Bingo: Radix-based Bias Factorization for Random Walk on Dynamic Graphs

Pinhuan Wang
Rutgers, The State University of New Jersey
Piscataway, NJ, USA
pw346@connect.rutgers.edu

Chengying Huan
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
huanchengying@nju.edu.cn

Zhibin Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
wzbwangzhibin@gmail.com

Chen Tian
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
tianchen@nju.edu.cn

Yuede Ji
The University of Texas at Arlington
Arlington, Texas, USA
yuede.ji@uta.edu

Hang Liu
Rutgers, The State University of New Jersey
Piscataway, NJ, USA
hang.liu@rutgers.edu

Abstract

Random walks are a primary means for extracting information from large-scale graphs. While most real-world graphs are inherently dynamic, state-of-the-art random walk engines failed to *efficiently* support such a critical use case. This paper takes the initiative to build a general random walk engine for dynamically changing graphs with two key principles: (i) This system should support both low-latency streaming updates and high-throughput batched updates. (ii) This system should achieve fast sampling speed while maintaining acceptable space consumption to support dynamic graph updates. Upholding both standards, we introduce Bingo, a GPU-based random walk engine for dynamically changing graphs. First, we propose a novel radix-based bias factorization algorithm to support constant time sampling complexity while supporting fast streaming updates. Second, we present a group-adaption design to reduce space consumption dramatically. Third, we incorporate GPU-aware designs to support high-throughput batched graph updates on massively parallel platforms. Together, Bingo outperforms existing efforts across various applications, settings, and datasets, achieving up to a 271.11x speedup compared to the state-of-the-art efforts.

CCS Concepts: • **Computing methodologies** → Parallel algorithms; • **Theory of computation** → **Dynamic graph algorithms**.

Keywords: Random Walk, Monte Carlo Sampling, GPUs.

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717456

ACM Reference Format:
Pinhuan Wang, Chengying Huan, Zhibin Wang, Chen Tian, Yuede Ji, and Hang Liu. 2025. Bingo: Radix-based Bias Factorization for Random Walk on Dynamic Graphs. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717456

1 Introduction

Random walks have drawn increasing attention in recent years since the representation learning of graphs (a.k.a, graph learning) entered the center stage of machine learning [18, 42, 70]. Below are two well-known scenarios: (i) In graph learning, a typical option is to use *random walks* to select a few subsets of vertices and edges from the original graph (each of the subsets is treated as a mini-batch) to train the graph neural networks [2, 9, 47, 50, 55]. This approach increases the scalability, generality, and efficiency of graph learning. However, random walks, unfortunately, take 96.2% of the end-to-end training time for graph learning, according to Gong et al's paper [15]. (ii) In friend recommendation of social media, one uses *random walks* to generate the node embeddings for the final recommendation. The random walk takes 3.5 hours (or 35% of the total time) on a graph snapshot of 227 million users and 2.71 billion edges [39]. In addition, in personalized PageRank [20], SimRank [25], and Random Walk Domination [32], we need to launch many *random walks* and use the visit frequency of each vertex across all these random walks as the major indicator to derive PageRank value, vertex similarity, and influence, respectively.

Perhaps the most practical use case of random walks would be extending their capabilities to real-world dynamic graphs. Using fraudulent detection of e-commerce platforms as an example, the transaction graph is changing constantly. The malicious users could commit a series of illicit activities if the graph updates are not immediately integrated

into the graph learning process [49]. Therefore, it is imperative to support random walks on dynamic graphs, which is already evident in Ant Finance [34]. Similar needs are also observed in weather forecast [35], Retrieval-Augmented Generation (RAG) of Large Language Models (LLMs) [30], friend/product recommendation [74], and human resource management [19] among many others [64, 71].

Unfortunately, there does not exist, to the best of our knowledge, a dedicated system that can *efficiently* support random walks on *dynamically changing* graphs. Below, we discuss related projects in four aspects: (i) Most existing CPU or GPU-based random walk and graph sampling systems only support random walk on static graphs (KnightKing [73], C-SAW [44], Graphwalker [62], and others [15, 24, 52]). Of note, TEA [22] focuses on a temporal graph, a special *static* graph with time-sensitive attributes. (ii) Existing dynamic graph analytical engines only support traditional graph algorithms, excluding random walks (see most recent LSGraph [48] and the prior ones [1, 23, 51, 65]). (iii) We do notice two recent projects that support updating random walk based on the graph updates [21, 45]. These two projects are orthogonal to Bingo. Particularly, they focus on managing a laundry list of random walks and expediting the process of finding the correct random walk to update. When updating the random walk, they simply rebuild the sampling space for updating, which is inefficient. (iv) We also want to clarify that - some recent efforts [39, 52] called higher-order Random Walk applications, such as Node2vec, as Dynamic Graph Random Walks. However, these random walk applications are performed on *static graphs* with biases that can change based on the random walk history. For more details on these related works, we refer the readers to Section 7.

This paper aims to design a system that could efficiently support various random walk applications on *dynamically changing graphs*. We identify two important design principles for building such a system:

First, this system should support both low-latency streaming updates and high-throughput batched updates. On the one hand, real-world graphs could experience important updates that should be incorporated immediately. Fraud detection, weather forecast, and RAG of LLM belong to this category. On the other hand, ingesting the system with a batch of graph updates is also commonplace. Certain graph systems, such as product recommendations, could require updating the graph daily with a large volume of updates. As we will demonstrate, batched updates offer more optimization opportunities for a higher ingestion rate.

Second, this system should achieve fast sampling speed while introducing acceptable memory consumption to support dynamic updates. It is anticipated that supporting dynamic graph updates could lead to more complex data structures that will consume more time for sampling and more memory for maintaining the transition probabilities. However, if the penalty on sampling speed and memory consumption is too high, one might simply adopt the sampling on static graphs approach by rebuilding the sampling space from scratch to cope with the dynamic updates. This principle will uphold the quality of this system.

We design and implement Bingo, a GPU-based random walk engine for dynamically changing graphs that adheres to the aforementioned two principles. Together, Bingo is up to 271.11× faster than the state-of-the-art that handles graph updates. For graph updates, the ingestion rate of Bingo can reach up to 226 million updates per second. Bingo features the following four major contributions:

- We introduce a novel radix-based bias factorization approach that decomposes the bias of each edge by their radix such that the updates are performed on each bias radix group, which is unbiased. This leads to constant time updating complexity. We also propose a hierarchical sampling algorithm for constant time sampling complexity (Section 4).
- Considering the memory consumption of our new data structure could be high, we present a group-adaption design to reduce the memory consumption significantly. Since our adaptation is designed according to the nature of various radix groups, this design preserves fast sampling time complexity (Section 5.1).
- When handling batched updates, Bingo processes all the requests in a massively parallel manner. Particularly, (i) we introduce a workflow to handle insertions and deletions and potential group rebuilding efficiently. (ii) Our novel delete-and-swap design enables massive parallel deletions (Section 5.2).
- We perform comprehensive evaluations of Bingo on various datasets, configurations, and random walk applications. Overall, Bingo constantly outperforms the state-of-the-art across all graph update settings with acceptable memory consumption. Further, our ingestion rate can reach 0.2 million (streaming updates) and 226 million (batched updates) updates per second across workloads of insertions, deletions, and mixed updates, respectively (Section 6).

