# LearnGraph: A Learning-Based Architecture for Dynamic Graph Processing

Lingling Zhang*, Yijian Wu, Hong Jiang‡, Ziyu Zhou, Tiancheng Lu
Capital Normal University, ‡University of Texas at Arlington
*Corresponding author: 7089@cnu.edu.cn

***Abstract*—Dynamic graph processing systems using conventional array-based architectures face significant throughput limitations due to inefficient memory access and index management. While learned indexes improve data structure access, they struggle with interconnected graph data. We present LearnGraph, a novel architecture with an adaptive tree-based memory manager that dynamically optimizes for graph topology and access patterns. Our design integrates two key components: a hierarchical learned index optimized for graph topology to predict vertex and edge locations, and an adaptive tree structure that automatically reorganizes memory regions based on access patterns. Evaluation results demonstrate that LearnGraph outperforms state-of-the-art dynamic graph systems, achieving 3.4× higher throughput on average and reducing processing time by 1.7× to 11× across standard graph workloads.**

## I. Introduction

Graphs are powerful tools for expressing object relationships through vertices and edges [1], [2]. With the exponential growth of dynamic graph data, traditional architectural approaches to graph storage and access are becoming increasingly inadequate [3], [4]. Efficient graph processing architectures are crucial for various applications [5], [6], from recommendation systems to natural language processing. Current graph processing architectures rely on conventional index structures using sorted arrays - for instance, Teseo [7] employs hash table-based adjacency lists, while Sortledton [8] uses compressed sparse row-like structures. However, these traditional architectures face fundamental limitations: first, maintaining sorted arrays becomes computationally prohibitive when processing millions of edge mutations per second [9]; second, accessing vertices or edges in large, dense arrays inherently incurs significant latency [10], [11]. These architectural constraints create performance bottlenecks in dynamic graph processing, suggesting the need for a fundamentally new approach to system design.

Learning-based architectures have emerged as a promising direction for overcoming traditional graph processing limitations, offering both space efficiency and rapid access performance [12], [13]. These architectures integrate learned models to predict data positions, replacing conventional index structures with intelligent position-mapping functions [14], [15]. By learning to map keys directly to memory locations within densely packed arrays, these architectures fundamentally reimagine data structure design. Compared to traditional B-tree architectures, learning-enhanced designs have demonstrated significant performance advantages, achieving at least 1.5× higher throughput on large datasets [16].

However, existing learning-based architectures prove inadequate for dynamic graphs due to two fundamental architectural limitations. First, these architectures are designed for independent data placement [17], [18], lacking mechanisms to optimize position prediction for interconnected data references. This architectural constraint fails to support efficient operations on connected data [19], which is crucial for graph applications where algorithms fundamentally depend on vertex relationships [2]. Second, current architectures employ static learning models [20], [21], making them unsuitable for dynamic graphs where vertex and edge distributions continuously evolve. These architectural limitations highlight the need for a new learning-enhanced system design specifically optimized for dynamic graph processing.

In this paper, we present LearnGraph, a novel graph processing architecture that integrates learned models into its fundamental data organization and access patterns. Our architecture departs from traditional sorted array structures by using learned models to predict and optimize data placement, access patterns, and memory management decisions. The system architecture is built around a learning-enhanced adaptive tree structure that continuously optimizes the trade-off between memory efficiency and access performance. Specifically, this paper makes the following contributions:

- **Learning-based hierarchical architecture:** LearnGraph employs a two-level learned model architecture - vertex position prediction and edge location optimization. The design combines lightweight neural models for vertex access with learned sparse arrays and bitmap-based indexing for edge position prediction, creating a learning-driven memory hierarchy.
- **Adaptive learning-guided memory management:** The architecture features a self-optimizing tree structure where learning models guide both data organization and access patterns. This learned tree maintains logical adjacency list views while dynamically adapting its structure through a cost-sensitive model that learns optimal node split/merge decisions. The learning-based adaptation ensures efficient memory utilization without compromising access performance or algorithm compatibility.
- **Comprehensive evaluation:** We systematically evaluate LearnGraph's architectural design using real-world and synthetic graph datasets, measuring operation throughput and end-to-end analytical processing performance. Our experimental

results demonstrate that LearnGraph's learning-based architecture achieves 3.4× higher throughput and reduces execution time by 3.5× on average compared to state-of-the-art systems Teseo [7] and Sortledton [8].

