return (x << 3) | (y << 2) | (z << 1);
}
```

Execution of Complex Distance Metrics on PIMs. Due to limited area inside memory chips, BLIMP architectures often suffer from constrained computational power on the PIM cores. This makes the computation of complex distance metrics on the PIM side comparatively slow. For example, on UPMEM machines, multiplication and division may take up to 32 cycles, much slower than simpler arithmetic operations (e.g., addition or bitwise AND/OR) [37], which significantly hinders the efficient computation of the ℓ_2-norm on PIM.

We propose an efficient execution flow for cases where a complex distance metric can be *anchored* by a simpler one. For example, the ℓ_2-norm can be anchored by the ℓ_1-norm since, for any $x \in \mathbb{R}^D$, $||x||_2/||x||_1 \in [1/\sqrt{D}, 1]$. As a result, in a kNN query, if the k-th nearest neighbor under ℓ_1-norm has distance x, then the k-th nearest neighbor under ℓ_2-norm must have ℓ_1-distance of at most $x\sqrt{D}$.

We decompose the candidate-finding process in Alg. 3 into two stages: coarse-grained filtering and fine-grained filtering. In the coarse-grained stage, PIM cores use a simple-to-compute (on PIM) distance metric (e.g., ℓ_1-norm) to determine a small candidate set that is guaranteed to contain all k nearest neighbors. The fine-grained stage is executed on the CPU, where a more complex distance metric (e.g., ℓ_2-norm) produces the accurate final results. For low-dimensional $D = O(1)$ and the ℓ_1- and ℓ_2-norm case (and assuming bounded ratio and bounded expansion constant), the candidate set returned by the coarse-grained filtering still has size $O(k)$, and Theorem 5.5 still holds.

Improved Direct API. We use a lightweight *Direct Interface/API* [50] to mitigate the overhead of the original UPMEM interface in small-batch scenarios. In the original UPMEM SDK [72], the PIM local memories are mapped to accessible regions in the virtual memory, and the UPMEM SDK communication APIs eventually translate into simple reads and writes to these regions. The Direct Interface [50] we use, in contrast, bypasses the intermediate SDK layers and directly manipulates the actual memory locations.

7 Evaluation

7.1 Experimental Setup

We evaluate PIM-zd-tree on an UPMEM®PIM-equipped server. The server features two Intel®Xeon Silver 4216 CPUs (32 threads total, 2.1 GHz, 22 MB LLC) and 12 memory channels, of which eight are populated with UPMEM DIMMs and four with standard DDR4 2400MT/s DRAM DIMMs. In total, the system includes 32 UPMEM ranks (2048 modules) providing 128 GB of PIM memory, with PIM cores running at 350 MHz.

Shared-Memory Competitors. We compare the throughput-optimized PIM-zd-tree against two state-of-the-art shared-memory implementations—zd-tree [12] and Pkd-tree [63].

We cannot evaluate shared-memory indexes on the UPMEM server because two-thirds of its memory channels are occupied by PIM-equipped DIMMs, which cannot serve as main memory. Running shared-memory indexes directly on this server would thus create an unfair limitation on memory bandwidth. Instead, we evaluate shared-memory indexes on a separate machine equipped with two Intel Xeon E5-2630 v4 CPUs, each with 10 cores at 2.20GHz and 25MB cache. Each socket has four memory channels, and no PIM-equipped DIMMs are present. This machine has similar performance to the CPU on the UPMEM server, and is used to evaluate the non-PIM baselines as we could not find an exact match.

Measurement. We evaluate on two metrics: (*i*) *Throughput*: Defined as the number of returned elements per second. For point operations (Insert, BoxCount), throughput is the number of operations executed per second, whereas for range operations (BoxFetch, *k*NN), it is the number of elements returned in the final output per second. (*ii*) *Per-Element Memory Traffic*: Defined as the total memory-bus communication (in bytes) incurred per returned element in the final output, including both CPU-DRAM and CPU-PIM communication. Memory traffic is a primary contributor to power consumption in index-based applications (see [37, 48, 66] for detailed studies on energy consumption).

7.2 End-to-End Comparison

Workload Setup. We begin with a microbenchmark using a uniformly random dataset. Each test first warms up the index by inserting 300 million uniformly random 3D data points. The benchmark then executes batches with same type of operations, each batch containing (*i*) 50 million point operations, or (ii) range operations that retrieve a total of 50 million elements in expectation. For BoxCount, BoxFetch and *k*NN queries, we evaluate each with three different query range sizes, covering on average 1, 10, and 100 data points.

In addition, we evaluate on two real-world datasets: COSMOS (CM) [78] and the Northern American region of OpenStreetMap (OSM) [38]. For each dataset, we use 80% of the data points for warmup and the remaining 20% for testing. These datasets exhibit real-world spatial skew. COSMOS, which captures astronomical objects in the galaxy, shows moderate skew, while OSM road network data for North America exhibits significantly stronger skew. We quantify their skew using Gini coefficients over the distribution of data points when each dataset is partitioned into $P = 2048$ bins. The resulting Gini coefficients are 0.287 and 0.967, corresponding approximately to Zipf distributions [100] with $\gamma = 0.455$ and 1.5, respectively.