2 Background

This section introduces the definition of dynamic graphs, along with popular random walk applications and the required background about sampling algorithms.

2.1 Dynamic Graph

As shown in Figure 1, different from static graphs, the states or number of vertices and edges may change over time in dynamic graphs, which is caused by a series of events like vertex update/insertion/deletion or edge update/insertion/deletion.

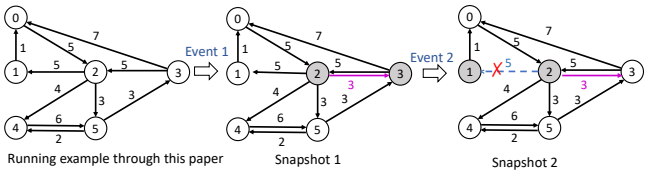


Figure 1. Running example. Event 1 contains one edge insertion and event 2 one edge deletion.

Formally, we define dynamic graphs using the graph snapshot model as follows [26]:

Definition 2.1. (Dynamic graph). A dynamic graph is a sequence of discrete graph snapshots, $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}$, where $t \in \mathbb{N}$ represents a timestamp, $\mathcal{V}^t = \{v_1^t, ..., v_n^t\}$ represents the vertices, and $\mathcal{E}^t = \{e_1^t, ..., e_m^t\}$ represents the edges.

Dynamic graphs otherwise behave similarly as static graphs, regardless of whether they have directed or undirected edges, or whether the edges are weighted.

2.2 Random Walk Applications

Random walks randomly extract paths from the original large graph. Random walks can be unbiased or biased. Here, we introduce two well-known random walk applications: DeepWalk and node2vec.

DeepWalk. DeepWalk [46] is a simple but popular random walk algorithm. It generates random walk paths through repeated sampling. In this process, the walkers stop when they reach the given path length. The paths are treated as sentences and used in the SkipGram model [41] to learn the latent representation. The original DeepWalk was unbiased and extended to a biased version later by Cochez et al [10].

Node2vec. Different from DeepWalk, node2vec [17, 75] is a more expressive higher-order algorithm, in which the transition probability also depends on the walk history. Supposing there is an undirected graph walker at vertex u, and the just visited vertex is w, the transition probability of an edge to the neighbor v will be multiplied with a factor $f(w, v)$ that depends upon the following conditions:

$$f(w,v) = \begin{cases} \frac{1}{p}, & \text{if } distance_{w,v} = 0, \\ 1, & \text{if } distance_{w,v} = 1, \\ \frac{1}{q}, & \text{if } distance_{w,v} = 2. \end{cases} \quad (1)$$

Here, p and q are user-defined hyper-parameters that influence the behavior of the walker. Specifically, smaller p increases the tendency for the walker to backtrack, while larger p decreases such a tendency. A lower q encourages the walker to explore far away, like Depth-First Search, while a larger q traps the walker around the group of vertices with strong connections. Finally, we normalize the transition probability to ensure the sum is 1.

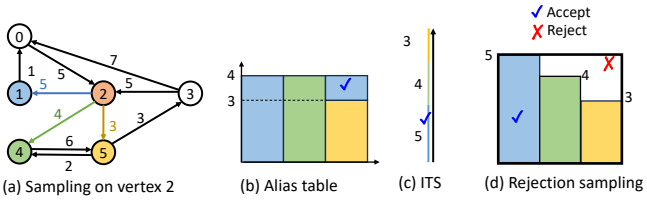


Figure 2. Three classical Monte Carlo sampling methods for sampling on vertex 2.

2.3 Monte Carlo Sampling Algorithms

Despite the large volume of random walk applications [71], they all share a common operation - a walker arrives at a vertex and selects among its neighbors for further exploration based on the transition probabilities. This process is called sampling. Sampling is called unbiased when all the candidates share identical probability; otherwise, it is biased sampling. Unbiased sampling is trivial, as we can easily pick the edge through random number generation. For biased sampling, the situation becomes more complex. Each candidate has a bias, which could be the weight or other information associated with the edge or vertex, depending on the specific application. Let w_i denote the bias of the i-th candidate. Without loss of generality, we consider biased sampling at vertex u, which has d neighbors denoted by $\{v_0, \ldots, v_{d-1}\}$. The transition probability of selecting neighbor v_i is:

$$P(v_i) = \frac{w_i}{\sum_{j=0}^{d-1} w_j}. \quad (2)$$

Figure 2 shows the three most common Monte Carlo sampling methods on a static graph. We describe them as follows:

Alias method splits all the d (d is the vertex degree) candidates into no more than $2d$ pieces and places them into d buckets, where two specifications are met: (i) Every bucket only contains up to 2 candidates. (ii) The volume of each bucket should be the same and equal to the average bias of the candidate set. The above structure is the so-called alias table. When sampling, we first select one bucket with equal probability, and then sample among up to 2 elements in that bucket. The total sampling time complexity is $O(1)$ and the time for alias table construction is $O(d)$. Figure 2(b) illustrates the alias table-based approach for vertex 2.

Inverse Transform Sampling (ITS) mainly samples based on an array C, which stores the information of Cumulative Distribution Function (CDF) for candidate transition probabilities. We arrange the candidates into a compact array and construct array C as the prefix sum of their biases, i.e. $c_i = \sum_{j=0}^{i} w_i$ with $c_0 = 0$. During the sampling process, we randomly generate a number x within the range $[0, c_d)$, then use binary search to determine the interval it belongs to. Specifically, if we find a value k such that $c_{k-1} \leq x < c_k$, then the edge labeled k is the result of sampling. Since we use binary search, the sampling time complexity is $O(\log d)$,

while the time complexity for constructing array C is $O(d)$. Figure 2(c) illustrates an example of the ITS method.

Rejection sampling is commonly used in cases with dynamic bias because it does not require maintaining a table or array like the alias method or ITS. It randomly selects one candidate i with equal probability then decides whether to accept or reject it. We randomly generate a number within the range $[0, max(w))$, where $max(w)$ indicates the maximum bias across all the candidates. If this number is less than w_i, we accept the candidate; otherwise, we reject it and repeat the above steps to select another candidate. The sampling time depends on the distribution of all the biases and the expected time complexity for rejection sampling is $O(\frac{d \cdot max(w)}{\sum_i w_i})$. In Figure 2(d), the uncolored area is the rejection area. If we select this area, we need to perform resampling.

