PimBeam: Efficient Regular Path Queries Over Graph Database Using Processing-in-Memory

Weihan Kong, Shengan Zheng, Yifan Hua, *Student Member, IEEE*, Ruoyan Ma, Yuheng Wen, Guifeng Wang, Cong Zhou, and Linpeng Huang, *Senior Member, IEEE*

***Abstract*—Regular path queries (RPQs) in graph databases are bottlenecked by the memory wall. Emerging processing-in-memory (PIM) technologies offer a promising solution to dispatch and execute path matching tasks in parallel within PIM modules. We present an efficient PIM-based data management system tailored for RPQs and graph updates. Our solution, called PimBeam, facilitates efficient batch RPQs and graph updates by implementing a PIM-friendly dynamic graph partitioning algorithm. This algorithm effectively addresses graph skewness issues while maintaining graph locality with low overhead for handling RPQs. PimBeam streamlines label filtering queries by adding a filtering module on the PIM side and leveraging the parallelism of PIM. For the graph updates, PimBeam enhances processing efficiency by amortizing the host CPU's update overhead to PIM modules. Evaluation results of PimBeam indicate 3.59x speedup for RPQs and 29.33x speedup for graph update on average over the state-of-the-art traditional graph database.**

***Index Terms*—Graph partition, load balance, path matching, processing-in-memory, regular path query.**

I. Introduction

THE rapid increase in the quantity and complexity of graph data has sparked interest in graph databases from both academia [1], [2], [3] and industries [4], [5]. In graph databases, regular path queries (RPQs) [6] are one of the most essential classes of queries. They have been widely used in various fields, including social networks analysis [7], biological networks [8], knowledge graphs [9], and the semantic web [10]. When evaluating RPQs, graph databases return all endpoint pairs of these matched paths.

Received 2 August 2024; revised 20 January 2025; accepted 27 February 2025. Date of publication 4 March 2025; date of current version 7 April 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4500303, in part by the National Natural Science Foundation of China (NSFC) under Grant 62332012, Grant 62227809, and Grant 62302290, in part by the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project under Grant 2021SHZDZX0102, and in part by the Natural Science Foundation of Shanghai under Grant 22ZR1435400. Recommended for acceptance by D. Li. *(Corresponding authors: Shengan Zheng; Linpeng Huang.)*

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

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

Digital Object Identifier 10.1109/TPDS.2025.3547365

Unfortunately, RPQs on traditional graph databases face bottlenecks due to the memory wall [11]. Processing RPQs involves extensive pointer chasing operations, which trigger numerous irregular and unpredictable memory access patterns when accessing neighboring nodes. This leads to a low cache hit rate and frequent DRAM memory access. The excessive data movement between DRAM and cache results in high access latency and high bandwidth consumption, not only limiting system performance but also incurring considerable energy costs.

The emerging processing-in-memory (PIM) technology [12], [13], [14], [15] is expected to solve this bottleneck. By embedding processors within memory modules, PIM technology allows computations to be performed closer to where the data resides. Specifically, PIM modules can perform data-intensive computations within memory modules that integrate computational resources, reducing data movement to the host CPU for processing, thereby accelerating task completion. This enhances the performance of applications with high data intensity or poor cache locality [16]. PIM systems have been widely used in the fields of graph analysis [17], [18], [19] and database indexing [20], [21], [22].

However, PIM systems encounter significant challenges when executing RPQs on graph databases. The first challenge is how to partition the graph precisely to achieve load balancing among PIM modules [23]. Graph partitioning plays a critical role in a wide range of applications, including distributed graph processing [24], [25], graph neural networks [26], and parallel computation frameworks [27]. Similarly, PIM scenario also requires well-designed graph partitioning algorithm to fully utilize the hardware's computational potential. Moreover, many real-world graphs [28], [29], [30] exhibit varying degrees of skewness, characterized by few high-degree nodes and many low-degree nodes. High-degree nodes demand more computing and bandwidth resources due to their extensive neighborhoods. As a result, PIM systems often experience load imbalances, where some PIM modules are overloaded with high-degree nodes while others remain underutilized. The second challenge is high communication overhead in PIM-based graph databases [31], which includes CPU-PIM communication (CPC) and inter-PIM communication (IPC). CPC overhead arises from task distribution and collection between PIM modules and the host. IPC overhead stems from data interaction between PIM modules, which in turn depends on the effectiveness of the partitioning algorithm. If the partitioning algorithm maintains good locality of the graph, IPC overhead can be significantly reduced; otherwise, numerous

next-hop queries across different PIM modules will result in high IPC costs. The third challenge is managing dynamic graphs with constant insertion and deletion of nodes [32], [33], requiring the graph storage engine to efficiently deal with frequent updates. Additionally, changes in nodes affect graph locality, necessitating reevaluation of graph locality and adjustment of node partitioning.

This paper proposes PimBeam, a PIM-based efficient graph database management system. PimBeam significantly accelerates path matching and graph update operations by leveraging the parallel capabilities of PIM modules. It employs a novel graph partitioning algorithm that utilizes known node degree information to maintain graph locality while achieving load balancing among PIM modules through dynamic capacity constraints. Instead of using a hash function to randomly assign graph nodes to PIM modules, we track and record previous partitioning decisions, optimizing subsequent allocations. This approach achieves more precise graph partitioning while maintaining low partitioning overhead.

To cope with the diversity of regular paths, PimBeam cleverly utilizes the parallelism of the PIM side by setting the filtering mechanism in PIM modules. To handle different types of workloads, PimBeam adopts a labor-division strategy that fully exploits the advantages of both the host CPU and PIM modules. Specifically, high-degree nodes with good locality access patterns are assigned to the host CPU, while the remaining low-degree nodes are allocated to PIM modules. The threshold for high-degree nodes can be adaptively adjusted based on the dataset. Thus, the PIM modules can overcome the load imbalance issue that stems from graph skewness by avoiding the high-degree nodes. To address changes in graph locality caused by continuous insertion and deletion of nodes, we propose a node migration scheme. This scheme enhances graph locality by migrating nodes with unreasonable partitions between PIM modules under the new state.

For graph updates, high-degree nodes are more likely to be frequently updated, consuming more computing resources. To tackle this challenge, PimBeam employs a heterogeneous graph storage for high-degree nodes, delegating complex update operations to PIM modules. The CPU side, based on synchronizing updates from the PIM modules, handles query operations with good cache locality.

To summarize, we make the following contributions:

1) We propose PimBeam, a graph database management system based on the PIM architecture that supports efficient batch processing of typical RPQs, extended RPQs with filtering labels, and graph updates through the collaboration of the PIM side and the host side.
2) We introduce a PIM-friendly dynamic graph partitioning algorithm that utilizes node degree information to handle graph skewness with low overhead while maintaining graph locality. Through this partitioning, PimBeam addresses load imbalance and communication bottlenecks during execution of RPQs in the PIM system.
3) We design a heterogeneous graph storage scheme for high-degree nodes, achieving rapid graph updates by amortizing the update cost of high-degree nodes on the host side to the PIM side.
4) We implement PimBeam on the commodity PIM system UPMEM [34]. Compared to the state-of-the-art traditional graph database RedisGraph [35], PimBeam is significantly faster, achieving speedup of 3.59x and 29.33x in RPQs and graph updates on average, respectively. Compared to other partitioning algorithms, PimBeam achieves a good balance between partitioning time and preserving graph locality.

II. Background

A. Regular Path Queries and Pointer Chasing

A regular path query seeks pairs of endpoints connected by a path that matches a specified regular expression [36]. A typical RPQ starts from a fixed node and queries several hops. More generally, for a directed graph with labels on its edges, a regular path query is just a regular expression over the set of edge labels. In some studies, the regular expression can also be extended to the labels of nodes [37], [38]. The answers to RPQs consist of endpoint node pairs connected by a path that conforms to the given regular expression.

Pointer chasing is a fundamental operation in many data structures such as linked list, skiplist [39], tree [40], and graph [41], [42]. For graph databases, the overhead of pointer chasing becomes a performance bottleneck. Because the nodes linked by pointers are scattered throughout memory, the actual storage locations of two logically adjacent nodes may be far apart. This results in a large number of memory accesses being required to find the next hop, leading to a low cache hit rate.

B. Hybrid Computing Systems and PIM Architecture

Hybrid computing systems have recently become widely adopted in high-performance computing (HPC) field. These systems integrate diverse computing units, such as CPUs, GPUs, and FPGAs, to capitalize on their distinct computational capabilities. To fully harness the strengths of such hybrid systems, considerable efforts have been devoted to adapting applications across various domains for execution on such heterogeneous architectures. For instance, Tallada et al. [43] proposed a runtime system to enhance CPU-GPU interoperability. Similarly, the Hybrid KNN-join algorithm [44] leverages both CPU and GPU features to optimize nearest neighbor searches, while parallel radix sorting algorithms designed for GPUs [45] demonstrate substantial improvements in data processing efficiency. All these systems typically optimize workload division by leveraging the strengths of each component in hybrid computing system.

The PIM architecture, as a form of hybrid computing system, provides a novel approach to enhancing the efficiency of pointer chasing by enabling its execution directly within memory devices. The PIM architecture comprises two components: the host side and the PIM side. The host side features a powerful CPU with multiple cores, large caches and DRAM, while the PIM side includes multiple PIM modules. Each PIM module contains an on-bank general-purpose processor with limited computing power, alongside smaller caches and local DRAM. Positioned closer to the memory, PIM processors access data faster than CPU cores. Moreover, PIM processors facilitate

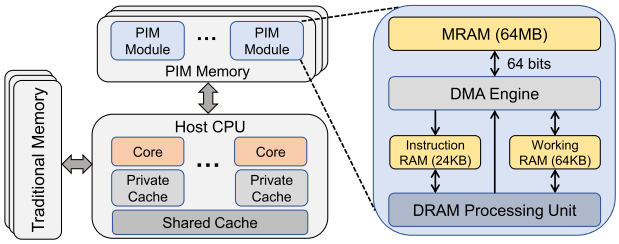
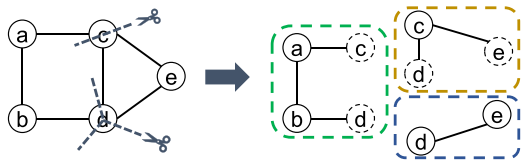
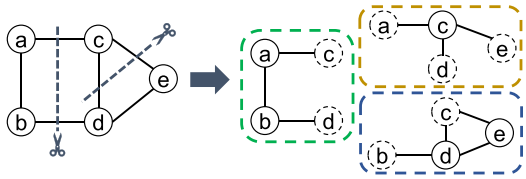


Fig. 1. UPMEM's architecture.