Main Results in Throughput. Fig. 5 compares the UPMEM-based throughput-optimized PIM-zd-tree with CPU-based Pkd-tree and zd-tree across ten types of operations. PIM-zd-tree achieves geometrically averaged speedups of 1.82×, 4.25×, 3.08×, and 1.46× over Pkd-tree for Insert, BoxCount, BoxFetch, and *k*NN. Against zd-tree, the corresponding speedups are 1.49×, 518×, 99×, and 3.46×. The geometrically averaged memory traffic reduction across all operations is 3.5× compared to Pkd-tree and 18.8× compared to zd-tree.

The few cases where PIM-zd-tree does not outperform Pkd-tree in throughput occur for *k*NN queries with large *k* values. This is because large *k*NN queries are more likely to cross the boundaries between PIM modules, resulting in multiple rounds of communication and incurring significant *mux switch overhead* [54] when switching control of PIM memory between CPU and PIM-core accesses.

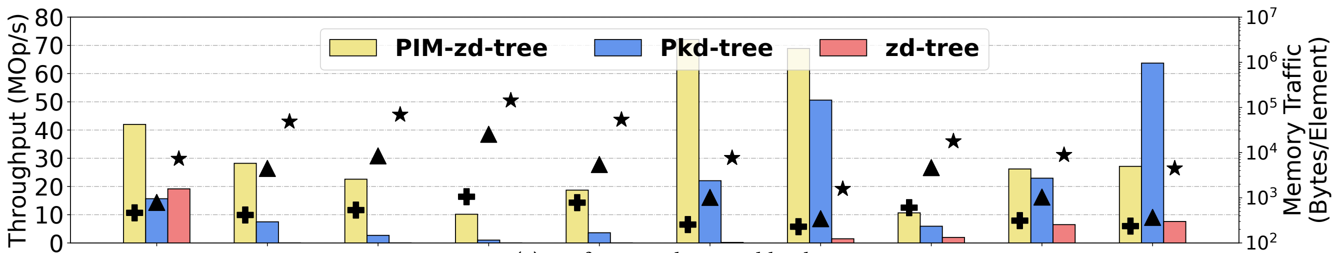
Latency Results. PIM-zd-tree also exhibits superior latency performance. For example, for 1-NN on OSM, excluding warmup, the P99 latencies of PIM-zd-tree, Pkd-tree, and zd-tree are 0.0325 s, 0.0449 s, and 0.210 s, respectively.

7.3 Ablation Study

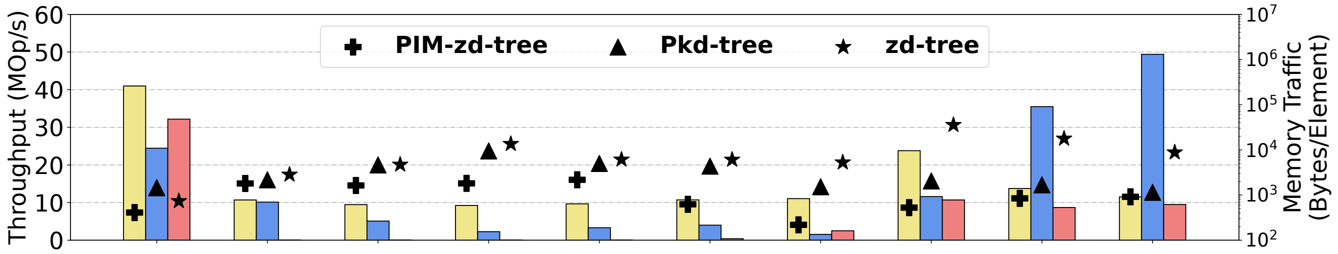
Breakdown of Time. Fig. 6 illustrates the time breakdown of CPU computation, PIM computation, and CPU-PIM communication. The Insert operation incurs significant CPU time, primarily due to preprocessing over the batch. In contrast, BoxFetch with size 100 exhibits high CPU-PIM communication time, as its computation is simple but the output size is large. For all other operations, the majority of the time is spent on PIM execution, which aligns with our design goal of offloading computation to PIM.

Sensitivity to Dimensions. We evaluate two-dimensional and three-dimensional uniform random workloads. The results show that 2D insertion throughput is only 1.02× higher than 3D, since execution is primarily bottlenecked by searches over fixed-length Morton keys. On the other hand, for box counts, box fetches, and *k*NN queries, 2D workloads achieve geometric-mean speedups of 1.49×, 1.22×, and 2.13× over 3D, respectively, due to the reduced cost of multi-dimensional vector computations and comparisons in these range queries.

