PIM-trie: A Skew-resistant Trie for Processing-in-Memory

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

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

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

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

Yan Gu
UC Riverside
Riverside, CA, USA
ygu@cs.ucr.edu

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

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

ABSTRACT

Memory latency and bandwidth are significant bottlenecks in designing in-memory indexes. *Processing-in-memory* (PIM), an emerging hardware design approach, alleviates this problem by embedding processors in memory modules, enabling low-latency memory access whose aggregated bandwidth scales linearly with the number of PIM modules. Despite recent work in balanced *comparison-based* indexes on PIM systems, building efficient *tries* for PIMs remains an open challenge due to tries' inherently unbalanced shape.

This paper presents the *PIM-trie*, the first batch-parallel radix-based index for PIM systems that provides load balance and low communication under adversary-controlled workloads. We introduce *trie matching*—matching a *query trie* of a batch against the *compressed data trie*—as a key building block for PIM-friendly index operations. Our algorithm combines (i) hash-based comparisons for coarse-grained work distribution/elimination and (ii) bit-by-bit comparisons for fine-grained matching. Combined with other techniques (*meta-block decomposition, selective recursive replication, differentiated verification*), PIM-trie supports LongestCommonPrefix, Insert, and Delete in $O(\log P)$ communication rounds per batch and $O(l/w)$ communication volume per string, where P is the number of PIM modules, l is the string length in bits, and w is the machine word size. Moreover, work and communication are load-balanced among modules *whp*, even under worst-case skew.

CCS CONCEPTS

• **Theory of computation** → **Data structures design and analysis**; *Pattern matching*; • **Hardware** → *Emerging architectures.*

KEYWORDS

processing-in-memory; trie; radix tree; in-memory index

ACM Reference Format:
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), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3558481.3591070

SPAA '23, June 17–19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591070

1 INTRODUCTION

As data-intensive applications have become increasingly prominent in the past decades, the ever-widening gap between computation speed and memory access speed has made *data movement* the dominant cost and main bottleneck in modern systems. This problem, often referred to as the *memory wall* [65], can now be potentially solved by emerging processing-in-memory technologies [42].

Processing-in-memory (PIM), a.k.a. *near-data-processing (NDP)*, enables computation to be executed on computation units embedded in memory modules. Instead of fetching data through the memory and cache hierarchy to the CPU as in traditional von Neumann systems, PIM pushes computation to the memory modules, reducing the energy consumption of data movement and leveraging the performance gains arising from having aggregated memory bandwidth that scales linearly with the number of modules.

Although the idea of PIM dates back to the 1970s [56], it is now re-gaining research attention due to the development of 3D-stacked memory [28], which enables the production of real PIM systems. At least hundreds of academic works on PIM architecture design have been published (see the references of [5, 32, 42]), while real-world commercialized PIM systems have also been launched [34, 58].

Most of the prior work focuses on systems improvements, in applications such as databases [17, 35], genome analysis [3, 64, 67], graph processing [1, 39, 43, 51, 69], neural networks [27, 33, 60, 61], security [2, 25, 38], and matrix multiplication [23, 66]. Very few works design PIM-friendly data structures from an algorithmic view [18, 19, 40], or provide good theoretical guarantees. The Processing-in-Memory (PIM) Model [29] is the latest (and maybe the only) model for analyzing parallel algorithms on PIM systems.

This paper focuses on the design of **PIM-friendly radix-based indexes** that repeatedly perform parallel execution of a batch of queries/operations (*batch-parallel*) and support keys of variable length bit-strings. Radix-based indexes support LongestCommonPrefix (LCP), Subtree Query, Insert and Delete operations.

Radix-based indexes (radix trees, tries) are important search structures introduced in textbooks [48], and widely used in linux kernels [45, 46], in-memory storage [68], and IP routing [54, 63]. They are the only family of search structures designed to inherently support variable-length keys. Moreover, they are often faster in practice than other search structures on shared-memory machines [6, 16, 36]. A recent convincing example is from SetBench [4], the most efficient open-source implementation of comparison-based trees. In their latest evaluation [55], radix-based ART-OLC [37] outperforms all other comparison-based structures in most cases.

Approaches	Space (in words)	IO rounds (per batch)			Communication (per operation, in words)		
		LCP	Insert/Delete	Subtree	LCP	Insert/Delete	Subtree
Distributed Radix Tree	$O(L_D/w + n_D)$	$O(l/s)$	$O(l/s)$	$O(n_D)$	$O(l/s)$	$O(l/s)$	$O(l/s + L_S/w + n_S)$
Distributed x-fast trie[#]	$O(L_D)$	$O(\log l)$	$O(\log l)$	$O(n_D)$	$O(\log l)$	$O(l)$	$O(L_S)$
PIM-trie	$O(L_D/w + n_D)$	$O(\log P)^*$	$O(\log P)^\dagger$	$O(\log P)^*$	$O(l/w)^*$	$O(l/w)^\dagger$	$O((l + L_S)/w + n_S)^*$

Table 1: Space, IO rounds, and communication of different approaches on the PIM Model. Key parameters are defined in Table 2. In addition, l is the bit-string length of the given operation, and s is the span of the radix tree (i.e., the fanout is 2^s). (#) denotes that the structure supports only fixed-sized strings with $l = O(w)$ bits, in which case $l/w = O(1)$. A Subtree query returns a trie S, with L_S and n_S as defined in Table 2. (*) denotes a *whp* bound (with high probability in P). (†) denotes an amortized bound.

Nevertheless, because the shape of a radix-based index depends on the set of strings being stored, it can be highly imbalanced with height up to the length of the longest string. This creates two main challenges for adapting existing radix trees or tries to the PIM setting: (C1) how to map their nodes/edges to PIM modules in a way that achieves good load balance across the modules, while minimizing the communication required to answer queries (as noted in [29], there is an inherent tension for PIM between load balance and low communication), and (C2) how to avoid serial bottlenecks when dealing with long strings. Table 1 (first two rows) summarizes the performance of taking *traditional radix trees* and *x-fast tries* and randomly hashing them to PIM modules, in terms of the space for each data structure, the number of communication rounds (*IO rounds*), and the total communication, for various operations (LCP, Insert/Delete, Subtree)—full details in Sections 2 and 3. The family of fast tries (*x-fast tries* [62], *y-fast tries* [62], *z-fast tries* [8]) are explicitly designed to address challenge C2, but x-fast tries and y-fast tries support only fixed-length keys, and how to address challenge C1 for z-fast tries is still an open problem.

To address these challenges, we present **PIM-trie**, a skew-resistant batch-parallel radix-based tree that has good asymptotic guarantees in the PIM Model (Table 1, third row). It is not only the first radix-based index designed for PIM systems, but also the first radix tree that asymptotically benefits from batch-parallel processing, for worst-case data and query skew. It builds upon the idea in z-fast tries of combining both a radix tree and a hash table, addressing challenge C1 and other challenges.

We introduce **trie matching** as the core idea in our algorithm design, which exploits the benefits of batch-parallel processing. The set of strings in the index is stored on PIM modules as a *data trie*, a hybrid of a radix tree and hash values to further exploit PIM parallelism. A *query trie* is constructed upon the batched operations, and the *matched trie* information between this query trie and the data trie is vital in all operations. The tree component of the data trie is decomposed into *blocks*. Each block is a sub-trie stored on the same PIM module, while distributed meta-blocks are used to organize the block metadata. We give a *selective recursive replication* of meta-blocks into child meta-blocks, to ensure load balance and to further reduce communication without increasing the space bound.

To resolve potential false-positive query results due to hash collisions, we introduce a *verification* procedure. It does not increase our asymptotic bounds due to our differentiated handling of *critical blocks*, whose number proved to be bounded, and *non-critical blocks* handled by the attached *last bytes*.

In summary, our PIM-trie supports batch-parallel operations (LongestCommonPrefix, Subtree Query, Insert and Delete) on variable-length bit strings. These operations have efficient bounds in the PIM Model even under adversary skew of query and data, outperforming PIM-based radix trees and fast tries.

Notation	Definition
P	Number of PIM modules
M	CPU cache size (in words)
w	Word size (in bits); values are $O(w)$ bits
N	Total size of data to be stored (in words)
n_T	Number of key-value pairs to be stored in trie T
L_T	Aggregate length of bit-string edges in trie T (in bits)
Q_T	Size of compressed trie T (in words); Note: $Q_T = O(L_T/w + n_T)$
k_T	Average length of bit-string keys in trie T (in bits)
k_{TM}	Maximum length of bit-string keys in trie T (in bits)
l_{TM}	Maximum edge length in trie T (in bits)
$X_D/X_Q/X_M$	Metric X of the data/query/match trie
K_B	Block size upper-bound (in words)
K_{MB}	Meta-block size upper-bound (in #*hash_values*)
K_{SMB}	Meta-block size lower-bound (in #*hash_values*)

Table 2: Definition of all notations.

The paper is organized as follows. Section 2 reviews the PIM Model and other preliminaries. In Section 3, we discuss the most closely related work, listing some of the building blocks we used from them and addressing their drawbacks in the settings of PIM-friendly radix-based trees. Section 4 describes the key techniques of PIM-trie and their analyses in the PIM Model. Section 5 introduces the procedures of the operations supported by PIM-trie and their asymptotic bounds. Section 6 presents conclusions.

2 PIM MODEL

We use the **Processing-in-Memory Model** (PIM Model) [29] as the theoretical abstraction of PIM systems. Prior work [30] has shown experimentally that the PIM Model is a good match for the well-studied commercial PIM system from UPMEM, which is an example of a class of PIM systems referred to as bank-level-in-memory-processing (BLIMP) [20]. We believe that the PIM Model is a good match for BLIMP systems and other near-data-processing (NDP) systems, although not a good match for processing-using-memory systems [42, 49] (because such systems can perform only a few operations in the memory module, not arbitrary code).

