RadixGraph: A Fast, Space-Optimized Data Structure for Dynamic Graph Storage

HAOXUAN XIE, Nanyang Technological University, Singapore
JUNFENG LIU, Nanyang Technological University, Singapore
SIQIANG LUO*, Nanyang Technological University, Singapore
KAI WANG†, Harbin Institute of Technology, China

Dynamic graphs model many real-world applications, and as their sizes grow, efficiently storing and updating them becomes critical. We present RadixGraph, a fast and memory-efficient data structure for dynamic graph storage. RadixGraph features a carefully designed radix-tree-based vertex index that strikes an optimal trade-off between query efficiency and space among all pointer-array-based radix trees. For edge storage, it employs a hybrid snapshot-log architecture that enables amortized $O(1)$ update time. RadixGraph supports millions of concurrent updates per second while maintaining competitive performance for graph analytics. Experimental results show that RadixGraph outperforms the most performant baseline by up to 16.27× across various datasets in ingesting graph updates, and reduces memory usage by an average of 40.1%. RadixGraph is open-source at https://github.com/ForwardStar/RadixGraph.

CCS Concepts: • **Information systems** → **Graph-based database models**; **Data structures**; *Main memory engines*.

Additional Key Words and Phrases: In-memory graph system, graph data structure, dynamic graph

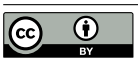
ACM Reference Format:
Haoxuan Xie, Junfeng Liu, Siqiang Luo, and Kai Wang. 2026. RadixGraph: A Fast, Space-Optimized Data Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 72 (February 2026), 28 pages. https://doi.org/10.1145/3786686

1 Introduction

Graphs are fundamental structures in computer science and widely used to model relationships and interactions between entities, including biological networks [29, 55, 60], financial networks [19, 40, 53] and social networks [17, 52, 75]. As real-world graphs evolve continuously, the storage and processing of dynamic graphs have become a significant area of focus for both academia and industry. For example, Facebook leverages dynamic graphs to model user relationships at a trillion-scale [20], while Twitter employs large-scale dynamic graphs to generate real-time recommendations [35]. These applications demand not only the management of large-scale graphs but also the efficient processing of updates to the evolving graph. Given the high computational cost of graph algorithms and the need for fast query efficiency, there is an urgent need to design in-memory dynamic graph data structures supporting fast query and update operations, while being space efficient [66].

*Corresponding author.
†Work done when the author was working as a research fellow at NTU.

Authors' Contact Information: Haoxuan Xie, haoxuan001@e.ntu.edu.sg, Nanyang Technological University, Singapore; Junfeng Liu, junfeng001@e.ntu.edu.sg, Nanyang Technological University, Singapore; Siqiang Luo, siqiang.luo@ntu.edu.sg, Nanyang Technological University, Singapore; Kai Wang, kai_wang@hit.edu.cn, Harbin Institute of Technology, China.



ACM 2836-6573/2026/2-ART72
https://doi.org/10.1145/3786686

Vertex index	Update	Query	Space
Static array [43, 79]	Fast	Fast	High
Multi-level vector [31, 78]	Moderate	Fast	Moderate
Tree-based [22–24]	Moderate	Moderate	Low

Edge index	Insert	Delete	Scan
PMA-based [4, 11, 69]	$O(lg^2(d))$	$O(lg^2(d))$	Fast
Log-based [43, 78, 79]	$O(1)$	$O(d)$	Fast
Tree-based [31, 51]	$O(lg(d))$	$O(lg(d))$	Moderate

Table 1. Comparison of existing vertex and edge indices. "Scan" refers to the efficiency of sequential scan, which requires scanning a set of edges and is a common access pattern in graph queries. d represents the average degree of vertices.

However, designing a graph data structure that supports dynamic updates is much more challenging than developing a static structure. It requires wise designs on both vertex index (using which we can locate an index) and edge index (using which we can store and retrieve neighbor edges of a vertex) to deliver both satisfying query and update performance. We summarize existing vertex and edge indices in Table 1 and discuss their limitations as follows.

Challenges of vertex index. Existing graph systems adopt various vertex indexing schemes, each reflecting a trade-off among **update efficiency**, **query performance**, and **space utilization**. Common designs include: (1) *static arrays* that directly index vertices of contiguous IDs, (2) *multi-level vectors* that are often coupled with hash tables for ID translation, and (3) *tree-based structures* such as B-trees [9] and adaptive radix trees (ART) [47]. *Static arrays* [43, 79] provide constant-time access but incur relatively high space overhead especially when vertex identifiers are non-contiguous. Unfortunately, in dynamic graphs, IDs are often hashed or generated as UUIDs (e.g., DataStax Enterprise Graph uses URIs[1]) which are inherently non-contiguous. *Multi-level vectors* [31, 78] mitigate the space issue by remapping arbitrary IDs into a compact range via hash tables, yet frequent reallocation and remapping operations during resizing can be a performance concern. *Tree-based structures* [22–24] are the state-of-the-art approaches that offer logarithmic-time lookups and better space elasticity, which achieves a better trade-off between space and operational costs. In this paper, we show that we can achieve an even better space and performance trade-off, by redesigning radix trees which inherently avoid costly operations such as node split and merge in typical tree-based approaches, while our new design on trie structure selection renders a nice space optimization.

Challenges of edge index. Existing dynamic graph systems generally adopt one of three edge storage paradigms: (1) Packed Memory Array (*PMA-based*), (2) *log-based*, or (3) *tree-based*, each embodying distinct trade-offs among **update efficiency**, **efficient sequential scan**, and **space overhead**. *PMA-based designs* [4, 11, 69] reserve gaps within arrays for edge insertions, and rebalance the gaps during updates. This offers excellent spatial locality and sequential scan performance but incurs non-trivial space and maintenance costs to manage reserved gaps within the array. *Log-based approaches* [43, 78, 79] mainly store edge logs within an array, which delivers efficient sequential scan, low memory overhead and high insertion throughput via append-only updates. In this design, outdated entries must be located and invalidated through log traversal, incurring extra costs. *Tree-based structures* [31, 51] organize adjacent edges of each vertex in a hierarchical layout chained with pointers, which offer a balanced trade-off between update efficiency and memory usage but reduce sequential scan performance compared to flat or contiguous storage.

The problem. *Can we design a dynamic graph data structure that minimizes vertex space with high update/query efficiency, provides $O(1)$ edge operations, and preserves efficient sequential access?*

Our solution. To address the problem, we propose RadixGraph, a highly performant for both updates and reads and space-efficient dynamic graph system that tackles the problem in the following ways:

[1]https://docs.datastax.com/en/dse/6.9/graph/using/vertex-and-edge-ids.html

SORT: a space-efficient and high-performance vertex index. As an efficient data structure for indexing, the radix tree has emerged as a promising solution. The classical radix tree has a well-bounded query latency, coupled with its ability to avoid node splitting and merging when inserting vertices. Empirically, radix-tree-based solutions (Spruce [65], RadixGraph) show higher vertex insertion throughput than other systems based on multi-level vectors or ARTs (Figure 8(d)). Specifically, a recent work, Spruce [65], employs the van Emde Boas tree (vEB-tree), which is a variant of the radix tree, to reduce the space cost by adjusting the fanouts of the trees. However, their adjustment is sub-optimal in the trade-offs between space cost and query performance, and how to optimally adjust the fanouts to achieve the best trade-off remains an open issue. To address this challenge, we firstly identify which possible radix tree structures can index a given ID universe, and formulate a constrained optimization program to select the best one that minimizes the space cost. Effectively solving the program gives us the space-optimized radix structure, dubbed as SORT.

Snapshot-Log edge storage architecture: $O(1)$ operation time for insert, update, and delete with sequential access support. Traditional approaches often face a trade-off between the efficiency of insertions, updates, deletions or sequential scans. To address this challenge, we introduce a snapshot-log edge storage architecture that separates the adjacency array into two distinct segments: a read-only snapshot and an updatable log. All updates and deletions are directly appended to the log segment as delta entries, which are then compacted into the snapshot segment once the log becomes full. By configuring the size of the new log segment after compaction to match that of the new snapshot, we ensure that the cost of this lazy compaction process remains bounded, achieving an $O(1)$ amortized cost per operation while preserving an optimal $O(d)$ get-neighbor query time and efficient sequential scan. To our best knowledge, this is the first edge structure that simultaneously supports $O(1)$ insertion, update, and deletion while maintaining sequential access. The $O(1)$ complexity is optimal and independent of vertex degree, allowing our design to remain both theoretically promising and practically simple.

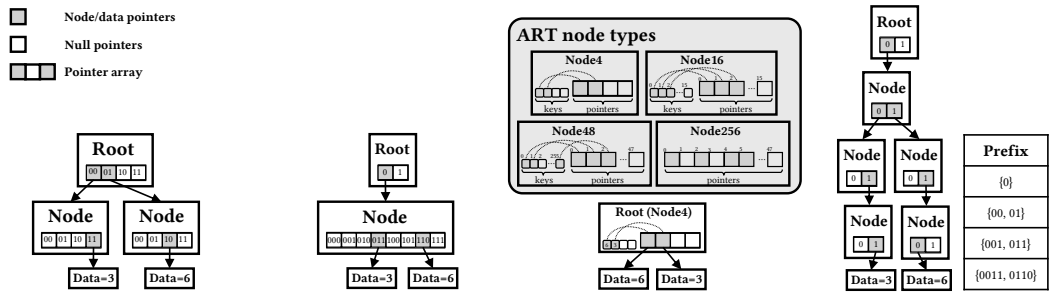
Contributions. We present RadixGraph, a novel in-memory data structure for dynamic graph storage that combines two innovative designs: a space-optimized vertex index (SORT) and a snapshot-log edge storage architecture. Empirically, we have used the widely adopted Graph Framework Evaluation (GFE) benchmark [22] to compare RadixGraph against the state-of-the-art in-memory graph storage structures. Results demonstrate that RadixGraph achieves up to 16.27× faster graph updates, and reduces memory consumption by an average of 40.1% compared to the strongest baseline.

2 Background

2.1 Radix tree and its variants

The radix tree (also known as trie) [13, 30, 42] is a highly efficient data structure for data management. Unlike other tree-like structures (e.g., B-tree [9], AVL tree [32] and red-black tree [8]), the radix tree does not explicitly store keys in its internal nodes. Instead, it distributes a key into individual parts across multiple nodes in different layers to reduce redundancy. Advanced radix tree techniques mainly contain path-compressed radix trees [46, 47, 59] and adaptive radix trees (ART) [47, 54].

Canonical l-layer radix tree. A canonical l-layer radix tree supports insertion and query operations in $O(l)$ time. Each layer partitions the key space by a fixed number of bits, and each node maintains a pointer array to their child nodes for further indexing. Specifically, if the i-th layer indexes a_i bits, then each node in that layer maintains a pointer array of size 2^{a_i}. Different ways of partitioning the key space lead to different radix tree structures. For example, Figures 1(a) and 1(b) illustrate two possible radix trees with $l = 2$ for indexing 4-bit integers. To retrieve the integer 3 (i.e., `0011`), the search in Figure 1(a) follows branch `00` in the first layer and then `11` in the second



(a) Two layers with 2+2 format (b) Two layers with 1+3 format (c) Adaptive radix tree (ART) (d) X-fast trie (bitwise trie)

Fig. 1. Different radix tree structures for storing 4-bit integers. Only integers 3 (i.e., `0011`) and 6 (i.e., `0110`) are inserted.

layer. In contrast, the search in Figure 1(b) follows branch `0` in the first layer and then `011` in the second layer. Given a key space $U = \{0, 1, \ldots, u-1\}$, a radix tree may occupy up to $O(u)$ space. In practice, however, it remains space-efficient because only keys that actually occur are materialized. Nevertheless, the null pointers within the pointer arrays can still incur memory overhead. For instance, the empty branches `10` and `11` in the root node of Figure 1(a) correspond to null pointers in the root node's pointer array, even though they do not store any valid entries.