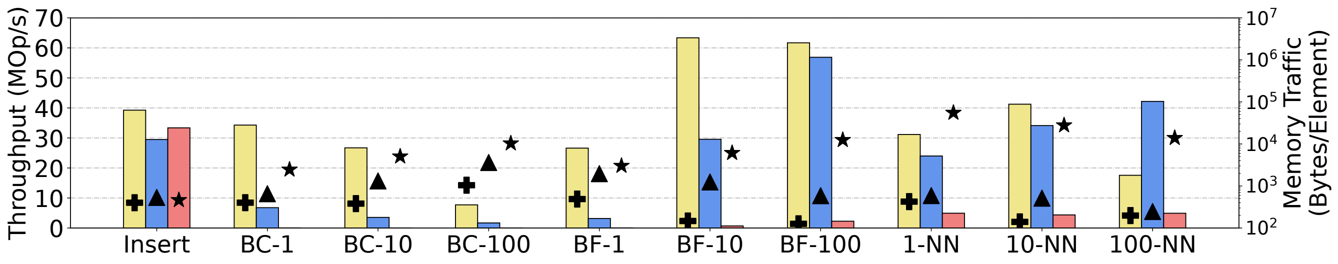
Sensitivity to Batch Sizes. Batch size plays a critical role in the execution of PIM-zd-tree. Larger batch sizes are preferred to amortize the mux switch overhead [54] and to achieve effective load balance. However, excessively large batches,



(a) Uniform random workloads.



(b) COSMOS dataset [78].



(c) OSM dataset [38].

Figure 5. Comparison of PIM-zd-tree, Pkd-tree, and zd-tree across three datasets on INSERT, BOXCOUNT (BC), BOXFETCH (BF) and nearest neighbor (NN) operations. The bar plots report throughput, while the scatter plots show memory traffic, measured as the number of bytes transmitted through the memory bus per element in the final output.

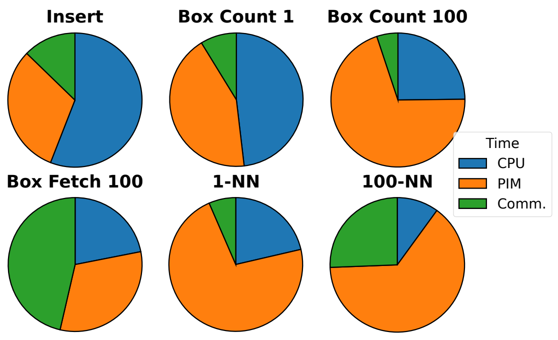


Figure 6. Runtime breakdown of different operations.

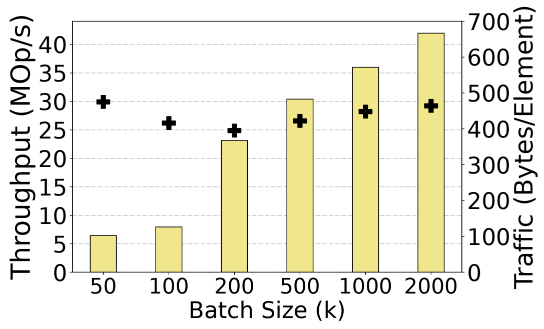


Figure 7. INSERT performance given different batch sizes.

combined with auxiliary structures, may exceed the capacity of the L3 cache, resulting in increased memory traffic.

Fig. 7 presents an ablation study on the impact of different batch sizes for INSERT operations. While increasing the batch size improves throughput, batch sizes exceeding 200k operations result in higher memory traffic per operation. This finding suggests that future systems with larger caches would be advantageous. Similar trends were observed for box and kNN queries, but are omitted here due to space limit.

Sensitivity to Dataset Sizes. One theoretical result we obtain is that, while search paths in shared-memory indexes have a length bounded by $O(\log n)$, the communication cost of PIM-zd-tree, as shown in §5, is bounded solely by the number of PIM modules P and is independent of n. Consequently, PIM-zd-tree is expected to maintain robust performance across datasets of varying sizes.

We evaluate the performance of the three methods under varying base dataset sizes during warmup, as shown in Fig. 8. The performance of PIM-zd-tree remains stable across dataset sizes, whereas the throughput of Pkd-tree and zd-tree degrades by 1.4× and 1.6×, respectively. Correspondingly, their memory traffic increases by 1.3× and 1.5×.

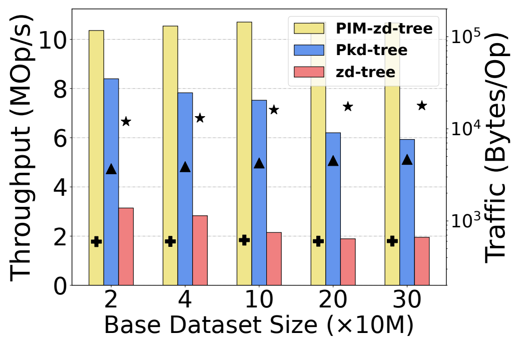


Figure 8. 1-NN throughput and memory traffic given different base dataset sizes during warmup before testing.