(a) Vertex partitioning. The number of edges after partitioning remains the same as the original graph.



(b) Edge partitioning. Each partition has to store the cut edges.

Fig. 2. Two ways of partitioning graph.

a range of interactions between the host CPU and the PIM modules, allowing the host CPU to transmit executable code to the PIM module, launch the code, and exchange data with the host. Previous work [20] has revealed that the two components of the PIM architecture are suited to different workloads. The host side with a traditional CPU handles skewed workloads optimally because the temporal and spatial locality in these workloads improves cache utilization efficiency. On the other hand, the PIM side with numerous but individually weak computational modules is well-suited for tasks with uniform distribution, but falters under skewed workloads.

Currently, UPMEM [34] is the only commercially available PIM product, featuring processing-in-memory DIMMs. Its architecture is presented in Fig. 1. Each PIM module has an on-bank processor (DRAM Processing Unit) and 64 MB of local memory (MRAM). A machine can support up to 20 UPMEM DIMMs, featuring 2560 PIM modules. In this setup, IPC is achieved by utilizing the CPC through CPU forwarding data. However, the bandwidth of IPC is expensive. Despite a system using 16 UPMEM DIMMs providing an intra-PIM bandwidth of 1.28 TB/s, the bandwidth for IPC communication is less than 2% of the intra-PIM bandwidth, approximately 25 GB/s [46].

C. Graph Partition

Graph partitioning includes two main approaches: vertex partitioning and edge partitioning [47], as shown in Fig. 2. Vertex partitioning aims to divide the set of vertices in a graph into several parts, where source vertex and its connected edges are stored together. Edge partitioning, on the other hand, separates the source vertex from its edges, which makes it appear as if the vertex is being cut. Our primary concern lies in vertex partitioning because in most real-world graphs, node degrees often follow a power-law distribution [48]. This means most vertices have a small number of edges, while only a few vertices have very high degrees. Distributing the edges of a vertex across multiple computing nodes will introduce significant additional communication overhead, diminishing the benefits of parallelism. In fact, many distributed database architectures [49], [50] also adopt vertex partitioning.

A common partitioning algorithm is the hash method. The hash partitioning method uses a hash function to compute a hash value for each vertex and maps it to a subgraph. Since the computation of a hash function is $O(1)$, the complexity of the hash partitioning algorithm is $O(|V|)$ for a graph with V vertices. Additionally, the hashing algorithm requires no extra storage space. Although the uniform distribution of the hash function's output ensures load balancing for each subgraph, it lacks consideration for graph locality.

Therefore, many partitioning algorithms for preserving graph locality have been proposed in recent years [25], [51], [52]. There are two prominent solutions: the greedy method and the adaptive method.

Linear Deterministic Greedy (LDG) [25] is a good representative of the greedy method. LDG uses a greedy heuristic that assigns a vertex to the partition containing most of its neighbors and imposes constraints on partition loads. This method achieves relatively balanced load distribution while preserving graph locality. However, the vertex assignment process could be time-consuming. It involves traversing each vertex's neighbors, computing the objective function for placing the vertex in each partition, and selecting the optimal partition for actual assignment. Moreover, LDG is unsuitable for dynamic graphs, as it requires prior knowledge of graph structure such as vertex and edge counts. Fennel [51] is an improvement over LDG designed for dynamic graph partitioning. It adopts a more sophisticated scoring function to achieve a trade-off between neighbor co-location and load balance, with the linear load penalty term replaced by a power-law relationship. However, the algorithm does not reduce the partitioning time complexity, and it involves many parameters, making the search for the optimal partitioning scheme potentially laborious.

For the adaptive method [52], new vertices are randomly assigned to computing nodes according to a hash function. It improves graph partitioning in a scalable manner through decentralized, iterative vertex migration, generating partitions with good locality while maintaining balance in terms of structural changes. This solution supports dynamic graph partitioning but incurs a significant communication burden during frequent vertex migrations.

D. Graph Database

A graph database uses the property graph model [53] to represent graph data. In this model, nodes represent distinct entities, and edges represent relationships between pairs of entities. Depending on the scenario, edges can be directed (such as in

a citation graph) or undirected (such as in a social relationship graph). Both nodes and edges can have labels and property-value pairs to describe their properties. When handling RPQs, paths composed of entities and relationships are treated as first-class citizens, and labels and property-value pairs can be seen as extensions. We can use an adjacency matrix to represent entities and their relationships in a directed property graph model.

While the Section II-C discusses various graph partitioning methods based on graph topology, the challenges of graph storage and database partitioning emerge after the graph has been partitioned. Even with an optimized graph partitioning scheme, managing and storing partitioned graphs in a distributed database system poses inherent challenges. These primarily arise from the metadata and communication overhead associated with partitioned subgraphs, as well as the necessity for effective database partitioning strategies to efficiently store, query, and update these subgraphs. Among the currently available distributed graph database products, there are two graph partitioning solutions:

1) *Primary-secondary replication:* Primary-secondary replication refers to replicating data from a primary database (primary) to one or more secondary databases (secondary). The most popular graph database, Neo4j [54], adopts this solution. In this approach, each computing node stores the global graph. Nevertheless, the PIM module is constrained by limited local memory capacity, which renders the storage of the complete global graph nearly unfeasible.
2) *Hash partition:* This is a widely used method that assigns graph nodes to computing nodes based on a consistent hash function. Representative distributed graph databases include G-Tran [2] and ByteGraph [4]. Although an excellent hash function can balance the number of nodes assigned to each PIM module, this random partitioning method ignores the locality within the graph. The nodes assigned to a single PIM module may have little relation to each other. This can lead to severe load imbalance between PIM modules during RPQs, resulting in high IPC overhead.

E. Matrix-Based Graph Operations

Most real-world graphs are sparse rather than dense [55], meaning that the majority of potential edges are absent. Consequently, graphs can be efficiently represented using sparse matrices, and graph algorithms can be implemented through operations on these sparse matrices. In graph database scenarios, graph pattern matching can be translated into a set of matrix multiplications. The GraphBLAS [56] is an established mathematical framework that defines a core set of graph operations based on matrices. These operations can be leveraged to implement a wide range of graph algorithms, enabling GraphBLAS to efficiently analyze complex graph structures. Due to the high serial and parallel processing performance of matrix-based operations, RedisGraph uses GraphBLAS to support efficient graph queries [35].

Similar to RedisGraph, PimBeam's execution plan can be abstracted into graph operations based on matrices, aiming to exploit the parallelism of PIM modules by leveraging the natural parallelism of matrix operations. Here is a simple path query example: if we want to find nodes that are 2-hops away from fixed source nodes (the batch 2-hop path query in Fig. 5(a)), the matrix-based execution plan would be $ans = Q \times Adj \times Adj$, where Q is a matrix containing information about source nodes, adj stands for the adjacency matrix of the graph, and ans is a matrix containing information about destination nodes. The rows of the Q identify a query in a batch of queries, and the columns represent source nodes. The rows of the ans identify a query in a batch of queries, and the columns represent destination nodes.

III. Design

In this section, we introduce the design of the PimBeam, supporting efficient batch RPQs and graph updates based on matrix operations. The key design is a PIM-friendly graph partitioning algorithm. It aims to minimize communication overhead, ensure load balancing during path matching, and feature a relatively low partitioning time.

A. System Architecture

Fig. 3 outlines the overall design of the PimBeam. PimBeam adopts an adjacency matrix to represent a property graph and uses sparse matrix-based operations to query the graph. The graph is partitioned across the host side and the PIM side. By leveraging both sets of resources, PimBeam achieves efficient RPQs and graph updates. As shown in Fig. 3, the main components of PimBeam include the Query Processor, the Storage Module, and the PIM Modules.

1) The Query Processor: When processing batch RPQs and graph updates, the Query Processor generates execution plans composed of matrix-based operators, and dispatches these operators to PIM modules for processing. A RPQ will be translated into a `smxm` operator for path matching and a `reduce` operator for reducing the result. A graph update is abstracted into `add` operator and `sub` operator. We identify the target PIM module for the given task associated with the specific node through the `node partition vector`. Similar to the map-reduce model, the matrix-based operators are mapped to P PIM modules for execution. These inherently parallel tasks take advantage of the high parallel intra-PIM bandwidth.

2) The Storage Module: To fully exploit the potential of the PIM architecture and achieve efficient path matching, the graph should be disjointly partitioned across multiple computing nodes. Specifically, P PIM modules are responsible for the storage and querying of low-degree nodes, while high-degree nodes are managed collaboratively by both PIM modules and the host CPU (detailed in Section III-D). As depicted in Fig. 3, when processing graph updates, if a vertex appears for the first time in the edge insertion stream, the *Graph Partitioner* will recognize it as a new node and make partitioning decisions based on *Graph Partition Info* and *PIM Balanced Info* (detailed in Section III-B). The partitioning result is stored in the `node partition vector`, allowing subsequent queries to quickly locate which PIM module the graph node belongs to. In addition, the Node

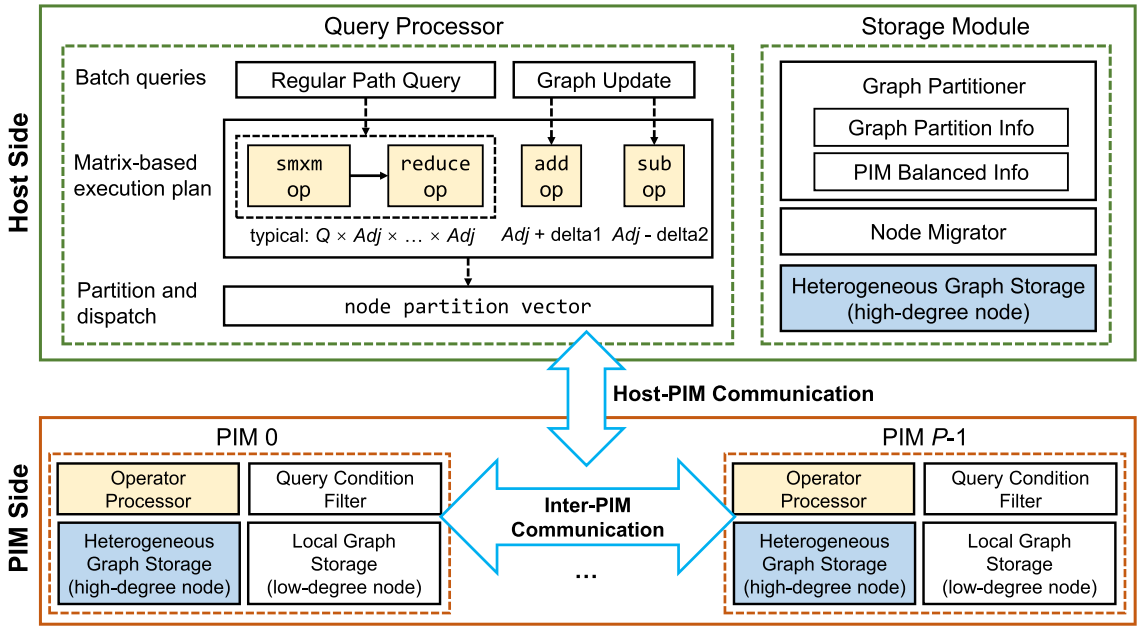


Fig. 3. The architecture of the PimBeam with P PIM modules.