Adaptive radix tree [47]. To mitigate the space wasted by null pointers in pointer arrays, the adaptive radix tree (ART) was proposed. ART typically indexes 8 bits per layer. Instead of allocating a fixed pointer array of size 2^8, ART defines four node types—`Node4`, `Node16`, `Node48`, and `Node256`—capable of storing up to 4, 16, 48, and 256 child pointers, respectively. The node type is dynamically chosen based on the number of non-empty child nodes. For example, Figure 1(c) illustrates an ART instance for indexing 4-bit integers. Since ART indexes 8 bits per layer, only one layer is required, and because there are only two keys stored, a `Node4` is allocated. ART significantly reduces space consumption compared to canonical radix trees. However, it incurs additional lookup cost for `Node4` and `Node16` types, as locating a child requires scanning the whole pointer array. Furthermore, insertion throughput may degrade due to node resizing operations when a node's capacity is exceeded.

X/Y-fast trie [71]. An X-fast trie is a special bitwise trie augmented with efficient successor query support. A bitwise trie corresponds to the case where $l = \lceil lg(u) \rceil$, where each layer indexes exactly one bit, as illustrated in Figure 1(d). To enable fast successor queries, the X-fast trie maintains a hash table for each layer, storing all prefixes that appear at that level. However, in workloads that do not require successor queries, the maintenance of these hash tables introduces additional space and update overhead. A Y-fast trie [71] extends the X-fast trie by combining it with $O(n/lg(u))$ red–black trees to store n elements. While this design improves space efficiency and supports $O(lglg(u))$ dynamic updates, the need to split and merge trees during updates can degrade practical performance and pose challenges to concurrency control.

2.2 Existing solutions and limitations

Numerous studies have investigated dynamic graph storage. In this work, we focus on *in-memory* dynamic graph systems, which typically consist of a vertex index and specially designed edge storage structures to support both vertex and edge updates.

Vertex index designs. The vertex index aims to support fast vertex updates with minimal memory overhead. Most existing dynamic graph systems employ a large pre-allocated vertex array for indexing [25, 27, 43, 51, 56, 79], typically of size 2^{32} that corresponds to the common range of vertex

identifiers. However, this approach results in substantial memory waste, as vertex identifiers in dynamic graphs are often non-contiguous, leading to fragmentation and inefficient space utilization. To address this issue, recent systems adopt adaptive radix trees (ART) [22], van Emde Boas trees (vEB-trees) [65], or multi-level vector indices (often integrated with an external hash map) [31, 78]. Although these adaptive structures provide more flexible indexing, they may incur additional time overhead from frequent structural modifications during insertions. For example, multi-level vector indices trigger hash map resizing and rehashing once the load factor is exceeded, while ART must resize and migrate node arrays when a node becomes full. In contrast, the vEB-tree achieves high throughput with its fixed structure, but this rigidity gives further room to optimize space efficiency and adaptability to different ID spaces.

Adjacency list-based edge structures. Dynamic graph systems based on adjacency lists primarily address two key challenges: (1) improving the efficiency of edge insertions and deletions, and (2) linking edge blocks to enhance read performance. Stinger [25] organizes edges into blocks to improve read efficiency. GraphOne [43] adopts a hybrid approach by combining an adjacency list with an edge log list, where new edges are initially logged and later batch-materialized into the adjacency list. Similarly, LiveGraph [79] and GTX [78] introduce the Transactional Edge Log (TEL) to manage edge modifications. However, they require traversing the entire adjacency list to locate and remove edges, leading to $O(d)$ complexity per update or deletion, where d is the degree of the vertex. To enhance efficiency, RisGraph [27] employs an indexed adjacency list combined with sparse arrays, significantly improving read performance while reducing traversal overhead. Sortledton [31] and GastCoCo [51] replace traditional edge lists with unrolled skip lists and B+ trees, respectively, achieving efficient edge insertions, updates, and deletions in $O(lg(d))$ time complexity. Similarly, Aspen [24] and CPAM [23] introduce C-tree, an improved version of the B+-tree that also supports $O(lg(d))$ time complexity, which enhances cache locality and is optimized for parallel batch-updates. Despite their efficient updates, these approaches introduce additional index structures for managing edge lists, which increase overall space consumption. Spruce [65] buffers newly inserted edges in a fixed-size array, which is merged into the sorted array once full. While this approach achieves high space efficiency, the merging process introduces an amortized $O(d)$ cost per edge insertion.

CSR-based edge structures. Dynamic graph systems based on CSR primarily aim to support efficient edge insertions and deletions while preserving cache-friendly access patterns. CSR++ [28] introduces a segmented CSR representation that divides the edge array into multiple segments, which reduces the need for full reconstruction. However, it still includes local shifting costs within segments during updates. LLAMA [56] takes a versioned approach by maintaining multiple graph snapshots, enabling efficient temporal queries but at the cost of increased memory usage. Another line of work replaces CSR's contiguous edge array with a Packed Memory Array (PMA) [4, 11, 69], which maintains edges in a partially sorted, dynamically resizable array. For instance, Teseo [22] stores PMAs in the leaf nodes of its vertex index, while Terrace [61] combines sorted arrays, PMAs, and B+ trees to handle vertices of varying degrees. PPCSR [70] further enhances the PMA to support concurrent operations. Compared with previous approaches, PMA provides efficient updates with complexity guarantees while preserving sequential access. The main challenge for PMA is the practical trade-off between space consumption and update efficiency. Under highly dynamic workloads, PMA requires rebalancing and can fragment memory or temporarily reduce update throughput. Although the theoretical space of most edge structures is $O(m)$, the practical trade-off may result in difference in actual space consumption.

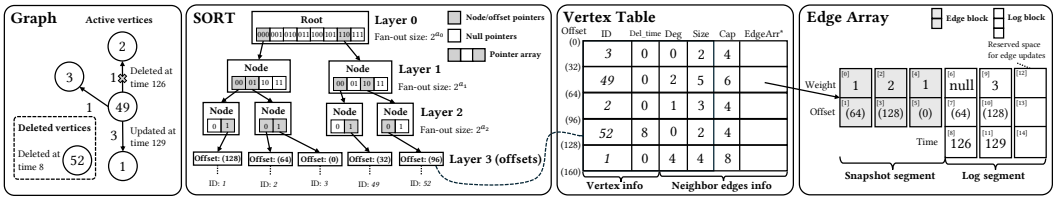


Fig. 2. The high-level overview of RadixGraph. Parenthesis "(x)" represents an offset corresponding to the location in the vertex table. The fan-out size 2^{a_i} is the pointer array size of a node at layer i and is predetermined by our optimizer. In this example, $a_0 = 3, a_1 = 2, a_2 = 1$.

3 Design of RadixGraph

Our graph system, RadixGraph, identifies a space-optimized radix tree (SORT) as the vertex index and stores the vertex information in a vertex table. RadixGraph also facilitates a novel snapshot-log edge structure supporting $O(1)$ time for edge insertions, updates and deletions. Figure 2 shows a high-level overview of RadixGraph. In this example, there are 5 vertices with IDs 3, 49, 2, 52 and 1 inserted sequentially, and their information is stored in the vertex table. Since the vertex IDs are 6-bit integers, the SORT has 3 layers that indexes 3 bits, 2 bits and 1 bit, respectively. The SORT indexes these vertex IDs and offers their locations (i.e., offsets) in the vertex table. For example, the binary format of ID 52 is `110100`, and its offset (96) can be thereby retrieved by traversing these bits in SORT. Each vertex is associated with an edge array which is equally partitioned into a snapshot segment and a log segment, where the snapshot segment stores the edges and the log segment stores the recent edge updates of that vertex. For example, in the edge array of vertex 49, the log block $(null, (64), 126)$ represents deleting the edge from vertex 49 to 2 at timestamp 126.

In the following, we will show the vertex storage structure in Section 3.1 and how to optimize SORT in Section 3.2. Then we introduce the edge storage in Sections 3.3, respectively. Moreover, we discuss the complexities and concurrency control in Section 3.4.

3.1 SORT: a Space-Optimized Radix Tree

Motivation. RadixGraph employs a radix-tree-like structure to map vertices to their locations in the vertex table. As discussed in Section 2.1, there are multiple possible structures for a radix tree. For canonical l-layer radix trees, a common practice is to assign the same fan-out size to different layers, and we denote it as uniform-tree. Given n integers in an integer universe $U = \{0, 1, \cdots, u-1\}$ and the number of layers l, the fan-out size of a uniform-tree is $2^{\lceil lg(u)/l \rceil}$. Moreover, the Van Emde Boas tree (vEB-tree) [68] recursively partitions the universe into $O(\sqrt{u})$ subtrees. For instance, when $u = 2^{64}$, the top-level fan-out is 2^{32}, followed by a second layer with fan-out 2^{16}, leading to a depth of $O(lglg(u))$. However, both of them apply a rigid fan-out size across layers, which often fails to strike an optimality balance between space usage and efficiency. To address this, RadixGraph introduces a novel **S**pace-**O**ptimized **R**adix **T**ree (SORT) that determines the optimal fan-out at each layer according to graph data, which minimizes the average space costs given the number of layers l of the radix tree. In Table 2, we set the same l for different radix tree structures to align the query efficiency, which is $O(l)$, and compare the performances of them. SORT exhibits the least memory usage since it optimizes the space cost, and this also benefits the insertion efficiency. For ART, although it introduces flexible node types to index its children and exhibits competitive space consumption compared with SORT, its efficiency is slower due to the need to transform between these node types. Detailed comparisons and analysis are provided in Sections 4.6 and 4.7. X/Y-fast trie can be viewed as an extended bitwise trie, which fits the canonical l-layer radix tree case when $l = O(lg(u))$. Therefore, the SORT optimization scheme can also be integrated for these

n	Uniform-tree		vEB-tree		SORT	
	Memory	Insertion	Memory	Insertion	Memory	Insertion
10^3	4.38MB	6.8ms	2.62MB	4.7ms	**0.85MB**	**2.8ms**
10^4	38.11MB	29ms	20.41MB	23ms	**5.23MB**	**16ms**
10^5	295.60MB	172ms	119.12MB	118ms	**32.09MB**	**86ms**

Table 2. Performances of different $O(lglg(u))$-layer radix tree structures for inserting n random IDs in $[0, 2^{32} - 1]$.

variants. However, they are mainly optimized for successor queries and require maintaining extra components to support them, which are redundant for our graph storage task since we do not require successor queries.

Structure of SORT. Figure 2 shows the high-level structure of a SORT. Specifically, if the depth of the tree is set as l ($l \geq 1$), then SORT begins with a root node at layer 0, and each node in layer i has an array of 2^{a_i} pointers, where a_i would be pre-determined by the optimizer, which will be introduced shortly in Section 3.2. Each pointer is either a null pointer or points to a child node at layer $(i + 1)$. Specifically, the leaf nodes at layer l store the byte offsets of their corresponding vertices, which correspond to their locations in the vertex table.

Algorithm 1: Insert-Vertex($N, i, v, a[]$)

Input: the current tree node N and its layer i and the vertex identifier v in binary format; a_i is pre-determined by the optimizer.

```
1  x ← the first a_i bits of v;
2  if N.children[x] is a null pointer then
3      if i < l − 1 then
4          Create a child node c, initialize its pointer array of size 2^{a_{i+1}} and store the pointer of
             c in N.children[x];
5      else
6          Store the vertex in a new entry of the vertex table;
7          Store the offset of the vertex in N.children[x];
8          return;
9  Delete the first a_i bits of v;
10 Insert-Vertex(N.children[x], i + 1, v, a);
```

Algorithm 1 details the process for inserting a vertex into SORT. The idea is simple yet effective: the vertex identifier is divided into individual segments in its binary format, and each segment is processed in a corresponding layer of the tree. At each layer, the value of the segment determines the specific child node to which the vertex is assigned. If the corresponding child node has not been created, we create a child node and store its pointer. The process continues recursively until the vertex is created and its offset is stored at the l-layer in the tree.

The process of retrieving a vertex from SORT follows a similar approach to Algorithm 1, which costs $O(l)$ time. By traversing the tree based on the segments of the vertex identifier, the algorithm either returns the desired vertex offset or returns a null value (e.g., -1) if the vertex has not been stored.