Table 3. Impact of implementation techniques on slowdown under uniform workloads. Results for box queries and *k*NN are reported as geometric means across three query sizes. *N.A.* indicates that the corresponding technique is not optimized for the given operation.

Technique	INSERT	BOXCOUNT	BOXFETCH	*k*NN
Lazy Counter	1.49×	N.A.	N.A.	N.A.
Fast z-order	1.99×	1.58×	1.31×	1.67×
Fast ℓ_2-norm	N.A.	N.A.	N.A.	1.58×
Direct API	1.06×	1.07×	1.09×	1.09×

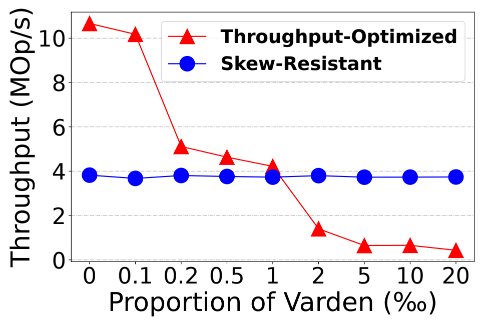


Figure 9. 1-NN throughputs of throughput-optimized and skew-resistant PIM-zd-tree, given combinations of Uniform+Varden [32].

Sensitivity to Optimizations. We evaluate the impact of four implementation techniques in PIM-zd-tree: lazy counter, fast z-order, fast ℓ_2-norm, and improved Direct API. Table 3 reports the slowdown observed when each technique is individually removed from the final design. All techniques provide substantial performance benefits, with the exception of Direct API. The limited impact of Direct API arises from our use of large batch sizes to maximize performance, which falls outside the scenarios for which it is primarily optimized.

Skew Resistance. We further evaluate the skew-resilience of PIM-zd-tree under non-uniform workloads. Specifically, we compare the performance of both the throughput-optimized and the skew-resistant versions on *k*NN under skewed conditions. The skewed workload is derived from Varden [32], an extremely skewed distribution generated via random walk.

In our experiments, we mix *k*NN queries generated from the skewed Varden distribution into batches of uniformly distributed *k*NN queries. Fig. 9 presents the throughputs across varying proportions of skewed queries. The skew-resistant version of PIM-zd-tree demonstrates highly stable performance, with fluctuations of no more than 4.1%. In contrast, while the throughput-optimized version performs well under workloads with low degrees of skew, it is outperformed by the skew-resistant variant when more than 0.1% of the workload is skewed, and its performance degrades by 10.66× when 2% of the queries originate from Varden distribution.

8 Related Work

Comparison with PIM-tree. PIM-zd-tree supports an adjustable design that spans the full spectrum between the range-partitioning and skew-resistant layouts, whereas PIM-tree [48, 50] targets only one end point (skew-resistance).

Moreover, if comparing only the skew-resistant invariant, PIM-tree enforces skew resistance by partitioning equal-height levels, which is effective for balanced trees but unsuitable for imbalanced zd-trees. In contrast, PIM-zd-tree partitions by subtree size and relies on a lazy counter mechanism to efficiently maintain approximate sizes, which is also required for *k*NN and orthogonal range queries. While PIM-zd-tree adopts push-pull search from PIM-tree, it employs a design-specific push-pull threshold and uses imbalanced chunking rather than the balanced chunking of PIM-tree.

Comparison with GPU-based Designs. Existing GPU-based spatial indexes, such as kd-trees [92] and R-trees [67], report relatively low performance, with construction throughput below 20 MOp/s in both works. Moreover, GPUs are significantly more energy-consuming than PIM systems. Therefore, while end-to-end comparison against GPU-based designs would be valuable future work, we do not expect such comparisons to change the main conclusions of this work.

9 Conclusion

We present PIM-zd-tree, the first space-partitioning index evaluated on a real-world PIM system. Our design introduces a provably-efficient tunable structure that adapts to different requirements in skew tolerance, communication, and space overheads. We further adopt implementation techniques that effectively translate theoretical efficiency into practical performance. PIM-zd-tree delivers up to 4.25× and 99× speedup over two shared-memory baselines, and reduces memory-channel traffic by an average of 3.5× and 18.8×.

Acknowledgments

This research was supported by NSF grants CCF-2119352, CCF-2339310, CCF-2403235, CNS-2211882 and CNS-2317194, and the Parallel Data Lab Consortium (Bloomberg, Datadog, Google, Intel, Jane Street, LayerZero, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, and Western Digital).