## II. Background and Motivation

### A. Graph Structure

A graph $G$ is represented by $G(V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Real-world graphs exhibit sparse connectivity patterns where $|E| << |V|^2$ [22], [23], a property that influences our learning model design. Traditional adjacency lists organize these relationships by grouping edges by vertex, forming neighborhood structures. As illustrated in Figure 1, (a) shows an example graph while (b) demonstrates its adjacency list representation, where each vertex maintains references to its edge array sized by vertex degree. While graph systems may use different physical storage strategies, they preserve this logical adjacency view [7] to support algorithm execution. Understanding these structural properties is crucial for designing learning models that can efficiently predict and optimize both vertex and edge access patterns.

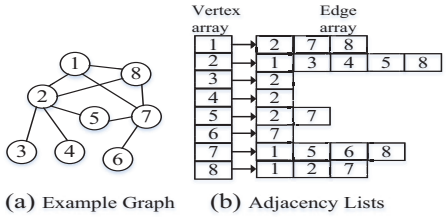



Fig. 1. Example graph and its adjacency lists.

### B. Related Work

Research on dynamic graph processing has explored various fundamental data structures: CSR combined with sparse arrays [24], trees [25], [26], and hash tables [27]. However, these basic approaches achieve limited throughput, particularly when handling frequent updates. To address this limitation, hybrid solutions emerged - systems like Terrace [28] and GraphOne [2] adopted multiple data structures to balance graph update and analytics performance. More recently, LSGraph [6] focused on data locality optimization in real-world streaming scenarios but at the cost of increased memory overhead and system complexity. Graph processing algorithms (e.g., PageRank and BFS) in modern streaming applications require both efficient sequential and random access patterns to vertices and edges [29]. To meet these requirements, various optimization strategies have been proposed: hybrid edge-vertex structures [2], sparse arrays with tree-based indexing [7], [26], and CSR-like organizations [8]. While these approaches achieve reasonable lookup performance, they face scalability challenges with large, dynamic graphs due to their reliance on maintaining dense array structures. These limitations motivate exploring learning-based approaches for dynamic graph processing.

### C. Motivation

Learning-based indexing represents an architectural innovation that achieves efficient data access through position prediction models. However, current architectures are fundamentally designed for independent data placement, lacking mechanisms to handle interconnected data structures. This architectural limitation prevents effective verification and navigation of relationships between data elements. Graph processing presents unique challenges for these architectures due to its inherently connected nature, where edges create complex relationships between vertices that must be efficiently accessed and maintained. While adapting existing architectures to treat edges as independent keys might seem viable, such an approach fails to capture the fundamental requirements of graph processing.

This architectural approach fails to efficiently support neighborhood access patterns, as treating edges independently fragments the critical vertex-neighbor relationships. Such a design cannot maintain coherent adjacency list views that graph algorithms fundamentally require. To quantify these architectural limitations, we evaluated a representative learning-based system, ALEX [30], against Teseo [7], a specialized graph processing architecture. Using five in-memory graphs (detailed in Section IV), our experiments in Figure 2 demonstrate that ALEX's architecture not only fails to provide benefits but actually degrades lookup throughput by up to 4× on large graphs compared to Teseo's graph-aware design. These results highlight the need for a learning-based architecture specifically designed to optimize neighborhood access patterns.

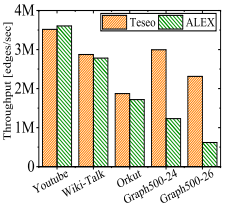



Fig. 2. Lookup throughput

Beyond their limitations with connected data, current learning-based architectures employ model designs unsuitable for dynamic graph processing. These architectures typically follow two approaches: (1) CDF-based designs that learn static data distributions [17], and (2) RMI-based hierarchical architectures where models form a fixed prediction hierarchy [30], [31]. Both architectural approaches have fundamental limitations: CDF-based designs assume fixed-length data structures, while RMI architectures restrict learning to single property mappings. Neither design can effectively adapt to the dynamic nature of graph data, where vertex and edge distributions continuously evolve. This architectural mismatch motivates the need for a new learning-enhanced design specifically optimized for dynamic graphs—one that maintains high-throughput operations on logical adjacency lists while adapting to evolving graph properties.