Vertex table. RadixGraph maintains an expandable vertex table with segmented storage, implemented using Intel's TBB concurrent vector [1]. When a segment reaches capacity, a new segment is allocated with twice the size of the previous one. The previous segments are kept such that related updates on vertices within them can process without being blocked. This segmented design

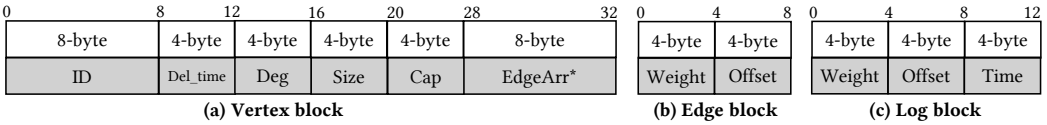


Fig. 3. The data layout of vertex, edge and log blocks.

ensures that once a vertex is inserted, its physical location remains fixed, avoiding data movement and preventing read-write conflicts. Figure 3 illustrates the layout of a vertex block, which consists of the vertex ID (i.e., *ID*), the deletion timestamp (i.e., *Del_time*), the degree of the vertex (i.e., *Deg*), the number of occupied blocks in the edge array (i.e., *Size*), the total number of blocks in the edge array (i.e., *Cap*), and a pointer to the edge array (i.e., *EdgeArr**). By default, *Del_time* is initialized to 0, indicating the vertex is active. When a vertex is deleted, RadixGraph sets *Del_time* to the current timestamp t, making the vertex invisible to transactions with timestamps greater than t.

For garbage collection, the offsets of deleted vertices are stored in a queue. Figure 2 shows an example where vertex 52 is deleted. The queue will store its offset (96), and when there is a vertex insertion, the offset is retrieved from the queue and the new vertex will reuse this offset and its corresponding slot in the vertex table. More specifically, when a new vertex is inserted, the system first checks the queue for reusable slots via atomic compare-and-swap (CAS) and only expands the vertex table when there are no available slots. Deleted vertices are only purged from the queue when all transactions before "Del_time" are finished for MVCC consistency.

3.2 Optimizing SORT configuration

Problem definition. Let the vertex identifier X be a discrete random variable uniformly distributed over the integer domain $\{0, 1, \ldots, 2^x - 1\}$, denoted as $X \sim \mathcal{U}_d(0, 2^x - 1)$. Given n distinct identifiers sampled from this domain and a fixed number l, we aim to find an optimal radix tree configuration within the family of canonical l-layer radix trees. Formally, our objective is to determine the structure $T^* \in \mathcal{T}_l$ that minimizes the average space consumption:

$$T^* = \arg\min_{T \in \mathcal{T}_l} \mathbb{E}_{X \sim \mathcal{U}_d(0,2^x-1)}[\text{Space}(T, X, n)],$$

where $\mathcal{T}_l$ denotes the set of canonical l-layer radix trees capable of indexing n distinct x-bit integers. Although our analysis assumes a uniform integer domain, the proposed method remains effective under skewed workloads. Empirical evaluation on such workloads is provided in Section 4.6.

Overview. We now present an overview for determining the optimal structural configuration of SORT. The objective is to minimize the average space cost of the tree under distinct and uniformly distributed integer identifiers, formulated as:

$$\min_{a_i \in \mathbb{N}} \quad \sum_{i=0}^{l-1} 2^{a_i} \cdot N(i) \cdot p(i), \quad \text{s.t.} \quad 2^{a_0 + a_1 + \cdots + a_{l-1}} \geq 2^x$$

Here, a_i is the logarithmic fan-out size of layer i for the SORT structure. Given a_i for all $0 \leq i \leq l-1$, $N(i)$ is defined as the maximum number of nodes that can be instantiated in layer i, and $p(i)$ denotes the probability that a node at layer i is instantiated. The optimization requires input l, x and n, where l is the number of layers, x is the bit-length of each vertex identifier and $n \geq 1$ is the total number of vertices.

The objective function expresses the total space consumption of SORT as a layer-wise summation, where $N(i) \cdot p(i)$ represents the expected number of instantiated nodes in layer i, and 2^{a_i} corresponds to the pointer array size of each node. Thus, the objective is the expectation of $\text{Space}(T, X, n)$, representing the average space usage of a canonical l-layer radix tree under uniform identifiers.

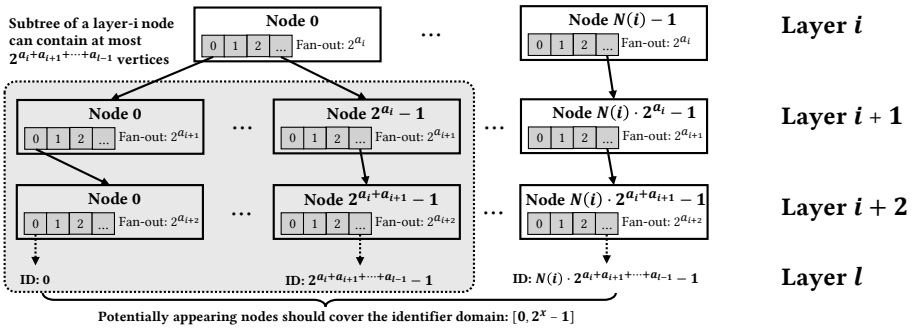


Fig. 4. The representation intervals of i-layer nodes.

The constraint defines the feasible space $\mathcal{T}_l$, ensuring that SORT spans the entire identifier domain $[0, 2^x - 1]$. In the following, we derive analytical formulations for $N(i)$ and $p(i)$, which enable a closed-form optimization of the above objective.

Objective function formulation. For layer i, as shown in Figure 2, each internal node contributes to a pointer array of size 2^{a_i}. Specifically, for layer 0, since $n \geq 1$, the root node must be created and contain an array of 2^{a_0} child pointers, and $N(0) = 1, p(0) = 1$. For a node at layer i ($i > 0$), its subtree can store at most $2^{a_i+a_{i+1}+\cdots+a_{l-1}}$ graph vertices, whose IDs span an interval $[L, R]$, and all intervals of nodes at layer i are pairwise disjoint. For example, as shown in Figure 4, where layer 0 to $l-1$ stores internal tree nodes and layer l stores graph vertices, the internal node 0 in layer i covers graph vertices within an ID interval $[0, 2^{a_i+a_{i+1}+\cdots+a_{l-1}} - 1]$, which are stored in layer l. For simplicity, we denote:

$$S_i = R - L + 1 = 2^{a_i+a_{i+1}+\cdots+a_{l-1}}$$

as the length of the node interval for layer i. Therefore, the root node (or equivalently, the whole SORT) can accommodate at most $S_0 = 2^{a_0+a_1+\cdots+a_{l-1}}$ graph vertices, which should cover the identifier domain $[0, 2^x - 1]$ and thus we derive the constraints of the optimization.

We observe the optimal configuration must satisfy $S_i \leq 2^x$ for $0 \leq i < l$. Otherwise, the covered ID interval of the leftmost node at layer i can already represent all the vertices within $[0, 2^x - 1]$. Then $N(i) = 1$ and $p(i) = 1$, and reducing a_i to $a_i - 1$ can result in a smaller objective value. Given the form of S_i and the fact that $S_i \leq 2^x$, this indicates that S_i always divides 2^x and a layer i tree node can thereby be classified as two cases: (1) its interval lies entirely within the domain $[0, 2^x - 1]$; (2) its interval lies completely outside this domain.

For case (2), the node would never be created since its interval exceeds the range of vertex identifiers, and thus does not contribute to the space cost. Therefore, the maximum number of tree nodes $N(i)$ that can appear in layer i corresponds to the nodes of case (1):

$$N(i) = \frac{2^x}{S_i} = 2^{x-(a_i+a_{i+1}+\cdots+a_{l-1})}$$

For case (1), we consider the complement event that the node is **not** created, which means that all n vertices are distributed in other $(2^x - S_i)$ possible IDs out of its corresponding interval. Since all vertex identifiers are distinct integers, the probability of the complement event is $\binom{2^x - S_i}{n} \Big/ \binom{2^x}{n}$ by hypergeometric distributions [26], and we immediately have the probability of this node being created as:

$$p(i) = 1 - \binom{2^x - S_i}{n} \Big/ \binom{2^x}{n} = 1 - \frac{(2^x - 2^{a_i+\cdots+a_{l-1}})!(2^x - n)!}{(2^x)!(2^x - 2^{a_i+\cdots+a_{l-1}} - n)!}$$

where $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ is the combination number representing the number of different ways to take b items from a distinct items. Note that $\binom{a}{b} = 0$ when $a < b$, meaning $p(i) = 1$ when $2^x - S_i < n$.

Putting everything together, now we are ready to formulate the complete optimization problem:

$$\min_{a_i \in \mathbb{N}} 2^{a_0} + \sum_{i=1}^{l-1} 2^{x-(a_{i+1}+\cdots+a_{l-1})} \left(1 - \frac{(2^x - 2^{a_i+\cdots+a_{l-1}})!(2^x - n)!}{(2^x)!(2^x - 2^{a_i+\cdots+a_{l-1}} - n)!}\right)$$
$$\text{s.t.} \quad a_0 + a_1 + \cdots + a_{l-1} \geq x$$

Solving the optimization problem. Directly solving the optimization problem via integer programming incurs exponential time complexity. Fortunately, by exploiting structural properties of the objective function, the problem can be simplified and solved in polynomial time.

To simplify the optimization problem, we define $s_i = \sum_{j=0}^{i} a_j$. Then $s_j \in \mathbb{N}, s_{l-1} \geq x$ and $s_j \leq s_{j+1}$ for all $0 \leq j < l-1$. We can rewrite the objective function as:

$$f(s_0, \cdots, s_{l-1}) = 2^{s_0} + \sum_{i=1}^{l-1} \frac{2^x}{2^{s_{l-1}-s_i}} \left(1 - \frac{(2^x - 2^{s_{l-1}-s_{i-1}})!(2^x - n)!}{(2^x)!(2^x - 2^{s_{l-1}-s_{i-1}} - n)!}\right)$$

We now present its optimality condition in Lemma 1, which shows the optimal tree structure should fit into the interval $[0, 2^x - 1]$ exactly. That is, its representation range should be $[0, 2^x - 1]$.

Lemma 1. *(Proof in Appendix A [73]) There exists an optimal solution $(s_0^*, s_1^*, \cdots, s_{l-1}^*)$, such that $s_{l-1}^* = x$.*

Then our problem becomes how to select $s_0, s_1, \cdots, s_{l-2}$ to optimize the internal structure of the tree, and the objective function f can be simplified by substituting s_{l-1} with x:

$$f(s_0, \cdots, x) = 2^{s_0} + \sum_{i=1}^{l-1} 2^{s_i} \left(1 - \frac{(2^x - 2^{x-s_{i-1}})!(2^x - n)!}{(2^x)!(2^x - 2^{x-s_{i-1}} - n)!}\right)$$

We observe that for the i-th summation term in f, its value only depends on s_i and s_{i-1}, which inspires us to solve the problem via **dynamic programming** that recursively solves the optimal value of the current state (s_i) from optimal previous states (s_{i-1})'s.

Specifically, we define an auxiliary function g, such that $g(i, j)$ represents the minimum average space cost for the first $(i+1)$ layers when $s_i = j$. In fact, $g(i, j)$ is the optimal value of considering 2^{s_0} and the summation of first i terms in f when $s_i = j$. Therefore, we could solve $g(i, j)$ via dynamic programming, and the transition is given by:

$$g(i, j) = \min_{0 \leq k \leq j} g(i-1, k) + 2^j \left(1 - \frac{(2^x - n)!(2^x - 2^{x-k})!}{(2^x)!(2^x - 2^{x-k} - n)!}\right) \tag{1}$$

with the initial states $g(0, j) = 2^j$ for all $0 \leq j \leq x$. The main idea of this dynamic programming, is to partition the tree by layers. For each layer i, we enumerate all possible values k for s_{i-1} and values j for s_i. Then we can derive the optimal inductions from layer $i-1$ to layer i for all $s_i = j$. The initial state $g(0, s_0)$ corresponds to 2^{s_0} and thereby equals 2^j when $s_0 = j$. Inductively solving $g(i, j)$ from $i = 0$ to $i = l-1$ gives us the globally optimal solution.