The PIM Model consists of a host CPU side and P PIM modules (the PIM side). The CPU side is a multicore processor with a shared on-chip cache of M words. Each PIM module combines a small memory of $O(N/P)$ words (where N denotes the problem size), called the PIM memory, and a weak but general-purpose compute unit called the PIM processor. The host CPU can load programs to PIM modules, launch them, and detect their completion. The host CPU can access both its on-chip cache and the local memory of PIM modules, but each PIM module can access only its own PIM memory. The host CPU communicates with PIM modules by directly reading/writing their respective local memories in parallel. The model assumes that programs run in BSP-like synchronous rounds [59], where at each round, the CPU side can (1) perform local computations, (2) write a buffer of data to each PIM module's local memory, (3) launch PIM programs and wait for their completion, and (4) read a buffer of data from each PIM module's local memory.

To analyze algorithms with both CPU and PIM sides, the model combines both shared-memory and distributed metrics. For local computations on the multicore CPU, it assumes a binary forking model [13] with a work-stealing scheduler [15] and measures the *CPU work* (total number of instructions of all cores) and *CPU depth* (work on the critical path). For CPU-PIM communication, it measures the number of *IO rounds* and the *IO time*, which is the maximum number of word-sized messages to/from any PIM module. For PIM programs, it measures the *PIM time*, which is the maximum work on any one PIM processor. For algorithms with multiple rounds, the maximums are derived separately for each round, and summed across rounds. Because both IO time and PIM time consider the *maximum* across all PIM modules, it is critical to design algorithms that ensure good load balance among PIM modules, even under adversarially chosen (skewed) workloads.

Definition 1 ([29]). *An algorithm is* ***PIM-balanced*** *if it takes $O(W/P)$ PIM time and $O(I/P)$ IO time, with W and I, respectively, the sums of PIM work and communication across all P PIM modules.*

In other words, each PIM module asymptotically performs an equal fraction of the total work and total amount of communication. In this paper, we will bound the total communication and prove PIM-balance *whp*, thereby bounding the IO time. We frequently use the following weighted balls-into-bins lemma to prove PIM-balance:

Lemma 2.1 ([29, 47]). *Placing weighted balls with total weight $W = \sum w_i$ and each $w_i < W/(P \log P)$ into P bins uniformly randomly yields $O(W/P)$ weight in each bin whp.*

3 RELATED WORKS

3.1 Tries and Variants

Trie. Tries are tree structures that store key-value pairs with bit-string keys. All descendants of an inner node in the tree share a common prefix formed by the path to this node. When searching through a trie, an inner node decides which child to traverse based on the query key. A binary trie, for instance, determines whether to go to the left or right child based on each bit of the query key. Binary tries do not necessarily perform well, since their heights are equal to the key length, k, in bits. Patricia tries [41] introduce the *path compression technique*, which reduces the tree height by omitting nodes with only one child and thus compressing the paths.

Radix Tree. Radix trees (or compressed tries) allow each node to represent s bits of a key instead of just one. They also use path compression. Each inner node can support an array of 2^s child pointers, and an s bit chunk of the key is used to index into each inner node when querying. The s here is called the *span*. The tree height is reduced to at most k/s. However, the 2^s-sized child array is often not fully utilized in practice, causing space inefficiency.

Adaptive radix trees (ARTs) [36] resolve this problem by dynamically adapting the node structure with a variable array size based on the number of children, enabling both large span and good space efficiency. The worst-case space overhead is proved to be a constant number of bytes per key-value pair for arbitrary long keys.

Height optimized tries (HOTs) [10, 11] introduce another solution that dynamically varies the number of bits considered at each node and introduces compound nodes, which enable a consistently high fanout and thereby good cache efficiency. However, there is no nontrivial bound on the height of a compound tree.

Fast Trie. Fast trie is another family of tries that leverage hashing to achieve logarithmic query costs relative to key length. The earliest work on x-fast tries [62] constructs hash tables on each level of the original binary trie. Meanwhile, each inner node in the trie that does not have a left (right) child stores a pointer to its predecessor (successor) leaf node. When querying, the x-fast trie carries out a binary search on the query string to find whether a prefix string exists in the hash table on the corresponding level. For a string with length k, this binary search returns the longest prefix as well as predecessor/successor queries in $O(\log k)$. However, the x-fast trie takes $O(nk)$ space as well as $O(k)$ update cost. The y-fast trie [62] was designed to reduce these costs. Buckets of comparison-based indexes with size $\Theta(k)$ are constructed near the leaves, and x-fast tries are used only in the top levels to index into the $O(n/k)$ buckets. In this way, y-fast tries achieve $O(n)$ space and $O(\log k)$ update cost, while keeping $O(\log k)$ query cost, for fixed-length keys.

The dynamic z-fast trie [8] was proposed to support arbitrary-length key strings. For machine word size w, it can handle strings of length up to 2^w, and queries/updates of key x are supported in $O(|x|/w + \log |x|)$ time and $O(|x|/B + \log |x|)$ I/Os (in the cache-oblivious model [21]). The key mechanism is fat binary search [7].

3.2 PIM-Friendly Indexes

Most indexes today are bottlenecked by memory bandwidth. Previous works try to overcome this bottleneck with the large aggregated memory bandwidth of PIM modules. However, the design of the index needs to be reconsidered to make effective use of many independent PIM modules. Two key challenges must be considered: (1) the algorithm should have low CPU-PIM communication, otherwise it will hit the memory wall like traditional algorithms for non-PIM indexes, and (2) the algorithm should have balanced communication, work, and space requirements across PIM modules, otherwise any stragglers will slow down the whole system. We divide previous PIM-friendly indexes into two categories, as follows.

Range-partitioned Indexes. Some prior works utilize PIM by using range partitioning [18, 19, 40]. The key space is divided into disjoint key ranges using a small set of separator keys that fit into the host CPU cache. Elements in the key ranges are then partitioned among PIM modules. The separators are managed by the CPU.

This type of index achieves low CPU-PIM communication—constant per element for both point and range operations—because after local CPU lookups, the operations are directly sent and executed by the corresponding PIM module. A limitation of this solution, however, is the load imbalance among PIM modules under skewed workloads. In the worst case, all queries target the key range of a single PIM module, serializing the entire batch of queries.

Skew-resistant Indexes. To solve this problem, two prior approaches focus on the load imbalance issue under skewed workloads [29, 30]. Both solutions build a batch-parallel PIM-optimized skiplist for integer keys that support operations with both accurate keys (get, update, insert, delete) and inaccurate keys (predecessor, range query). They first randomly distribute skiplist nodes to PIM modules for skew resistance, then horizontally divide the index into multiple layers and use different replication policies in each layer to reduce communication. The PIM-tree data-structure [30], for example, is a three-layer comparison-based index for PIMs. It uses full replication in the top layer, a partial selective replication method called *shadow subtrees* in the middle layer and purely distributed storage in the bottom layer. This solution is not readily applicable for tries, however, because tries can be arbitrarily unbalanced.

3.3 Building Blocks

Although the approach for PIM-trees [30] cannot directly be used for tries, two key ideas—*push-pull search* and *selective replication*—can be adapted, so we describe them here in more detail.

Push-Pull Search. *Push-Pull Search* is introduced to avoid imbalance among PIM modules, as randomly distributing nodes in a tree does not guarantee load balance without further design. In traditional non-PIM skip lists, point queries are executed by pointer chasing from root to leaf, forming a *search path.* A straightforward but inefficient query algorithm for PIM is to arbitrarily distribute the nodes among the PIM modules and visit them one by one along the search path by remote accesses. In addition to being communication inefficient, this approach suffers from load imbalance in skewed workloads: many queries can share nodes on their search paths, causing imbalance across PIM modules. For example, predecessor queries with different keys but the same answer will have exactly the same search path, causing contention on every node.

In Push-Pull search, this straightforward algorithm is called the *Push method*. The *Pull method* is then introduced to alleviate the load imbalance: when the number of queries to the same node exceeds a fixed number, the node is fetched to the CPU side and comparisons are executed on the CPU side instead. The combination of the Push and Pull methods guarantees load balance for any workload, although by itself this approach does not reduce communication.

Selective Replication. Selective replication is a replication strategy used in PIM-based balanced search trees, guaranteeing load-balance and constant communication per tree search. For each of the selected randomly-distributed inner nodes in a PIM-based tree, its entire subtree is replicated and stored on the same PIM module as the node. When a search query reaches this inner node, all further searching downwards can be carried out locally on the same PIM module, avoiding any further pointer chasing across modules. Selective replication is applicable only when the subtree(s) can fit into the local memory of a single PIM module.

The primary benefit of selective replication is that it guarantees load balance when combined with Push-Pull search. When there are few searches to a subtree, these queries are *pushed* through the selective replicas and only constant communication is needed per search. When the number of searches on a subtree is sufficiently large and can cause load imbalance if all executed by the same replica, the node on the current level is *pulled* to the CPU for searches to proceed. Selective replicas are used in the nodes on the next layer if the search distribution is relatively balanced; otherwise nodes are recursively pulled. This pulling incurs only amortized constant communication per query.

Applying selective replication to all nodes in the bottom $\log P$ levels of a balanced tree would increase by a $\Theta(\log P)$ factor both (i) the space and (ii) the work required to update the replicas under insertions/deletions. Instead, PIM-trees do not replicate their bottom $O(\log\log P)$ levels. This guarantees linear space, no asymptotic increase in insertion/deletion costs, and $O(\log\log P)$ IO rounds.

3.4 Limitations of Prior Work

We cannot simply use algorithms from prior PIM-friendly indexes to get ideal tries. Range-partitioning indexes are not suitable for skewed workloads because they suffer from severe load imbalance. Techniques in prior skew-resistant indexes cannot be applied to tries directly, because they require a balanced tree with (i) limited height and (ii) a constant factor decrease in size from the bottom level leaves to the top level root. Specifically, as tries are not balanced, they cannot be horizontally divided into layers with decreasing size bounds. Directly applying the PIM-tree replication strategy can cause a factor of P (or even n) space amplification.