## III. LearnGraph Framework

### A. Overview

LearnGraph implements a learning-driven two-layer architecture that optimizes graph processing through intelligent position prediction and adaptive memory management. As illustrated in Figure 3, at the top, the Model Layer integrates

two specialized learned models: a Vertex Model that learns and predicts storage positions by adapting to vertex insertion patterns, and an Edge Model that learns position prediction based on dynamic neighborhood relationships. The Data Layer beneath implements a learning-guided hierarchical tree structure with three specialized node types: Internal Nodes that maintain model parameters and learned routing information, Vertex Nodes using learned position-aware fixed-size arrays, and Edge Nodes employing fixed-size arrays and learned sparse arrays with bitmap tracking for optimized edge operations. This hierarchical design enables the learning models to continuously adapt and optimize both data placement and access patterns throughout the graph structure.

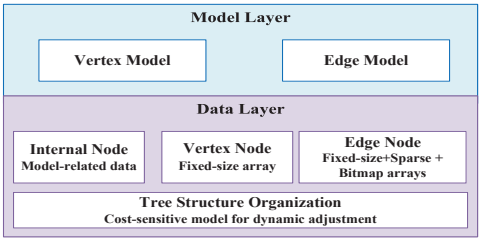
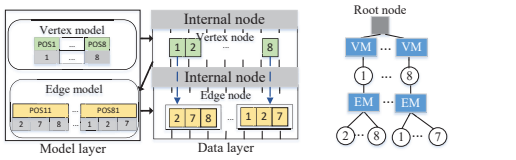



Fig. 3. LearnGraph Architecture.

Figure 4(a) illustrates LearnGraph's memory architecture using the example graph from Figure 1(a). The architecture uses learned models at two levels: a vertex model for position prediction and edge models for neighbor access optimization. Figure 4(b) shows the logical tree structure, where the root node employs the learned vertex model for indexing, while vertex nodes maintain edge models for efficient neighbor management. This architecture enables efficient graph access through continuous model adaptation. The following subsections detail how our two-layer design achieves high performance through learned vertex prediction and neighbor access optimization. To ensure usability, we provide modular interfaces that allow existing graph algorithms to leverage LearnGraph's optimizations without modification.




(a) In-memory structure of the example graph using LearnGraph (b) Logical view of LearnGraph

Fig. 4. The in-memory architecture and logic view of the example graph in Figure 1(a) using LearnGraph.

## B. Model Layer

LearnGraph introduces a novel learning model architecture for dynamic graph processing, as traditional learned indexes like recursive model indexes (RMI) and CDF-based approaches prove insufficient for dynamic graphs (Section II). Our architecture builds upon RMI principles due to their inherent adaptability to growing and evolving data, but fundamentally reimagines their implementation for graph workloads. The architecture employs specialized linear regression models ($F(key) = a(key) + b$) for RMI-based insertions, with two key architectural innovations over existing RMI designs:

**Graph-aware parameter optimization:** Unlike existing RMI implementations like ALEX [30] that use arbitrary parameters leading to non-uniform memory distribution, LearnGraph's architecture carefully determines $a$ and $b$ based on graph properties. This design ensures uniform data distribution, reducing memory management overhead and improving access patterns.

**Training-free model adaptation:** While traditional RMI approaches rely heavily on training data selection and distribution, our architecture eliminates training set dependencies entirely. Instead, the vertex model learns from insertion order patterns, while the edge model adapts based on current neighborhood structures.

To support efficient graph operations, the architecture maintains two coordinated learning models: a vertex position predictor using insertion patterns and an edge locator leveraging neighborhood information. This dual-model design enables accurate position prediction for both vertices and edges while maintaining adaptivity to graph mutations.