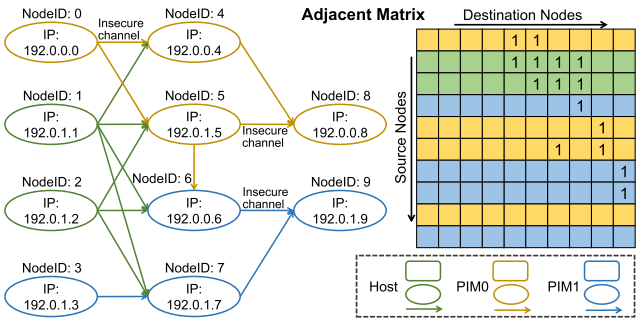


Fig. 4. Example of partitioning a routing connection graph in property graph and adjacency matrix view. The label of the node is the network segment, with two subnets 192.0.0.* and 192.0.1.*. The label of the edge is the channel security. The channels from nodes 0 to 4, 5 to 8, and 6 to 9 are insecure, while others are secure.

Migrator is responsible for relocating new high-degree nodes to the host side and migrating incorrectly partitioned nodes to appropriate partitions.

3) The PIM Modules: Each PIM module contains an Operator Processor, a Query Condition Filter, a Local Graph Storage Module, and a Heterogeneous Graph Storage Module.

The Operator Processor is responsible for parsing and processing operators received from the host CPU. For the `add` and `sub` op, the Operator Processor performs relevant updates on nodes, edges, and properties. For the `smxm` op, the Operator Processor produces intermediate results after execution. These intermediate results then pass through the Query Condition Filter, which can filter them based on certain labels or property values. For the `reduce` op, the Operator Processor needs to remove duplicate results and then return the final results to the host side.

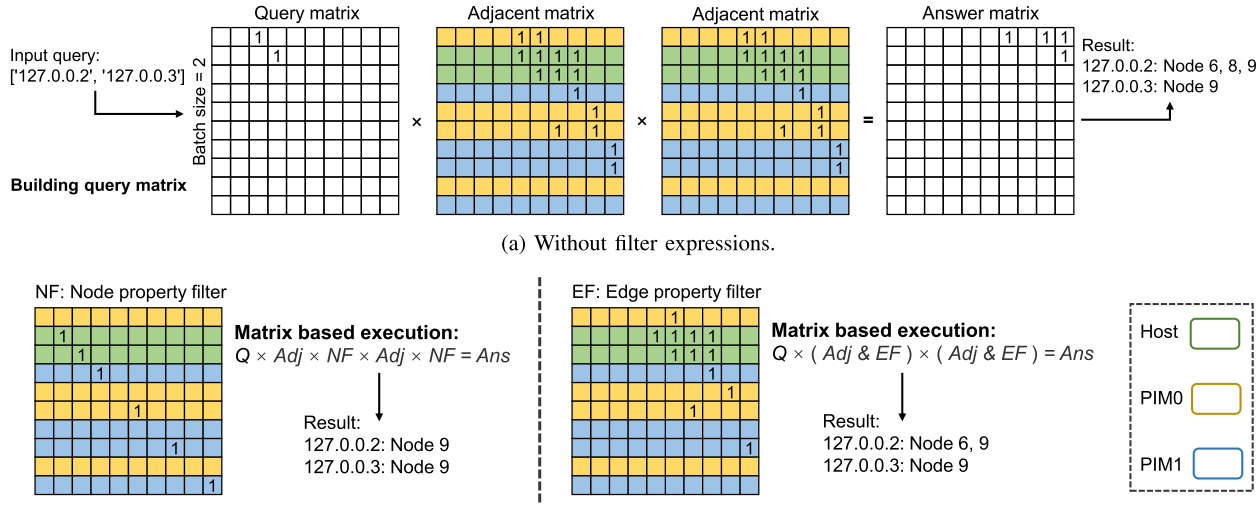
According to the Fig. 4, the adjacency matrix is partitioned by row across $1 + P$ computing nodes, with each computing node maintaining a segment of the adjacency matrix. Due to the good concurrency and scalability of the hash map, each PIM module uses a hash map to store the corresponding segment of the adjacency matrix in the Local Graph Storage. The hash map stores a mapping from row IDs (NodeID) to row data (the columns, which are the next-hop NodeIDs). However, simple local graph storage cannot meet the demands of frequent updates from the host side. By introducing heterogeneous graph storage, the storage of adjacency matrix segments maintained by the host CPU is optimized.

In the remainder of this section, we will provide a detailed description of the PIM-friendly dynamic graph partitioning algorithm (Section III-B), extended RPQs with label filtering (Section III-C), and optimizations for graph storage (Section III-D).

B. Graph Partition

PimBeam prioritizes graph partitioning to achieve load balancing among PIM modules with low overhead while maintaining graph locality. In this section, we propose a PIM-friendly graph partitioning algorithm consisting of the *labor-division* method and the *multi-neighbor adaptive* method. For the labor-division method, PimBeam handles high-degree and low-degree nodes differently to address graph skewness and leverage the strengths of both the host CPU and the PIM modules. For the multi-neighbor adaptive method, PimBeam attempts to allocate nodes to the same PIM module as their highest-degree neighbor with low overhead, aiming to reduce the IPC overhead during path matching. Additionally, we developed a *load balancing* strategy to prevent certain PIM modules from being overloaded.

1) Labor-Divison Method: To address load imbalance caused by graph skewness and leverage the strengths of the host side and the PIM side, we propose a labor-division approach that the host side handles high-degree nodes and the PIM side deals



(a) Without filter expressions.

(b) With filter expressions. The left side shows the node property filter and the query only through subnet 192.0.1.*. The right side shows the edge property filter and the query without passing through the insecure channel.

Fig. 5. The matrix-based execution plan of a batch 2-hop path query.

with low-degree nodes. As depicted in Fig. 4, the out-degrees of nodes 1 and 2 are relatively high, so they are assigned to the CPU for processing. The remaining nodes are partitioned into PIM0 and PIM1.

PimBeam leverages the strengths of both the host side and the PIM side. The two components of the PIM architecture, the host side and the PIM side, have different preferences for workloads. The distributed PIM side prioritizes workloads with uniformly random memory access to exploit high parallel intra-PIM bandwidth, while the host side, with its powerful CPU, prefers continuous and skewed memory access workloads with good locality. Accordingly, low-degree nodes with fewer neighbors are likely to be accessed randomly, satisfying the PIM side's preference. High-degree nodes tend to be accessed more frequently, and retrieving the NodeIDs of their numerous next-hop nodes exhibits sequential memory access, satisfying the host side's taste. Therefore, the labor-division method aligns perfectly with the PIM architecture. Through dispatching path matching tasks associated with high-degree nodes to the CPU side and low-degree nodes to the PIM side, the PimBeam can leverage the advantages of both sets of resources.

The degree of nodes in the graph is not constant. As the graph grows, a low-degree node might gain more connections and become a high-degree node. PimBeam can smartly identify and migrate high-degree nodes to the host side using the Node Migrator, ensuring a disjoint workload distribution tailored to the capabilities of the PIM side and the host side. We have designed a high-degree node identification scheme, with a default high-degree threshold Th set at 16. Nodes with a degree below Th have lower access frequencies and are suitable for handling by PIM modules. During the initial establishment or update phase of the graph, if the average degree of all nodes in the graph exceeds 4, this graph is considered dense. At this point, we iteratively increase Th by 16 and measure the completion time of the query task until we find the optimal threshold or until only the top 0.10% highest-degree nodes remain on the host side. During this process, the Node Migrator will migrate new high-degree nodes from the PIM side to the host side. This approach retains high-degree nodes on the host side while limiting the number of nodes assigned to the host. It allocates relative low-degree nodes in the dense graph to the PIM modules, preventing them from idling. Since PIM modules are now allocated a moderate workload and no longer process high-degree nodes, the load imbalance caused by graph skew naturally dissipates.

2) Multi-Neighbor Adaptive Method: To maintain graph locality with low overhead and support dynamic graph changes, we propose a multi-neighbor aware partitioning method for assigning low-degree nodes among PIM modules and an adaptive method for node migration to enhance locality.

We first introduce our multi-neighbor aware partitioning method. When a new graph node is received, we examine its existing neighbor nodes in the current batch, select the neighbor with the highest degree located in a PIM, and assign the new node to the same PIM module. If, during the neighbor traversal, it is determined that the number of the new node's neighbors exceeds the high-degree threshold, the node is assigned to the host side. If no neighbors are found for the node, a hash algorithm is utilized to randomly assign it to a PIM module. This approach allows for node assignment upon the insertion of the first edge, rather than waiting until the entire graph is constructed, thus meeting the requirements of dynamic graph partitioning in graph databases.