Deploying prior radix-based indexes on a PIM system is also non-trivial. Table 1 illustrates the CPU-PIM communication bounds for queries by different approaches, showing that simple transformations of traditional indexes fail to reach competitive bounds. The first approach is to build a PIM-friendly radix tree by distributing tree nodes uniformly randomly to PIM modules, in order to mitigate load imbalance. The CPU-PIM communication required by this approach, however, is not smaller than the CPU-Memory communication for traditional in-memory radix trees, providing no benefits for having PIM modules. Moreover, querying a string of l bits can take $O(l/s)$ words communication. This bound is worse than ours because the radix span s must be several times smaller than w, since inner nodes support 2^s child pointers. The number of IO rounds is also higher. The second approach is to adapt x-fast tries to PIM systems by using PIM hash tables [30], but such tries can support only integers of $l = O(w)$ bits and require $O(l)$ space (in words) per integer. We could reduce space consumption to $O(l/w)$ words if an efficient distribution strategy were proposed for the buckets in y-fast tries, but y-fast tries still support only integer keys. The z-fast trie algorithm is a serial algorithm with good bounds in both work and space by combining a radix tree and a hash table. We are motivated by this combination, and realize that a hash table can act like a load distributor that parallelizes the workload to utilize PIM modules. This new use does not exist in serial z-fast tries. Also, all these algorithms have bad IO rounds bounds for SUBTREE queries.

4 OUR APPROACHES

Overview. PIM-tries are batch-parallel skew-resistant PIM-friendly binary radix trees supporting LONGESTCOMMONPREFIX (LCP) queries, INSERT, DELETE, and SUBTREE QUERY for keys of arbitrary length bit-strings. Being a batch-parallel algorithm, PIM-tries take a batch of same-type operations as input, and execute them in parallel as in [30, 50]. Minimum batch sizes are required for load balance, and batch sizes are assumed to be $O(M)$ so that a batch fits in the CPU cache. Motivated by radix trees, fast tries, and skew-resistant indexes, we design PIM-tries with three key optimizations: (1) a new execution layout to avoid replicated computation in the input batch, as we build a query trie upon the queried keys then do parallel lookup for the entire query trie; (2) using hash value comparisons for parallelism, load distribution, and work elimination; and (3) a hash value manager supporting effective hash value comparison by a recursive decomposition strategy over dynamic tries.

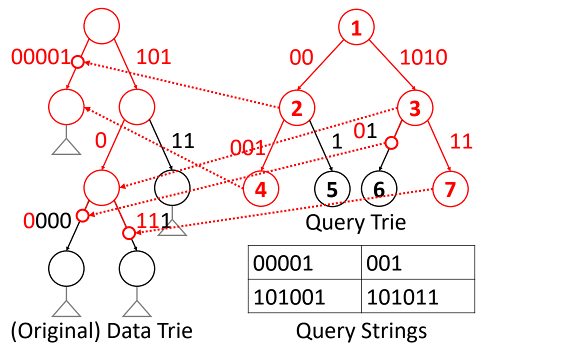
The hash value comparison in PIM-tries may generate false positives in the case of hash collisions. The collision probability can be reduced by using more bits while hashing; we also introduce a verification process to eliminate false positives.

Basic Structures and Terminology. Before introducing PIM-tries, we clarify the underlying trie structure and terminologies used. We use a binary radix tree (i.e., a binary compressed trie) as the basic structure in our paper. The term "trie" refers to a binary radix tree rather than its standard definition, an uncompressed k-ary search tree, for simplicity. For a standard radix tree storing n (key, value) pairs, path compression omits all nodes with only one child except those being the end of a key, leaving only $O(n)$ nodes and edges in total. We call these nodes *compressed nodes*, and edges *compressed edges* as they remain after path compression. A compressed node either has two children, or is the endpoint of a stored key, or both. Compressed nodes do not represent all prefixes stored in a radix tree, and those not included are also valid prefixes required for LCP, Insert, etc. We introduce *hidden nodes* to represent these implicit prefixes. There can be multiple hidden nodes on each compressed node. Combining both types of nodes, we have a bijection between all nodes and all valid prefixes. The *node depth* of a node is the length of its represented string (in bits). The word "node" refers to both types of nodes unless explicitly specified.

All compressed nodes/edges physically exist in PIM-tries and are each referred to by a (PIM module ID, local memory address) pair (called a *PIM address*). Every node has pointers to all its adjacent edges, and vice versa. Hidden nodes do not exist physically, so we refer to them by pairing the address of its host edge and its position on the edge (in bits). The node representing the key of a (key, value) pair holds the value (assumed to take $O(1)$ words) locally.

Although nodes and edges are distributed in CPU and PIM modules, pointers between them are never remote pointers, because PIM-tries always store a trie at CPU or at a single PIM module. A PIM-trie achieves this by decomposing itself into a bunch of unconnected small tries called *blocks* and distributing in block granularity. Every trie T with n_T strings stores all its compressed nodes, compressed edges, and an array of pointers to all compressed nodes to enable efficient parallel tree operations. For example, the treefix operations [53], including rootfix operations and leaffix operations, can be executed in $O(n_T)$ work and $O(\log n_T)$ depth *whp*. This array also enables efficient decomposition—given a set of K partition nodes, we can generate K stand-alone tries, where each node contains the ID of its corresponding node in the original trie—in $O(n_T)$ work and $O(\log n_T)$ depth *whp*. We use parameter L to denote the total number of trie nodes, which is also the aggregated lengths of all edges. The space consumption for T is $Q_T = O(L_T/w + n_T)$.

PIM-tries use hashing as a key technique. The term *node hash* represents the hash value of the string represented by a node. Hash values are stored in hash tables [24] of linear space and *whp* $O(k)$ work (and $O(\log^* k)$ PRAM depth or $O(\log k)$ binary forking depth) for batched lookups, inserts and deletes with batch size k.



00001	001
101001	101011

Figure 1: The figure displays an example of an LCP query with a data trie and a query trie generated from a batch of query strings. Useless hidden nodes are omitted, and values are drawn as attached gray triangles. The matched trie is marked in red, and may end on hidden nodes. Matching results are represented as dashed pointers.

```
Algorithm 1: QTrieConstruct
  Input: The batch of operation keys, $Q$
  Output: The query trie, $QT$
1 $Q_{Sorted}$ = StringSort($Q$)
2 $lcp$ = AdjacentLCPArray($Q_{Sorted}$)
3 return $QT$ = PatriciaGenerate($Q_{Sorted}$, $lcp$)
```

4.1 Query Trie and Trie Matching

PIM-trie relies on two key structures: the *data trie* and the *query trie*. The data trie is the main structure containing all data stored in the index, and the query trie is a novel structure containing all keys considered by a batch of operations. A query trie and a data trie (before future optimizations) are shown in Figure 1. Processing operation batches by using a whole query trie rather than one-by-one enables PIM-trie to avoid processing shared common prefixes among strings in the batch. This idea improves both computation and communication, and also has benefits in reducing contention.

Query Trie Construction. We build the query trie as a preprocessing step for every new batch. It is built and stored in the CPU cache. The construction algorithm is shown in Algorithm 1, and the theoretical guarantees in Lemma 4.1.

Lemma 4.1. *Constructing a query trie can be done in $O(n_Q(1 + k_Q/w)\log\log n_Q)$ CPU work and $O(\log^2 n_Q)$ CPU depth whp.*

Proof. Regarding construction in Algorithm 1, string sorting [26] is $O(n_Q(1 + k_Q/w)\log\log n_Q)$ work and $O(\log^2 n_Q/\log\log n_Q)$ depth on the binary forking model. Constructing the LCP array between adjacent string pairs [52] is $O(n_Q k_Q/w)$ work and $O(\log^2 n_Q)$ depth. Constructing a Patricia trie with the LCP array [14] costs $O(n_Q k_Q/w)$ work and $O(\log^2 n_Q)$ depth *whp*. □

Trie Matching Operation. We use *trie matching* operation as a key subroutine in our method. It compares the query trie generated by the current batch with the data trie to derive a shared part between them. This shared part, called the *matched trie*, represents all common prefixes between two tries, and is stored as a collection of node reference pairs between query trie nodes and data trie nodes. Figure 1 shows an example, with the matched trie in red.

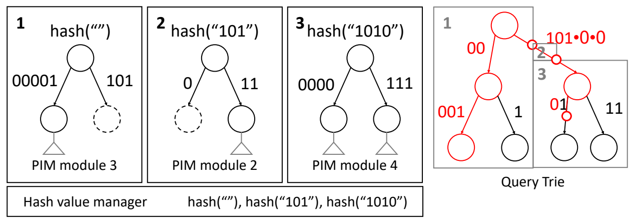


Figure 2: Left side: The data trie in Figure 1 decomposed and distributed randomly among PIM modules with mirror nodes marked as dashed circles and the hash value manager omitted. Right side: A query trie decomposed by data trie block root hashes, with each block in a gray box marked with its matching block ID. The final matched trie is in red.

Let L_M denote the total number of matched trie pairs. The full matching information contains $L_M = O(L_Q)$ pairs, taking $O(L_Q)$ communication to record data trie node references, thus breaking our bounds. Therefore, we only derive $O(n_Q)$ matchings for compressed nodes of the matched trie and the query trie. As shown in Figure 1, a trie matching operation builds red dashed arrows.

4.2 Hybrid Hash Trie

The PIM-trie stores string data among the PIM modules and enables efficient search by using a hybrid method of trie and hash comparisons. We combine both methods because using only one would fail to fully utilize the PIM system. For instance, simply randomly distributing the trie nodes to the PIM modules suffers from (serially) pointer chasing $O(n_D)$ steps when processing highly-skewed data.

Using hash comparisons only, on the other hand, is also insufficient. Assuming a fast trie structure where we calculate the node hash for every data trie node, and store a pointer to the node in the hash table, trie matching is executed by doing a hash join: it looks up query trie nodes in the hash table, then all matched node pairs will be found because they have the same node hash. However, this solution brings a dilemma in case of path compression: if we only store compressed nodes of the data trie in the hash table, we miss potential matched node pairs if one data trie hidden node matches with one query trie node; if we store all data trie nodes, the hash table will take $O(L_D)$ words space, $O(w)$ times more than storing only the trie. A similar dilemma also exists on the query trie side, as we miss possible answers if we look up only compressed nodes, or cause too much communication otherwise. One example is shown in Figure 1, where compressed nodes 1, 3, 4 match with compressed nodes while 2 matches with a hidden node; common prefix "10100" are represented by hidden nodes in both tries. In this approach, we get worse space and communication bounds for a correct result.