Theorem 1. *(Proof in Appendix A [73]) The optimal value of f is equal to $g(l-1, x)$.*

Tuning the depth of SORT. Generally, the depth l of the tree should not exceed x. Otherwise, there exists $a_i = 0$ for some i and these layers can be safely pruned to reduce space costs as discussed above. For dense universe (i.e., $n \approx 2^x$), the optimal l tends to be smaller since most possible

identifiers are present, and we can afford a large fan-out (i.e., large a_i) without wasting much space. For sparse universe (i.e., $n \ll 2^x$), the optimal l tends to be larger and a_i should be smaller to reduce excessive empty pointers. Specifically, when $n = 2^x$, the optimal setting for minimizing space is to set $l = 1$ and $a_0 = x$, where the total space is $O(n)$. Since the lookup efficiency of the vertex index depends on l, in practice, we can set l based on the required lookup efficiency to compute an optimal setting of a_i, and then prune all layers with $a_i = 0$ to derive an appropriate setting for the tree. For example, in our experiments, we set $l = O(lglg(u))$, where $[0, u-1]$ is the vertex ID universe to ensure a fair comparison with Spruce, which also utilizes a radix-tree-like structure (vEB-tree) with query complexity $O(lglg(u))$.

Implementation and adaptation. Given input n (the number of integers) and x (the bit-length of vertex IDs), we implement the iterative process in Equation 1 by for-loop structures, where the outermost loop enumerates i from 0 to $l-1$, the middle loop enumerates j from 0 to x, and the innermost loop enumerates k from 0 to j. This implementation is efficient and takes less than 1 second (5‰ of the graph construction time) to compute the optimal setting on our largest experimental dataset *twitter-2010*. As n evolves, the optimal configuration of SORT may gradually shift. However, in real-world graphs, vertex insertions are relatively infrequent compared to edge updates. For instance, Twitter's daily active users increased from 115 million to 237.8 million between 2017 and 2023[2], less than a double growth over six years. Empirically, we observe that SORT's optimal configuration remains largely stable with respect to n; the adjustment intervals grow roughly exponentially with powers of 2 in n (see Section 4.6). Moreover, the memory overhead remains small even if the configuration is not updated immediately.

To minimize disruption, we recompute the optimal configuration asynchronously and trigger an update only when n changes exponentially. When an update is required, we employ a lazy transformation strategy: subtrees with unchanged fanouts are preserved by reusing their existing pointers, while only the affected upper levels are reconstructed. As shown in Section 4.6, even when SORT is aggressively adapted whenever its configuration changes, each transformation completes within about 1 second, which clearly shows the transformation costs are negligible relative to overall system performance.

Supplementary information. Due to the space limit, we refer readers to the appendices in our extended version [73] for detailed illustrations, proofs and complexity analysis.

3.3 Snapshot-log edge storage

RadixGraph adopts an enhanced log-based structure for edge storage, since it exhibits high edge insertion throughput. Other log-based graph systems such as LiveGraph and GTX [78, 79] also achieve constant-time edge insertions by appending edges to a per-vertex log array, and they support multi-version concurrency control (MVCC) by maintaining historical edge versions. However, this comes at the cost of low update and deletion efficiency, as each modification requires traversing the whole log array to locate and invalidate the previous version.

RadixGraph maintains an edge array per vertex that is evenly partitioned into a snapshot segment and a log segment. For instance, in Figure 2, vertex 49 maintains an edge array of 6 blocks, with 3 blocks in the snapshot segment and 3 in the log segment. The key benefit of this layout is that it enables an out-of-place update strategy which avoids costly traversal during updates: edge updates are first appended to the log segment and are later compacted in batches. When the log segment becomes full (i.e., when the "Size" field reaches the "Cap" field), a compaction is triggered that combines the snapshot and log segments to produce a new snapshot segment containing only the latest versions of edges. The size of this snapshot segment equals the vertex degree d (recorded

[2]https://famewall.io/statistics/twitter-stats

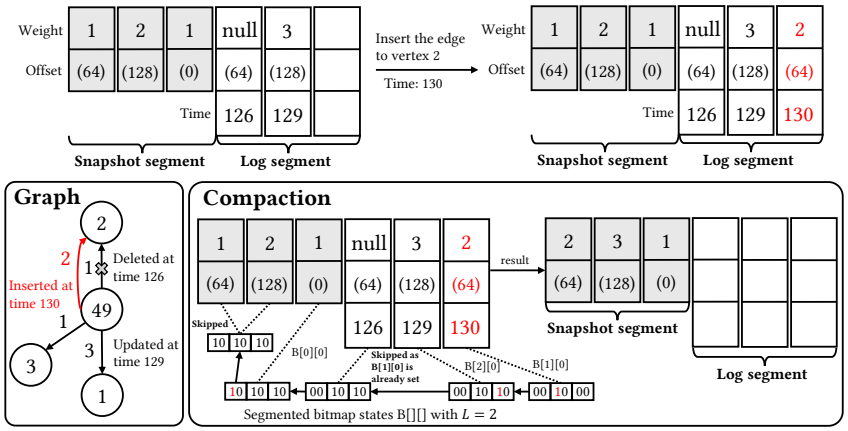


Fig. 5. Example edge update and compaction processes. Offset (x) corresponds to the bit $B[(x/32)/L)][(x/32) \bmod L]$.

in the "Deg" field of the vertex block). Since the entire edge array has capacity $2d$, the number of outdated entries is always bounded by $O(d)$. Consequently, all edge operations run in amortized $O(1)$ time: appending an update to the log segment costs $O(1)$, and although each compaction takes $O(d)$ time, this cost is amortized over the d updates processed by the compaction. As a result, the amortized cost of compaction is $O(1)$ per update.

Edge array. Each vertex is associated with an edge array composed of two segments: a snapshot segment storing edge blocks and a log segment storing log blocks. Figure 3 illustrates the layout of these edge and log blocks. Each edge block stores its metadata (e.g., *Weight*) along with an *Offset* field. The *Offset* field stores the destination vertex's position in the vertex table to track its information, and source vertex is not stored in the edge and log blocks since its information is already presented in the corresponding vertex block. For log blocks, an additional *Time* field is included to support edge versioning. Specifically, if the log entry represents an edge insertion or update, the *Weight* field holds the updated weight. If the log entry corresponds to a deletion, the *Weight* field is set to NULL (e.g., 0). For each operation, the source and destination vertices are first located or inserted into SORT and the vertex table. Then, a log entry is appended to the log segment according to the operation type. Therefore, each directed edge consumes only 1 block; for undirected edges, it will consume 2 blocks since both directions of the edge are inserted.

- Insertion: Adds an edge to the graph, appending a log with the edge's properties and the byte offset of the destination vertex;
- Update: Modifies the properties of an existing edge, appending a log with the edge's updated properties and the byte offset of the destination vertex;
- Deletion: Removes an edge, appending a log with a NULL property and the byte offset of the destination vertex.

Figure 5 illustrates an example of the edge update process, where the insertion of edge from vertex 49 to vertex 2 is firstly appended to the log segment and then triggers a compaction since the edge array is full. More specifically, when the log segment is not full, the update process is lock-free and costs $O(1)$ time: each edge log obtains a unique slot via an atomic *fetch_add* on the *Size* field. When the edge array becomes full (i.e., "Size"=="Cap"), a lock is acquired to perform log compaction. Specifically, if the current degree of the vertex is d, the capacity of the new array after compaction is $2d$, reserving space for d additional log entries.

Log compaction. As edge logs are appended independently and may create multiple entries for the same destination, the compaction procedure must identify and retain only the latest valid edge among these duplicates. To address this, we introduce *duplicate checker* for each thread to efficiently identify the latest version of an edge. Algorithm 2 shows the process. During log compaction, we perform a reverse iteration on the array, such that edge logs are traversed from most recent ones to least recent ones. For each edge log, if the duplicate checker finds that its destination vertex has not been visited, this means the edge is the latest version in the current graph, and the edge is thereby added to A; otherwise, it is skipped. After processing each edge log, the duplicate checker sets the destination vertex as visited. This ensures outdated edge logs to be skipped and only latest edges are retrieved from the array. After the whole edge array is processed, we reset all vertices as unvisited in the duplicate checker.

Algorithm 2: LogCompaction(O_u)

Input: the target vertex offset O_u.

```
C ← the duplicate checker of current thread;
A ← an edge array of size 2 × V[O_u].Deg;
E_u ← V[O_u].EdgeArr, s ← V[O_u].Size, cnt ← 0;
for i = s − 1, s − 2, ··· , 0 do
    O_v ← E_u[i].Offset;
    if C finds v not visited and E_u[i].Weight ≠ null then
        A[cnt++] ← {E_u[i].Weight, O_u};
    Mark v as visited in C;
for i = 0, 1, ··· , s − 1 do
    O_v ← E_u[i].Offset;
    Mark v as unvisited in C;
V[O_u].EdgeArr ← A, V[O_u].Size ← cnt;
```

The duplicate checker employs a segmented bitmap to track visited vertices, where each vertex is mapped to a unique bit. The segmented bitmap includes a set of bitmaps (i.e., segments) of fixed length L. Before each compaction, it allocates additional segments as needed to ensure the total number of bits is at least n. Let B denote the segmented bitmap where $B[i][j]$ corresponds to the j-th bit of the i-th segment and O_v be the offset of the destination vertex v. Since each vertex occupies a 32-byte block in the vertex table, the logical ID of vertex v is $O_v/32$, which is mapped into $[0, n-1]$. Accordingly, the corresponding bit of vertex v is located at $B[(O_v/32)/L)][(O_v/32) \bmod L]$. If this bit equals 1, v has been visited; otherwise, it is unvisited. We provide a complete example about the bitmap states during the compaction in Figure 5. Initially, the bitmap has 3 segments and all bits are 0. For offset (64), its corresponding segment is $64/32/L = 1$, and corresponding bit in that segment is $(64/32) \bmod L = 0$. Therefore, $B[1][0]$ is set as 1. Later, when traversing all blocks with offset (64), they are skipped since the corresponding bit has already been marked.

Theorem 2. *Under a single-threaded execution model, the amortized time complexity of edge insertion, update and deletion is* $O(1)$.

Proof Sketch. We prove the theorem by discussing the two cases of edge operations: (1) when the log segment is not full, appending an edge log to the segment clearly costs $O(1)$ time; (2) when a compaction is required, the compaction process costs $O(d)$, and is amortized to $O(1)$ for each edge operation. The complete proof can be found in Appendix B of our extended version [73]. □

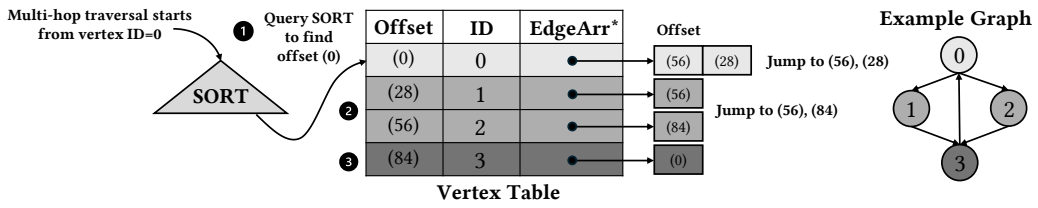


Fig. 6. An example of traversing multiple hops starting from $ID = 0$; Only step-① needs to query SORT.

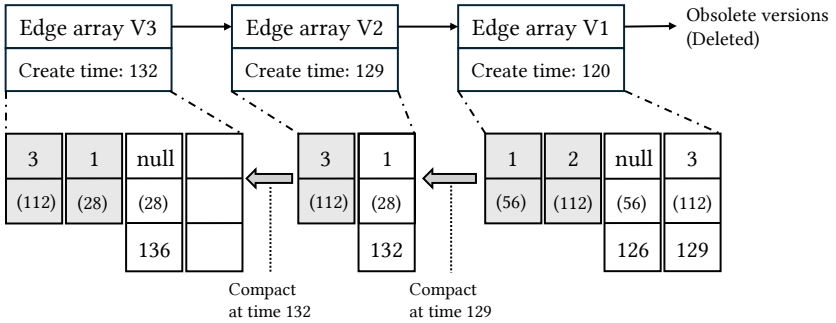


Fig. 7. An example multi-versioned edge arrays.

We remark that the concurrency overhead is low, as synchronization is only required in vertex-local compactions and does not block updates on other vertices.

Edge chain. Existing graph systems typically store the destination vertex IDs within their edge structures. Consequently, during graph traversals (e.g., multi-hop queries), each visited vertex must be looked up in the vertex index to retrieve its corresponding neighbor edges. For example, when performing a breadth-first search (BFS) [21] starting from vertex 0 on the graph shown in Figure 6, the system must perform four vertex index lookups—one for each of the visited vertices: 0, 1, 2, and 3.