References

[1] Evangelos Anagnostopoulos, Ioannis Z. Emiris, and Ioannis Psarros. 2015. Low-Quality Dimension Reduction and High-Dimensional Approximate Nearest Neighbor. In *31st International Symposium on Computational Geometry (SoCG 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 34)*, Lars Arge and János Pach (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 436–450. https://doi.org/10.4230/LIPIcs.SOCG.2015.436

[2] Evangelos Anagnostopoulos, Ioannis Z. Emiris, and Ioannis Psarros. 2018. Randomized Embeddings with Slack and High-Dimensional Approximate Nearest Neighbor. *ACM Trans. Algorithms* 14, 2, Article 18 (April 2018), 21 pages. https://doi.org/10.1145/3178540

[3] Md Tanvir Arafin and Zhaojun Lu. 2020. Security Challenges of Processing-In-Memory Systems. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI (GLSVLSI '20)*. 229–234. https://doi.org/10.1145/3386263.3411365

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 119–129. https://doi.org/10.1145/277651.277678

[5] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM* 45, 6 (Nov. 1998), 891–923. https://doi.org/10.1145/293347.293348

[6] Kazi Asifuzzaman, Narasinga Rao Miniskar, Aaron R. Young, Frank Liu, and Jeffrey S. Vetter. 2023. A survey on processing-in-memory techniques: Advances and challenges. *Memories - Materials, Devices, Circuits and Systems* 4 (2023), 100022. https://doi.org/10.1016/j.memori.2022.100022

[7] Jeroen Baert. 2013. Morton encoding/decoding through bit interleaving: Implementations. https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/. Accessed December 2025.

[8] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

[9] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN '23)*. 44–52. https://doi.org/10.1145/3592980.3595312

[10] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/1143844.1143857

[11] Wenhao Bi, Junwen Ma, Xudong Zhu, Weixiang Wang, and An Zhang. 2022. Cloud service selection based on weighted KD tree nearest neighbor search. *Applied Soft Computing* 131 (2022), 109780. https://doi.org/10.1016/j.asoc.2022.109780

[12] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208. https://doi.org/10.1137/1.9781611977042.16

[13] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/3350755.3400227

[14] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998). https://doi.org/10.1137/S0097539793259471

[15] Pedro Carrinho, Oscar Ferraz, João Dinis Ferreira, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2024. Processing Multi-Layer Perceptrons In-Memory. In *2024 IEEE Workshop on Signal Processing Systems (SiPS)*. 7–12. https://doi.org/10.1109/SiPS62058.2024.00010

[16] Pedro Carrinho, Hamid Moghadaspour, Oscar Ferraz, João Dinis Ferreira, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2026. An Experimental Exploration of In-Memory Computing for Multi-Layer Perceptrons. *Journal of Signal Processing Systems* 98, 1 (2026), 1. https://doi.org/10.1007/s11265-025-01974-7

[17] Timothy M. Chan. 2008. Well-separated pair decomposition in linear time? *Inform. Process. Lett.* 107, 5 (2008), 138–141. https://doi.org/10.1016/j.ipl.2008.02.008

[18] Deting Chen, Yu Huang, Yi Huang, Binbin Lin, Yi Zhang, Long Zheng, Xiaofei Liao, and Hai Jin. 2025. Prism: Practical In-Memory Acceleration for Subgraph Matching at Scale. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC67600.2025.11196211

[19] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2025. Accelerating RNA-Seq Quantification on a Real Processing-in-Memory System. *IEEE Trans. Comput.* 74, 7 (2025), 2334–2347. https://doi.org/10.1109/TC.2025.3558075

[20] Mingkai Chen, Cheng Liu, Shengwen Liang, Lei He, Ying Wang, Lei Zhang, Huawei Li, and Xiaowei Li. 2024. An Energy-Efficient In-Memory Accelerator for Graph Construction and Updating. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 6 (2024), 1781–1793. https://doi.org/10.1109/TCAD.2024.3355038

[21] Qile P. Chen, Bai Xue, and J. Ilja Siepmann. 2017. Using the k-d Tree Data Structure to Accelerate Monte Carlo Simulations. *Journal of Chemical Theory and Computation* 13, 4 (2017), 1556–1565. https://doi.org/10.1021/acs.jctc.6b01222

[22] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. 2025. UpANNS: Enhancing Billion-Scale ANNS Efficiency with Real-World PIM Architecture. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 789–804. https://doi.org/10.1145/3712285.3759777