Tree Decomposition with Hash Values. PIM-tries combine the trie structure with hashing. We decompose the data trie into blocks of similar sizes, compute node hashes for their roots as metadata, and distribute these blocks uniformly randomly to PIM modules. An example is shown on the left side of Figure 2 where the data trie in Figure 1 is decomposed and distributed. Each (data trie) block contains the root node hash and the trie block. This decomposition replicates block root nodes as *mirror nodes* in the block containing the node's parent, represented by dashed circles in Figure 2. We omit details about metadata management by the hash value manager (see Section 4.4) and focus on trie blocks in this section.

Differently from the approach without hashing, we avoid remote pointer chasing by using block root hashes instead, abandoning all inter-block remote pointers. The trie matching algorithm starts with a comparison between block root hashes and hashes of all nodes in the query trie, and a trie matching operation between the current block and the subtree of V is triggered if and only if the block root matches the query trie node V. Furthermore, trie matching operations over all blocks can be run in parallel, even without waiting for the results of their parent blocks.

Block Size and Blocking Algorithm. Data tries are divided into blocks of $O(K_B) = O(\log^2 P)$ words, making $O(Q_D/K_B)$ blocks. All block roots are compressed nodes, and long compressed edges of more than K_B words are cut into pieces by adding compressed nodes in the middle to avoid oversized blocks, introducing $O(L_D/(w \cdot K_B))$ new compressed nodes.

For the blocking algorithm, PIM-tries reduce the problem to a parallel tree partitioning problem with weighted nodes, where the weight of each node is the total size of itself and two child edges. We use the parallel tree partitioning algorithm of [9], but extend it to a weighted version. To divide the tree of n nodes into blocks of B nodes, the unweighted algorithm (1) generates the Euler tour of a tree, (2) marks one out of every B nodes as base nodes, and (3) marks all lowest common ancestors of base nodes. The marked node set makes an ideal partition for the tree. For the weighted version, we assign the node weight to the Euler tour array, calculate the prefix sum, and pick nodes whose prefix sum exceeds a multiple of K_B as base nodes. This algorithm generates roots for $O(Q_D/K_B)$ blocks of size less than K_B in $O(n)$ work and $O(\log n)$ *whp* depth.

Lemma 4.2. *PIM-tries take $O(L_D/w + n_D)$ words space.*

Proof. Every PIM-trie contains multiple distributed trie blocks and a hash value manager. The aggregated size of the trie blocks is $O(L_D/w + n_D)$, as the data trie takes $O(L_D/w + n_D)$ words space before decomposition. Only $O(L_D/(w \cdot K_B) + n_D)$ additional space is required after, including $O(L_D/(w \cdot K_B))$ new compressed nodes as long edge cuts and $O(1)$ sized data per block. The space complexity of the hash value manager is proved in Lemma 4.7. □

4.3 Trie Matching Algorithm

The data trie block root hashes enable parallel block matching by decomposing the query trie, but matching all blocks in parallel breaks our communication bound. In the worst case, each query trie node can match with a different data trie block root, dividing the query trie into $O(L_Q)$ blocks for parallel matching, breaking our bounds even if only one word communication is used per block.

However, most blocks are not critical. Though there can be up to $O(L_Q)$ blocks, all but $O(n_Q)$ blocks are simply edges connecting two matched hidden nodes as ends and contain no compressed nodes, as the query trie only has $O(n_Q)$ compressed nodes. We call these blocks *non-critical blocks*. For example, a query trie after root hash matching is shown in Figure 2, where blocks are in gray boxes, and nodes representing (ϵ, "101", and "1010") are roots of block 1, 2, and 3, respectively. Blocks 1 and 3 are *critical blocks*, but block 2 is not. During the trie matching process, we ignore non-critical blocks and only match critical blocks unless verification is required. The verification process will be discuss later in Section 4.4.3.

Algorithm 2: Trie Matching

```
Input: The Query trie, QT, in CPU cache. The hash value
       manager, HVM.
Output: The matched trie as node reference pairs, Match[..]
1  QueryBlockRoot[..] = HVM.MatchCriticalBlock(QT)
2  QueryBlock[..] = Span(QueryBlockRoot[..])
   // only for $O(n_Q)$ critical blocks
3  parallel for $i \leftarrow 1$ to #CriticalBlocks do
4      DataBlock[i] = QueryBlock[i].matchBlockReference
5      PIMID[i] = DataBlock[i].PIMID
6  parallel for $i \leftarrow 1$ to #CriticalBlocks do
7      if QueryBlock[i].size < $K_B$ then
8          Send QueryBlock[i]: CPU $\rightarrow PIM_{PIMID[i]}$
           // executed on PIM modules locally
9          Match[i][..] = Match(QueryBlock[i], DataBlock[i])
10         Fetch Match[i][..]: $PIM_{PIMID[i]} \rightarrow CPU$
11     else
12         Fetch DataBlock[i]: $PIM_{PIMID[i]} \rightarrow CPU$
           // executed on CPU locally
13         Match[i][..] = Match(QueryBlock[i], DataBlock[i])
14 return Union_i(Match[i][..])
```

Algorithm 2 depicts pseudo-code for our trie matching algorithm. The hash value manager generates critical block roots together with references to their matching data trie blocks, and stores them on the CPU side (Line 1). We then expand these roots to stand-alone blocks (Line 2) on the CPU side. Blocks generated by this expansion can be larger than the actual critical block by absorbing all child non-critical blocks (for example, block 2 will be absorbed into block 1 in Figure 2), but these additional bits will be filtered out automatically in the local trie matching process. Since actual data trie blocks are in PIM memory, we use the Push-Pull technique to decide where to match local trees (Line 6-13). We get the final result by merging results in different blocks (Line 14). The actual Push-Pull process takes 5 rounds: (1) push small query trie blocks (Line 8), (2) PIM calculation (Line 9), (3) fetch results (Line 10), (4) pull for small data trie blocks (Line 12), and (5) CPU calculation (Line 13). In this paper, we merge these rounds into a single one in pseudo-codes for simplicity. A serial depth-first search is used as the local matching algorithm between blocks, and details about this algorithm will be introduced at the end of Section 4.4.2. A minimal batch size is required for the Push-Pull technique. The hash value manager and the solution to hash collisions will be introduced later in Section 4.4.

Theorem 4.3. *For a query trie with size $Q_Q = \Omega(P\log^5 P)$, the trie matching algorithm requires $O(Q_Q/P)$ IO time, $O(\log P)$ IO rounds, $O(Q_Q)$ CPU work, $O(\log^2 P + \log w \log P + \log(n_Q l_{QM}))$ CPU depth, and $O(Q_Q \log w/P)$ PIM time, with all bounds holding whp.*

Proof. We prove IO bounds for Algorithm 2 here and leave the work bounds to Lemma 4.11. Bounds for the hash value manager will be proven in Lemmas 4.8 and 4.10. All bounds hold *whp* because of the verification costs for potential hash collisions.

According to the Push-Pull technique, the communication of each query trie critical block is the minimal between its own size and the block size limit K_B. Therefore, the total communication is no more than the sum of the sizes of all query trie critical blocks, which is $O(Q_Q) = O(L_Q/w + n_Q)$, because only $O(n_Q)$ additional data is introduced ($O(1)$ per critical block) after decomposing a query trie of size $O(L_Q/w + n_Q)$. For load balance, note that data trie blocks are uniformly randomly distributed, so messages are between CPU and uniformly random target PIM modules with maximum size $K_B = O(Q_Q/P\log P)$. Combining these conditions ensures $O(Q_Q/P)$ IO time *whp* according to Lemma 2.1. □

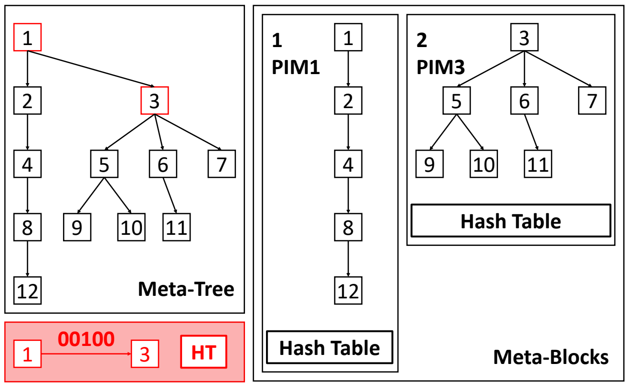


Figure 3: A meta-tree example on the top left, where meta-block roots are marked in red. The replicated master-tree and its hash table are drawn in red on the bottom left. Meta-blocks generated from the meta-tree are shown on the right.

4.4 Hash Value Manager

The hash value manager manages the metadata of data trie blocks: their root hashes. It is used in the first step of trie matching, where we derive critical blocks by matching between hash values of all $O(L_Q)$ query trie nodes and that of all $O(Q_D/K_B)$ data trie block roots in $O(L_Q/w + n_Q)$ words communication. Designing a hash value manager is challenging when we want both low communication and load balance. If it calculates node hashes on the CPU side, sending all $O(L_Q)$ node hashes breaks our communication bound. Another solution is to send the trie and calculate node hashes on the PIM side. This approach overcomes the communication problem, but it brings additional challenges as it requires the matching pairs of trie nodes and block root hashes to be on the same PIM module.

We observed that we need sufficient locality for low communication, and sufficient randomness for load balance. The selective replication technique (Section 3.3) is such combination, but its design is only applicable to balanced search trees, not to tries that can have much larger height. Motivated by this technique, we design a new distribution strategy in the hash value manager.

The hash value manager organizes block root hashes into a *meta-tree*, a directed tree with nodes representing blocks and edges representing block connections: if block A contains a mirror node of the root of block B, nodes for A and B are connected in the meta-tree. Every meta-tree node contains the root hash and the PIM address for the block it represents. The meta-tree does not physically exist as a whole tree, but instead as similar-sized pieces distributed randomly in PIM modules, forming *meta-blocks*. Each meta-block also maintains a hash table that maps block root hashes to its meta-tree nodes. Similarly, a master-tree and its hash table are built to organize meta-blocks. The master-tree physically exists and is replicated in all PIM modules. An example is shown on the

Algorithm 3: HashMatching: to decompose a trie according to node hashes in a hash table

```
Input: A trie. A hash table.
Output: A set of partition nodes.
1 GenerateHashValues(Trie)
2 for compressed edges in Trie do
3     for compressed nodes on the edge bottom-up do
4         if node.hash in $HT$ then
              // Find critical block root
5             Set node.matchBlockReference
6             Root.append(node)
7             break
8 return Root[..]
```

left side of Figure 3. A meta-tree with 12 nodes is shown in the top left corner. It is decomposed into two meta-blocks with their IDs and PIM IDs attached, shown on the right side. The master-tree and its hash table are drawn in red on the bottom left corner. As for Figure 2, its meta-tree can be represented as $\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3}$.