The time complexity of our method is relatively low. Considering a straightforward greedy algorithm, it finds the first neighbor of a given node. The time complexity $Cost$ for partitioning a graph with n nodes is $O(n)$. The multi-neighbor aware partitioning method appears to have a significantly higher time complexity at first glance. But since the number of neighbors a node traverses is at most equal to the threshold th, the added time complexity is no more than $th \times Cost$. Therefore, the time complexity of our approach is of the same order as that of the greedy method.

When performing RPQs, PimBeam employs an adaptive method to augment graph locality that might be compromised for low partitioning overhead. During path matching, PIM modules

simultaneously detect incorrectly partitioned nodes that miss most next-hop nodes in the local PIM module (with a next-hop hit rate below 25%), effectively overlapping detection overhead with path matching query processing. The host CPU then migrates these nodes to the partition containing the largest number of their neighbors among all partitions. With more neighbors on the same module, PimBeam further enhances graph locality.

It is noteworthy that the labor-division approach (Section III-B1) facilitates easier partitioning of the graph on the PIM side, because high-degree nodes have already been migrated to the host side. Ideally, a graph without high-degree nodes would fragment into multiple disconnected subgraphs. In most scenarios, our multi-neighbor aware partitioning method effectively maintains graph locality among PIM modules, leaving only a few nodes requiring migration. Ultimately, the host CPU only needs to deal with a small amount of migration overhead.

3) Load Balancing: Based on the multi-neighbor aware partitioning method, we record the load information of PIM modules during partitioning, facilitating load balancing among PIM modules.

During the allocation of graph nodes, we use dynamically allocated node capacity constraints to enforce load balancing among PIM modules. We first determine a $node\_bound$, which is the ideal number of nodes in each PIM for the system under universal workloads (default is 8192in our system). The dynamic capacity constraint is set to 1.05x the average number of assigned nodes among PIM modules. When the average number of nodes per PIM module exceeds the $node\_bound$, we gradually increase the constraint until it reaches a maximum of 1.10x the average number. The constraint value is re-evaluated after each batch update. We point out that decreasing the proportion of the capacity constraint can facilitate load balance but at the expense of decreased graph locality.

Under the load balancing strategy mentioned above, PimBeam may encounter some new nodes that cannot be assigned to their target PIM modules due to exceeding constraints. When this occurs, we apply our multi-neighbor aware partitioning method among the remaining PIM modules that have not exceeded the capacity constraint. This approach avoids assigning most of the graph nodes to a few PIM modules, thereby preventing load imbalance, while keeping a reasonable proportion of subgraph locality.

Eventually, our PIM-friendly graph partitioning algorithm achieves load balance among PIM modules and maintains graph locality with low overhead and low partitioning time. Additionally, it is flexible enough to support dynamic graphs in graph databases.

C. Query With Label Filtering

Simply performing path queries from a fixed start node in a breadth-first search manner is only a basic part of RPQs, and the forms of regular paths can be very diverse. A common extension to regular path queries involves filtering by edge labels and node labels, as illustrated in Fig. 5(b). By incorporating node property filter matrices or edge property filter matrices into the query process, path matching with filtering conditions can be easily extended based on a matrix-based approach. PimBeam stores property values and entities together, transforms complex properties into simple values through mapping, and leverages the inherent advantages of the PIM side to achieve efficient query filtering.

For graphs with labels and property values, PimBeam adopts a method where property values and entities are stored together. This allows for seamless access to properties when finding nodes or edges during the next-hop discovery process. Property values can be compared with filtering conditions to determine whether the current edge or node matches the query criteria. For complex property values, PimBeam maps them into numerical values, allowing PIM modules to focus solely on basic path searches and simple condition comparisons. In this manner, the entire filtering process only requires the time for condition comparison in PimBeam, eliminating the need for additional property value lookup time.

PimBeam's filtering operation can reduce the communication overhead. Regardless of filtering labels of nodes or edges, each hop of path matching yields fewer intermediate results compared to the original queries. Moreover, as the query hops increase, the number of filtered-out nodes grows exponentially, leaving relatively few results. Similarly, after the PIM modules complete the reduction, the host side will receive fewer final results from the PIM side. Therefore, in the case of label filtering queries, the CPC and IPC overheads are reduced. This avoids the disadvantage of lower CPC and IPC bandwidth in the PIM architecture, making the PIM modules highly suitable for handling label filtering queries.

By storing data efficiently and leveraging the advantage of reduced communication overhead through filtering, PimBeam is ideally suited for handling queries with label filtering and can effectively manage complex RPQs.

D. Heterogeneous Graph Storage

In PimBeam, every computing node needs to manage graph nodes assigned to it and maintain an adjacent matrix segment to store these graph data. PIM modules can leverage high parallel intra-PIM bandwidth to provide high performance for querying and updating low-degree nodes. While high-degree nodes on the host side can benefit from cache locality, efficiently accessing them still poses a challenge. Furthermore, updating high-degree nodes remains difficult due to the substantial memory management involved. This challenge exerts significant pressure on the host CPU. To alleviate this load, PimBeam uses heterogeneous graph storage for high-degree nodes, achieving efficient query and update simultaneously by amortizing the host side's update cost to the PIM side.

When querying high-degree nodes, PimBeam can effectively leverage cache locality. On the host side, the most efficient approach for graph querying is to store the next-hop data of high-degree nodes in a contiguous memory array. Fig. 6 demonstrates our organization of high-degree nodes. For each of these nodes, we allocate an array *cols_vector* to store the next-hop NodeIDs, with unused positions filled with -1. Consequently,

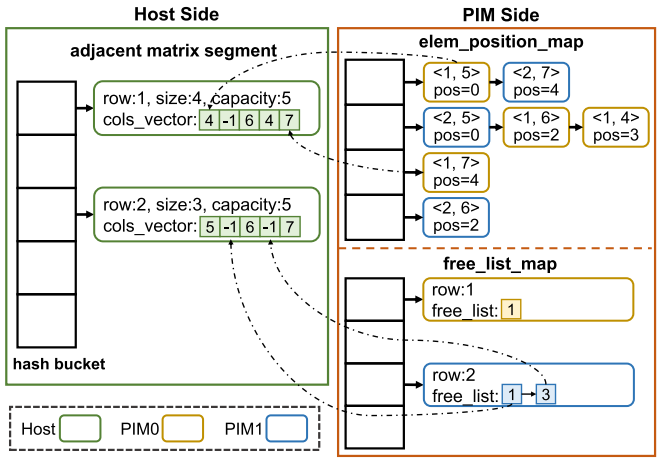


Fig. 6. Heterogeneous graph storage for high-degree nodes.

when accessing a high-degree node, PimBeam needs only a single memory fetch to obtain its next-hop data, greatly enhancing memory access locality on the host side.

When inserting or deleting an edge where the row is a high-degree node, PimBeam must traverse corresponding *cols_vector* to determine if the edge already exists. To avoid time-consuming traversals, PimBeam maintains two supplementary hash maps on the PIM side for storing high-degree nodes: *elem_position_map* and *free_list_map*, as depicted in Fig. 6. The *elem_position_map* in PIM modules stores the mapping from the edge to its position in *cols_vector* on the host side. We can effectively distribute the edges originating from high-degree nodes across different PIM modules using their hash values, thus preventing any PIM module from becoming overloaded. The *free_list_map* stores free positions in *cols_vector*. For each insertion or deletion of a $\langle \text{row}, \text{col} \rangle$ tuple where a high-degree node is the row, we need to pop an available position from or push a newly freed position back into the *free_list_map*.

For example, inserting an edge $\langle 1, 2 \rangle$ as shown in Fig. 6 follows these steps: First, the edge $\langle 1, 2 \rangle$ is hashed and assigned to PIM 0. PIM 0 queries its *elem_position_map* to confirm that the edge does not exist. Then, *free_list_map* pops an available position 1, and it is returned as *pos* to the host side. Next, PIM 0 updates *elem_position_map* by inserting ($edge = \langle 1, 2 \rangle$, $pos = 1$). Finally, the host CPU writes 2 to position 1in the *cols_vector* of row 1. This way, for graph updates, the host CPU only needs to perform the simple task of writing data to a specific position within the *cols_vector*, while the PIM side undertakes complex operations such as edge retrieval and space management.

Ultimately, this forms a heterogeneous graph storage for high-degree nodes, where the CPU side utilizes sequential access for memory locality, and the PIM side handles the majority of the update expenses.

IV. Evaluation

A. Experiment Setup

1) Datasets: We evaluate our PimBeam using 15 datasets from the Stanford Network Analysis Platform (SNAP) [57] and a social network graph with properties from the Linked Data Benchmark Council Social Network Benchmark (LDBC SNB) [58]. These datasets are widely recognized in prior research on graph databases [59], [60], [61], offering diverse graph structures and realistic properties for evaluation. Given the definition of RPQs, we treat all the datasets as directed graphs.

SNAP [57] is a general-purpose network analysis and graph mining library that includes many real-world graphs from various domains, such as citation networks, web graphs, product co-purchasing networks, and road networks. Trace IDs #1-#15 in Table I show these 15 large-scale graphs from SNAP, each with more than 200 K nodes.

LDBC SNB [58] is a benchmark designed to evaluate graph database performance in social network scenarios, supporting the generation of property graph datasets of varying sizes. We generated a dataset with the scale factor set to 30, which has 157,444 nodes and 6,017,657 edges, making it a fairly dense graph compared with the SNAP datasets. We selected the node label *language* and the edge label *location* for label filtering RPQ experiments. The *language* label has 71 different values, and the *location* label has 1,343 different values.

2) Compared Systems: We select RedisGraph [35] as the baseline system for the pure CPU approach. RedisGraph is the first queryable property graph database that uses matrices to represent adjacency matrices and leverages linear algebra for querying graphs. It provides a fast and efficient way to store, manage, and process graphs [62]. To the best of our knowledge, RedisGraph is the only state-of-the-art baseline focus on accelerating RPQs for comparison.

We select PIM-hash, PIM-greedy, PIM-LDG, and PIM-only as comparative systems for PIM approaches. The PIM-hash system distributes low-degree nodes in the graph to PIM modules using a hash function, as hash partitioning schemes are widely adopted in distributed graph databases [2], [4]. The PIM-greedy system employs a radical greedy method proposed by [63]. Specifically, when a new node is received, it is assigned to the PIM module of its first neighbor. If the first neighbor remains unassigned, hash partitioning is used. The PIM-LDG system adopts the partitioning strategy from LDG [25], which considers the weighted degree information of all neighboring nodes within each PIM module. New nodes are assigned to the PIM module with the highest score, while the module capacity is capped at 1.1x the average number of nodes. The PIM-only system adopts the proposed dynamic graph partitioning method but excludes the involvement of the host CPU for high-degree nodes. In contrast, all other compared systems employ our proposed labor-division approach, in which highly connected nodes are handled by the CPU, and low-degree nodes are allocated to the PIM side.