In contrast, the edge array in RadixGraph stores vertex offsets instead of vertex IDs. By storing destination vertex's offset, the edges form a chain structure that directly represents the topological structure of the original graph, as shown in Figure 6. This effectively creates a direct path from one vertex to its neighbors, allowing traversal to proceed seamlessly along this chain. For the same BFS process starting from vertex 0, RadixGraph only needs to perform a one-time lookup in the SORT to locate the starting vertex. Once the offset of the starting vertex is known, the traversal can continue through the edge chain by following the offsets embedded in the edge blocks without further vertex index accesses.

Version management. Many graph analytics workloads require access to a consistent snapshot of the graph. RadixGraph supports multi-version concurrency control (MVCC) [12] by associating timestamps with both the snapshot segment and each appended log block. Multiple versioned edge arrays are organized as a linked list, as illustrated in Figure 7. A versioned array becomes eligible for deletion only after it has been compacted or its associated vertex has been deleted, and it is no longer accessed by any readers. For a graph analytics workload timestamped t, the system first traverses the linked list to locate the most recent versioned array created before t, then filters the log segment to include only updates with timestamps less than or equal to t. This design allows multiple analytics tasks to operate concurrently on consistent, historical views of the graph.

Operation	Teseo	Spruce	GTX	RadixGraph
locate_v	$O(lg(u))$	$O(lglg(u))$	$O(1)$	$O(lglg(u))$
insert_v	$O(lg(u))$	$O(lglg(u))$	$O(1)$	$O(lglg(u))$
delete_v	$O(lg(u)+d)$	$O(lglg(u)+d)$	$O(d^2)$	$O(lglg(u)+d)$
insert_e	$O(lg^2(d))$	$O(d)$	$O(1)$	$O(1)$
delete_e	$O(lg^2(d))$	$O(lg(d))$	$O(d)$	$O(1)$
update_e	$O(lg^2(d))$	$O(lg(d))$	$O(d)$	$O(1)$
get_ngbrs	$O(d)$	$O(d)$	$O(d)$	$O(d)$

Table 3. Time complexity of fundamental operations. n is the number of vertices, u is the size of the range of vertex identifiers and d is the average degree of vertices. "get_ngbrs" represents the get-neighbors operation.

3.4 Discussions

Operation complexity. The fundamental operations in a dynamic graph system include (1) vertex-oriented operations: locate, insert or delete a vertex, (2) edge-oriented operations: insert, delete or update an edge, and (3) query operation: get neighbors of a vertex.

We compare the time complexities of RadixGraph on fundamental operations with recent state-of-the-art works in Table 3. With l layers in the SORT, all vertex-related operations can be performed in $O(l)$ time. Note that l is a hyperparamter, and we set $l = O(lglg(u))$ to achieve the same complexity as Spruce's vertex index while minimizing its average space cost compared to Spruce. While GTX achieves better complexity in some vertex operations, it relies on a concurrent hashmap for indexing vertices, which becomes significantly slower as n grows large. For edge-related operations, RadixGraph requires only $O(1)$ time complexity for inserting, updating, or deleting an edge as shown in Theorem 2. For get-neighbors operation, RadixGraph costs $O(d)$ by performing a process similar to the log compaction.

Memory consumption. Let n denote the number of vertices and m the number of edges. When vertex IDs are not excessively sparse, the worst-case space complexity of SORT grows linearly with n, say, kn for some k. The empirical verification of this property is presented in Section 4.6 "SORT space consumption as n increases", while the detailed theoretical analysis is provided in Appendix C of our extended version [73]. The vertex table consumes at most $64n$ bytes of memory since each vertex block costs 32 bytes and the vertex table size is at most twice the number of vertices. For the edge arrays, each edge block occupies 8 bytes, so the total memory of the snapshot segments does not exceed $8m$ bytes. Since the log segment size is set equal to the snapshot segment size and each log block costs 12 bytes, the total memory of the log segments is bounded by $12m$ bytes. Each duplicate checker maintains a segment bitmap of $L\lceil\frac{n}{L}\rceil$ bits, requiring $L\lceil\frac{n}{L}\rceil/8$ bytes. Let t be the maximum number of worker threads allowed for concurrent compactions (typically a constant such as 32 or 64), the total bitmap space is $t \cdot L\lceil\frac{n}{L}\rceil/8$ bytes. Therefore, the overall space required to store the graph is $O(kn + 64n + 8m + 12m + tL\lceil\frac{n}{L}\rceil/8) = O(m)$.

Concurrency control. Concurrency control is another critical factor in the performance of a graph system. Our vertex index concurrency protocol builds on Read-Optimized Write EXclusion (ROWEX) [48, 65]. RadixGraph employs an atomic bitmap in each internal node of the SORT, where an i-layer node maintains a bitmap of size 2^{a_i}. Each bit in the bitmap uniquely corresponds to a child node. When a new child node is created, the corresponding bit in its parent node is atomically set to 1 using an atomic compare-and-swap (CAS) operation [38], indicating an ongoing creation process, and is reset to 0 once the creation completes. For readers, they simply read the data atomically. To support concurrent edge operations, RadixGraph maintains an atomic *Size* field in its vertex block, and each log entry obtains a unique identifier by an atomic add operation on *Size*. For readers, they can perform latch-free read on the edge array.

Datasets	#Vertices	#Edges	Avg. Deg.	Max. Deg.
livejournal (lj)	4M	34M	17.35	14,815
dota	61K	51M	1663.24	17,004
orkut	3M	117M	76.28	33,313
g500-24 (g24)	9M	260M	58.70	406,416
uni-24 (u24)	9M	260M	58.70	103
twitter-2010 (twitter)	41M	1.46B	70.51	3,081,112

Table 4. Datasets used in our experiments. "K", "M", "B" represent "thousand", "million" and "billion".

Supporting graph types. RadixGraph is primarily designed for directed weighted graphs. However, the "Weight" field in edge and log blocks can be replaced with other properties to support other types of graphs like *temporal graphs* [72] and *labelled graphs* [5] as long as a "NULL" value is reserved for the field. RadixGraph can also be easily extended to support *multi-graphs*. In that case, each vertex block should maintain a counter to represent the number of edges inserted in its edge array, and each edge and log block should maintain an extra "ID" field. Upon inserting a new edge to that vertex, the edge gets a unique ID by incrementing the counter. During log compactions, edge and log blocks with the same IDs will be compacted and only the latest version is kept. If maintaining *vertex labels* [15] with MVCC consistency is required, we can also extend the vertex block to store a pointer referring the versioned label chain of the vertex.

4 Experiments

4.1 Setup

System environment. We perform our experiments on a dual-socket machine equipped with Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz processors and 250GB RAM. Each CPU has 48MB L3 Cache, 16 cores, and supports at most 32 threads. Unless otherwise specified, all experiments are executed with 64 threads running concurrently. All the codes were compiled with GCC 11.4.0 on Ubuntu Linux with O3 optimization.

Graph data. We use various graph datasets from SNAP [49] and LDBC graph analytics [39], as shown in Table 4. Two synthetic datasets include (1) *g500-24*, power-law graphs, and (2) *uni-24*, uniform-law graphs. These datasets provide diverse structural properties under different graph topologies. Four real-world datasets include (1) *livejournal*, the LiveJournal social network [14]; (2) *dota*, the relationships between game entities [34]; (3) *orkut*, a relationship graph capturing ground-truth communities from the Orkut social network [64]; and (4) *twitter-2010*, Twitter follower network as of 2010 [44]. Note that all the graphs are treated as undirected graphs, and each edge would be inserted in both directions.

Graph benchmark. Recently, some benchmarks are developed to evaluate in-memory dynamic graph systems [10, 22, 39, 67]. We evaluate RadixGraph and its competitors based on GFE (Graph Framework Evaluation) driver [22], a C++ driver to evaluate updates and analytics in dynamic structural graphs. For graph analytics, we implement parallel k-hop neighbor queries and graph algorithms with GAPBS (GAP Benchmark Suite) [10], including breadth-first search (BFS), single-source shortest paths (SSSP), PageRank (PR), weakly connected components (WCC), triangle counting (TC) and betweeness centraility (BC).

Competitors. We choose recent state-of-the-art in-memory dynamic graph systems for comparisons: Teseo [22], Sortledton [31], Spruce [65] and GTX [78], all implemented in C++ with parallelism support. We also compare RadixGraph with Terrace [61], Aspen [24] and CPAM [23], which are optimized for batch updates. Note that some methods failed to complete on some datasets either due to exceptions or not finishing within 24 hours, and hence omitted from the figures.

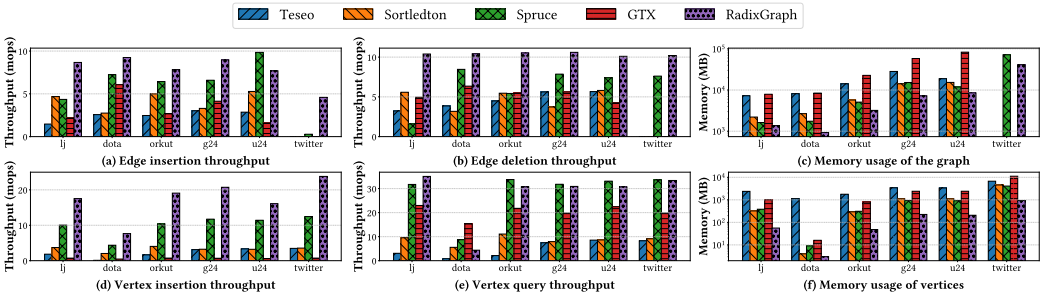


Fig. 8. Throughput of edge and vertex operations and memory consumptions.

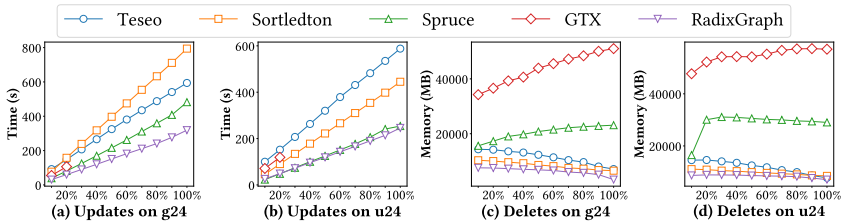


Fig. 9. The time cost of mixed edge updates and memory footprint of large-scale edge deletions for all methods.

We also note that these baselines may have different levels of transactional supports which correlates with operational performance or memory consumption. For example, GTX has a stronger transactional guarantee with snapshot isolation and serializable transactional support.

4.2 Dynamic operations

Edge insertion throughput. Figure 8(a) shows the insertion throughput of all methods across all datasets. On the twitter dataset, Sortledton, Teseo and GTX fail to finish in our experimental environment and thus are omitted. In this experiment, edges are randomly permuted and inserted in a round-robin manner and vertices are inserted alongside if they do not exist in the system. RadixGraph outperforms all other systems on most datasets, achieving up to 16.27× higher throughput than Spruce on the twitter dataset. Although Spruce performs best in the uniform graph (u24), its throughput degrades significantly on the twitter dataset. This is because Spruce appends neighbor edges to a fixed-size buffer, which is merged into a sorted array only when full. While this strategy yields near-constant insertion time when merges are infrequent, high-degree vertices in the twitter graph (up to 3 million neighbors) trigger frequent merges, resulting in degraded $O(d)$ insertion complexity. GTX employs a delta-chain index per vertex to enhance concurrency and provide transactional support. This design is beneficial for dense graphs like dota, where concurrent writes to the same vertex are common, allowing GTX to outperform Sortledton and Teseo, which rely on per-vertex latches. However, in sparser graphs with lower average degree, the delta-chain is underutilized, and transactional support inherently requires trade-offs with performance. Overall, RadixGraph generally has a good performance on different types of datasets thanks to its superior complexity, compact edge structure and latch-free log append process.