3 Motivation and Bingo Overview

Motivation. None of the existing Monte Carlo sampling methods, to the best of our knowledge, is well-suited for sampling on dynamically changing graphs. First, the Alias method [60, 61, 73] requires $O(d)$ time complexity for accommodating one edge update, where d is the degree of the affected vertex, although it enjoys $O(1)$ sampling time complexity. Second, while rejection sampling presents constant time complexity for updating, its rejection rate could be high, see [24, 73]. Third, ITS sampling also enjoys fast updating time but suffers from non-trivial $O(\log d)$ sampling complexity, which is especially true for real-world graphs that reach billions of neighbors [15, 44].

Bingo overview. Figure 3 illustrates the major data structures, sampling, and graph updating workflow of Bingo. Particularly, the sampling space is partitioned into various groups. We further build a inter-group sampling space for Bingo to choose the group of interest for sampling.

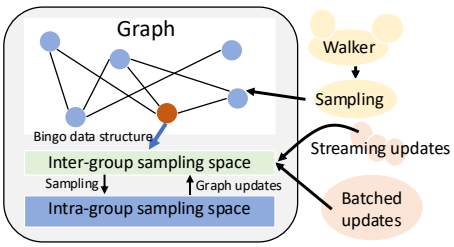


Figure 3. Bingo workflow. The right top is sampling while the right bottom is updating the graph.

Bingo features two functionalities: random walk query and graph update. (i) During random walk, when a walker reaches a vertex, it samples from inter-group to intra-group sampling spaces. First, it samples on the per-vertex inter-group sampling space to decide which group to further sample. Second, Bingo moves to that particular group for sampling. This process repeats until the end of the walk. (ii) The updating procedure follows the opposite direction. That is, it begins by deciding which groups will experience updates for a graph update. Subsequently, those groups will be updated. Finally, the per-vertex inter-group sampling space is updated. When batched updates are experienced, we will only update the inter-group space once.

4 Bingo: Radix-based Bias Decomposition for Random Walk

4.1 Bingo: Bias Decomposition Algorithm

High-level intuition: When handling vertices with many neighbors, the complexity of rebuilding alias tables or other auxiliary structures increases linearly with the number of neighbors, making traditional sampling techniques computationally expensive. The core idea behind Bingo is to apply a transformation that mitigates this growth by leveraging the binary representation of sampling biases w_i. Specifically, by decomposing biases into buckets corresponding to powers of 2, we effectively reduce the number of values involved in recomputing the alias table from the total number of neighbors to the logarithm of the maximum bias value. This transformation significantly reduces the complexity of incorporating the graph updates.

Sampling space construction: We follow a two-step approach for sampling space construction. (i) We perform a radix-based bias decomposition, decomposing each bias into sub-biases according to the bit positions. Formally, for bias w_i, we have a decomposition function D that decomposes w_i into a set of K sub-biases, where K is the number of bits. We term this set as $D(w_i)$. To put it more concretely, we have

$$D(w_i) = \{2^k | w_i \wedge 2^k \neq 0, k = 0, 1, \ldots, K-1\}, \quad (3)$$

where $\wedge$ represents bitwise AND in this paper. (ii) We reorganize all the $D(w_i)$'s according to the bit positions, k, by grouping the sub-biases with the same bit position into the same group. We define the reorganized bias group as $W(p_k)$, where p_k is the k-th bit position:

$$W(p_k) = \sum_{i=0}^{d-1} w_i \wedge 2^k. \quad (4)$$

Notice that if $w_i \wedge 2^k = 0$, w_i does not contribute to $W(p_k)$, which is equivalent to the fact that 2^k is not in $D(w_i)$.

Hierarchical sampling. Bingo performs a two-stage-based sampling on the aforementioned data structure as follows: During stage (i), termed inter-group sampling, we select a group of interest via Monte Carlo sampling. Of note, the bias of a group is the sum of biases in this group. Therefore, the transition probability for group p_k is:

$$P(p_k) = \frac{W(p_k)}{\sum_{j=0}^{K-1} W(p_j)}. \quad (5)$$

At stage (ii), we proceed to intra-group sampling. Since group p_k maintains all the neighbors whose bias bit at radix position k is 1, neighbors belonging to the same sub-bias group have equal transition probability.

Thus, we can perform unbiased sampling by randomly picking a vertex v_i from group p_k. The probability is:

$$P(v_i|p_k) = \frac{w_i \wedge 2^k}{W(p_k)}. \tag{6}$$

The corresponding original edge of this sub-bias is the final selected edge.

Theorem 4.1. *(Correctness). BINGO ensures that the probability of choosing each neighbor remains the same before and after radix-based bias factorization, i.e., Equation (2) holds for BINGO's sampling.*

Proof. Although we have transformed the sampling space, the probability of selecting each neighbor remains unchanged because the total bias associated with each neighbor is preserved. Formally, for a vertex v_i to be chosen, it can be chosen through all the groups v_i belongs to. Therefore, we arrive at:

$$P(v_i) = \sum_{k=0}^{K-1} (P(p_k) \cdot P(v_i|p_k)). \tag{7}$$

Replacing $P(p_k)$ by Equations 5 and $P(v_i|p_k)$ by Equation 6, we arrive at:

$$P(v_i) = \sum_{k=0}^{K-1} \frac{w_i \wedge 2^k}{\sum_{j=0}^{K-1} W(p_j)} = \frac{\sum_{k=0}^{K-1} w_i \wedge 2^k}{\sum_{i=0}^{d-1} w_i} = \frac{w_i}{\sum_{i=0}^{d-1} w_i}, \tag{8}$$

which is the same as when we perform sampling before radix-based bias factorization. □

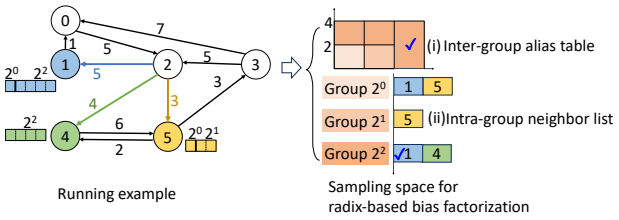


Figure 4. BINGO on the running example.