[23] Yewang Chen, Lida Zhou, Yi Tang, Jai Puneet Singh, Nizar Bouguila, Cheng Wang, Huazhen Wang, and Jixiang Du. 2019. Fast neighbor search by using revised k-d tree. *Information Sciences* 472 (2019), 145–162. https://doi.org/10.1016/j.ins.2018.09.012

[24] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2022. HybriDS: Cache-Conscious Concurrent Data Structures for Near-Memory Processing Architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA) *(SPAA '22)*. Association for Computing Machinery, New York, NY, USA, 321–332. https://doi.org/10.1145/3490148.3538591

[25] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (Phoenix, AZ, USA) *(SPAA '19)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/3323165.3323191

[26] B. Choi, B. Chang, and I. Ihm. 2013. Improving Memory Space Efficiency of Kd-tree for Real-time Ray Tracing. *Computer Graphics Forum* 32, 7 (2013), 335–344. https://doi.org/10.1111/cgf.12241

[27] Michael Connor and Piyush Kumar. 2010. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics* 16, 4 (2010), 599–608. https://doi.org/10.1109/TVCG.2010.9

[28] Florestan De Moor, Meven Mognol, Charles Deltel, Erwan Drezen, Julien Legriel, and Dominique Lavenier. 2024. MiMyCS: A Processing-in-Memory Read Mapper for Compressing Next-Gen Sequencing

Datasets. In *2024 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 6716–6723. https://doi.org/10.1109/BIBM62325.2024.10821790

[29] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3627535.3638483

[30] Dina Fakhry, Mohamed Abdelsalam, M. Watheq El-Kharashi, and Mona Safar. 2023. An HBM3 Processing-In-Memory Architecture for Security and Data Integrity: Case Study. In *Green Sustainability: Towards Innovative Digital Transformation*, Dalia Magdi, Ahmed Abou El-Fetouh, Mohamed Mamdouh, and Amit Joshi (Eds.). 281–293. https://doi.org/10.1007/978-981-99-4764-5_18

[31] Oscar Ferraz, Gabriel Falcao, and Vitor Silva. 2024. In-Memory Bit Flipping LDPC Decoding. In *2024 32nd European Signal Processing Conference (EUSIPCO)*. 706–710. https://doi.org/10.23919/EUSIPCO63174.2024.10715253

[32] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 14 (July 2017), 45 pages. https://doi.org/10.1145/3083897

[33] Sahar Ghoflsaz Ghinani, Jingyao Zhang, and Elaheh Sadredini. 2025. Enabling Low-Cost Secure Computing on Untrusted In-Memory Architectures. arXiv:2501.17292 [cs.CR] https://arxiv.org/abs/2501.17292

[34] Seyedeh Gol Ara Ghoreishi, Charles Boateng, Sonia Moshfeghi, Muhammad Tanveer Jan, Joshua Conniff, Kwangsoo Yang, Jinwoo Jang, Borko Furht, David Newman, Ruth Tappen, Monica Rosselli, and Kelley L. Jackson. 2025. Quad-Tree-Based Driver Classification Using Deep Learning for Mild Cognitive Impairment Detection. *IEEE Access* 13 (2025), 63129–63142. https://doi.org/10.1109/ACCESS.2025.3558706

[35] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/2755573.2755597

[36] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. 2023. Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 211–215. https://doi.org/10.1109/IISWC59245.2023.00030

[37] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. https://doi.org/10.1109/ACCESS.2022.3174101

[38] Mordechai Haklay and Patrick Weber. 2008. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing* 7, 4 (2008), 12–18. https://doi.org/10.1109/MPRV.2008.80

[39] Xu-Qiang Hu and Yu-Ping Wang. 2023. QuadSampling: A Novel Sampling Method for Remote Implicit Neural 3D Reconstruction Based on Quad-Tree. In *International Conference on Computer-Aided Design and Computer Graphics*. Springer, 314–328. https://doi.org/10.1007/978-981-99-9666-7_21

[40] Yuan Huang, Zhiqin Zhao, Conghui Qi, Zaiping Nie, and Qing Huo Liu. 2018. Fast Point-Based KD-Tree Construction Method for Hybrid High Frequency Method in Electromagnetic Scattering. *IEEE Access* 6 (2018), 38348–38355. https://doi.org/10.1109/ACCESS.2018.2853659

[41] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 263–279. https://doi.org/10.1109/HPCA57654.2024.00029

[42] Intel. 2025. Intel In-Memory Analytics Accelerator (Intel IAA). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/in-memory-analytics-accelerator.html. Accessed December 2025.