Edge deletion throughput. Figure 8(b) presents the deletion throughput of all methods across all graphs. This experiment directly follows the insertion benchmark, with edges deleted in the same round-robin order as they were inserted. We focus on evaluating the performance of the

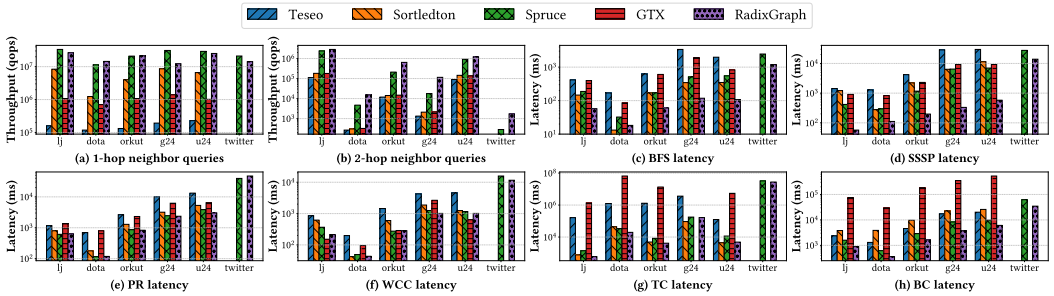


Fig. 10. Throughput of hop queries and latency of graph algorithms.

deletion phase in isolation. RadixGraph achieves the highest deletion throughput across all datasets, outperforming the second-best system by 1.23×-1.93× on the datasets, respectively. In general, we observe that most graph systems achieve higher throughput for deletions than for insertions. For Spruce, this is due to its deletion complexity being $O(lg(d))$, compared to its insertion complexity of $O(d)$. For the other systems, the primary reason is that edge deletions do not require writing edge properties, reducing the I/O and processing overhead.

Vertex operations. Figures 8(d) and 8(e) show the vertex insertion and query throughput of all methods. RadixGraph achieves the highest vertex insertion performance across all datasets, with Spruce being the second best. This advantage comes from the radix-tree-like structure, which avoids costly split, merge, or resize operations during updates. Compared with Spruce which employs a vEB-tree as its vertex index, RadixGraph benefits from its space-optimization model that minimizes memory consumption. The reduced space usage leads to fewer pointer allocations, thereby accelerating insertions. For vertex queries, RadixGraph and Spruce also outperform other methods in most cases, owing to their superior $O(lglg(u))$ query complexity. Although multi-level vector structures have a theoretical $O(1)$ query time, they require maintaining an external hash map for ID translation. Each query must first access the hash map before traversing multiple vector levels, and this complex design introduces overhead that ultimately degrades throughput. GTX has the unique benefit to support transactional and serializable graph operations; as a result, the additional coordination and validation required by its concurrency control mechanisms introduce extra overhead during vertex insertions, naturally leading to lower throughput even though it employs a similar vertex index as Sortledton's.

Mixed edge updates. Figure 9(a) and (b) show the update time footprint of all methods on synthetic datasets. This experiment follows the same setting as tested in the Teseo and Sortledton papers [22, 31]. GTX encounters an OOM exception on both datasets when processing 20% operations, and thus its curve contains only two data points. On g500-24, RadixGraph consistently outperforms the other methods. On uni-24, RadixGraph and Spruce exhibit similar performances for low vertex degrees. RadixGraph's curves show no latency spikes, demonstrating stable $O(1)$ amortized complexity and the log compactions do not significantly influence the efficiency.

4.3 Memory consumption

Storing the whole graph. Figure 8(c) shows the physical memory usage of storing dynamic graphs by edge insertions across different methods. On the twitter dataset, only Spruce and RadixGraph construct the graph successfully so we only report their results. We measure the memory difference before and after graph construction. RadixGraph demonstrates the lowest memory consumption on all datasets, using an average of 40.1% less space than the second-lowest memory consumption.

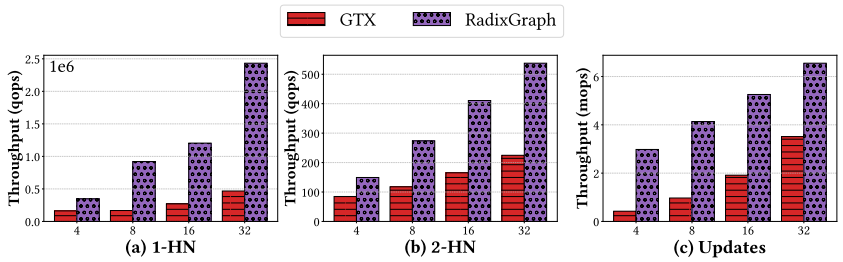


Fig. 11. The throughput of concurrent reads and writes on *dota* dataset over different number of threads. (a) refers to 1-hop neighbor query throughput; (b) refers to 2-hop neighbor query throughput; (c) refers to graph update throughput.

Although Spruce also employs a radix-tree-based structure (i.e., vEB-tree) for its vertex index, its structure remains fixed regardless of graph size, whereas RadixGraph optimizes space efficiency through our proposed integer programming model. GTX consumes the largest space on most datasets, since it uses 64-byte edge deltas and a delta-chain index per vertex. This exchanges the memory consumption with better transactional supports.

Storing vertices. Figure 8(f) reports the physical memory usage for storing only the vertices of each dataset. RadixGraph achieves the lowest memory consumption across all datasets. In contrast, Teseo exhibits consistently high memory usage regardless of graph size, as it stores both vertices and edges within the leaf nodes of its ART. Consequently, inserting a vertex immediately triggers the allocation of edge segments, even when no edges exist. Although Spruce also employs a radix-tree-like structure as its vertex index, its space efficiency is only comparable to that of Sortledton and does not show significant advantages. This is because Spruce's radix tree is not optimized for minimal space usage, unlike RadixGraph's structure, which is tuned through its optimization model.

Memory footprints of edge deletions. Figures 9(c) and (d) show the memory footprints during edge deletions on synthetic datasets. The memory usage of RadixGraph gradually decreases, with a more pronounced drop in the later stages of deletion. This behavior arises because in RadixGraph, the log segment size equals the snapshot segment size. Therefore, an edge array is compacted and recycled only after all its edges have been deleted (i.e., the log segment is fully filled). This observation suggests that, for delete-heavy workloads, reducing the log segment size relative to the snapshot segment size could enable more aggressive compaction and lower space overhead. Spruce initializes a linked list of versioned edges at the start of deletions and records each deleted edge in this list without employing any garbage collection mechanism. As a result, its memory footprint may even increase during edge deletions. GTX exhibits similar behavior, as its lazy garbage collection delays memory reclamation to reduce runtime overhead, resulting in temporary memory growth under delete-heavy workloads.

4.4 Graph analytics

In this subsection, we evaluate the efficiency of graph analytical tasks across different methods. Note that on the Twitter dataset, only Spruce and RadixGraph construct the graph and execute graph analytics successfully so we only report their results.

Neighbor queries. Figure 10(a) and (b) show the throughput of k-hop neighbor queries. The experiment first constructs the graph by inserting all edges, and then executes k-hop neighbor queries from every vertex in parallel. Our evaluation focuses on measuring the query throughput.

For 1-hop queries, RadixGraph and Spruce achieve the highest performance due to their compact edge array designs, which enable efficient sequential scans for neighbor retrieval. For 2-hop queries, RadixGraph outperforms Spruce by up to 6.11×, owing to its edge chain structure that avoids redundant vertex lookups when traversing the neighbors of 1-hop neighbors. Note that the time complexity of a 2-hop query is $O(d^2)$, so its throughput decreases on dense and power-law graphs (e.g., dota, g24, twitter) compared to uniform-law graphs (e.g., u24).

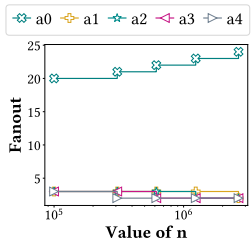
Graph algorithms. We implement two parallel single-source traversal algorithms (BFS and SSSP) and four parallel iterative graph algorithms (PR, WCC, TC and BC) based on the GAP Benchmark Suite (GAPBS). Single-source traversal algorithms start from a given vertex and explore its reachable subgraph, while iterative algorithms perform repeated computations over the entire graph. Figures 10(c) and (d) present the performance of each method on BFS and SSSP using a logarithmic scale. RadixGraph significantly reduces BFS and SSSP execution time on most datasets. This improvement is primarily due to RadixGraph's edge chain design, which enables direct traversal from a vertex without repetitive vertex index lookups. In contrast, prior methods require vertex index lookups at each step. Figures 10(e), (f), (g) and (h) show the performance of the systems on PR, WCC, TC and BC. The complexity of TC algorithm is much higher than other algorithms and GTX fails to finish within 24 hours on g24 dataset since its stronger transactional support degrades performance. For PR and WCC, although RadixGraph does not benefit from the same lookup avoidance in these algorithms (as they need to iterate the whole graph), it still delivers competitive performance across all datasets due to its compact edge array design. BC involves BFS executions in its process, so RadixGraph also benefits from the BFS speedup and outperforms other baselines. For TC, although it involves two-hop neighbor queries, the dominant cost lies in computing neighbor intersections rather than neighbor retrievals, so RadixGraph gains limited benefits in this case.

4.5 Concurrent reads and writes

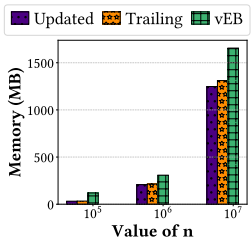
Real-world workloads often involve concurrent reads and writes. RadixGraph supports MVCC that allows read transactions to proceed without blocking or conflicting with concurrent write transactions. To evaluate performance under such workloads, we generate update operations following the procedure described in the mixed updates experiment and execute them concurrently with neighbor queries. For comparison, we include GTX, a state-of-the-art transactional graph system that also supports MVCC. Other systems are excluded from this experiment, as they either encounter deadlocks or segmentation faults under concurrent workloads, as also reported in the GTX paper [78]. Figure 11(a) and (b) show the concurrent read throughput for 1-hop and 2-hop neighbor queries, respectively, with a varying number of readers and fixed 32 writers. Figure 11(c) presents the concurrent write throughput when varying the number of writers while holding the number of readers constant at 32. RadixGraph shows strong scalability for both concurrent read efficiency and write efficiency.

4.6 Case study of SORT

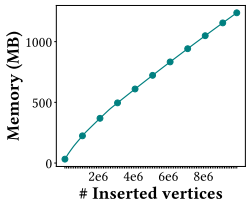
Robustness of SORT over n. We examine how the number of vertices n affects the optimal fanout configuration and memory consumption of SORT. We fix $u = 2^{32}$, meaning vertex IDs are distributed within the range $[0, 2^{32} - 1]$, and set the number of layers in SORT to $l = lglg(u) = 5$. Figure 12(a) shows how the optimal fanouts evolve as n increases. We observe that the changes occur at exponentially spaced intervals of n, with only five configuration shifts between 10^5 and 10^7. This suggests that the fanout can remain fixed over a broad range of n and only needs to be updated when n grows by roughly an order of magnitude. Additionally, the fanout values



(a) SORT fanouts



(b) Mem-costs



(c) Mem-footprint

n	Time
3.1E5	0.23s
6.2E5	0.46s
1.2E6	0.93s
2.7E6	1.96s

(d) Adapt-costs

Fig. 12. The (a) optimal configurations, (b) memory consumptions for different n; and (c) the memory footprint, (d) the transformation costs during insertions.

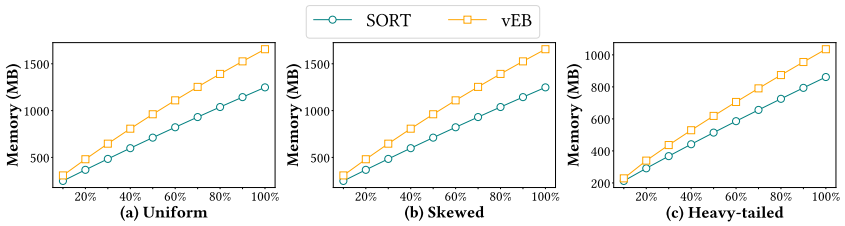


Fig. 13. The memory footprints of inserting 10^7 IDs into SORT and vEB-tree under different workloads.

a_i vary only slightly across different n, indicating that the optimal configuration is stable and robust. For example, when n changes from 50000 to 300000, the optimal configuration changes from $\{19, 4, 3, 3, 3\}$ to $\{20, 3, 3, 3, 3\}$. Figure 12(b) presents the memory usage across different n values under various radix tree configurations. The "Updated" bars represent the most recently computed optimal configuration, while the "Trailing" bars reflect the previously optimal configuration that has not yet been updated. The results show that using the updated configuration yields approximately 5% lower memory consumption than the trailing one. Nonetheless, the difference is modest, and both configurations consistently outperform the vEB-tree baseline.