BINGO running example. Figure 4 exemplifies how to implement BINGO for the running example. Using vertex 2 as an example, following the $(src, dst, bias)$ format, there are three candidate edges from vertex 2, i.e., $(2, 1, 5)$, $(2, 4, 4)$, and $(2, 5, 3)$. Applying binary factorization to the biases of all three neighbors, we get group 2^0 contains $\{1, 5\}$, group 2^1 contains $\{5\}$, and group 2^2 contains $\{1, 4\}$. Here, we directly use the out-neighbor ID to represent the edge. Therefore, the biases of these three groups are 2, 2, and 8. We adopt the alias table method to build the (i) inter-group sampling space, which is shown in the top right of Figure 4.

During sampling, step (i) selects group 2^2 in the inter-group alias table. Subsequently, step (ii) relies on unbiased sampling to select neighbor 1 in group 2^2.

Remarks on BINGO sampling benefits. BINGO enjoys fast sampling speed. Particularly, in stage (i), we adopt the alias method in stage(i), which offers constant time complexity. Further, because the number of groups is small, updating the alias table will be fast. For stage (ii), while the sampling space is large, it is uniform sampling. Therefore, the sampling speed is, again, fast (i.e., constant time complexity).

4.2 BINGO for Streaming Updates

Now, we introduce our designs to achieve fast graph updates (i.e., insertion and deletion) in BINGO.

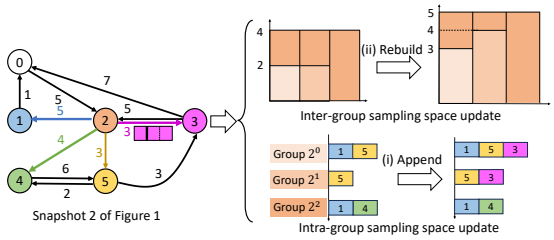


Figure 5. BINGO insertion operation.

Insertion: Figure 5 exemplifies the insertion operation of BINGO. We first split the newly inserted edge following Equation 3 and reorganize these sub-biases following Equation 4. In this example, the new edge (2, 3, 3)'s bias is decomposed into $3 = 2^0 + 2^1$. We thus split this edge into two sub-biases, one belonging to group 2^0, and the other to 2^1.

During insertion, we perform the intra-group updates before the inter-group one: First, one can simply (i) append the new edge to the end of each neighbor list array. Second, the inter-group alias table is updated based on the new biases in all groups, which is shown as (ii) rebuild in Figure 5.

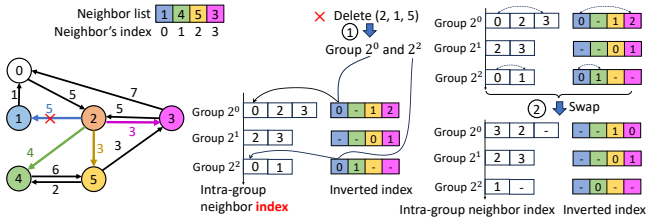


Figure 6. BINGO deletion operation.

Deletion: Figure 6 illustrates the edge deletion operation of BINGO. This process contains four steps: (i) we perform a radix-based bias decomposition to identify which groups this edge has contributed sub-biases to. (ii) One needs to locate this edge in those identified groups. (iii) We swap this edge with the edge in the tail of that group to maintain a compact intra-group neighbor list for $O(1)$ unbiased sampling. (iv) We

rebuild the inter-group sampling space similar to insertion in Figure 5 thus omitted for brevity.

We introduce two design changes to achieve a near-constant time complexity for deletion. First, we store the neighbor index in each group as opposed to the neighbor ID, which was the case in Figures 4 and 5, because the neighbor index can locate the edge in the neighbor list in O(1) complexity. This will help the swap operation in step (iii). Second, we introduce an inverted index that tracks where each neighbor index was stored in each group. Formally, this inverted index maintains a mapping from the neighbor index to the position in each group. This reduces the complexity of locating edges in step (ii) from $O(d)$ to $O(1)$.

Figure 6 exemplifies our deletion operation. Assuming we want to delete edge (2,1,5). In step (i), we identify this edge contributes to groups 2^0 and 2^2. (ii) Because the edge (2,1,5)'s index is 0, we obtain the location of this edge in these two groups at the first entry of the corresponding inverted index, both of which are 0 in the inverted index. (iii) We swap them with the tail of that group. Using group 2^0 as an example, the neighbor index 0 will swap with neighbor index 3. Because we store the neighbor index in the intra-group neighbor index, we use this index to locate that we should update the pink location to 0. In a nutshell, one can use the content in the intra-group neighbor index to locate where we should change in the inverted index and vice versa.

In addition to insert/delete edges, other graph updates, e.g., deleting a vertex, and updating the edge bias, can be either implemented with insertion and/or deletion operations or supported straightforwardly. We omitted their descriptions for the sake of space constraints.

4.3 Bingo for Floating-Point Biases

While one might suggest that radix sort [40] is the closest related work to Bingo when handling floating-point biases, Bingo, faces more stringent requirements. That is, radix sort only cares about whether a bias is bigger or smaller than the other bias to sort out the order while Bingo requires to know one bias is exactly how much bigger or smaller than the other to build the sampling space. For more details about radix sort of floating-point values, we refer the readers to [56]. In that regard, Bingo cannot adopt radix sort ideas to handle floating-point biases in sampling.

Bingo's approach. Bingo handles floating-point biases in a four-step approach. (i) We empirically determine an amortization factor λ, which is used to round floating-point values to proportional integer values. (ii) For each bias, we decompose this bias into an integer part and a decimal part. (iii) We further perform radix decomposition for the integer part of all the biases (similar to our integer alone case) while leaving the decimal part in one group. If the decimal part is taken, we will adopt ITS or rejection sampling. (iv) We perform the hierarchical sampling to choose the edge of interest. Of note, λ is properly chosen such that the sum of the decimal parts is very low. Then the sampling time complexity remains low. See Section 4.4 for detailed discussion.

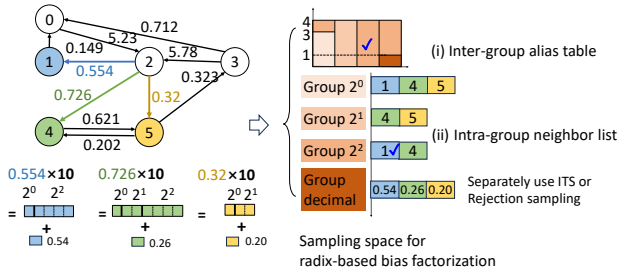


Figure 7. Bingo with floating-point biases.