[43] J. JaJa. 1992. *Introduction to Parallel Algorithms.*

[44] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*. 87–88. https://doi.org/10.1109/VLSIT.2012.6242474

[45] Shouyan Jiang, Liguo Sun, Ean Tat Ooi, Mohsen Ghaemian, and Chengbin Du. 2022. Automatic mesoscopic fracture modelling of concrete based on enriched SBFEM space and quad-tree mesh. *Construction and Building Materials* 350 (2022), 128890. https://doi.org/10.1016/j.conbuildmat.2022.128890

[46] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive k-d tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. 1–5. https://doi.org/10.1109/DSIA.2017.8339084

[47] Hongbo Kang, Phillip B. Gibbons, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/3409964.3461816

[48] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proc. VLDB Endow.* 16, 4 (dec 2022), 946–958. https://doi.org/10.14778/3574245.3574275

[49] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2023. PIM-Trie: A Skew-Resistant Trie for Processing-in-Memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23)*. 1–14. https://doi.org/10.1145/3558481.3591070

[50] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2025. PIM-tree: A Skew-resistant Index for Processing-in-Memory: H. Kang et al. *The VLDB Journal* 34, 6 (2025), 66. https://doi.org/10.1007/s00778-025-00937-5

[51] Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang. 2013. gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *Journal of Systems Architecture* 59, 3 (2013), 166–175. https://doi.org/10.1016/j.sysarc.2011.06.003

[52] David R. Karger and Matthias Ruhl. 2002. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02)*. Association for Computing Machinery, New York, NY, USA, 741–750. https://doi.org/10.1145/509907.510013

[53] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of first-order queries on classes of structures with bounded expansion. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (New York, New York, USA) *(PODS '13)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/2463664.2463667

[54] Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. 2025. No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory. *Proc. VLDB Endow.* (2025), 4241–4254. https://doi.org/10.14778/3749646.3749690

[55] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 627–642. https://doi.org/10.1109/MICRO61859.2024.00053

[56] Vincent T. Lee, Amrita Mazumdar, Carlo C. del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. 2018. Application Codesign of Near-Data Processing for Similarity Search. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 896–907. https://doi.org/10.1109/IPDPS.2018.00099

[57] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[58] Chun-Chien Liu, Chun-Feng Wu, and Yunho Jin. 2025. UPVSS: Jointly Managing Vector Similarity Search with Near-Memory Processing Systems. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7. https://doi.org/10.1109/DAC63849.2025.11132577

[59] Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1595–1598.

[60] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/3087556.3087582

[61] Magdalen Dobson Manohar, Yuanhao Wei, and Guy E. Blelloch. 2025. CLEANN: Lock-Free Augmented Trees for Low-Dimensional k-Nearest Neighbor Search. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*. 131–143. https://doi.org/10.1145/3694906.3743339

[62] Ziyang Men, Bo Huang, Yan Gu, and Yihan Sun. 2026. Parallel Dynamic Spatial Indexes. In *Proceedings of the 31st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Sydney, Australia) *(PPoPP '26)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3774934.3786412

[63] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *Proc. ACM Manag. Data* 3, 1, Article 62 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709712

[64] Meven Mognol, Dominique Lavenier, and Julien Legriel. 2024. Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24)*. 1062–1071. https://doi.org/10.1145/3673038.3673094

[65] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842. https://doi.org/10.1145/359619.359627

[66] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2023. *A Modern Primer on Processing in Memory*. 171–243. https://doi.org/10.1007/978-981-16-7487-7_7

[67] Jian Nong, Xi He, Jia Chen, and Yanyan Liang. 2024. Efficient Parallel Processing of R-Tree on GPUs. *Mathematics* 12, 13 (2024), 2115. https://doi.org/10.3390/math12132115

[68] Mohammed Otair. 2013. Approximate k-nearest neighbour based spatial clustering using k-d tree. arXiv:1303.1951 [cs.DB] https://arxiv.org/abs/1303.1951

[69] Gonçalo Perrolas, Milad Niknejad, Ricardo Ribeiro, and Alexandre Bernardino. 2022. Scalable Fire and Smoke Segmentation from Aerial Images Using Convolutional Neural Networks and Quad-Tree Search. *Sensors* 22, 5 (2022), 1701. https://doi.org/10.3390/s22051701

[70] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. 2020. QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 180–192. https://doi.org/10.1109/HPCA47549.2020.00024

[71] KH Vijayendra Prasad and P Sasikumar. 2024. Energy-efficient quad tree-based clustering using edge-assisted UAV-relay to enhance network lifetime in WSN. *Scientific Reports* 14, 1 (2024), 17160. https://doi.org/10.1038/s41598-024-68085-4

[72] Qualcomm. 2025. UPMEM Technology. https://www.upmem.com/technology/. Accessed August 2025.

[73] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*. 1378–1388. https://doi.org/10.1145/3292500.3330875

[74] W. Saftly, M. Baes, and P. Camps. 2014. Hierarchical octree and k-d tree grids for 3D radiative transfer simulations. *A&A* 561 (2014), A77. https://doi.org/10.1051/0004-6361/201322593

[75] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June 1984), 187–260. https://doi.org/10.1145/356924.356930

[76] Samsung. 2025. Samsung PIM Technology. https://semiconductor.samsung.com/technologies/memory/pim/. Accessed December 2025.

[77] Peter Sanders. 1996. On the Competitive Analysis of Randomized Static Load Balancing. In *Workshop on Randomized Parallel Algorithms (RANDOM)*.