*1) Vertex model:* For a vertex $\mu$ to be inserted, let $Num(\mu)$ represent its insertion order and $D$ denote the capacity of a vertex or edge node. The vertex model determines the target vertex node for inserting $\mu$ using a function $F(\mu)$ as defined below.

$$F(\mu) = \lfloor \frac{Num(\mu)}{D} \rfloor + \lceil \frac{Num(\mu)}{D} \rceil \times D \tag{1}$$

*2) Edge model:* Edges fundamentally represent vertex neighborhoods. In undirected graphs, inserting an edge involves adding each vertex as a neighbor of the other, while in directed graphs, it requires adding just one vertex as a neighbor. Let $CurDeg(\mu)$ denote the current neighbor count of vertex $\mu$. The edge model determines which edge node will store $\mu$'s neighbor $\nu$, using a function $M[e(\mu, \nu), \nu]$ defined in Equation (2).

$$M[e(\mu, \nu), \nu] = \lfloor \frac{CurDeg(\mu)}{D} \rfloor + \lceil \frac{CurDeg(\mu)}{D} \rceil \times D \tag{2}$$

*3) The specific insertion position:* Vertex insertion follows a two-step process: first determining the target vertex node, then identifying the specific array position within that node. For a vertex $\mu$, if $\lfloor \frac{Num(\mu)}{D} \rfloor = T$, then $Num(\mu)$ falls within $[TD, (T+1)D)$. This means $\mu$ will be stored in the $T^{th}$ vertex node, which has capacity $D$. After determining the vertex node, the specific array position is calculated using $Mod(\frac{Num(\mu)}{D})$, which gives the remainder when $Num(\mu)$ is divided by $D$. Edge insertion follows a similar process for determining the target edge node. Once the edge node is identified, exponential search locates the specific insertion position as detailed in Section III-D2.

## C. Data Layer

LearnGraph organizes in-memory graph data using a tree structure comprising internal nodes, vertex nodes, and edge

nodes. As shown in Figure 5, each vertex or edge node contains a model value ($'MV'$) used to determine element positions within that node. The data structure employs vertex arrays, sparse arrays, and bitmap arrays to store both vertex-related and edge-related data. To represent vertex neighborhoods, vertices in vertex nodes maintain neighbor reference arrays that point to their corresponding edge nodes.

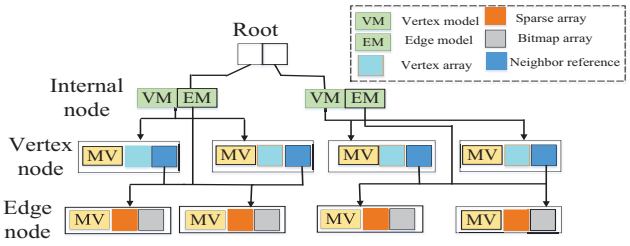



Fig. 5. The structure of the data layer in LearnGraph.

*1) Internal node:* An internal node maintains references to vertex nodes, edge nodes, and other internal nodes as its children. The model value range of a parent internal node encompasses those of all its child data nodes. Beginning from the second level of the tree (with total height $K$), we constrain the maximum number of internal nodes at the $i^{th}$ level to $t^{i-1}$, where $t$ is an integer $\geq 2$ and $2 \leq i \leq K-2$. Internal nodes at the same level maintain an equal number of references to next-level nodes, with this reference count being dynamically configurable in LearnGraph. These internal nodes serve to implement recursive model indexes, providing flexible management of vertex and edge nodes.

*2) Vertex node and edge node:* Vertex and edge nodes store model values for position prediction. Each vertex node maintains a fixed-size array of capacity $D$ to store vertices and a separate array for neighbor references. Since many nodes have few neighbors in a graph, we use a fixed-size array of capacity 10 to store the earliest arriving neighbors in sorted order. When neighbor counts exceed this fixed array size, edge nodes switch to sparse arrays that contain deliberate gaps to accommodate insertions. Following established practices [7], [30], we maintain 75% occupancy in sparse arrays, reserving the remaining space as gaps to optimize search operations. Each gap is populated with its closest right neighbor, enabling efficient exponential search with $O(\log D)$ time complexity for vertex lookups, where $D$ is the sparse array size [30]. Edge nodes also maintain a bitmap array matching the sparse array's size to mark occupied positions, allowing searches to skip gaps efficiently.

### D. Basic algorithms

The algorithms for vertex operations (lookups, insertions, and deletions) in LearnGraph are straightforward since vertex positions can be directly computed from their arrival order. Edge operations, particularly in learned sparse arrays with bitmap tracking, require more detailed examination. While operations on neighbors stored in fixed-size arrays are efficiently handled using quicksort, we focus on the algorithms for edge lookup, insertion, and deletion in learned sparse arrays.