Figure 7 illustrates how Bingo works on the floating-point biases. Still using vertex 2 as an example, there are three candidate edges from vertex 2, i.e., $(2, 1, 0.554)$, $(2, 4, 0.726)$, and $(2, 5, 0.320)$. We multiply all the biases by 10. Then we get edges with new biases, $(2, 1, 5.54)$, $(2, 4, 7.26)$, and $(2, 5, 3.20)$. Applying binary factorization to the integer part of biases of all three edges, we get groups 2^0, 2^1 and 2^2, respectively, containing neighbors $\{1, 4, 5\}$, $\{4, 5\}$, and $\{1, 4\}$. Besides, we collect the decimal part as a new group, i.e., $\{1, 4, 5\}$. We adopt the alias table method to build the (i) inter-group sampling space, which is shown in the top right of Figure 7. Since the decimal group only takes up a small area of the alias table, for most of the cases, our sampling happens in the integer part, whose sampling complexity is constant.

4.4 Complexity Analysis

Table 1 presents the complexity of Bingo vs four common Monte Carlo sampling methods. The superiority of Bingo's time complexity can be clearly observed from this table. However, Bingo consumes more memory than others. We will introduce a series of optimizations for memory consumption in the following section.

Time complexity. For a vertex with a degree of d, we assume each neighbor has the bias of w_i. (i) Sampling: Given that both inter- and intra-group sampling (alias table sampling and unbiased sampling) have a time complexity of $O(1)$, the total sampling time remains $O(1)$. (ii) Insertion: For each of the K groups, we preform $O(1)$ insertions. The size of K could be derived by $K = log(max(w_i))$. Additionally, we need to reconstruct the alias tables for each of these groups. Therefore, the overall time complexity of insertion is $O(K)$. (iii) Deletion: as deletion operations within each group are identical, each single deletion step in each group only consumes $O(1)$ constant time, and the time complexity of deletion is also $O(K)$.

For floating-point biases, we only need to analyze the sampling of the intra-group part (inter-group sampling complexity is O(1)). Let W_I and W_D denote the sums of the biases

Table 1. Complexity comparison for a vertex: BINGO vs. Alias, ITS, and Rejection sampling, where K and d are the numbers of groups and degree of the vertex.

Name	Insertion	Deletion	Sampling	Memory
BINGO	$O(K)$	$O(K)$	O(1)	$O(d \cdot K)$
Alias Method	$O(d)$	$O(d)$	$O(1)$	$O(d)$
ITS	$O(1)$	$O(d)$	$O(log_2 d)$	$O(d)$
Rejection Sampling	$O(1)$	$O(d)$	$O(\frac{D \cdot max(w)}{\sum_i w_i})$	$O(d)$

for, respectively, integer and decimal parts. The sampling time complexity becomes $O(\frac{W_D}{W_I+W_D} \cdot \frac{d \cdot max(w_i)}{\sum_i w_i} + \frac{W_I}{W_I+W_D})$, where use rejection sampling as an example. In this expression, the first term means inter-group sampling selects the decimal group, while the second term means the integer group. Through adjusting λ, we can ensure that $\frac{W_D}{W_I+W_D} < \frac{1}{d}$, which keeps our sampling complexity as $O(1)$. In Figure 7, we set $\lambda = 10$ which leads $\frac{W_D}{W_I+W_D} = \frac{1}{16} < \frac{1}{d} = \frac{1}{3}$. This λ ensures $O(1)$ sampling complexity.

The updating time complexity of BINGO is only related to the number of groups, i.e., K, which is small (usually no more than 32 or 64), which is the number of bits for integer and long integer.A larger radix base can further reduce K, which is briefly discussed in Section 9.2 (see supplement material).

Space complexity. Our naive design repeats the inverted index K times, i.e., the number of groups, with the size of each inverted index as d. This leads to a memory consumption of $d \cdot K$. Further, each edge appears in $t = popc(w_i)$ groups, i.e., the number of nonzero bits in bias w_i. This leads to a memory consumption of $d \cdot t$ for the intra-group neighbor index list. Combined, our method amplifies the memory consumption from d to $d \cdot (K + t)$. Since each vertex might not appear in all groups, we derive $t \leq K$. Therefore, the space complexity of BINGO is $O(d \cdot K)$.

5 BINGO System Implementation and Optimizations

This section implements BINGO on GPUs with two optimizations, i.e., memory consumption optimization (Section 5.1) and batched graph updates optimization (Section 5.2).

5.1 Adaptive Group Representation

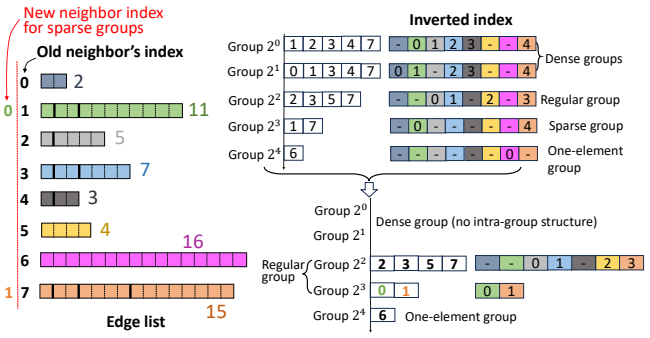


Figure 8. BINGO adaptive group representation.

Figure 8 presents our solutions to reduce both $K + t$ and d (see Section 4.4), which reduces the memory consumption for BINGO. Our intuition is as follows: we first separate dense and sparse groups from the regular group, which permits simpler data structures (i.e., less memory consumption). Additionally, we observe that one-element groups constitute a significant proportion of the data (see Figure 11), and handling them separately enables further memory savings through simplified representation. Based on the cardinality of different groups G_i, we divide them into four categories: dense, one-element, sparse, and regular groups, following Equation 9:

$$\text{Group } G_i \in \begin{cases} \text{Dense group}, & \frac{|G_i|}{d} > \alpha\%; \\ \text{One-element group}, & |G_i| = 1; \\ \text{Sparse group}, & \frac{|G_i|}{d} < \beta\% \text{ and } |G_i| \neq 1; \\ \text{Regular group}, & \text{Otherwise.} \end{cases} \tag{9}$$

Based on our heuristic study, we set $\alpha = 40$ and $\beta = 10$ in our design for the optimal performance.

Dense group is often the groups with less significance. Using group 2^0, i.e., the group with the least significant bit as an example, a neighbor v_i falls into this group when $w_i \bmod 2 = 1$, or simply with an odd bias. Statistically, half of the neighbors (50%) has a chance of falling into the 2^0 group.

We propose a radical change for dense groups, i.e., we maintain neither the intra-group neighbor index list nor the inverted index for a dense group. This helps reduce the memory cost for both t and K (mentioned above). Besides, these groups are not accessed frequently because they often appear in the groups with less significant bits. Therefore, we save the memory for the largest intra-group neighbor index lists and the inverted indices with potentially small or no sampling efficiency impacts.