We set the meta-block size threshold as $K_{MB} = P$. Meta-trees have degree up to $O(K_B) = O(\log^2 P)$, because all $O(K_B)$ leaves of a data trie block can be mirror nodes of child block roots.

Trie Matching. Every meta-block represents multiple blocks, forming a single connected component in the data trie. Therefore, the data trie now has a two-layer decomposition: first by meta-block roots, then by block roots. The trie matching algorithm runs by applying the same decomposition to the query trie: it first divides the query trie into meta-blocks, then into blocks, and finally performs the actual matching. There is also the idea of critical query trie meta-blocks. The first step that derives critical meta-blocks is shown in Algorithm 4, where we first divide the query trie into $O(P \log P)$ similar sized master-blocks (Line 1), then do HashMatching between the hash table of the master-tree and the block (Algorithm 3) at PIM modules to generate query trie meta-block roots (Line 2-6).

Algorithm 4 is load balanced because it sends similar-sized blocks to random PIM modules, but a load balanced matching between meta-blocks to divide query trie into blocks is hard in case of large query trie meta-blocks. The load balanced matching algorithm will be introduced in Section 4.4.1; optimizations to reduce work in Section 4.4.2; and verification in Section 4.4.3.

Hash Function. PIM-tries have requirements on the hash function, and the minimum of which is *incremental*, because after decomposing a query trie into blocks, the full string of a node may not exist in the block, therefore the hash function needs to generate node hashes from a prefix hash (the block root hash) and a suffix string (see Definition 2). Many commonly used incremental hash functions in practice are even further *binary associatively incremental* (see Definition 3), such as rolling polynomial hashing [31] and CRC [44]. This property enables internal parallelism in node hash generation by parallel prefix sum [12] and rootfix scan [53].

Definition 2. *A hash function $h(\cdot)$ for bit-strings is* ***incremental****, if for any bit-string $C = AB$ concatenated by bit-strings A and B, it gives a function $f(\cdot,\cdot)$ that can output $h(C) = f(h(A), B)$, using only the hash value $h(A)$ and the bit-string B.*

Algorithm 4: MatchCriticalMetaBlock: decompose query trie to critical meta-blocks

```
Input: The Query trie, QT, in CPU cache.
Output: QueryMetaBlockRoot[..]
// MasterBlock[..]: QT divided into $O(P \log P)$
   similar sized blocks
1 MasterBlock[..] = Decompose(QT, QT.size/$(P \log P)$)
2 parallel for $i \leftarrow 1$ to #MasterBlock do
3     Send MasterBlock[i]: CPU $\rightarrow$ PIM$_{random(1..P)}$
      // Executed on PIM
4     QueryMetaBlockRoot[i][..] =
        HashMatching(MasterBlock[i], MasterTree.HashTable)
5     Fetch QueryMetaBlockRoot[i][..]: PIM $\rightarrow$ CPU
6 return Union$_i$(QueryMetaBlockRoot[i][..])
```

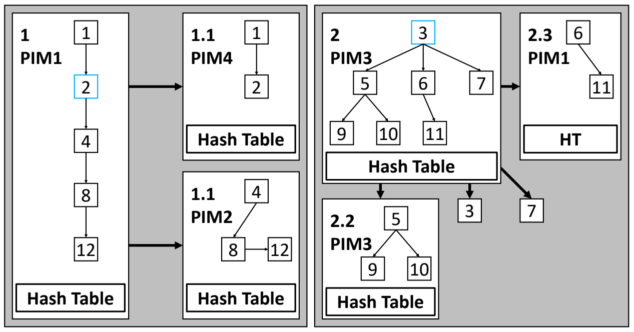


Figure 4: Meta-block trees generated from Figure 3 with cut nodes for meta-blocks in blue.

Definition 3. *A hash function $h(\cdot)$ for bit-strings is* ***binary associatively incremental****, if for any bit-string $C = AB$ concatenated by A and B, it gives a binary associative operation $\oplus$ that outputs $h(C) = h(A) \oplus h(B)$, using only the hash values $h(A)$ and $h(B)$ and their lengths, without knowing bit-strings A or B.*

Lemma 4.4. *Hashing all keys in an unprocessed batch with a binary associatively incremental hashing can be done in $O(n_Q(1 + k_Q/w))$ CPU work and $O(\log(n_Q k_{QM}))$ CPU depth, by applying parallel prefix sum to each of the keys in word granularity.*

4.4.1 Recursive Meta-block Decomposition Trying to match between meta-blocks directly at PIM modules can cause load imbalance, because we may send a large critical meta-block to a single PIM module. For example, if no hash match is found between the query trie and any meta-block roots, the whole query trie will be sent to the PIM module of the root meta-block. Similar problems occurs in block matching, where the Push-Pull technique is applied as a solution. However, the same solution does not work for meta-blocks because of the size difference ($\log^2 P$ vs. P, respectively). Simply fetching a meta-block of P words to CPU can cause imbalance given our $\Omega(P \log^5 P)$ limit on batch sizes.

To solve this size problem, we divide meta-blocks into smaller meta-blocks in order to re-enable the Push-Pull technique. We recursively decompose meta-blocks while ensuring that each still represents a connected component in the data trie. In each step, we select the node that minimizes the size of the child meta-block, then cut all its child edges. We recursively apply the division until the size drops below threshold $K_{SMB} = K_B = \log^2 P$. In two examples in Figure 4, we set $K_{MB} = 7$ and $K_{SMB} = 3$; each meta-block tree is in a gray box; and the cut node is marked in blue. Child meta-blocks are distributed and stored at random PIM modules, and a *meta-block tree* is generated, connecting them by remote pointers. The height of this tree is $O(\log P)$ according to Lemmas 4.5 and 4.6. Only root meta-blocks in meta-block trees are inserted to the master-tree.

Lemma 4.5. *For any unweighted directed out-tree of n nodes, there exists a cut node such that if we remove all its out-edges, the maximum remaining tree contains no more than $(n+1)/2$ nodes.*

Lemma 4.6. *Assuming a meta-block tree with $O(K_{MB})$ nodes and a pre-defined constant $\alpha < 1$, if it is true for all meta-blocks in the meta-block tree that all its children have size less than α fraction of its own size, the meta-block tree height is $O(\log(K_{MB}))$.*

Lemma 4.7. *The hash value manager takes $O(Q_D) = O(L_D/w + n_D)$ words of space.*

Proof. The size of the hash value manager is linear to the number of root hashes it stores after internal replication, including root hashes of $O(Q_D/K_B) = O(Q_D/\log^2 P)$ different blocks and $O(Q_D/K_B/K_{MB}) = O(Q_D/(P\log^2 P))$ different root meta-blocks. Block root hashes are replicated $O(\log K_{MB}) = O(\log P)$ times according to the height of the meta-block tree, and meta-block tree root hashes are replicated P times in the master-tree. Thus, the hash value manager contains $O(Q_D)$ hash values with $O(Q_D)$ space. □

Note that after this decomposition, the master-tree and the meta-block tree combine to form a hierarchical decomposition of the data trie of bounded height. The data trie is first decomposed into meta-blocks by the master-tree, then further divided by root hashes of meta-block tree nodes layer by layer into blocks. With this balanced hierarchy, we can now apply the Push-Pull strategy to divide large query trie meta-blocks into blocks without load imbalance: whenever its size exceeds the threshold $\log^4 P$, we pull the matching meta-block's $O(\log^2 P)$ child root hashes to divide it into smaller ones. We do this recursively until the block is small enough for a push, or the meta-block tree leaves are reached and then pulled. The full algorithm for deriving critical blocks from query trie is described in Algorithm 5, where the recursive decomposition is in Line 4-16, and Push-Pull in Line 17-27.

Lemma 4.8. *IO bounds in Theorem 4.3 hold for Algorithm 5.*

Proof. We prove the IO time separately for push and pull. In push rounds, blocks are sent from CPU to PIM for HashMatching, where the query trie is sent only twice (once in Algorithm 4) in blocks of $O(Q_Q/P\log P)$ words, guaranteeing load balance. There are $O(\log P)$ pull rounds according to the meta-block tree height, and we pull $O(\log^2 P)$ data for each of the $O(Q_Q/\log^4 P)$ oversized blocks in each round, adding up to $O(Q_Q/P\log P)$ total IO time. □

The key challenge in meta-block design is its low-cost compatibility to Inserts/Deletes. Newly generated/removed blocks may break the assumption in Lemma 4.6. We will discuss in Section 5.2.