3) Configuration: We conduct our experiments on a dual-socket server equipped with two Intel(R) Xeon(R) Silver 4216 CPUs and 20 UPMEM DIMMs. Each CPU has 16 cores, operates at 2.10 GHz, and features a 22 MB L3 cache. The server has six memory channels per socket: two DDR4-3200 64 GB memory modules occupy one channel, while the remaining five channels are populated with ten UPMEM DIMMs. Each UPMEM DIMM has two ranks, each containing 64 PIM modules,

TABLE I
THE REAL-WORLD GRAPHS FROM SNAP DATASET USED IN OUR EXPERIMENTS

Name	roadNet-CA	roadNet-PA	roadNet-TX	cit-patents	com-youtube	com-DBLP	com-amazon	wiki-Talk
Trace ID	#1	#2	#3	#4	#5	#6	#7	#8
Nodes	1,965,206	1,088,092	1,379,917	3,774,768	1,134,890	317,080	334,863	2,394,385
High-degree nodes (%)	0	0	0	0.10	2.07	3.10	0.62	0.27
Name	email-EuAll	web-Google	web-NotreDame	web-Stanford	amazon0312	amazon0505	amazon0601	
Trace ID	#9	#10	#11	#12	#13	#14	#15	
Nodes	265,214	875,713	325,729	281,903	262,111	410,236	403,394	
High-degree nodes (%)	0.29	1.29	2.86	1.12	0	0	0	

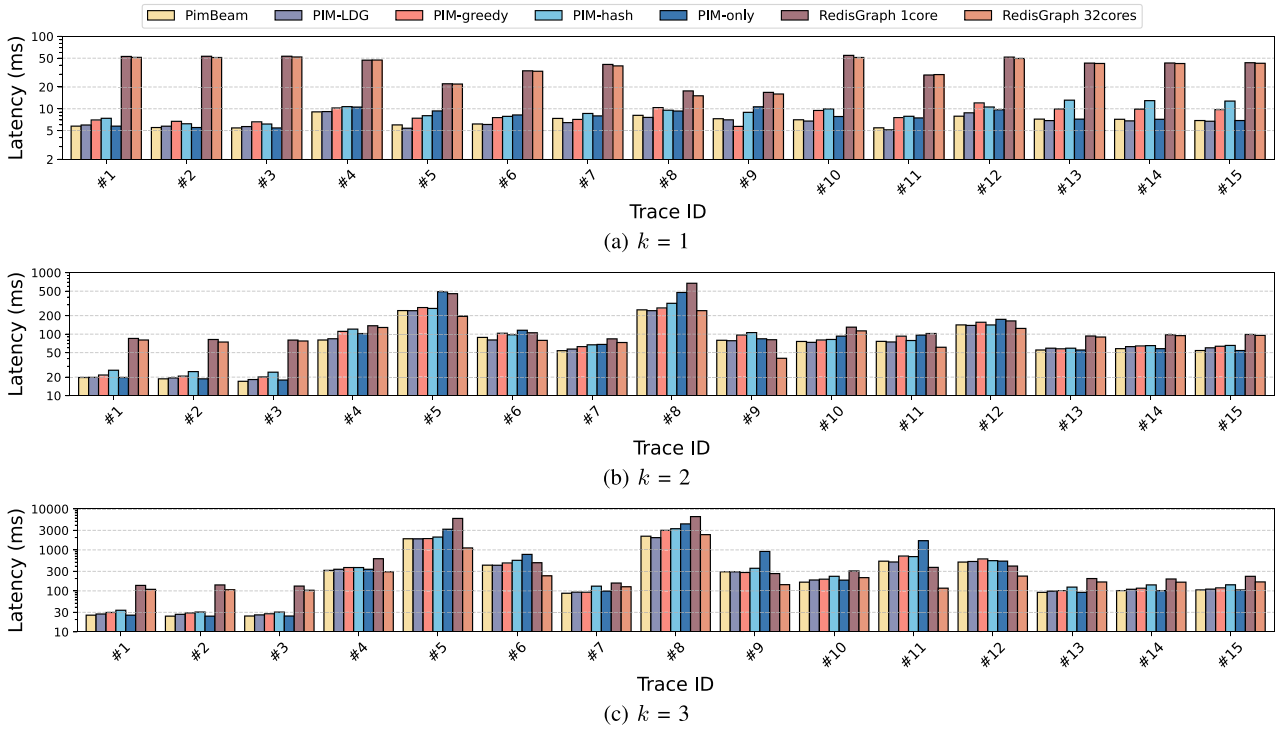


Fig. 7. Run-time of short k-hop path queries (log scale). Datasets #1-#3 and #13-#15 contain no high-degree nodes, so the results of PIM-only are identical to those of PimBeam.

for a total of 2560 PIM modules. PimBeam and other PIM-based approaches utilize one dedicated CPU core and 64 PIM modules. RedisGraph is evaluated under two configurations: single-core and 32-core, the latter utilizing all available CPU cores.

4) Workload Setup: For simplicity, the edge stream of each dataset is processed in random order to construct the initial graph. Our evaluation mainly focuses on typical RPQs, specifically the k -hop path queries with a fixed starting node. Starting nodes are randomly selected, and RPQs are processed in batches with a batch size of 64K. We perform extended label filtering RPQs on the social network dataset of LDBC SNB. To show graph update performance, we randomly select 64K edges for insertions and deletions.

B. Performance of Short Path RPQs

Fig. 7 shows the run-time for processing short k-hop path queries on PimBeam, PIM-hash, PIM-greedy, PIM-LDG, PIM-only, and RedisGraph with 1 core and 32 cores. Across 15 real-world graphs in the SNAP datasets, PimBeam performs the best, achieving a 3.59x speedup over the state-of-the-art database RedisGraph-1core on average, as shown in Table II. By assigning path matching tasks to PIM modules and reducing data movement, PimBeam breaks the memory wall bottleneck during path matching, achieving high performance. Compared to other PIM-based schemes, PimBeam also has advantages, being 0.84%, 15.60%, 22.56%, and 10.86% faster than PIM-LDG, PIM-greedy, PIM-hash and PIM-only, respectively (derived from Table II).

The query advantage decreases as the number of hops increases. Specifically, the advantage of PimBeam is more pronounced when path hops are smaller, as IPC cost is lower at this time. Particularly, when $k = 1$, IPC expense is zero, and the query process only involves CPC expense, without needing reduction. At this point, PimBeam has the greatest advantage, being faster than RedisGraph-1core on all datasets with an average speedup of 6.06x (shown in Table II). And RedisGraph

TABLE II
MINIMUM, MAXIMUM, AND MEAN SPEEDUP OF PIM-BASED SCHEMES ON 15 SNAP DATASETS FOR SHORT k-HOP QUERIES ($k = 1, 2, 3$)

Hop	Metric	PimBeam	PIM-LDG	PIM-greedy	PIM-hash	PIM-only	Redis-32
1	min	2.17	2.33	1.70	1.85	1.58	0.99
	max	9.79	9.37	8.06	8.64	9.79	1.17
	mean	6.06	6.14	4.96	4.61	5.51	1.04
2	min	1.02	1.04	0.84	0.77	0.91	1.02
	max	4.47	4.38	3.98	3.33	4.47	2.79
	mean	2.17	2.14	1.93	1.78	1.90	1.41
3	min	0.71	0.74	0.53	0.55	0.22	1.19
	max	5.77	5.21	4.82	4.59	5.77	5.24
	mean	2.53	2.41	2.20	1.96	2.20	1.95
mean		3.59	3.56	3.03	2.78	3.20	1.47

Results are normalized to RedisGraph-1core.

executed with 32 cores only exhibits a minor performance improvement over 1 core, since the query load is low and multi-core competition incurs noticeable overhead. As a result, PimBeam is faster than RedisGraph-32cores across all datasets, achieving an average speedup of 5.83x. As k increases, even though our partitioning ensures graph locality, there are still some cases where the next-hop of some nodes are not within the current PIM module. At this time, the number of nodes queried in the next-hop grow exponentially, rapidly increasing IPC overhead. Therefore, as shown in Table II, for $k = 2$ and $k = 3$, the average speedup of PimBeam over RedisGraph-1core decreases to 2.17x and 2.53x. In comparison to RedisGraph-32cores, the average speedup of PimBeam decreases to 1.54x and 1.30x, respectively.

The skewness of the dataset affects the performance of PimBeam. For datasets with lower skewness, such as datasets #1-#3, PimBeam demonstrates significant advantages, achieving a run-time at least 4.32x that of RedisGraph (shown in Fig. 7(b) for dataset #1). Under these circumstances, PimBeam also achieves at least a 3.59x speedup over RedisGraph-32cores (shown in Fig. 7(b) for dataset #2). When it comes to highly skewed datasets, PimBeam performs worse than RedisGraph-1core in few cases. This is observed only in dataset #9, #11, and #12 when $k = 3$. And it shows weaknesses compared to RedisGraph-32cores on more datasets, especially datasets #11 and #12 under 3-hop queries (depicted in Fig. 7(c)). Leveraging all powerful CPU cores and the 44MB L3 cache clearly benefits RedisGraph-32cores in handling skewed datasets. Fortunately, the performance of PimBeam is faster than RedisGraph in most cases, demonstrating that our partitioning scheme effectively maintains graph locality in most scenarios.

Compared to PIM-hash, PimBeam has shorter query times. As derived from Table II, for $k = 1, 2, 3$, the improvements are 23.93%, 17.97%, and 13.04%, respectively. The hash partition scheme only balances the number of nodes between PIM modules, but it cannot maintain the locality of the graph. PimBeam, on the other hand, addresses the skewness problem of the graph by being aware of node distribution, achieving load balancing between PIM modules.