We adopt a rejection sampling on the original neighbor list for dense groups. Even if dense groups are selected, we can still maintain a low rejection ratio because the rejection ratio is below (1-α%)=60%. Particularly, our rejection sampling works as follows: (i) Our inter-group sampling selects a particular dense group. (ii) For intra-group sampling, we select a neighbor from the original neighbor list. Further, we use that neighbor's bias to AND (i.e., &) the group radix of the chosen dense group. If the result is not zero, this sampling is accepted. Otherwise, we repeat the sampling from the intra-group sampling.

In Fig 8, group 2^0 and group 2^1 are dense groups, since they contained 62.5% of edges. If inter-group sampling selects group 2^0, we will further randomly select one neighbor from the original neighbor list. Assuming we selected the 5-th neighbor, we use its bias 4 to AND 2^0, which returns 0. This means our sampling is rejected. We will repeat this process until we find a neighbor in group 2^0.

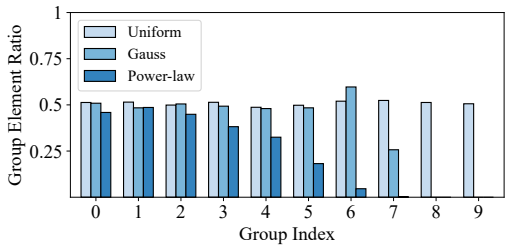


Figure 9. Group element ratio of various distributions.

One-element group. With skewed bias distributions, chances are group(s) with the most significant values might only contain one element. For those groups, there is no need to maintain either the inverted index or the intra-group neighbor index list. Group 2^4 in Figure 8 belongs to this case.

Sparse group. Our sparse group optimizations mainly focus on reducing the memory consumption of the inverted index. As shown in Figure 9, for two out of three cases, groups with higher powers tend to have fewer edges, which means the inverted indexes in these groups can be very sparse. Consequently, we maintain a new neighbor list that only contains edges with larger biases. This leads to a much smaller neighbor list, thus a smaller inverted index for sparse groups.

As shown in Figure 8, neighbor indices 1, 6, and 7 are the edges with bias larger than 8. Since neighbor 7 also belongs to the one-element group, we exclude it from the sparse group. Therefore, our new neighbor list contains {1, 7}, whose cardinality is 2. In this case, our inverted index has a size of 2 for group 2^3, which is significantly smaller than the original inverted index with a size of 8.

Regular group. After filtering out dense, one-element, and sparse groups, we might still have a few groups. These groups require the full inverted index and the original intra-group neighbor index list. In Figure 8, group 2^2 is such a group.

5.2 BINGO's Parallel Batched Graph Updates

Figure 10(a) presents the overall workflow of our parallel batched graph updates. On the CPU platform, we first put the graph updates of the same vertex together. Subsequently, we move these ordered update requests to GPU. For each vertex v_i, we, in order, perform three steps, i.e., insert, delete, and rebuild. Of note, one might insert a just deleted edge back; we thus allow duplicated insertions of the same edge with a time stamp. When deletion happens to a duplicated edge, we delete the earlier version first.

During insertion, we first insert each edge into the graph following the dynamic graph format [4]. Subsequently, we perform four types of insertions differently. For dense group, we simply do nothing because it does not maintain any sampling-related data structures. One-element group: We derive whether this group evolves into a sparse/regular/dense group based on all the insertions. Subsequently, it follows how an insertion is done to a sparse/regular/dense group to perform this particular insertion. Sparse group: We first

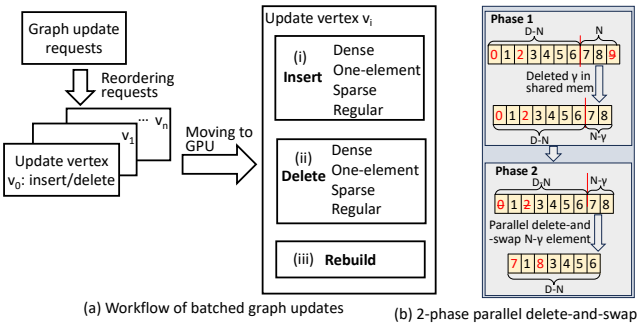


Figure 10. Parallel batched graph updates.

append this new neighbor to the sparse group neighbor list. Second, we append to the intra-group neighbor index list and the inverted index of this group for this specific neighbor. Regular group: we append the edge to the intra-group neighbor index list and update the inverted index.

Deletion is more complex than insertion. On-element group: We simply remove this group. Dense group: we will perform a rebuild if this dense group evolves into a different group type (details discussed later). Sparse/Regular group: Since parallel delete-and-swap could introduce bubbles in the intra-group neighbor index list that prevent unbiased sampling, we introduce a design to support parallel delete-and-swap as below:

Figure 10(b) exhibits our two-phase parallel delete-and-swap N elements operation, which is a key contribution to our parallel deletion. The biggest concern of the parallel delete-and-swap is that the elements from the tail might also be deleted. Therefore, if one uses the concurrently deleted tail element to fill in the entry that is to be deleted, one fills the to-be-deleted entry with a voided entry. For instance, one wants to delete entry 0 and use entry 9 to fill in entry 0. However, what if entry 9 will also be deleted? In this case, we cannot use entry 9 to fill entry 0.

In our design, we perform this deletion in two phases. (i) We load N elements from the tail to GPU's shared memory, if fit. Otherwise, we keep them in global memory. Subsequently, we will delete all the elements that are supposed to be deleted in these N elements. We assume we have deleted γ elements. (ii) Since we have already deleted γ elements from the list, we only need to delete $N - \gamma$ more elements from the list. In phase (i), we are sure the remaining $N - \gamma$ elements will not be deleted. Therefore, we are guaranteed to delete $N - \gamma$ elements at the front with these $N - \gamma$ will-not-deleted elements to fill the entry.

Our rebuild mainly tackles group-type transformations of the following two cases: (i) from other group formats to a regular group, and (ii) From other group formats to a sparse group format. The reason for case (i) is one needs to rebuild the entire intra-group neighbor index list and inverted index, which is expensive during insertion or deletion. For case (ii), one needs to scan the original group to rebuild the new

neighbor index, which is again better performed after all the insertions and deletions are accomplished.

6 Evaluation

Bingo is implemented using CUDA/C++ with approximately 2,000 lines of code. It is designed with an efficient GPU-based architecture to handle graph processing and random walks. Below are the implementation details: (i) Bingo stores the graph and related metadata directly on the GPU, ensuring fast access and efficient computations. (ii) Before each random walk computation, Bingo integrates all the graph updates, maintaining the correct order of operations to ensure consistency. This correctness is enforced for both streaming and batched random walks. (iii) Bingo performs random walks in a step-by-step manner, where each step involves sampling to select the next node for the next step of random walks. (iv) Bingo treats each vertex as an individual object and utilizes eight main kernels to support various random walk applications, i.e., streaming_insert, streaming_delete, batched_insert, batched_delete, random_walk_node2vec, random_walk_deepwalk, random_walk_ppr, and random_walk _simple_sampling.