*1) Lookup:* Edge lookup begins by verifying that both vertices of the edge exist in vertex nodes. If confirmed, we locate the root node of vertex neighbors using one vertex's starting pointer. Since LearnGraph implements model-based insertions, we first check the edge's model value. A non-existent model value indicates the edge hasn't been inserted, resulting in a null record return. Otherwise, edge models are recursively applied from the root node, predicting positions at each level until reaching the edge node. If the edge is found at the predicted position, the lookup returns that record. If not, exponential search is initiated from the predicted position to locate the edge's actual position.

*2) Insert:* The edge insertion process in LearnGraph follows a structured approach utilizing both sparse arrays and bitmap arrays. As shown in Figure 6, starting from the initial state, where the sparse array maintains ordered destination vertex IDs of edges (e.g., edges (0, 2), (0, 5), (0, 8), (0, 9), where Vertex 0 is the source vertex) with intentional gaps and a corresponding bitmap array (1, 1, 0, 1, 0, 1) tracking occupied positions, the LearnGraph system employs a four-step process for edge insertion: First, the system verifies both vertices exist in vertex nodes using vertex models, and inserts any missing vertices into vertex nodes at positions determined according to Section III-B3. Second, the edge model is applied to compute the model value and predict the target position - insertion proceeds only if no model value exists. Third, for each neighbor vertex insertion (one for directed graphs, two for undirected graphs), the system traverses internal nodes following edge models from the root node to reach the target edge node.

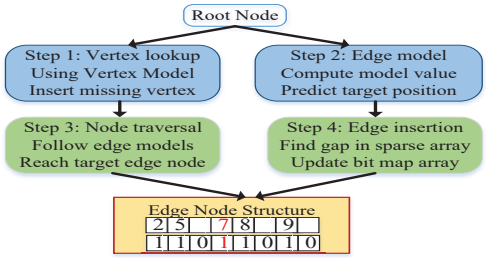



Fig. 6. Insertion process using LearnGraph.

Finally, when inserting an edge, e.g., (0, 7), representing a connection from vertex 0 to vertex 7, if inserting at the predicted position maintains the edge node's sparse array ordering, the insertion succeeds immediately; otherwise, exponential search locates the correct insertion position. When inserting at the final position between existing destination vertices (between 5 and 8), the system identifies an available gap using the bitmap array (0 indicates available gap) for direct placement, or shifts elements to the nearest rightward gap if the position is occupied. Once the position is determined, the destination vertex ID 7 is inserted into the sparse array, and the corresponding bitmap position is updated from 0 to 1 to reflect the occupied status. If a model value falls outside internal

node ranges, the ranges and references must be adjusted, though this is typically prevented by setting sufficiently large initial ranges. This process maintains both the sorted order of edge destinations and the target 75% utilization rate, while preserving gaps for future insertions, ensuring efficient edge search and insertion operations through the combination of model-based prediction and bitmap-assisted position tracking.

*3) Delete and update:* Edge deletion employs the lookup algorithm to locate edge positions within the edge node. Once found, the position is converted to a gap by clearing it and setting the corresponding bitmap position to '0', followed by removing the edge's model values. Edge updates are straightforward, simply writing new edge values into positions identified by the lookup algorithm.

### E. Cost-sensitive model

First, we analyze the time complexity and overhead of constructing our learning-based architecture before considering the cost-sensitive model. This analysis applies to any input graph with $|V|$ nodes and $|E|$ edges. The time complexity is primarily determined by data access operations. Our tree-based data structure requires $O(\log K)$ time for internal lookups and $O(\log D)$ time for vertex searches. In terms of space complexity, internal nodes consume $O((|V|+2|E|)/H)$ space, while vertex and edge nodes require $O((|V|+2|E|)/(0.75D))$ space, plus negligible overhead from a few small fixed-size arrays that store the first-arrival neighboring nodes. Here, $K$ denotes the tree height, $D$ represents the fixed-size array capacity, and $H$ (64MB) specifies the maximum size of each internal node's subtree.