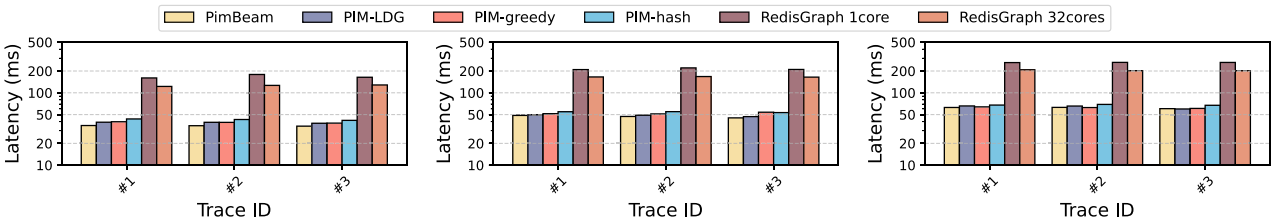
Compared to PIM-greedy, PimBeam also has obvious advantages. It improves by 18.15%, 11.06%, and 13.04% for $k = 1, 2, 3$, respectively (derived from Table II). This indicates that PimBeam's partitioning method of choosing the PIM module with the highest-degree neighbor of a node is more reasonable than simply using the radical greedy method. From another perspective, for a randomly ordered input edge stream, PIM-greedy essentially assigns new nodes randomly to the PIM module of one of their neighbors, while PimBeam selects the neighbor with the highest degree. Therefore, in the vast majority of cases, the query performance of PimBeam is comparable to or better than that of PIM-greedy. In fact, across all experiments, PimBeam's performance is only 4.28% lower than PIM-greedy for $k = 3$ on dataset #9, and 3.17% and 27.70% lower for $k = 1$ on datasets #7 and #9, respectively. In other cases, PimBeam consistently outperforms PIM-greedy.

Compared to PIM-LDG, PimBeam is slightly ahead. For $k = 1, 2, 3$, it improves by -1.32%, 1.38%, and 4.74%, respectively (derived from Table II). PIM-LDG, by examining the weighted degrees of all neighbors of a node in every PIM module, theoretically maintains the best graph locality among all schemes. Therefore, it performs the best in 1-hop scenario. As the number of query hops increases, PimBeam's node migration strategy further enhances the locality of subgraphs. Consequently, the performance of PIM-LDG starts to fall behind that of PimBeam. Moreover, traversing neighbor nodes and calculating the score function in each PIM module takes too long, making its graph construction time significantly longer than the PimBeam (detailed time analysis in Section IV-I).

Compared to PIM-only, PimBeam demonstrates significant advantages through its labor-division strategy. For $k = 1, 2, 3$, it achieves improvements of 9.08%, 12.44%, and 10.86%, respectively (derived from Table II). This is primarily because PIM-only does not involve host CPU in the query process. While the PIM side is well-suited for handling pointer chasing, it struggles with high-degree nodes, whose neighbors are frequently accessed. In such cases, the weaker cores on the PIM side are clearly outperformed by the powerful CPU with a large cache, particularly in highly skewed datasets (e.g., datasets #8, #9, and #11in 3-hop queries, as depicted in Fig. 7(c)).

C. Performance of Long Path RPQs

For long path queries, we only perform k-hop path queries on the road network graph (datasets #1-#3), since every PIM module has limited memory. The results are shown in Fig. 8. High skewness or dense graphs could lead to insufficient memory when querying long path queries, making it infeasible to execute such queries on these types of graphs. As k increases from 4 to 6 and 8, PimBeam achieves a 4.80x, 4.55x, and 4.24x speedup over RedisGraph-1core, respectively. When compared to RedisGraph-32cores, PimBeam achieves speedup of 3.59x, 3.54x, and 3.29x, respectively. It can be observed that the performance of PimBeam deteriorates compared to RedisGraph (1 core and 32 cores) as k increases. Compared with the other three PIM-based schemes (PIM-hash, PIM-greedy, PIM-LDG), PimBeam improves performance by 13.50%, 6.93%, and 5.22% on average, respectively. The advantage of PimBeam also shrinks as k increases. The reason is that the number of matching paths grows exponentially with k. On the one hand, this leads to an



(a) $k = 4$ (b) $k = 6$ (c) $k = 8$

Fig. 8. Run-time of multiple k-hop path queries (log scale).

TABLE III
NORMALIZED RUN-TIME ACROSS DIFFERENT THRESHOLDS

Trace ID	Threshold					
	8	16	32	64	256	PIM-only
4	1.07	1.02	1.01	**1.00**	1.05	1.06
5	1.07	**1.00**	1.01	1.11	1.38	1.71
6	1.00	**1.03**	1.29	1.35	/	1.89
7	1.04	**1.00**	1.10	1.11	1.12	1.13
8	1.09	1.00	1.02	**1.05**	1.20	2.01
9	1.00	**1.01**	1.16	1.69	3.10	3.18
10	1.08	**1.00**	1.12	1.13	1.13	1.12
11	1.00	**1.03**	1.57	2.38	2.85	3.25
12	1.02	1.03	**1.00**	1.06	/	1.05

Bolded values indicate the results under our selection. For some datasets, results for a threshold of 256 are unavailable, as all graph nodes are assigned to the PIM side, rendering it equivalent to PIM-only.

increase in IPC overhead; on the other hand, it results in a large number of query results, making reduction a bottleneck.

D. High-Degree Node Threshold Selection

We explore various high-degree thresholds on SNAP datasets containing nodes with degrees exceeding 16 (datasets #4-#12). Thresholds of 8, 16, 32, 64, 256, and infinity are evaluated for high-degree nodes, with the results presented in Table III. As the threshold increases from 8 to infinity, the run-time exhibits two distinct patterns: it either initially decreases and then increases, or it continuously increases.

For most datasets (e.g., datasets #4, #5, #8), the run-time follows the first pattern. At lower thresholds, a significant portion of the workload is assigned to host CPU, resulting in slower overall query times. As the threshold increases, part of this workload shifts to the PIM modules, reducing the burden on the host and subsequently decreasing run-time. However, when the threshold reaches the upper limit, the run-time increases again. This is due to the limited cache capacity on the PIM side, which is less efficient at handling high-degree nodes. Consequently, the system's performance becomes constrained by the PIM side. Optimal run-time is achieved only at intermediate threshold values, where the workload is balanced between the host and PIM sides. For these graphs, our threshold selection method successfully identifies the minimum run-time point for all datasets except for dataset #8.

For other graphs (datasets #6, #9, #11), the run-time continuously increases as the threshold rises. This reveals the load distribution between the host and PIM sides is more balanced at a threshold of 8 compared to 16. Nevertheless, with a difference of less than 3%, the default threshold of 16 remains a reasonable choice, offering satisfactory performance.

Dataset #8 poses a unique challenge, as our threshold selection method fails to identify its optimal threshold. This dataset contains several ultra-high-degree nodes with degrees exceeding 10,000 yet lacks overall density. Under random graph construction, the early insertion of these ultra-high-degree nodes often causes the threshold to increase, thus preventing the selection of the optimal threshold of 16. Identifying an effective threshold selection method for datasets with such complexity and variability remains intrinsically challenging. Nevertheless, our method demonstrates robust performance across the majority of datasets, achieving results within 5% of the optimum.

E. Performance of IPC Communication

One of the factors for evaluating the effectiveness of partitioning schemes is the IPC cost. Fig. 9 shows the fraction of total run-time for IPC operations of PIM-based schemes processing 3-hop queries. Among all PIM-based schemes, PimBeam exhibits the lowest average IPC fraction of total run-time, at only 9.35%, compared to PIM-greedy and PIM-LDG, which account for 9.50% and 9.54%, respectively. The higher IPC cost of PIM-greedy can be attributed to its classification strategy, which only considers the first neighbor node. For PIM-LDG, the relatively high communication overhead in some datasets (e.g., dataset #9) is likely due to the absence of a dynamic node migration strategy. As a result, the lack of subsequent adjustments to the partitioning state leads to increased communication costs. PIM-hash, on the other hand, distributes all nodes randomly, resulting in significantly higher communication overhead in many datasets, particularly in complex situations (especially datasets #5 and #8). Consequently, PIM-hash achieves the worst performance, with an average IPC fraction of 22.68%.

F. Load Distribution

We further analyze the load distribution between the host and the PIM side during path matching. Fig. 11 presents the query load distribution for PimBeam when processing 3-hop queries on datasets containing high-degree nodes. The host manages 0.33% to 4.44% of the query workload for high-degree nodes, with the low-degree node workload offloaded to the PIM side. Combined with the experimental results on threshold

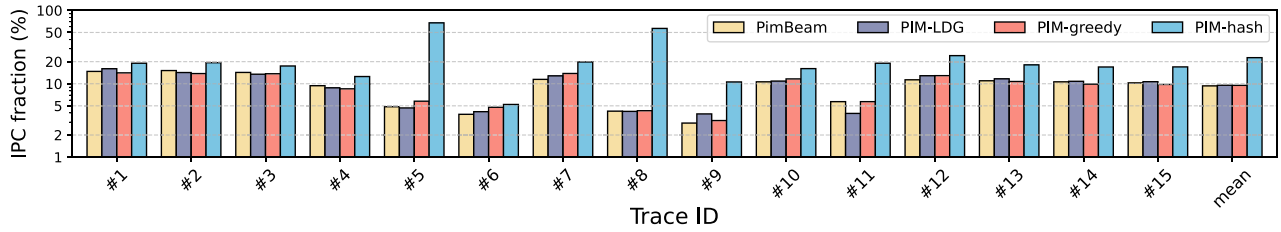
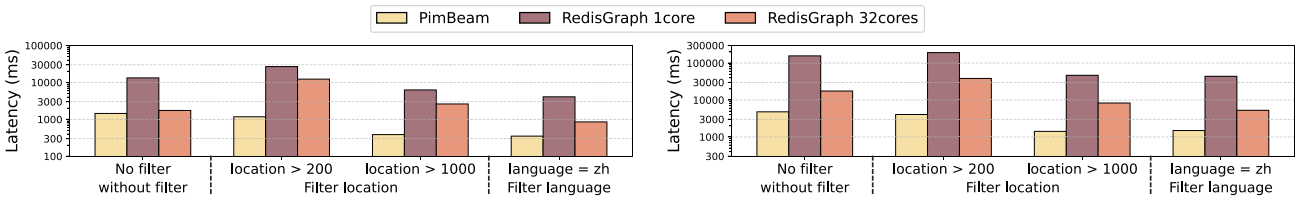


Fig. 9. The fraction of total run-time for IPC operations of different architectures processing 3-hop queries.