6.1 Experimental Setup

The evaluation is performed on a Linux server equipped with two 2.8GHz Intel(R) Xeon(R) Silver 4309Y CPUs, 16 physical cores, 512 GB of memory, and four NVIDIA A100 GPUs, each with 80 GB of HBM2e memory.

Graph Applications. We examine Bingo and the state-of-the-art with three applications: biased DeepWalk, node2vec, Personalized PageRank (PPR). For all of them, we initialize the vertex count number of random walkers. Further, for DeepWalk and node2vec, we set the default walk length as 80. We set the hyper-parameters $p = 0.5$ and $q = 2$ for node2vec. The parameters for DeepWalk and node2vec are identical to that from KnightKing [73]. We put the termination probability of PPR as 1/80, which offers an expected walk length of 80.

Table 2. Graph dataset ($K = 10^3$, $M = 10^6$).

Dataset	Abbr.	Vertex count	Edge count	Avg degree	Max degree
Amazon	AM	403.4K	3.4M	8.4	10
Google	GO	875.7K	5.1M	5.8	456
Citation	CT	3.8M	16.5M	4.4	770
LiveJournal	LJ	4.8M	68.5M	14.3	20.3K
Twitter	TW	41.7M	1,468.4M	35.2	770.2K

Datasets. TABLE 2 presents the five real-world graph datasets, retrieved from Konect [28] and SNAP [29], which we use for evaluation. We follow a three-step design to create dynamic updates: (i) we split the original graph dataset into two sets: A (original edges - $10 \cdot BATCHSIZE$ edges) and B ($10 \cdot BATCHSIZE$ edges) randomly. (ii) We randomly determine whether we want to delete or insert an edge. (iii) If we want to delete an edge, we will delete a randomly selected edge from set A. Otherwise, we randomly choose an edge from the set B and add that to set A. We perform this $10 \cdot BATCHSIZE$ times to generate a sequence of $10 \cdot BATCHSIZE$ updates. In this paper, we set $BATCHSIZE = 100k$. We use the edges in set A as the edges to initialize the test.

Dynamic updates. We generate three types of updating situations for each dataset in TABLE 2: "Insertion", "Deletion", and "Mixed", which contain insertion only, deletion only, and mixed graph updates with an equal number of insertions and deletions, respectively. Except for Section 6.2, which uses all these three update situations, other sections only use the "Mixed" update situation (if not stated otherwise).

Bias. We generate the bias for most of the tests based on the degree of vertices, which naturally follow power law distribution (given all datasets are real-world graphs [5]). We also evaluate Bingo under floating-point bias and bias with different distributions in Section 6.4.

Evaluation Workflow. We perform (i) $BATCHSIZE$ number of updates and (ii) graph application computation. Since we generate 10 graph updates of $BATCHSIZE$, we repeat the aforementioned steps (i) and (ii) 10 rounds. We report the total time of these 10 rounds. By default, we conduct tests on GPU-based systems using a single GPU and on CPU-based systems using all the 16 physical cores of a single machine.

6.2 Bingo vs. the State-Of-The-Art (SOTA)

Choice of SOTA. We compare Bingo with three representative projects: KnightKing [73], gSampler [15] and FlowWalker [39]. KnightKing is the first general-purpose CPU-based distributed graph random walk engine that employs the alias method for static biased sampling and the rejection method for dynamic biased sampling. Because Flashmob [72], a more recent work, does not support biased sampling, we choose KnightKing as the CPU-based SOTA. gSampler is one of the most recent GPU-based graph sampling systems, featuring matrix-based APIs for efficient execution. In addition, FlowWalker, another recent GPU-based sampling framework, is based on parallel reservoir sampling and serves as a strong baseline for comparison. Since the above systems only support static or streaming graphs, we develop their open-source code for our evaluation. Specifically, we reload or reconstruct the corresponding structure after each round of updates. Notice that we do not conduct comparative experiments with the dynamic graph processing systems, e.g., Wharf [45] or Hornet [4] (see Section 7.2), as Bingo focuses on sampling, which is different from these projects. We will analyze the difference between Bingo and these projects in Section 7.

TABLE 3 provides a comprehensive analysis of Bingo vs the SOTA on various datasets and applications. For ease of viewing, we color-coded the shortest runtime and largest

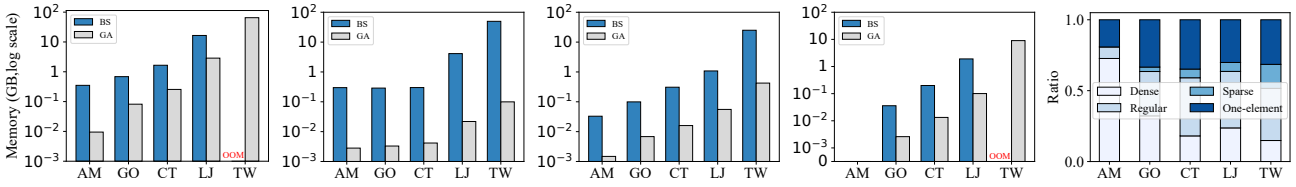
Table 3. BINGO vs. SOTA on time and memory consumption.