4.4.2 Optimizations to Reduce Work

The work inefficiency of Algorithm 5 comes from two sides: (1) it calculates hashes for all $O(L_Q)$ nodes in HashMatching; and (2) it decomposes large query trie blocks of size $O(Q_Q)$ in linear work in each pull round, adding up to $O(Q_Q\log P)$ total work. In this section, we introduce optimizations that reduce the CPU work to $O(Q_Q)$ *whp* and the PIM

Algorithm 5: MatchCriticalBlock: decompose query trie to critical blocks

```
Input: The Query trie, QT, in CPU cache.
Output: QueryBlockRoot[..]
1  QueryMetaBlockRoot[..] = MatchCriticalMetaBlock(QT)
2  QueryMetaBlock[..] = Span(QueryMetaBlockRoot[..])
3  Threshold = $K_{SMB} \cdot \log^2 P = \log^4 P$
4  while QueryMetaBlock.size > 0 do
5      parallel for QMB in QueryMetaBlock[..] do
6          if QMB.size ≤ Threshold then
7              QueryMetaBlockSmall.append(QMB)
8          else if QMB.matchingMetaBlock is not a leaf then
9              DMB = QMB.matchDataMetaBlockReference
10             hashes[..] = DMB.ChildRootHash[..]
11             references[..] = DMB.ChildMetaBlock[..]
12             Fetch hashes, references: PIM → CPU
13             Root[..] = HashMatching(QMB,
                 HashTable(hashes[..], references[..]))
14             Block[..] = Span(Root[..])
15             QueryMetaBlockNext.append(Block[..])
16     QueryMetaBlock[..] = QueryMetaBlockNext[..]
   // Pull for QueryMetaBlocks.size > threshold
17 parallel for QMB in QueryMetaBlock[..] do
18     DMB = QMB.matchDataMetaBlockReference
19     Fetch DMB: PIM_{DMB} → CPU
20     Root[..] = HashMatching(QMB, DMB.HashTable)
21     QueryBlockRoot[..].append(Roots)
   // Push for QueryMetaBlocks.size ≤ threshold
22 parallel for QMB in QueryMetaBlockSmall[..] do
23     DMB = QMB.matchDataMetaBlockReference
24     Send QMB: CPU → PIM_{DMB}
25     Root[..] = HashMatching(QMB, DMB.HashTable)
26     Fetch Root[..]: PIM_{DMB} → CPU
27     QueryBlockRoot[..].append(Root)
28 return QueryBlockRoot[..]
```

time to $O(Q_Q \log w/P)$ *whp* to meet our bounds. We first reduce the number of hashes by string alignment, then use a batch-parallel euler-tour tree [57] to maintain query trie blocks. A local matching algorithm for blocks is introduced at the end, which follows similar ideas of our optimizations in work reduction.

Efficient HashMatching. To reduce the number of hashes computed, we compute for only a small proportion of nodes to cut the trie into fixed-length tries, then obtain the matching using fast tries.

To be specific, we select *pivot nodes* on the query trie—nodes whose depth (string length) is a multiple of w bits. We generate all $O(L_T/w)$ pivot node hashes on CPU when generating the query trie. We define the bottommost pivot ancestor of a node as its *host pivot*, and a pivot node is a host pivot of an edge if it is the host pivot of any node on this edge. We will miss matching points if we only consider pivot nodes, therefore additional steps are introduced for a complete result, starting by a data augmentation on block root

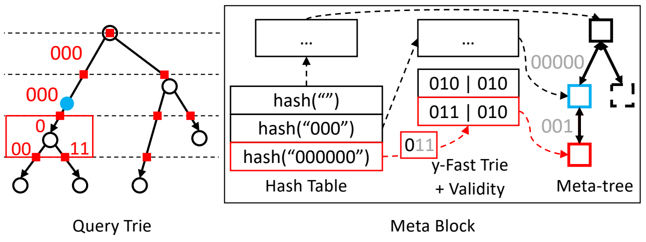


Figure 5: An example for HashMatching. Pivot nodes appear as red squares. One pivot-rooted block is outlined in red. The blue circle represents a critical block root, which matches the blue meta-tree node. On the right is the two-layer index, including the hash table and y-fast tries combined with validity vectors. Gray strings in the meta-tree do not exist physically.

hashes. For each meta-tree node, assuming that it represents a data trie block B, whose root represents a string S, so that it physically contains $hash(S)$ and $address(B)$, we now add to this node: (1) the length of S, (2) the hash value of the longest prefix of S whose length is a multiple of w, denoted as $hash(S_{pre})$, and (3) the suffix of S after S_{pre}, denoted as S_{rem}. Note that $|S_{rem}| < w$.

The HashMatching algorithm is modified to utilize this additional data. Our modification is different for CPU and PIM side execution, as they have different inputs. In CPU side execution (pull), $O(\log^2 P)$ data about child meta-blocks are fetched to CPU to divide a large query trie block. We augment the query trie for efficient division: (1) a hash table is built to map pivot hashes to pivot nodes, (2) a z-fast trie of height w is built for every pivot node as shortcuts to all its hosting compressed nodes. (Z-fast tries are compressed binary radix trees supporting lookup with cost $O(\log h)$ *whp* for height h and linear additional space.) For the fetched information of every block root in a pull round, we first lookup its $hash(S_{pre})$ in the hash table, then its S_{rem} in the relative z-fast trie to find the exact match position, each taking $O(\log w)$ work *whp*.

In PIM side execution (push), query trie blocks of size $O(\log^4 P)$ are sent to meta-blocks for critical block roots. To support this operation, we change the hash table in each meta-block to a two-layered index, where the first layer maps $hash(S_{pre})$ to the second layer, and the second maps from S_{rem} to meta-tree nodes. During the matching process, we lookup all pivot hashes in the hash table, then for every edge, we pick its bottommost hit host pivot node as its critical pivot. The match point is no more than w bits deeper than the critical pivot, so we build the string S'_{rem} going downward from the critical pivot until gathering w bits or reaching the end of the edge. However, the critical block root may not be on this string (see Figure 5 for an example), making it impossible for us to directly find the node by a simple lookup of S'_{rem} in the second layer.

To solve this problem, we do not directly find the critical block root, but find it or one of its direct children. The second layer index can be described as follows: (1) it maintains a set K of strings, all less than w bits, (2) For a query string Q, it returns string K_i, whose LCP with Q ($LCP(K_i, Q)$) is the longest among all strings in K, and there is no K_j who has the same LCP but is a prefix of K_i, in $O(\log w)$. The last requirement ensures that we find a direct child rather than an arbitrary node in the subtree. We store S_{rem} of block roots in this structure. Once we query S'_{rem}, it will return S_{rem} of either the critical block root or its direct child, which leads us to the node by an additional hash table from S_{rem} to meta-tree node addresses.

The second layer index combines a y-fast trie and a table. Every S_{rem} of a block root is padded into two (w-bit) integers, $S0$ and $S1$, by adding 0s and 1s at the end, and both are inserted into the y-fast trie. As different strings may be padded into the same integer, for every padded integer, a w-bit validity vector is stored in the table to record all valid prefixes. For a query, its string Q is padded similarly into $Q0$ and $Q1$, then we lookup their predecessor and successor in the y-fast trie. Taking the predecessor of $Q0$, $Q0_p$, as an example, we calculate its LCP with Q, then do a binary search on the validity vector of $Q0_p$ to find the shortest valid string longer than the LCP, or the longest valid string shorter than the LCP if no one is longer, and take it as the result. A similar process is executed for both the predecessor and the successor of $Q0$ and $Q1$, and the shortest one of those with the longest LCP is the final result. It takes $O(\log w)$ time *whp* to lookup, insert, delete a string in this structure, and we are guaranteed to find the correct string by a single query.

Figure 5 gives an example of this case when the critical block root is an ancestor of the critical pivot, but we still find its child through the two-layer index. In this example, we have $w = 3$. We first lookup hash("000000") in the hash table, then pad S'_{rem} = "0" into "011" (also "000") and lookup in the y-fast trie to find a predecessor (an exact match in this case), and finally do binary search on the validity vector to find S_{rem} = "01" for the child of our target meta-tree node (marked in blue). The whole process is marked in red.

Lemma 4.9. *Hashing all nodes in an already constructed trie with a binary associatively incremental hashing can be done in expected* $O(L_Q/w + n_Q)$ *CPU work and* $O(\log(n_Q l_{QM}))$ *depth whp.*

Proof. We first apply parallel prefix sum [12] to hash pivots on each edge locally, in $O(L_Q/w)$ work and $O(\log(n_Q l_{QM}))$ depth; then we use rootfix scan [53] on the trie, which costs $O(n_Q)$ work in expectation and $O(\log n_Q)$ depth *whp*. □

Efficient Block Partition. According to the height of meta-block trees, we use $O(\log P)$ pull rounds in Algorithm 5 (Line 4-16) to divide oversized blocks into smaller ones that will not cause load imbalance. In each round, there are $O(Q_Q/\log^4 P)$ oversized blocks with total size $O(Q_Q)$, and we fetch $O(\log^2 P)$ child meta-block root hashes and references to divide each block, adding up to $O(Q_Q/\log^2 P)$ divisions. Although the position of all divisions can be found in $O(Q_Q \log w/\log^2 P)$ time, it takes $O(Q_Q)$ work each round if we partition the oversized block physically into multiple stand-alone tries, adding up to $O(Q_Q \log w)$ total CPU work.

We realize that this tree decomposition problem is a dynamic forest problem with edge deletions and subtree size queries, and choose the batch-parallel Euler tour tree [57] as the solution. The basic idea of this data structure is to use augmented skip lists to maintain the Euler tours, which supports edge insertions and deletions by splits and merges of skip lists and subtree queries by values augmented on skip list nodes. For a forest with n nodes, this algorithm solves a batch of k edge insertions, edge deletions, or subtree size queries in $O(k \log n)$ work and $O(\log n)$ depth *whp*.

Lemma 4.10. *Work bounds in Theorem 4.3 hold for Algorithm 5.*

Proof. Pre-processing all pivot hashes takes $O(Q_Q)$ CPU work and $O(\log(l_{QM} n_Q))$ depth. We then prove work bounds separately for push and pull in Algorithm 5. For push, the total work (number of instructions) on PIM modules is $O(L_Q/w + n_Q \log w)$ *whp* because the y-fast trie lookup with $O(\log w)$ *whp* cost only happens once per edge, but the total PIM time is $O(Q_Q \log w/P)$ because of the load imbalance when some block has more nodes than the others. For pull, there are $O(\log P)$ rounds. In each round we pull $O(Q_Q/\log^2 P)$ node hashes to divide oversized query trie meta-blocks. Locating these divisions takes $O(Q_Q \log w/\log^2 P)$ work *whp* and $O(\log w)$ depth *whp* by z-fast tries, and dividing the meta-block takes $O(Q_Q/\log P)$ work *whp* and $O(\log P)$ depth *whp* by the dynamic forest algorithm. Adding these all up gives our bound. □

Efficient Local Matching. Using a serial depth-first search as the local trie matching algorithm takes work linear in the total size of the two blocks, which can be $O(w)$ times more than the size of a small query block, breaking our bounds in PIM time. The method to reduce it to $O(\log w)$ is similar to that used in the hash value manager: we pick pivot nodes in the data block (with length $kw, k \in \mathbb{Z}$), and build a z-fast trie of height w bits on every pivot as a shortcut to compressed nodes hosted by this pivot and pivots of the next level (with length $(k+1)w$). Trie Matching is executed by a DFS for all pivot nodes and compressed nodes in the query block.