(a) $k = 2$ (b) $k = 3$

Fig. 10. Run-time of k-hop path queries on the social network graph generated by LDBC SNB with label filtering (log scale).

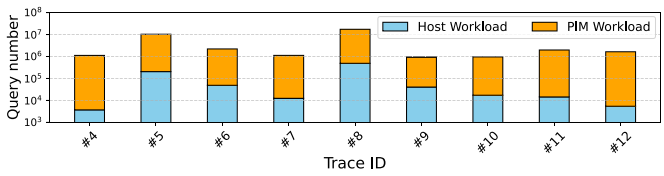


Fig. 11. Load distribution for host and PIM sides executing 3-hop queries across SNAP datasets with high-degree nodes.

exploration (Section IV-D), it can be concluded that the PIM side is ill-suited for handling high-degree nodes. Therefore, our labor-division scheme combines complementary capabilities of the PIM and host sides, significantly alleviating the load on the PIM side.

G. Performance of Label Filtering RPQs

We conduct typical and label filtering RPQs on the social network graph generated by LDBC SNB. The results are shown in Fig. 10. When typical RPQs are performed, PimBeam achieves a 9.18x and 32.78x speedup over RedisGraph-1core for 2-hop and 3-hop queries, respectively. Compared to RedisGraph-32cores, PimBeam also demonstrates a 1.21x and 3.66x speedup for 2-hop and 3-hop queries, respectively. It can be observed that RedisGraph-32cores effectively leverages the advantages of multi-core parallelism on relatively dense graphs like social networks. However, due to the lower skewness of this graph, PimBeam still exhibits a more pronounced advantage over RedisGraph.

When executing filtering node label *language*, we select to filter the Chinese-speaking population ($property\_value =' zh'$), which accounts for 17.4% of the entire social network. PimBeam significantly reduces the query time for 2-hop and 3-hop queries, with performance being 11.61x and 29.43x that of RedisGraph-1core filtering the same property value. Compared to RedisGraph-32cores, PimBeam achieves 2.43x and 3.53x speedup for 2-hop and 3-hop queries, respectively, demonstrating strong superiority.

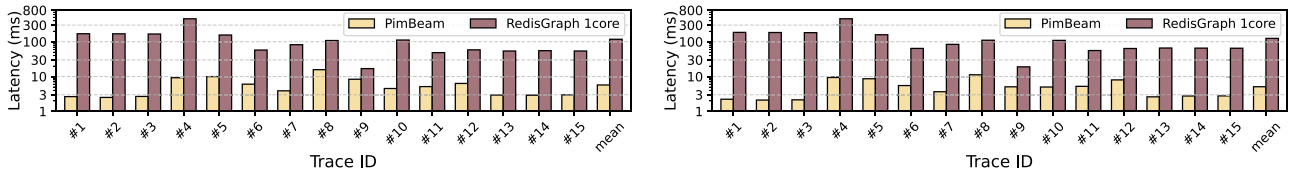
When executing filtering edge label *location*, we set 2 regular paths to pass through edges with property values greater than 200 and 1000, which respectively correspond to retaining most and fewer of the typical RPQ results. When retaining most of the results, PimBeam's execution speed increased by 16.86%, while retaining fewer property values led to a substantial increase of 71.89%. On average, PimBeam's performance reached 29.78x and 8.14x that of RedisGraph performed at 1-core and 32-core. We can find that the more nodes or edges are filtered out, the shorter PimBeam's query time is. In contrast, RedisGraph's performance declines due to the introduction of matrix operations related to labels and property values.

In summary, PimBeam is capable of simultaneously filtering nodes and edges across multiple PIM modules, greatly improving filtering efficiency. Moreover, filtering can reduce CPC and IPC overhead, making PimBeam significantly outperform in the extended RPQ tasks.

H. Performance of Graph Updates

Fig. 12 shows the run-time for graph updates, inserting 64 k edges and deleting 64 k edges, on 15 real-world graphs from the SNAP datasets. Compared to RedisGraph-1core, the insertion throughput of PimBeam is up to 67.36x and 26.45x on average. The deletion throughput is up to 87.02x, and the average is 32.22x. Exempt from IPC and reduction stages, the graph update workloads can fully utilize the intra-PIM bandwidth, resulting in exceptional performance.

For the graphs with no high-degree nodes (datasets #1-#3 and #13-#15), the advantage of PimBeam is very large, with at least 18.17x and 23.38x boosts for insertions and deletions, respectively. For the graphs with a high proportion of data stored



(a) insert (b) delete

Fig. 12. Run-time of graph updates (log scale).

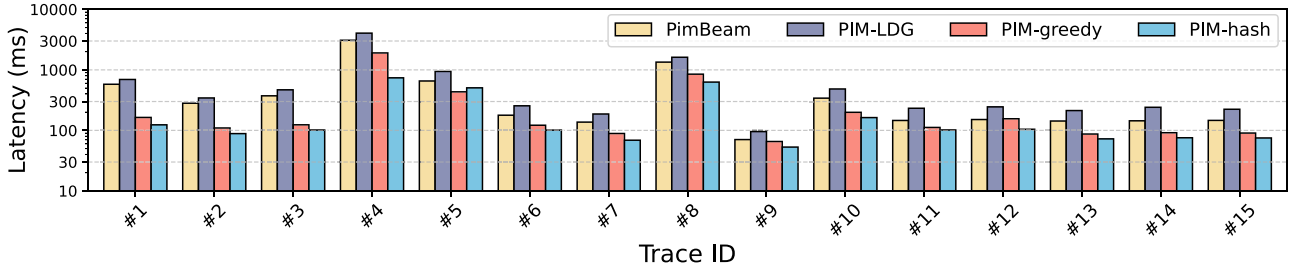


Fig. 13. Graph partitioning time for different PIM-based architectures (log scale).

on the host side (datasets #5, #6, #10, and #11) or with very high-degree nodes (dataset #8), the load of the host becomes the bottleneck of graph updates. Leveraging heterogeneous graph storage for high-degree nodes, PimBeam can well amortize the host side's update cost to the PIM side, and still achieve good graph update performance, with an average speedup of 13.24x and 14.47x for insertions and deletions, respectively.

I. Graph Partition Time

Before conducting RPQs and update experiments, it is necessary to load the edge stream and construct the initial state of the graph. The graph construction time includes the time for graph partitioning and the time for inserting all nodes and edges into subgraphs in computing nodes. An ideal partitioning method should achieve high-quality partitioning results while featuring a low partitioning time. Fig. 13 compares the graph construction times of PimBeam with PIM-hash, PIM-greedy, and PIM-LDG schemes.

We consider PIM-greedy as the baseline, as it is the fastest method for partitioning while ensuring some graph locality. In contrast, PIM-hash achieves the fastest graph construction speed, reducing time by 20.18%, as it only hashes the NodeID of a new node to assign its PIM module, with no regard for graph locality.

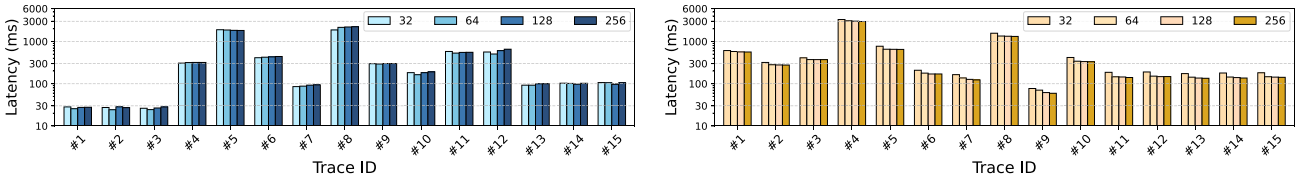
PimBeam identifies the neighbor of a new node with the highest degree, but since the number of neighbors traversed does not exceed the high-degree node threshold, the additional partitioning time is only a constant factor. This results in PimBeam's graph construction time being 78.24% longer than that of PIM-greedy. Given that PimBeam reduces the average time by 15.60% compared to PIM-greedy for typical RPQs within 3-hop paths (shown in Table II), this additional construction time is acceptable. PIM-LDG, which traverses neighbors and considers the linear weighted degrees of neighboring nodes in all partitions, is highly time-consuming, increasing time by 144.28% compared to PIM-greedy. Considering that the query efficiency of PIM-LDG is slightly inferior to that of PimBeam, PimBeam offers the best overall performance.

J. Performance With Different Number of PIM Modules

It might be intuitive to assume that the speedup ratio of PimBeam could improve with an increasing number of PIM modules, given that the UPMEM machine is equipped with 2560 modules. To investigate this, we measured the run-time of PimBeam for 3-hop queries using 32, 64, 128, and 256 PIM modules, as shown in Fig. 14(a).

The results show that increasing the number of PIM modules has minimal impact on run-time: the longest run-time (with 256 modules) is only 6.41% longer than the shortest run-time (with 32 modules). While increasing the number of PIM modules reduces the workload per module, it simultaneously increases the likelihood that next-hop nodes will be distributed across multiple PIM modules. This increases IPC overhead, which offsets the benefits of having more PIM workers. This insight suggest that the number of PIM modules should be allocated judiciously based on the characteristics of the specific task. Indiscriminately utilizing a large number of PIM workers could result in inefficient resource utilization.

Graph construction time is also affected by the number of PIM modules, as shown in Fig. 14(b). As the number of PIM modules increases, the graph construction time continuously decreases. This is because with more PIM modules, each module has fewer nodes to insert, thereby reducing the load for spatial retrieval and management. However, 64 marks a turning point: with 32 PIM modules, the construction time is 12.67% longer than that with 64; whereas with 128 and 256 PIM modules, the



(a) Run-time (b) Graph partitioning time

Fig. 14. Run-time and partitioning time for 3-hop queries in PimBeam across varying PIM module numbers (log scale).

time reductions are only 1.93% and 3.12%, respectively, which is almost negligible.

On the other hand, each PIM module in the UPMEM machine has a fixed memory capacity of 64 MB. For skewed graphs with a large number of nodes and edges, such as dataset #4, using fewer PIM modules (such as 16) may lead to memory overflow in some high-load PIM modules during multi-hop query phases.