Transformation cost of SORT. We evaluate the time overhead of continuously transforming SORT as the number of vertices n increases from 10^5 to 10^7. Figure 12(d) summarizes the results, where n represents the exact point where the optimal configuration changes. As shown in the table, configuration updates occur approximately when n doubles, and each transformation completes within a few seconds. This confirms that SORT's reconfiguration overhead has negligible impact on overall system performance.

SORT space consumption as n increases. We further examine whether the space consumption of SORT scales linearly with the number of vertices n. This question is crucial, as radix trees generally lack a deterministic worst-case bound with respect to n. To investigate, we vary n within the range $[10^5, 10^7]$, sampling 100 evenly spaced points, and record the corresponding memory usage. Figure 12(c) demonstrates the results which show an approximately linear growth in memory consumption with n. This indicates that SORT's practical space complexity is close to $O(n)$. We also provide a theoretical analysis of SORT's worst-case space complexity, which confirms that its space remains near-linear in n when vertex IDs are not excessively sparse. Due to space constraints, the detailed analysis is included in Appendix C of our extended version [73].

n	Insertion				Query				Memory (KB)			
	$u = 2^{24}$		$u = 2^{32}$		$u = 2^{24}$		$u = 2^{32}$		$u = 2^{24}$		$u = 2^{32}$	
	SORT	ART	SORT	ART	SORT	ART	SORT	ART	SORT	ART	SORT	ART
10^4	**2.5E5**	1.7E4	6.0E4	**1.7E5**	**1.9E7**	1.2E6	**6.6E6**	1.2E6	2.6E3	**5.1E2**	3.1E3	**5.1E2**
10^5	**8.6E5**	1.7E5	**2.9E5**	1.8E5	**1.2E8**	2.4E6	**7.1E7**	2.5E6	4.5E4	**1.6E4**	6.7E4	**1.6E4**
10^6	**7.4E5**	1.5E5	**5.0E5**	1.7E5	**1.4E8**	3.5E6	**7.0E7**	3.2E6	**1.6E5**	2.7E5	6.4E5	**2.0E5**
10^7	**2.6E6**	1.4E5	**1.7E6**	1.1E5	**7.7E8**	3.7E6	**9.8E7**	4.2E6	**1.6E5**	2.7E6	**1.3E6**	2.9E6

Table 5. Insertion, query throughput and memory consumption of SORT and ART under different n and u.

Graphs	ART v.s. SORT			Slowdown w/o edge chain			
	Insert	Delete	Memory	2-hop	BFS	SSSP	BC
lj	8.58×	32.4×	1.86×	1.78×	27.67×	5.90×	1.56×
orkut	18.89×	28.33×	1.24×	1.72×	1.36×	1.14×	1.37×
dota	55.91×	23.74×	1.01×	1.04×	1.01×	1.36×	1.33×
g24	7.44×	5.50×	1.16×	1.05×	2.33×	1.00×	1.37×
u24	10.99×	30.72×	1.15×	2.10×	1.43×	1.19×	1.27×
twitter	12.81×	1.91×	1.30×	1.32×	1.73×	1.07×	1.12×

Table 6. Ablation study of RadixGraph. Left part is the relative time and memory costs of ART compared with SORT.

SORT performance under non-uniform workloads. We evaluate the memory efficiency of SORT and the vEB-tree under three types of workloads: (1) *Uniform*: IDs are uniformly distributed within $[0, 2^{32}-1]$, which matches our theoretical analysis; (2) *Skewed*: IDs are uniformly distributed within $[0, \frac{2^{32}-1}{1.5}]$, making the most significant bit more likely to be 0; (3) *Heavy-tailed*: IDs follow a reciprocal distribution [37] over $[0, 2^{32}-1]$ (i.e., $p(i) \propto \frac{1}{i}$), where smaller IDs occur much more frequently. For each workload, we generate 10^7 distinct IDs and insert them into SORT and vEB-tree, respectively. Figure 13 presents the memory footprints during insertions. SORT consistently outperforms the vEB-tree across all workloads, though the performance gains decrease as the distribution becomes more skewed.

Comparing SORT and ART. Table 5 presents the insertion, query throughput and memory consumption of SORT and ART under varying configurations, where n denotes the number of inserted elements and u represents the size of the ID universe. For ART, we adopt unodb[3] , an implementation that follows the designs proposed in prior work for both sequential and parallel settings [47, 48]. We evaluate the parallel version. Overall, SORT achieves higher throughput and better scalability than ART. In terms of memory efficiency, ART performs better when the ID universe is sparse (large $\frac{u}{n}$), whereas SORT is more efficient when the universe is dense (small $\frac{u}{n}$). This is because in sparse universes, even an optimized SORT configuration leaves many array slots unoccupied. These observations suggest that incorporating adaptive strategies into SORT could further improve its efficiency under sparse conditions.

4.7 Ablation study for RadixGraph

Swapping SORT with ART. To further compare SORT and ART, we integrate unodb (the ART implementation) as an alternative vertex index for RadixGraph. Then we evaluate and compare RadixGraph using ART and SORT to assess their relative performance. Table 6 presents the results and we summarize several insights:

- **ART is memory-efficient and competitive with SORT.** Across all datasets, the ART-based RadixGraph consumes slightly more memory than the SORT-based version. The four adpative

[3]https://github.com/laurynas-biveinis/unodb

	Batch	LJ				Orkut				Twitter			
		RadixGraph	Terrace	Aspen	CPAM	RadixGraph	Terrace	Aspen	CPAM	RadixGraph	Terrace	Aspen	CPAM
Insertion throughput	10	**6.58E5**	1.42E5	1.03E5	8.54E4	**4.89E5**	8.56E4	1.21E5	8.92E4	**1.09E5**	3.15E4	9.54E4	8.92E4
	10^2	**4.47E6**	2.79E5	6.37E5	4.17E5	**7.23E6**	2.18E5	9.97E5	4.48E5	**9.34E5**	2.80E5	**9.34E5**	4.86E5
	10^3	**3.36E7**	1.61E6	3.22E6	8.55E5	**3.38E7**	5.68E5	2.80E6	7.60E5	1.32E6	9.87E5	**3.45E6**	6.10E5
	10^4	**6.18E7**	2.74E6	6.10E6	2.61E6	**5.69E7**	2.67E6	6.57E6	1.99E6	**3.64E6**	2.45E6	*Seg fault*	6.60E5
Deletion throughput	10	**2.75E5**	2.29E5	9.57E4	8.32E4	**5.07E5**	2.55E5	1.27E5	9.10E4	**1.18E5**	1.09E5	1.02E5	8.85E4
	10^2	**5.04E6**	1.01E6	6.09E5	4.33E5	**8.08E6**	6.10E5	9.55E5	4.61E5	**2.22E6**	7.20E5	9.77E5	5.29E5
	10^3	**6.10E7**	1.48E6	3.43E6	8.52E5	**7.07E7**	9.18E5	2.86E6	7.79E5	**1.96E7**	1.91E6	3.15E6	6.19E5
	10^4	**2.01E8**	2.81E6	6.92E6	2.71E6	**5.76E7**	1.41E6	8.07E6	2.09E6	**9.65E7**	5.44E6	*Seg fault*	6.67E5
Memory	/	**0.77G**	1.51G	3.52G	2.57G	**2.93G**	4.30G	4.64G	6.98G	**26.25G**	47.06G	66.78G	69.60G

Table 7. Throughput for inserting and deleting edges with varying batch sizes in the LJ, Orkut and Twitter graphs and memory consumptions of different methods.

node types: *Node4*, *Node16*, *Node48*, and *Node256*, corresponding to fixed child array sizes of 4, 16, 48, and 256, allow ART to reduce wasted space. However, the coarse granularity can still lead to memory overhead. For instance, a node with 49 children must upgrade to a *Node256*, allocating space for 256 entries while storing only 49, thereby wasting memory. Although finer granularity (e.g., introducing more node types) could mitigate this issue, it would also increase transformation overhead due to more frequent node-type conversions. Nevertheless, on most datasets, the memory usage of the ART-based RadixGraph remains close to that of the SORT-based version, indicating that ART's space efficiency is still competitive overall.

- **Query costs of ART are higher than SORT, resulting in lower throughput.** The ART-based RadixGraph exhibits significantly higher insertion and deletion times than the SORT-based version across all datasets (Table 6), indicating that both insertion and query operations in ART incur overhead. During insertions, ART often needs to resize its internal arrays when a node changes type (e.g., from *Node4* to *Node16*), which involves memory reallocation and data copying. For queries, ART performs a sequential scan (for *Node4* and *Node16*) within each node to locate the corresponding child pointer. These factors collectively lead to slower update and query performance compared with SORT. However, the overall slowdown ratio of ART is much more than the "Insertion" part of Table 5, and closer to the "Query" part. The reason is that in graph systems, edge updates occur much more frequently than vertex insertions. Therefore, a query-efficient vertex index can significantly improve the overall throughput.

Graph analytics without edge chain. We evaluate the impact of disabling the edge chain structure in RadixGraph to assess its contribution to graph analytics performance. For 1-hop queries as well as PR, WCC, and LCC, the performance difference is negligible, as these tasks are largely insensitive to vertex-index lookup costs. For other analytical workloads, the observed slowdowns are summarized in Table 6. The results show that the edge chain generally improves performance.

4.8 Batch updates

Another line of edge storage research focuses on optimizing batch operations [23, 24, 61], whereas RadixGraph is primarily designed for single-edge operations. In this subsection, we compare RadixGraph with these systems in terms of end-to-end efficiency and memory consumption under varying batch sizes to highlight the advantages of RadixGraph's snapshot-log edge structure.

Table 7 summarizes the results. We observe that RadixGraph scales effectively as the batch size increases, even though RadixGraph is not explicitly optimized for batch processing. This scalability arises because RadixGraph's $O(1)$ edge operation cost is independent of vertex degree, and larger batch sizes naturally improve cache locality. On the Twitter dataset, Aspen exhibits competitive performance for batch sizes 10^2 and 10^3, but fails with a segmentation fault at batch size 10^4. In

contrast, RadixGraph outperforms other systems over most datasets and batch sizes. RadixGraph also demonstrates lowest memory consumption towards all datasets.

We note that single-edge and batch operations correspond to different real-world scenarios: single-edge operations are common in streaming or highly dynamic graphs, while batch operations are typical in bulk graph updates or offline analytics.

5 Related works

Indices for vertices and edges. Various in-memory graph systems have been developed for dynamic graph storage and processing [22–25, 27, 31, 43, 45, 51, 56, 61, 65, 69, 79]. Different indexing techniques have been explored for both vertex and edge management. Hash tables provide efficient lookups by mapping a large key space to a smaller one [16, 41, 62]. Coupled with hashmaps, the multi-level vector stores vertices based on their logical IDs from 0 to $n - 1$ [31, 78]. B+ trees support logarithmic time operations [6, 18, 63] and balanced binary search trees (BSTs) maintain elements in order and can be viewed as the binary equivalent of B-trees [7, 33, 36]. Radix trees offer near-constant time operations and eliminate the need for complex splitting and merging [30, 47, 59]. Recent works have further optimized radix-based structures for dynamic graphs. Teseo [22] employs an Adaptive Radix Tree (ART) [47] as its primary index, while Spruce [65] adopts the van Emde Boas tree (vEB-tree) [68] for vertex indexing.

Graph databases. Beyond in-memory dynamic graph systems, another approach focuses on developing graph databases that leverage external memory for storage. Neo4j [2] and OrientDB [3] use linked list-based storage, where each vertex maintains its adjacency list as a linked list. SQLG [57] adopts a relational model, storing vertices and edges as tables within a relational database. More recent efforts, such as Aster [58] and LSMGraph [76], integrate Log-Structured Merge (LSM) trees [74] with graph data structures to achieve high write throughput while maintaining efficient query performance. An industrial database, BG3 [77], adopt the Bw-tree [50] instead of LSM-tree to utilize cheap cloud storage and minimize costs. These databases are designed for persistent storage and can serve as backends for in-memory dynamic graph systems.