Since vertex operations involve simple computations with minimal overhead, we focus our cost-sensitive model on edge operations. The model relies on historical statistics of edge lookups and insertions. For any vertex $\mu$, LearnGraph monitors two key metrics: $numTra_\mu$, which counts nodes traversed from root to edge nodes, and $numSearch_\mu$, which counts exponential searches in edge nodes. The ratio between these metrics guides our dynamic tree structure adjustments, with $numTra_\mu$ indicating memory usage for storing $\mu$'s neighbors and tree height, while $numSearch_\mu$ reflects the computational costs of neighbor operations and tree width.

Using a threshold $\epsilon = 0.2$, we implement a dynamic adjustment strategy. When $\frac{numTra_\mu}{numSearch_\mu} \leq \epsilon$, we optimize for computation time by increasing the RMI-based edge models' height from $K$ to $K+1$ and evenly distributing the $K^{th}$ level node references across new $(K+1)^{th}$ level nodes. Conversely, when the ratio exceeds $\epsilon$, we consolidate the structure by removing the last level of internal nodes and connecting second-last level nodes directly to edge nodes.

## IV. Evaluation

We perform experiments on a machine equipped with Intel(R) Xeon(R) Silver 4110 CPU @2.10GHz and 256GB memory. Our evaluation examines both operation throughputs (lookups, inserts, and deletes) and processing times for five common graph algorithms: breadth-first search (BFS), PageRank (PR), local triangle counting (LCC), weakly connected components (WCC), and single-source shortest path (SSSP). We compare **LearnGraph** against two state-of-the-art graph structures: **Teseo** [7] and **Sortledton** [8], using three real-world datasets and two synthetic graphs as shown in Table I. While all three systems support multithreading, we evaluate LearnGraph (with maximum array size 100 for vertex and edge nodes) and Teseo using single-thread execution to simplify comparisons and minimize hardware configuration effects. Sortledton, being specifically designed for multithreaded environments, is evaluated with multiple threads for optimal performance.

TABLE I
SUMMARY OF GRAPH DATASETS, WHERE $|V|$ DENOTES THE NUMBER OF THE VERTICES IN THE VERTEX SET V, $|E|$ THE TOTAL NUMBER OF THE EDGES IN THE EDGE SET E.

| Graph | $\|V\|$ | $\|E\|$ |
|---|---|---|
| Youtube | 1,134,890 | 2,987,624 |
| Wiki-Talk | 2,394,385 | 5,021,410 |
| Orkut | 3,072,441 | 117,184,899 |
| Graph500-24 | 8,870,942 | 26,037,952 |
| Graph500-26 | 32,804,978 | 1,051,922,853 |

### A. Throughput

**Lookup throughput**. We evaluate lookup performance by measuring the total time required to search all edges in random order across each input graph. As shown in Figure 7, LearnGraph achieves an average $2.1X$ higher throughput compared to Teseo and Sortledton across the five datasets. This performance advantage stems from LearnGraph's architectural approach: while Teseo and Sortledton rely on a two-step process of vertex table lookup followed by binary search through adjacency list-like structures, LearnGraph employs learning models to directly predict vertex and edge positions, using exponential search only for final verification.

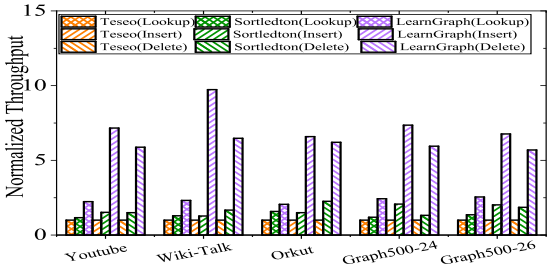
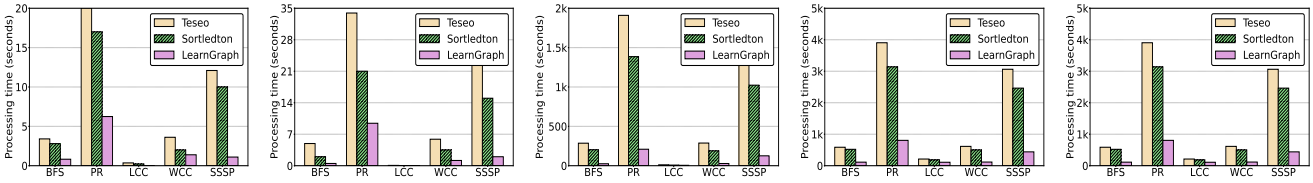


Fig. 7. The throughput of LearnGraph compared to competitors.