In conclusion, we consider 64 PIM modules to be a relatively balanced choice, and the number of PIM modules can be appropriately increased based on the task load.

V. Conclusion

This paper introduces PimBeam, a state-of-the-art PIM-based graph database management system for accelerating path matching. PimBeam successfully supports efficient batch of typical and extended RPQs and graph updates while having low partitioning time. The dynamic graph partitioning algorithm in PimBeam successfully tackles graph skewness and preserves graph locality with low overhead. With PIM-friendly graph partitioning, PimBeam addresses the challenges of load imbalance and the communication bottleneck during performing RPQs. The optimizations for graph storage enable frequent graph updates by amortizing the host side's update cost to the PIM side. Evaluation on multiple datasets against RedisGraph and other PIM-based partitioning algorithms shows superiority of PimBeam. It can achieve high path matching and graph update performance while have relative fast graph construction.

References

[1] M. Besta et al., "The graph database interface: Scaling online transactional and analytical graph workloads to hundreds of thousands of cores," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 1–18.

[2] H. Chen et al., "G-tran: A high performance distributed graph database with a decentralized architecture," in *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 2545–2558, 2022.

[3] W. Martens, M. Niewerth, T. Popp, S. Vansummeren, and D. Vrgoc, "Representing paths in graph database pattern matching," 2022, *arXiv:2207.13541*.

[4] C. Li et al., "Bytegraph: A high-performance distributed graph database in bytedance," in *Proc. VLDB Endowment*, vol. 15, no. 12, pp. 3306–3318, 2022.

[5] Y. Tian, "The world of graph databases from an industry perspective," *ACM SIGMOD Rec.*, vol. 51, no. 4, pp. 60–67, 2023.

[6] S. Abiteboul and V. Vianu, "Regular path queries with constraints," in *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst.*, 1997, pp. 122–133.

[7] R. Ronen and O. Shmueli, "SoQL: A language for querying and creating data in social networks," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2009, pp. 1595–1602.

[8] U. Leser, "A query language for biological networks," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii33–ii39, 2005.

[9] X. Wang et al., "FPIRPQ: Accelerating regular path queries on knowledge graphs," *World Wide Web*, vol. 26, no. 2, pp. 661–681, 2023.

[10] M. Arenas, C. Gutierrez, D. P. Miranker, J. Pérez, and J. F. Sequeda, "Querying semantic data on the web?," *ACM SIGMOD Rec.*, vol. 41, no. 4, pp. 6–17, 2013.

[11] S. L. Xi, A. Augusta, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proc. 11th Int. Workshop Data Manage. New Hardware*, 2015, pp. 1–10.

[12] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Berlin, Germany: Springer, 2022, pp. 171–243.

[13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 105–117.

[14] B. Asgari et al., "FAFNIR: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *Proc. 2021 IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 908–920.

[15] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. IEEE 2015 Int. Conf. Parallel Archit. Compilation*, 2015, pp. 113–124.

[16] C. Giannoula et al., "SynCron: Efficient synchronization support for near-data-processing architectures," in *Proc. 2021 IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 263–276.

[17] Y. Huang et al., "A heterogeneous PIM hardware-software co-design for energy-efficient graph processing," in *Proc. 2020 IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 684–695.

[18] M. Zhang et al., "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *Proc. 2018 IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 544–557.

[19] S. Cai, B. Tian, H. Zhang, and M. Gao, "PimPam: Efficient graph pattern matching on real processing-in-memory hardware," in *Proc. ACM Manage. Data*, vol. 2, no. 3, pp. 1–25, 2024.

[20] H. Kang et al., "PIM-tree: A skew-resistant index for processing-in-memory," 2022. [Online]. Available: https://arxiv.org/abs/2211.10516

[21] H. Kang et al., "PIM-trie: A skew-resistant trie for processing-in-memory," in *Proc. 35th ACM Symp. Parallelism Algorithms Archit.*, 2023, pp. 1–14.

[22] Y. Hua et al., "RADAR: A skew-resistant and hotness-aware ordered index design for processing-in-memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 9, pp. 1598–1614, Sep. 2024.

[23] X. Zhao, S. Chen, and Y. Kang, "Load balanced PIM-based graph processing," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 29, pp. 1–22, 2024.

[24] M. Onizuka, T. Fujimori, and H. Shiokawa, "Graph partitioning for distributed graph processing," *Data Sci. Eng.*, vol. 2, pp. 94–105, 2017.

[25] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2012, pp. 1222–1230.

[26] T. Kawamoto, M. Tsubaki, and T. Obuchi, "Mean-field theory of graph neural networks in graph partitioning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 4366–4376.

[27] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, no. 12, pp. 1519–1534, 2000.

[28] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 251–262, 1999.

[29] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Phys.*, vol. 46, no. 5, pp. 323–351, 2005.

[30] E. Papalexakis, B. Hooi, K. Pelechrinis, and C. Faloutsos, "Power-hop: A pervasive observation for real complex networks," *PLoS One*, vol. 11, no. 3, 2016, Art. no. e0151027.

[31] Newton, V. Singh, and T. E. Carlson, "Pim-graphscc: Pim-based graph processing using graph's community structures," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 151–154, Jul./Dec. 2020.
[32] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proc. 2012 ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 145–156.
[33] A. G. Labouseur et al., "The G* graph database: Efficiently managing large distributed dynamic graphs," *Distrib. Parallel Databases*, vol. 33, pp. 479–514, 2015.
[34] UPMEM, "UPMEM technology," 2024. Accessed: Jul. 19, 2024. [Online]. Available: https://www.upmem.com/technology/
[35] P. Cailliau et al., "Redisgraph graphblas enabled graph database," in *Proc. 2019 IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2019, pp. 285–286.
[36] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi, "Answering regular path queries using views," in *Proc. IEEE 16th Int. Conf. Data Eng.*, 2000, pp. 389–398.
[37] L. Libkin and D. Vrgoč, "Regular path queries on graphs with data," in *Proc. 15th Int. Conf. Database Theory*, 2012, pp. 74–85.
[38] A. Koschmieder and U. Leser, "Regular path queries on large graphs," in *Proc. 24th Int. Conf. Sci. Statist. Database Manage.*, Springer, 2012, pp. 177–194.
[39] J. Zhang et al., "S3: A scalable in-memory skip-list index for key-value store," in *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2183–2194, 2019.
[40] J. Hu, Z. Wei, J. Chen, and D. Feng, "RWORT: A read and write optimized radix tree for persistent memory," in *Proc. IEEE 41st Int. Conf. Comput. Des.*, 2023, pp. 194–197.
[41] S. Assadi, G. Kol, R. R. Saxena, and H. Yu, "Multi-pass graph streaming lower bounds for cycle counting, max-cut, matching size, and other problems," in *Proc. IEEE 61st Annu. Symp. Found. Comput. Sci.*, 2020, pp. 354–364.
[42] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 311–326.
[43] M. Gonzalez Tallada and E. Morancho, "Heterogeneous programming using openmp and cuda/hip for hybrid CPU-GPU scientific applications," *Int. J. High Perform. Comput. Appl.*, vol. 37, no. 5, pp. 626–646, 2023.
[44] M. Gowanlock, "Hybrid KNN-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features," *J. Parallel Distrib. Comput.*, vol. 149, pp. 119–137, 2021.
[45] S.-Y. Xiao, C.-L. Li, B.-Y. Guo, and H. Xiao, "A radix sorting parallel algorithm suitable for graphic processing unit computing," *Concurrency Comput. Pract. Experience*, vol. 33, no. 6, 2021, Art. no. e5818.
[46] Z. Zhou, C. Li, F. Yang, and G. Sun, "Dimm-link: Enabling efficient inter-dimm communication for near-memory processing," in *Proc. 2023 IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 302–316.
[47] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," in *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.
[48] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, pp. 2–es, 2007.
[49] L.-Y. Ho, J.-J. Wu, and P. Liu, "Distributed graph database for large-scale social computing," in *Proc. 2012 IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 455–462.
[50] D. Dai, W. Zhang, and Y. Chen, "IOGP: An incremental online graph partitioning algorithm for distributed graph databases," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 219–230.
[51] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.
[52] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–2.
[53] R. Angles, "The property graph database model," in *Proc. Alberto Mendelzon Workshop Foundations Data Manage.*, 2018, 2100, pp. 1–10.
[54] Neo4j, Neo4j - graph database platform. Accessed: Jul. 19, 2024. [Online]. Available: https://neo4j.com/
[55] I. J. Farkas, I. Derényi, A.-L. Barabási, and T. Vicsek, "Spectra of "real-world" graphs: Beyond the semicircle law," *Phys. Rev. E*, vol. 64, no. 2, 2001, Art. no. 026704.
[56] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *Procedia Comput. Sci.*, vol. 51, pp. 2453–2462, 2015.
[57] J. Leskovec and S. Group, "Stanford network analysis project (SNAP)," 2024. Accessed: Jul. 24, 2024. [Online]. Available: http://snap.stanford.edu/snap/
[58] LDBC Council, "Ldbc social network benchmark (SNB)," 2024. Accessed: Jul. 24, 2024. [Online]. Available: https://ldbcouncil.org/benchmarks/snb/
[59] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "Jgrapht—a Java library for graph data structures and algorithms," *ACM Trans. Math. Softw.*, vol. 46, no. 2, pp. 1–29, 2020.
[60] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat, "gMark: Schema-driven generation of graphs and queries," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 4, pp. 856–869, Apr. 2017.
[61] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. 2017 IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.
[62] R. Labs, "New redisgraph 1.0 achieves 600x faster performance for graph databases," 2023. Accessed: Jul. 24, 2024. [Online]. Available: https://redis.com/blog/new-redisgraph-1-0-achieves-600x-faster-performance-graph-databases/
[63] R. Ma et al., "Accelerating regular path queries over graph database with processing-in-memory," in *Proc. 61st ACM/IEEE Des. Automat. Conf.*, 2024, pp. 1–6.

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

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

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

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

Yuheng Wen is currently working toward the PhD degree in Shanghai Jiao Tong University. His research interests include near data processing and processing-in-memory systems.

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

Guifeng Wang is currently working toward the PhD degree in Shanghai Jiao Tong University. His research interests include computational storage and key-value store.

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