Lemma 4.11. *Work bounds in Theorem 4.3 hold for Algorithm 2.*

Proof. The pull side on the CPU takes work linear in the query trie and depth linear in the block size $O(\log^2 P)$. For PIM work, the number of z-fast trie lookups is the total number of pivot nodes and compressed nodes in the query block, which is linear in its size. □

4.4.3 Verification Hash collisions can cause false positives and thus incorrect results. We design a verification process to effectively correct this. By setting a hash length of $5\log_2 N = O(w)$ bits and triggering a global re-hash once the tree size is squared, hash collisions occur with probability $O(1/N)$. A global PIM hash table [30] ensures no hash collision between data trie block roots by triggering global re-hashes once a collision is found. Given the infrequency of re-hashing, the maintenance cost does not change our (amortized) Insert bounds.

A collision between a query trie and a data trie can cause an incorrectly partitioned query trie and thus a wrong matching process. The key challenge is to detect these collisions. Once a collision is detected, we rectify the partitioning, then redo the matching process, until no collision exists. The number of redo rounds is linear in the number of simultaneous collisions, whose probability decreases exponentially to $O(1/N)$, making all bounds in Theorem 4.3 *whp.*

All critical blocks go through the bit-by-bit matching process, and these results report hash collisions. However, we cannot afford to verify all $O(L_Q)$ non-critical blocks in this way, because it will take $O(L_Q)$ communication, which breaks our bounds. We instead verify them during HashMatching by the hash value manager. Notice that non-critical blocks only exist as consecutive sequences in the middle of edges, and every such sequence has an critical block root as the bottommost end. For PIM side HashMatching, after finding a critical block root and its matching meta-tree node V, it locates the matching meta-tree nodes of these non-critical blocks, which are the closest ancestors of V. We augment the meta-tree node by (1) the last w bits of its represented string S, denoted as S_{last}, (2) a shortcut pointer to an arbitrary meta-tree ancestor whose depth is $k \in [w/2, w]$ bits smaller than the node's depth. Short non-critical blocks with less than w bits are verified by S_{last}, chasing shortcut pointers for speedup, and long ones are returned as critical blocks. CPU side HashMatching matches all blocks, so short non-critical blocks are directly verified by S_{last}. The bounds in Theorem 4.3 still hold with verification introduced.

5 PIM-TRIE OPERATIONS

In this section, we introduce operations supported by PIM-trie, and show how trie matching can be used to efficiently implement them.

5.1 LongestCommonPrefix

A LongestCommonPrefix (LCP) query finds the LCP between the query string and all strings stored in the PIM-trie. Because the result is a prefix of the query string, we only need the length. This query can be easily solved by trie matching results: the length is the node depth of the bottommost matching ancestor of the string's representing query trie node. For example, in Figure 1, string "101001" is represented by node 6. Its LCP length 5 can be found by the node depth of its ancestor.

To find LCPs for a batch of strings, we build the query trie, perform trie matching, then run a parallel rootfix on the query trie for results. Combining these bounds gives its final bound.

5.2 Insert and Delete

Insert. To Insert a batch of strings, we insert the unmatched subtrees of the query trie into the data trie, which updates both distributed data trie blocks and the hash value manager.

Update blocks. Unmatched query trie subtrees are inserted to the matching node of their roots. For example in Figure 2, the unmatched subtrees are in black and should be inserted into block 1 and 3. The required information is derived by the matched trie after a trie matching. The in-block insert process is the same as in traditional tries, and is executed on CPU (for large subtrees) or PIM (for small subtrees) based on the Push-Pull threshold. New blocks may be oversized after insertion. In this case, they are sent to CPU and re-partitioned by the blocking algorithm mentioned in Section 4.2, where additional info for block roots is also derived. Newly generated blocks are re-distributed to random PIM modules.

Update the hash value manager. Root hashes (and other auxiliary data) of the newly generated block should be inserted into the hash value manager. As all new blocks are children of the existing blocks, only meta-blocks holding the parent block's root hash need insertion, and we send new root hashes to them.

Insertions to the meta-block tree brings new problems: the cut vertex may no longer be the optimal cut in the new subtree, breaking the balanced structure of the meta-block tree and its height bound. For example, adversaries can insert many new blocks at the end of a flat list of blocks, degenerating the meta-block tree into a flat list.

Motivated by the scapegoat tree [22], we keep meta-block tree height bounds by rebuilding in case of sufficient imbalance. To be specific, for any meta-block in the meta-block tree with a child whose size exceeds α fraction of its own size, a new cut vertex is selected and its meta-block subtree is rebuilt completely. The rebuilt process is always executed on the CPU side in $O(n \log n)$ work and $O(\log^2 n)$ depth *whp* for a meta-block with n nodes. Setting α as a predefined parameter larger than 0.5, this rebalancing protocol ensures that the tree height bound holds according to Lemma 4.6.

Besides imbalanced meta-blocks, oversized ones also need maintenance to meet size limits for roots and leaves of the meta-block tree. For a leaf meta-block exceeding K_{SMB} nodes, we select its cut vertex to generate its children in the meta-block tree. For a root meta-block exceeding K_{MB} nodes, all its children are upgraded to roots of separate meta-block trees, and their root hashes are inserted to the master-tree. This policy may bring $O(\log^2 P)$ undersized root meta-block to the master-tree, but is still $O(Q_D)$ space.

Delete. To Delete a batch of strings, we remove the matched trie from the data trie. The process is similar to Insert, by updating the distributed trie blocks first and then the hash value manager.

For block-level deletion, query trie blocks are sent to data trie blocks for local deletion. A special notice is required for mirror nodes (block root replicas), as they should not be deleted until the entire subtree of its representing block root is completely removed. For example in Figure 2, we should not remove the mirror node in block 1 unless both blocks 2 and 3 are removed completely. To solve this, we record each query trie block attempting to completely delete its matching data trie block; then use a parallel leaffix operation to find completely deleted subtrees; finally do the local deletion. Undersized blocks are merged into their parents. Another challenge is that non-critical blocks may also need deletion. Long non-critical blocks are treated as critical, and short ones never completely delete a block, so only those with a critical block as a child need deletion, and the number of those is linear to that of other blocks.

For hash value manager updates, block removal is applied at meta-blocks as node removal. The rebalance protocol is the same as Insert. Undersized internal nodes in the meta-block tree delete all their children, and undersized root meta-blocks are merged into their parent meta-blocks in the master-tree if both are undersized.

Load Balance. In case when a skewed Insert/Delete batch is applied all into the same meta-block, there can be $O(Q_Q/\log^2 P)$ updates to a single meta-block in one round, which is severely load-imbalanced for $\Omega(P \log^5 P)$ batch sizes.

Note that the structure of the meta-block tree is a selective replicated one, as every meta-block tree node caches the information in its subtree in its meta-tree piece. The data on each meta-block can be categorized into: $O(\log^2 P)$ critical information, including its own root hash, pointers and root hashes of child meta-block, and meta-tree pieces for leaves; and $O(P)$ non-critical infor, including meta-tree pieces for internal nodes. If we set the threshold in Algorithm 5 to 0, we get correct trie matching results using only critical information, though with asymptotically more communication.

Our solution to load imbalance is similar to [29] – always keeping critical information up-to-date and delaying non-critical info updates on some meta-blocks with unfinished markers. Whenever the number of unfinished nodes exceeds $P \log P$, additional update rounds are activated until the number drops below. With a bounded number of unfinished nodes, Theorem 4.3 still holds.

Compared to [29], there are two new challenges in PIM-trie. First, reading meta-blocks of $O(P)$ words to CPU for merging/repartitioning causes load imbalance. We solve this by not fetching these blocks directly, but merging their child meta-blocks stored in different PIM modules. Second, while y-fast trie insertions(deletions) take amortized $O(\log w)$ time, they take worst-case $O(w)$, which can cause PIM time imbalance. They can be de-amortized by using a weight balanced tree as the internal binary search tree, and de-amortizing the internal x-fast trie by adding a layer of indirection, borrowing ideas from concurrent data structures.

Theorem 5.1. *For an Insert/Delete trie with size $Q_Q = \Omega(P \log^5 P)$, PIM-tries require $O(Q_Q/P)$ amortized IO time, $O(\log P)$ amortized IO rounds, $O(Q_Q)$ amortized CPU work, $O(\log^2 P + \log w \log P + \log(n_Q l_{QM}))$ span whp, and $O(Q_Q \log w/P)$ amortized PIM time.*

5.3 Subtree Query

A Subtree Query returns a trie including all (key, value) pairs whose key contains the given string as a prefix. Traditional indexes solve them by pointer chasing, which may take $O(n_D)$ IO rounds in the worst case. PIM-tries solve them in $O(\log P)$ IO rounds.

Using trie matching, we find a target node for every non-trivial Subtree Query; its result is the subtree of this node. A primary observation is that if a block (meta-block) root is in the subtree, the whole block is in this subtree. Thus, the result subtree consists of the subtree within the target node's block, and the children of this block. We can fetch and merge all these components by slightly modify the trie matching to return all child block references rather than only the matching block. This takes no additional IO rounds. To ensure load balance, we do not fetch large meta-tree subtrees (containing block references) directly from a single meta-block, but by merging its children in $O(\log P)$ more IO rounds.

Theorem 5.2. *For a Subtree Query trie with size $Q_Q = \Theta(L_Q/w + n_Q) = \Omega(P \log^5 P)$, and result size $Q_R = \Theta(L_R/w + n_R)$, PIM-tries requires $O((Q_Q + Q_R)/P)$ IO time, $O(\log P)$ IO rounds, $O(Q_Q + Q_R)$ CPU work, $O(\log^2 P + \log w \log P + \log(n_Q l_{QM}))$ CPU depth, and $O((Q_Q \log w + Q_R)/P)$ PIM time, with all bounds holding whp.*

6 CONCLUSION

This paper presents *PIM-trie*, the first batch-parallel radix-based index designed for processing-in-memory (PIM) systems. It supports LCP, Subtree, Insert, and Delete on variable-length strings, and simultaneously achieves high load balance, low communication, and low space overhead with good theoretical bounds in the PIM Model, regardless of adversary skew in data and queries. Key techniques introduced in this paper include (1) *trie matching* between a *query trie* and data trie, (2) a *block-wise decomposition* and *selective recursive replication* of the data stored on the PIM side supported by *hashing management*, and (3) a *verification* procedure with differentiated handling of *critical* and *non-critical blocks.* Future work includes designing PIM-friendly algorithms and data structures supported by these key methods (such as suffix trees and graph processing), as well as implementing PIM-trie on real PIM systems.