**Insert throughput**. We evaluate insertion performance by measuring random-order edge insertions into the input graph. Figure 7 demonstrates that LearnGraph achieves at least $7X$ higher insertion throughput than Teseo and Sortledton across the five datasets. This significant performance gap arises from fundamental structural differences: Teseo utilizes a modified B+ tree (fat tree) with leaf nodes storing the vertex table alongside a hash table, while Sortledton employs dual-level




(a) Youtube (b) Wiki-Talk (c) Orkut (d) Graph500-24 (e) Graph500-26

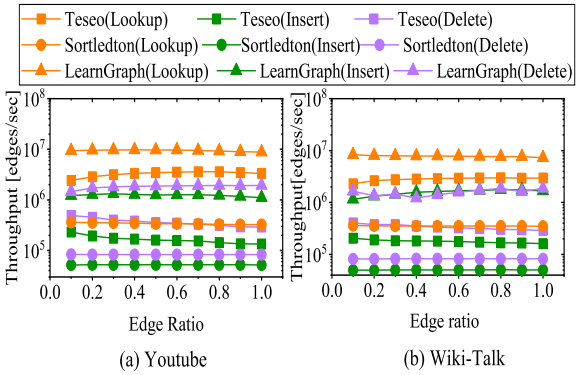
Fig. 8. Processing time of graph algorithms with different analytic systems.

adjacency list-like and CSR-like structures. Both competitors require extensive structure traversal to locate proper insertion positions, whereas LearnGraph leverages vertex and edge models to directly predict insertion positions, resulting in substantially shorter traversal paths.

**Delete throughput**. Figure 7 demonstrates that LearnGraph achieves at least $5X$ higher deletion throughput compared to existing graph analytic systems. This superior performance is attributed to LearnGraph's learning model approach, which quickly predicts vertex and edge positions, in contrast to Teseo and Sortledton's requirement to traverse vertex tables and associated structures for element location.

### B. Processing time of Graph analysis

Figure 8 demonstrates that LearnGraph processes the five algorithms on average $3.5X$ faster and reduces processing time by 1.7× to 11× over Teseo and Sortledton across all datasets. Most notably, LearnGraph executes BFS up to 10X faster than its competitors, as BFS and LCC operations require extensive vertex and neighbor traversal, which LearnGraph accomplishes in $O(\log D)$ time using learning models, while Sortledton requires $O(D)$ time with vertex table scans. For PR and WCC, which also depend heavily on vertex and neighbor access, LearnGraph achieves an average 2.5× speedup over Teseo and Sortledton, whose performance is limited by hashmap lookup and skip list traversal costs. SSSP execution is at least $5X$ faster in LearnGraph, benefiting from its superior lookup throughput. These performance improvements across various graph algorithms demonstrate LearnGraph's effectiveness in accelerating operation throughput.




(a) Youtube (b) Wiki-Talk

Fig. 9. LearnGraph keeps high and steady throughput with a growing ratio of edges.

### C. Evaluation on Cost-sensitive Model

We evaluate our cost-sensitive model's effectiveness through scalability analysis, measuring edge-process throughput as a function of edge ratio in dynamic graph processing. Edge-process throughput represents operations (lookups, insertions, and deletions) per second on sequentially processed edges, while edge ratio indicates the proportion of processed edges to total graph edges. Figure 9 demonstrates that LearnGraph achieves 3.4× higher throughput on average compared to Teseo and Sortledton across increasing edge ratios on Youtube and Wiki-Talk datasets. This performance advantage stems from our cost-sensitive model's ability to adapt to varying workload characteristics. Particularly in Youtube and Wiki-Talk graphs, where neighbor counts [32], [33] exceed the capacity of an edge node rooted at an internal node, the cost-sensitive model stabilizes throughput fluctuations during edge insertions and deletions by dynamically adjusting data structure configurations and optimizing memory allocation patterns. Teseo achieves moderately higher throughput than Sortledton, primarily because Sortledton's versioning-based concurrency mechanisms introduce unnecessary overhead during sequential edge processing. Overall, Figure 9 validates that LearnGraph's cost-sensitive model enables consistent and superior operation throughput when handling dynamic graphs through balanced operation costs across different graph regions.