6 Conclusion

We presented RadixGraph, a fast and space-efficient data structure for in-memory dynamic graph systems. RadixGraph combines a space-optimized vertex index (SORT) with a compact edge structure that supports $O(1)$ dynamic edge operations while ensuring fast query performance. A potential direction for future work includes extending RadixGraph for transactional support, whose importance has been noted in GTX [78]. While RadixGraph already supports multi-version concurrency control (MVCC), how to provide stronger transactional guarantees on top of the existing design remains an open problem. In particular, supporting stronger isolation levels would require coordinating the visibility and ordering of updates that span multiple vertex-local edge arrays (for example, a transaction involving multiple operations). This requires shared commit timestamps, detecting write-write conflicts, and other mechanisms to ensure consistent and correct transaction execution.

7 Acknowledgement

This research is supported by the National Research Foundation, Singapore under its Frontier CRP Grant (NRF-F-CRP-2024-0005), and NTU SUG-NAP (022029-00001). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

References

[1] 2024. Intel® Threading Building Blocks (TBB). https://www.threadingbuildingblocks.org/

[2] 2024. Neo4j Graph Database Platform. https://neo4j.com/product/neo4j-graph-database

[3] 2024. OrientDB: Multi-Model Database. https://orientdb.org

[4] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 71–80. doi:10.1109/CCGrid54584.2022.00016

[5] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1, Article 1 (Feb. 2008), 39 pages. doi:10.1145/1322432.1322433

[6] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: approximate tree indexing. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1881–1892. doi:10.14778/2733085.2733094

[7] Rudolf Bayer. 1971. Binary B-trees for virtual memory. In *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (San Diego, California) *(SIGFIDET '71)*. Association for Computing Machinery, New York, NY, USA, 219–235. doi:10.1145/1734714.1734731

[8] Rudolf Bayer. 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (1972), 290–306. https://doi.org/10.1007/BF00289509

[9] R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. doi:10.1145/1734663.1734671

[10] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619

[11] Michael A. Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 26–es. doi:10.1145/1292609.1292616

[12] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., USA.

[13] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. 2011. Efficient in-memory indexing with generalized prefix trees. In *Datenbanksysteme für Business, Technologie und Web (BTW)*. Gesellschaft für Informatik e.V., Bonn, 227–246.

[14] P. Boldi and S. Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. doi:10.1145/988672.988752

[15] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Synthesis Lectures on Data Management, Vol. 1. Springer. doi:10.1007/978-3-031-01864-0

[16] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton tables: fast hash tables for in-memory data-intensive computing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 281–294.

[17] Pritish Chakraborty, Sayan Ranu, Krishna Sri Ipsit Mantri, and Abir De. 2023. Learning and Maximizing Influence in Social Networks Under Capacity Constraints. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining* (Singapore, Singapore) *(WSDM '23)*. Association for Computing Machinery, New York, NY, USA, 733–741. doi:10.1145/3539597.3570433

[18] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. doi:10.14778/2752939.2752947

[19] Dawei Cheng, Yao Zou, Sheng Xiang, and Changjun Jiang. 2025. Graph neural networks for financial fraud detection: a review. *Frontiers of Computer Science* 19, 9 (22 Jan 2025), 199609. doi:10.1007/s11704-024-40474-y

[20] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. doi:10.14778/2824032.2824077

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[22] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1053–1066. doi:10.14778/3447689.3447708

[23] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 108–121. doi:10.1145/3519939.3523733

[24] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 918–934.

doi:10.1145/3314221.3314598

[25] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. doi:10.1109/HPEC.2012.6408680

[26] William Feller. 1957. *An Introduction to Probability Theory and Its Applications*. Wiley. https://books.google.com.sg/books?id=BsSwAAAAIAAJ

[27] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. doi:10.1145/3448016.3457263

[28] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. 2021. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 184)*, Quentin Bramas, Rotem Oshman, and Paolo Romano (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:16. doi:10.4230/LIPIcs.OPODIS.2020.17

[29] Duncan T. Forster, Sheena C. Li, Yoko Yashiroda, Mami Yoshimura, Zhijian Li, Luis Alberto Vega Isuhuaylas, Kaori Itto-Nakama, Daisuke Yamanaka, Yoshikazu Ohya, Hiroyuki Osada, Bo Wang, Gary D. Bader, and Charles Boone. 2022. BIONIC: biological network integration using convolutions. *Nature Methods* 19, 10 (01 Oct 2022), 1250–1261. doi:10.1038/s41592-022-01616-x

[30] Edward Fredkin. 1960. Trie memory. *Commun. ACM* 3, 9 (Sept. 1960), 490–499. doi:10.1145/367390.367400

[31] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (Feb. 2022), 1173–1186. doi:10.14778/3514061.3514065

[32] Adel'son-Vel'skii G. M. and Landis E. M. 1963. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR* (1963), 263–266. Issue 2. http://mi.mathnet.ru/dan26964

[33] Kamaledin Ghiasi-Shirazi, Taraneh Ghandi, Ali Taghizadeh, and Ali Rahimi-Baigi. 2024. Revisiting 2–3 red–black trees with a pedagogically sound yet efficient deletion algorithm: parity-seeking. *Acta Inf.* 61, 3 (March 2024), 199–229. doi:10.1007/s00236-023-00452-6

[34] Yong Guo and Alexandru Iosup. 2012. The Game Trace Archive. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. 1–6. doi:10.1109/NetGames.2012.6404027

[35] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: online motif detection in large dynamic graphs. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1379–1380. doi:10.14778/2733004.2733010

[36] Bernhard Haeupler, Siddhartha Sen, and Robert E. Tarjan. 2015. Rank-Balanced Trees. *ACM Trans. Algorithms* 11, 4, Article 30 (June 2015), 26 pages. doi:10.1145/2689412

[37] R. W. Hamming. 1970. On the Distribution of Numbers. *The Bell System Technical Journal* 49, 8 (1970), 1609–1625.

[38] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[39] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. doi:10.14778/3007263.3007270

[40] Samira Khodabandehlou and Alireza Hashemi Golpayegani. 2024. FiFrauD: Unsupervised Financial Fraud Detection in Dynamic Graph Streams. *ACM Trans. Knowl. Discov. Data* 18, 5, Article 111 (Feb. 2024), 29 pages. doi:10.1145/3641857

[41] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. 2020. Cuckoo index: a lightweight secondary index structure. *Proc. VLDB Endow.* 13, 13 (Sept. 2020), 3559–3572. doi:10.14778/3424573.3424577

[42] Donald E. Knuth. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA.

[43] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[44] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. doi:10.1145/1772690.1772751

[45] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 31–46.

[46] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies* (Santa clara, CA, USA) *(FAST'17)*. USENIX Association, USA, 257–270.

[47] V. Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. *Proceedings - International Conference on Data Engineering*, 38–49. doi:10.1109/ICDE.2013.6544812

[48] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN'16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. doi:10.1145/2933349.2933352

[49] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[50] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313. doi:10.1109/ICDE.2013.6544834

[51] Hongfu Li, Qian Tao, Song Yu, Shufeng Gong, Yanfeng Zhang, Feng Yao, Wenyuan Yu, Ge Yu, and Jingren Zhou. 2025. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. *Proc. VLDB Endow.* 17, 13 (Feb. 2025), 4827–4839. doi:10.14778/3704965.3704986

[52] Jiuqiang Li and Hongjun Wang. 2024. Graph Diffusive Self-Supervised Learning for Social Recommendation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Washington DC, USA) *(SIGIR '24)*. Association for Computing Machinery, New York, NY, USA, 2442–2446. doi:10.1145/3626772.3657962

[53] Junhong Lin, Xiaojie Guo, Yada Zhu, Samuel Mitchell, Erik Altman, and Julian Shun. 2024. FraudGT: A Simple, Effective, and Efficient Graph Transformer for Financial Fraud Detection. In *Proceedings of the 5th ACM International Conference on AI in Finance* (Brooklyn, NY, USA) *(ICAIF '24)*. Association for Computing Machinery, New York, NY, USA, 292–300. doi:10.1145/3677052.3698648

[54] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael Lyu, and Yangfan Zhou. 2024. A Memory-Disaggregated Radix Tree. *ACM Trans. Storage* 20, 3, Article 15 (June 2024), 41 pages. doi:10.1145/3664289

[55] Anjun Ma, Xiaoying Wang, Jingxian Li, Cankun Wang, Tong Xiao, Yuntao Liu, Hao Cheng, Juexin Wang, Yang Li, Yuzhou Chang, Jinpu Li, Duolin Wang, Yuexu Jiang, Li Su, Gang Xin, Shaopeng Gu, Zihai Li, Bingqiang Liu, Dong Xu, and Qin Ma. 2023. Single-cell biological network inference using a heterogeneous graph transformer. *Nature Communications* 14, 1 (21 Feb 2023), 964. doi:10.1038/s41467-023-36559-0

[56] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374. doi:10.1109/ICDE.2015.7113298

[57] Pieter Martin. 2024. SQLG: A SQL-Based Graph Database. https://github.com/pietermartin/sqlg

[58] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proc. ACM Manag. Data* 3, 1, Article 12 (Feb. 2025), 26 pages. doi:10.1145/3709662

[59] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. doi:10.1145/321479.321481

[60] Giulia Muzio, Leslie O'Bray, and Karsten Borgwardt. 2020. Biological network analysis with deep learning. *Briefings in Bioinformatics* 22, 2 (Nov 2020), 1515–1530. doi:10.1093/bib/bbaa257

[61] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. doi:10.1145/3448016.3457313

[62] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 96–107. doi:10.14778/2850583.2850585

[63] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2011. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 286–297. doi:10.14778/2095686.2095688

[64] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 4292–4293.

[65] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1, Article 27 (March 2024), 26 pages. doi:10.1145/3639282

[66] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Rec.* 53, 2 (July 2024), 21–37. doi:10.1145/3685980.3685984

[67] Jixian Su, Chiyu Hao, Shixuan Sun, Hao Zhang, Sen Gao, Jiaxin Jiang, Yao Chen, Chenyi Zhang, Bingsheng He, and Minyi Guo. 2025. Revisiting the Design of In-Memory Dynamic Graph Storage. *Proc. ACM Manag. Data* 3, 1, Article 70

(Feb. 2025), 27 pages. doi:10.1145/3709720

[68] P. van Emde Boas. 1975. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (SFCS '75)*. IEEE Computer Society, USA, 75–84. doi:10.1109/SFCS.1975.26

[69] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2018.8547566

[70] Brian Wheatman and Helen Xu. 2021. A Parallel Packed Memory Array to Store Dynamic Graphs. In *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. 31–45. doi:10.1137/1.9781611976472.3

[71] Dan E. Willard. 1983. Log-logarithmic worst-case range queries are possible in space O(N). *Inform. Process. Lett.* 17, 2 (1983), 81–84. doi:10.1016/0020-0190(83)90075-3

[72] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proc. VLDB Endow.* 7, 9 (May 2014), 721–732. doi:10.14778/2732939.2732945

[73] Haoxuan Xie, Junfeng Liu, Siqiang Luo, and Kai Wang. 2026. RadixGraph: A Fast, Space-Optimized Data Structure for Dynamic Graph Storage (Extended Version). arXiv:2601.01444 [cs.DB] https://arxiv.org/abs/2601.01444

[74] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the design of LSM-tree Based OLTP storage engine with persistent memory. *Proc. VLDB Endow.* 14, 10 (June 2021), 1872–1885. doi:10.14778/3467861.3467875

[75] Carl Yang, Jieyu Zhang, Haonan Wang, Sha Li, Myungwan Kim, Matt Walker, Yiou Xiao, and Jiawei Han. 2020. Relation Learning on Social Networks with Multi-Modal Graph Edge Variational Autoencoders. In *Proceedings of the 13th International Conference on Web Search and Data Mining* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 699–707. doi:10.1145/3336191.3371829

[76] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6, Article 243 (Dec. 2024), 28 pages. doi:10.1145/3698818

[77] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 360–372. doi:10.1145/3626246.3653373

[78] Libin Zhou, Lu Xing, Yeasir Rayhan, and Walid G. Aref. 2025. GTX: A Write-Optimized Latch-free Graph Data System with Transactional Support. *Proc. ACM Manag. Data* 3, 3, Article 168 (June 2025), 27 pages. doi:10.1145/3725305

[79] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.* 13, 7 (March 2020), 1020–1034. doi:10.14778/3384345.3384351

Received July 2025; revised October 2025; accepted November 2025