[78] Nick Scoville, H Aussel, Marcella Brusa, Peter Capak, C Marcella Carollo, M Elvis, M Giavalisco, L Guzzo, G Hasinger, C Impey, et al. 2007. The cosmic evolution survey (COSMOS): overview. *The Astrophysical Journal Supplement Series* 172, 1 (2007), 1. https://doi.org/10.1086/516585

[79] Luc Segoufin and Alexandre Vigny. 2017. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *ICDT*. Venise, Italy. https://inria.hal.science/hal-01589303

[80] Yunxiao Shan, Shu Li, Fuxiang Li, Yuxin Cui, Shuai Li, Ming Zhou, and Xiang Li. 2022. A Density Peaks Clustering Algorithm With Sparse Search and K-d Tree. *IEEE Access* 10 (2022), 74883–74901. https://doi.org/10.1109/ACCESS.2022.3190958

[81] Shunchen Shi, Xueqi Li, Zhaowu Pan, Peiheng Zhang, and Ninghui Sun. 2024. CoPIM: A Collaborative Scheduling Framework for Commodity Processing-in-memory Systems. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*. 44–51. https://doi.org/10.1109/ICCD63220.2024.00018

[82] John Sietsma. 2019. Morton Order - Introduction. https://johnsietsma.com/2019/12/05/morton-order-introduction/. Accessed December 2025.

[83] Guy L. Steele and Jean-Baptiste Tristan. 2016. Adding approximate counters. *SIGPLAN Not.* 51, 8, Article 15 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851147

[84] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2017. Adding Approximate Counters. *ACM Trans. Parallel Comput.* 4, 1, Article 5 (Oct. 2017), 45 pages. https://doi.org/10.1145/3132167

[85] Guy L Steele Jr and Jean-Baptiste Tristan. 2018. Method and system for latent dirichlet allocation2 computation using approximate counters. US Patent 10,147,044.

[86] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78. https://doi.org/10.1109/TC.1970.5008902

[87] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th International Middleware Conference (Middleware '24)*. 417–430. https://doi.org/10.1145/3652892.3700782

[88] Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. 3–17. https://doi.org/10.1145/3582016.3582026

[89] Vijay R Tiwari. 2023. Developments in KD tree and KNN searches. *International Journal of Computer Applications* 975 (2023), 8887. https://doi.org/10.5120/ijca2023922879

[90] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/

79173.79181

[91] Yi Wang, Dun Liu, Hongmei Zhao, Yali Li, Weimeng Song, Menglin Liu, Lei Tian, and Xiaohao Yan. 2022. Rapid citrus harvesting motion planning with pre-harvesting point and quad-tree. *Computers and Electronics in Agriculture* 202 (2022), 107348. https://doi.org/10.1016/j.compag.2022.107348

[92] David Wehr and Rafael Radkowski. 2018. Parallel kd-tree construction on the gpu with an adaptive split and sort strategy. *International Journal of Parallel Programming* 46, 6 (2018), 1139–1156. https://doi.org/10.1007/s10766-018-0571-0

[93] Jingyi Xu, Sehoon Kim, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Memory-Efficient Hardware Performance Counters with Approximate-Counting Algorithms. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 226–228. https://doi.org/10.1109/ISPASS51385.2021.00041

[94] Zhou Yijun, Xi Jiadong, and Luo Chen. 2021. A Fast Bi-Directional A* Algorithm Based on Quad-Tree Decomposition and Hierarchical Map. *IEEE Access* 9 (2021), 102877–102885. https://doi.org/10.1109/ACCESS.2021.3094854

[95] Yiwei Zhao, Jinhui Chen, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Jorge Tomas Gomez, Jae-Sun Seo, Barbara De Salvo, Chiao Liu, Phillip B. Gibbons, and Ziyun Li. 2025. H4H: Hybrid Convolution-Transformer Architecture Search for NPU-CIM Heterogeneous Systems for AR/VR Applications. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC '25)*. 1133–1141. https://doi.org/10.1145/3658617.3697627

[96] Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*. 350–366. https://doi.org/10.1145/3694906.3743318

[97] Yiwei Zhao, Ziyun Li, Win-San Khwa, Xiaoyu Sun, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Yi-Lun Lu, Jorge Tomas Gomez, Jae-Sun Seo, Phillip B. Gibbons, Barbara De Salvo, and Chiao Liu. 2024. Neural Architecture Search of Hybrid Models for NPU-CIM Heterogeneous AR/VR Devices. arXiv:2410.08326 [cs.CV]

[98] Yiwei Zhao, Qiushi Lin, Hongbo Kang, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2025. TD-Orch: Scalable Load-Balancing for Distributed Systems with Applications to Graph Processing. arXiv:2511.11843 [cs.DC]

[99] Yue Zhao, Yunhai Wang, Jian Zhang, Chi-Wing Fu, Mingliang Xu, and Dominik Moritz. 2022. KD-Box: Line-segment-based KD-tree for Interactive Exploration of Large-scale Time-Series Data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 890–900. https://doi.org/10.1109/TVCG.2021.3114865

[100] George Kingsley Zipf. 2016. *Human behavior and the principle of least effort: An introduction to human ecology.* Ravenio Books.

Received 2025-09-01; accepted 2025-11-10