## V. Conclusion

This paper introduces LearnGraph, a learning-based architecture for dynamic graph processing. Our system integrates learned models into the system architecture through a hierarchical approach: a vertex prediction model for optimal data placement and an edge model for efficient neighbor access, both built into an adaptive tree structure. The architecture employs a learning-based cost model that continuously optimizes tree organization based on operation patterns. Experimental evaluation across five diverse graph datasets demonstrates that LearnGraph's architecture outperforms state-of-the-art systems Teseo and Sortledton, achieving throughput improvements of 2.1× for lookups, 7× for insertions, and 5× for deletions. When evaluating five common graph algorithms, LearnGraph's learning-enhanced architecture delivers performance gains of 3.5× over existing competitors.

## VI. Acknowledgement

We would like to thank the anonymous reviewers. This work is supported by the NSFC (Natural Science Foundation of China) under Grant No.62302043.

## REFERENCES


[1] Zhuo *et al.*, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[2] Kumar *et al.*, "Graphone: A data store for real-time analytics on evolving graphs," in *ACM Transactions on Storage*, 2020.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows," *Proceedings of the VLDB Endowment*, vol. 11, no. 6, pp. 691–704, 2018.

[4] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1860–1876, 2021.

[5] Asgari *et al.*, "Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[6] H. Qi, Y. Wu, L. He, Y. Zhang, K. Luo, M. Cai, H. Jin, Z. Zhang, and J. Zhao, "Lsgraph: a locality-centric high-performance streaming graph engine," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 33–49.

[7] Leo *et al.*, "Teseo and the analysis of structural dynamic graphs," in *Proceedings of the VLDB Endowment*, 2021.

[8] Fuchs *et al.*, "Sortledton: a universal, transactional graph data structure," in *Proceedings of the VLDB Endowment*, 2022.

[9] Sahu *et al.*, "The ubiquity of large graphs and surprising challenges of graph processing," in *Proceedings of the VLDB Endowment*, 2017.

[10] M. N. Bojnordi and F. Nasrullah, "Retagger: An efficient controller for dram cache architectures," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[11] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 581–594.

[12] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "{ROLEX}: A scalable {RDMA-oriented} learned {Key-Value} store for disaggregated memory systems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 99–114.

[13] H. Qi, Y. Zhang, L. He, K. Luo, J. Huang, H. Lu, J. Zhao, and H. Jin, "Psminer: A pattern-aware accelerator for high-performance streaming graph pattern mining," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[14] Nathan *et al.*, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020.

[15] Wu *et al.*, "Updatable learned index with precise positions," in *arXiv preprint arXiv:2104.05520*, 2021.

[16] Kraska *et al.*, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018.

[17] Tang *et al.*, "Xindex: a scalable learned index for multicore data storage," in *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2020.

[18] Wei *et al.*, "Fast {RDMA-based} ordered {Key-Value} store using remote learned cache," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[19] Ma *et al.*, "Film: A fully learned index for larger-than-memory databases," in *Proceedings of the VLDB Endowment*, 2022.

[20] Yu *et al.*, "Treeline: an update-in-place key-value store for modern storage," in *Proceedings of the VLDB Endowment*, 2022.

[21] Yang *et al.*, "{GL-Cache}: Group-level learning for efficient and high-performance caching," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023.

[22] Zhang *et al.*, "Depgraph: A dependency-driven accelerator for efficient iterative graph processing," in *2021 IEEE International Symposium on High-Performance Computer Architecture*, 2021.

[23] P. Kumar and H. H. Huang, "G-store: high-performance graph store for trillion-edge processing," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 830–841.

[24] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 4, pp. 26–es, 2007.

[25] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *arXiv preprint arXiv:1912.12740*, 2019.

[26] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, 2019, pp. 918–934.

[27] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic graphs on the gpu," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 739–748.

[28] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 1372–1385.

[29] Zhao *et al.*, "Tdgraph: a topology-driven accelerator for high-performance streaming graph processing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[30] Ding *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[31] Li *et al.*, "Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems," in *Proceedings of the VLDB Endowment*, 2021.

[32] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD workshop on mining data semantics*, 2012, pp. 1–8.

[33] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting positive and negative links in online social networks," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 641–650.