ACKNOWLEDGMENTS

This research was supported by National Key Research & Development Program of China (2022YFB2404202), Natural Science Foundation of China (62141216), NSF grants CCF-1910030, CCF-1919223, CCF-2028949, and CCF-2103483, VMware University Research Fund Award, Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma), Tsinghua University Initiative Scientific Research Program, and Beijing HaiZhi XingTu Technology Co., Ltd.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).* 105–117. https://doi.org/10.1145/2749469.2750386

[2] Shahanur Alam, Chris Yakopcic, and Tarek M. Taha. 2022. Memristor Based Federated Learning for Network Security on the Edge using Processing in Memory (PIM) Computing. In *2022 International Joint Conference on Neural Networks (IJCNN).* 1–8. https://doi.org/10.1109/IJCNN55064.2022.9891986

[3] Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. 2020. PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly. In *2020 57th ACM/IEEE Design Automation Conference (DAC).* 1–6. https://doi.org/10.1109/DAC18072.2020.9218653

[4] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 295–306. https://www.usenix.org/conference/atc18/presentation/arbel-raviv

[5] 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

[6] Nikolas Askitis and Ranjan Sinha. 2007. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *Computer Science 2007. Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007). Ballarat, Victoria, Australia, January 30 - February 2, 2007. Proceedings (CRPIT, Vol. 62)*, Gillian Dobbie (Ed.). Australian Computer Society, 97–105. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV62Askitis.html

[7] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2009. Monotone Minimal Perfect Hashing: Searching a Sorted Table with O(1) Accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (New York, New York) *(SODA '09)*. Society for Industrial and Applied Mathematics, USA, 785–794.

[8] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2010. Dynamic Z-Fast Tries. In *String Processing and Information Retrieval*, Edgar Chavez and Stefano Lonardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–172.

[9] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2016. Parallel Algorithms for Asymmetric Read-Write Costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 145–156.

[10] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[11] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. 47, 1, Article 3 (apr 2022), 46 pages. https://doi.org/10.1145/3506692

[12] Guy E. Blelloch. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, John Reif (Ed.). Morgan Kaufmann.

[13] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2020), 89–102.

[14] G. E. Blelloch and J. Shun. 2011. A Simple Parallel Cartesian Tree Algorithm and its Application to Suffix Tree Construction. In *ALENEX*.

[15] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999).

[16] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany (LNI, Vol. P-180)*, Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz (Eds.). GI, 227–246. https://dl.gi.de/20.500.12116/19581

[17] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for near-Data Accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 629–642. https://doi.org/10.1145/3307650.3322266

[18] 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.* 321–332.

[19] 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 *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 297–308.

[20] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 231–244. https://doi.org/10.1145/3470496.3527431

[21] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS).* 285–298.

[22] Igal Galperin and Ronald L. Rivest. 1993. Scapegoat Trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Austin, Texas, USA) *(SODA '93)*. Society for Industrial and Applied Mathematics, USA, 165–174.

[23] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 21 (feb 2022), 49 pages. https://doi.org/10.1145/3508041

[24] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS).*

[25] Saransh Gupta and Tajana Šimunić Rosing. 2021. Invited: Accelerating Fully Homomorphic Encryption with Processing in Memory. In *2021 58th ACM/IEEE Design Automation Conference (DAC).* 1335–1338. https://doi.org/10.1109/DAC18074.2021.9586285

[26] Torben Hagerup. 1994. Optimal Parallel String Algorithms: Sorting, Merging and Computing the Minimum. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '94)*. Association for Computing Machinery, New York, NY, USA, 382–391. https://doi.org/10.1145/195058.195202

[27] Jaehoon Heo, Junsoo Kim, Sukbin Lim, Wontak Han, and Joo-Young Kim. 2022. T-PIM: An Energy-Efficient Processing-in-Memory Accelerator for End-to-End On-Device Training. *IEEE Journal of Solid-State Circuits* (2022), 1–14. https://doi.org/10.1109/JSSC.2022.3220195

[28] 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

[29] 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.* 295–306.

[30] 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. *PVLDB* 16, 4 (2022), 946–958. https://doi.org/10.14778/3574245.3574275

[31] Richard M. Karp and Michael O. Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31, 2 (1987), 249–260. https://doi.org/10.1147/rd.312.0249

[32] Donghyuk Kim, Chengshuo Yu, Shanshan Xie, Yuzong Chen, Joo-Young Kim, Bongjin Kim, Jaydeep P. Kulkarni, and Tony Tae-Hyoung Kim. 2022. An Overview of Processing-in-Memory Circuits for Artificial Intelligence and Machine Learning. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 2 (2022), 338–353. https://doi.org/10.1109/JETCAS.2022.3160455

[33] Ji-Hoon Kim, Juhyoung Lee, Jinsu Lee, Jaehoon Heo, and Joo-Young Kim. 2021. Z-PIM: A Sparsity-Aware Processing-in-Memory Architecture With Fully Variable Weight Bit-Precision for Energy-Efficient Deep Neural Networks. *IEEE Journal of Solid-State Circuits* 56, 4 (2021), 1093–1104. https://doi.org/10.1109/JSSC.2020.3039206

[34] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 64. 350–352.

[35] Donghun Lee, Jinin So, MINSEON AHN, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Rebholz Oliver, Vishnu Charan Thummala, Ravi shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. 2022. Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM). In *Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 9 pages. https://doi.org/10.1145/3533737.3535093

[36] 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

[37] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[38] Dai Li, Akhil Pakala, and Kaiyuan Yang. 2022. MeNTT: A Compact and Efficient Processing-in-Memory Number Theoretic Transform (NTT) Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 5 (2022), 579–588.

[39] Xing Li, Rachata Ausavarungnirun, Xiao Liu, Xueyuan Liu, Xuan Zhang, Heng Lu, Zhuoran Song, Naifeng Jing, and Xiaoyao Liang. 2022. Gzippo: Highly-Compact Processing-in-Memory Graph Accelerator Alleviating Sparsity and Redundancy. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 115, 9 pages. https://doi.org/10.1145/3508352.3549372

[40] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-memory Computing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[41] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)* 15, 4 (Oct. 1968), 514–534.

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

[43] Newton, Virendra Singh, and Trevor E. Carlson. 2020. PIM-GraphSCC: PIM-Based Graph Processing Using Graph's Community Structures. *IEEE Comput. Archit. Lett.* 19, 2 (jul 2020), 151–154. https://doi.org/10.1109/LCA.2020.3039498

[44] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. *Proceedings of the IRE* 49, 1 (1961), 228–235. https://doi.org/10.1109/JRPROC.1961.287814

[45] Nick Piggin. 2006. A lockless pagecache in linux—introduction, progress, performance. In *Linux Symposium*. 241.

[46] Sonny Rao, Dominique Heger, and Steven Pratt. 2005. Examining Linux 2.6 page-cache performance. In *Linux Symposium*. 79.

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

[48] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms, 4th Edition*. Addison-Wesley.

[49] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie S. Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. ACM, 273–287. https://doi.org/10.1145/3123939.3124544

[50] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[51] Wonbo Shim and Shimeng Yu. 2022. GP3D: 3D NAND Based In-Memory Graph Processing Accelerator. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 2 (2022), 500–507. https://doi.org/10.1109/JETCAS.2022.3155654

[52] Julian Shun. 2014. Fast Parallel Computation of Longest Common Prefixes. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 387–398. https://doi.org/10.1109/SC.2014.37

[53] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2015. Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.

[54] Keith Sklower. 1991. A Tree-Based Packet Routing Table for Berkeley Unix. In *Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991*. USENIX Association, 93–104.

[55] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-trees with fast, durable updates. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 416–430. https://doi.org/10.1145/3503221.3508441

[56] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970), 73–78.

[57] Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. 2019. Batch-parallel Euler Tour trees. In *SIAM Meeting on Algorithm Engineering and Experiments (ALENEX)*. 92–106.

[58] UPMEM. 2023. UPMEM Technology. https://www.upmem.com/technology/. Accessed January 5, 2023.

[59] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[60] Junpeng Wang, Haitao Du, Bo Ding, Qi Xu, Song Chen, and Yi Kang. 2022. DDAM: Data Distribution-Aware Mapping of CNNs on Processing-In-Memory Systems. *ACM Trans. Des. Autom. Electron. Syst.* (dec 2022). https://doi.org/10.1145/3576196

[61] Zhao Wang, Yijin Guan, Guangyu Sun, Dimin Niu, Yuhao Wang, Hongzhong Zheng, and Yinhe Han. 2020. GNN-PIM: A Processing-in-Memory Architecture for Graph Neural Networks. In *Advanced Computer Architecture*, Dezun Dong, Xiaoli Gong, Cunlu Li, Dongsheng Li, and Junjie Wu (Eds.). Springer Singapore, Singapore, 73–86.

[62] Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inform. Process. Lett.* 17, 2 (1983), 81–84.

[63] Gary R. Wright and W. Richard Stevens. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley.

[64] Ting Wu, Chin-Fu Nien, Kuang-Chao Chou, and Hsiang-Yun Cheng. 2022. RePAIR: A ReRAM-based Processing-in-Memory Accelerator for Indel Realignment. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 400–405. https://doi.org/10.23919/DATE54114.2022.9774715

[65] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (mar 1995), 20–24. https://doi.org/10.1145/216585.216588

[66] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583. https://doi.org/10.1109/HPCA51647.2021.00055

[67] Fan Zhang, Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. 2021. PIM-Quantifier: A Processing-in-Memory Platform for mRNA Quantification. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 43–48. https://doi.org/10.1109/DAC18074.2021.9586144

[68] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1567–1581. https://doi.org/10.1145/2882903.2915222

[69] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053