Algorithms	Frameworks	Runtime (s), Memory Consumption (GB)					Avg. speedup
		Amazon	Google	Citation	LiveJournal	Twitter	
DeepWalk Insertion	BINGO	0.51, 0.70	1.01, 1.54	0.62, 2.09	1.99, 6.34	19.63, 51.22	-
	KnightKing	2.64, 0.17	4.39, 0.17	12.63, 0.80	70.92, 2.54	1114.26, 46.14	24.46
	gSampler	1.55, 1.25	2.44, 1.99	3.81, 7.95	22.97, 8.80	403.91, 71.24	8.74
	FlowWalker	0.55, 0.70	1.85, 1.02	4.26, 2.99	11.01, 4.04	25125.48, 43.97	259.05
DeepWalk Deletion	BINGO	0.25, 0.70	0.32, 1.44	0.16, 2.09	1.46, 6.24	19.11, 51.12	-
	KnightKing	2.48, 0.17	4.43, 0.17	12.41, 0.78	71.45, 2.54	1136.11, 46.14	41.94
	gSampler	1.47, 1.01	2.33, 1.51	3.44, 5.54	21.46, 5.92	376.54, 60.61	13.81
	FlowWalker	0.56, 0.70	1.84, 1.02	4.20, 2.99	10.44, 4.04	25112.71, 43.97	271.11
DeepWalk Mixed	BINGO	0.25, 0.87	0.64, 1.38	0.44, 4.62	1.96, 7.25	18.94, 51.22	-
	KnightKing	2.59, 0.17	4.52, 0.22	12.40, 0.78	67.43, 2.53	1006.66, 46.21	26.63
	gSampler	3.30, 1.25	2.64, 2.14	4.39, 7.72	17.60, 10.08	307.16, 73.19	10.50
	FlowWalker	0.57, 0.70	1.94, 1.02	4.33, 2.99	10.72, 4.04	25137.66, 43.97	269.57
node2vec Insertion	BINGO	0.63, 0.79	1.18, 1.64	0.66, 2.29	6.35, 7.03	66.75, 66.55	-
	KnightKing	13.75, 0.33	15.47, 0.44	58.23, 1.00	202.23, 2.86	2526.14, 49.11	38.57
	gSampler	4.32, 1.29	4.61, 2.05	12.12, 8.24	48.09, 10.79	695.43, 60.01	9.42
	FlowWalker	0.64, 0.70	2.92, 1.02	4.60, 2.99	18.57, 4.04	60108.03, 43.97	182.78
node2vec Deletion	BINGO	0.35, 0.79	0.57, 1.54	0.18, 2.29	5.92, 6.93	66.51, 66.55	-
	KnightKing	13.66, 0.33	12.94, 0.44	55.50, 1.00	176.38, 2.86	2550.16, 49.11	87.64
	gSampler	3.81, 1.06	4.14, 1.59	10.88, 5.85	43.42, 7.39	672.04, 60.01	19.21
	FlowWalker	0.66, 0.71	2.94, 1.03	4.60, 2.99	18.62, 4.04	60007.51, 43.97	187.60
node2vec Mixed	BINGO	0.27, 0.97	0.81, 1.58	0.46, 5.40	6.14, 8.13	66.64, 66.55	-
	KnightKing	18.34, 0.33	29.99, 0.43	36.49, 1.00	109.78, 2.86	2510.34, 49.22	47.97
	gSampler	6.01, 1.29	4.36, 2.19	13.87, 8.02	44.03, 12.07	671.59, 72.88	15.01
	FlowWalker	0.66, 0.70	2.99, 1.02	4,64, 2.99	18.70, 4.04	59840.22, 43.97	183.45
PPR Insertion	BINGO	0.56, 0.70	1.10, 1.54	0.62, 2.09	2.01, 6.34	20.21, 51.22	-
	KnightKing	2.67, 0.29	4.52, 0.37	11.86, 0.78	74.08, 2.54	1172.27, 46.14	24.57
	gSampler	1.67, 1.43	2.42, 2.35	3.86, 9.74	28.36, 10.60	519.07, 52.76	10.24
	FlowWalker	0.57, 0.70	1.93, 1.02	3.80, 2.99	17.25, 4.04	24230.61, 43.97	243.28
PPR Deletion	BINGO	0.27, 0.70	0.34, 1.44	0.16, 2.09	1.47, 6.24	19.69, 51.12	-
	KnightKing	2.56, 0.29	4.13, 0.36	11.32, 0.78	69.85, 2.54	1122.42, 46.14	39.38
	gSampler	1.33, 1.13	2.39, 1.75	3.56, 6.74	21.97, 7.12	476.82, 72.54	14.67
	FlowWalker	0.53, 0.70	1.94, 1.02	3.78, 2.99	17.04, 4.04	24181.06, 43.97	254.19
PPR Mixed	BINGO	0.26, 0.87	0.66, 1.38	0.44, 4.62	1.74, 7.35	19.96, 51.22	-
	KnightKing	2.58, 0.29	4.25, 0.37	11.08, 0.77	64.76, 2.53	989.14, 46.21	25.66
	gSampler	3.39, 1.44	2.58, 2.53	4.52, 9.45	19.73, 12.19	559.58, 53.97	13.32
	FlowWalker	0.55, 0.70	1.97, 1.02	3.86, 2.99	17.23, 4.04	24243.25, 43.97	247.67

memory consumption. Briefly, BINGO consistently outperforms KnightKing, gSampler, and FlowWalker by 24.46 – 112.28×, 8.74 – 25.66× and 182.78 – 271.11×, respectively.

When it comes to runtime, we observe four different insights: (i) BINGO outperforms all SOTA across all graph applications on all graph datasets and updating cases. (ii) BINGO enjoys more speedup on larger graphs with bigger degrees. Using DeepWalk-Insertion as an example, our speedup climbs from 5.17× (Amazon) to 1279.9× (Twitter). (iii) BINGO offers higher speedup on deletion than insertion, with mixed operations sitting in the middle. The reason lies in the fact that dynamic arrays may need to allocate memory immediately during insertion. In contrast, memory released during deletion can be managed offline without incurring immediate overhead in our custom memory pool. In short, deletion is more friendly to BINGO than insertion.

Regarding memory consumption, we witness two key insights: (i) gSampler often consumes the most memory, followed by BINGO, then KnightKing and FlowWalker. The reason is that gSampler relies on matrix APIs, which will factor out a laundry list of memory costs. BINGO would consume more memory than KnightKing and FlowWalker because



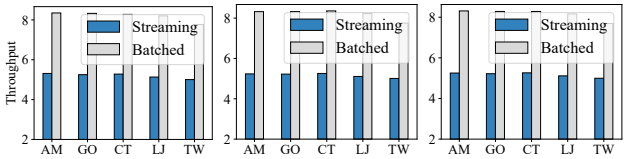
(a) Overall. **(b)** Dense group. **(c)** One-element group. **(d)** Sparse group. **(e)** Ratio of various groups.

Figure 11. Adaptive group representation impacts on memory consumption of Bingo, where BS, GA, respectively, represent "BaSeline" and "Group Adaption optimization". In BS, we use regular group format for all groups.

these two SOTAs rely on relatively simpler sampling space data structures or do not require such structures at all. (ii) We notice a downside of Bingo. That is, Bingo consumes the most memory for three cases, i.e., DeepWalk-Deletion (Livejournal), node2vec-Insertion (Twitter), node2vec-Deletion (Twitter). The major reason is that these graphs contain more vertices with higher bias. That will render more regular groups in Bingo, which consumes more memory. Given Bingo outperforms the SOTA on larger graphs, we can slightly adjust α and β in Equation 9 when memory consumption is a concern. Further, Bingo is a distributed random walk system and can support higher radix factorizations, both of which can help mitigate the memory consumption problem faced by Bingo.

6.3 Impacts of System Optimizations

Adaptive group representation. Figure 11 illustrates the memory consumption savings brought by the adaptive group representation in Bingo. Figure 11(a) shows that overall this optimization, on average, reduces the memory consumption from 14.6× (GO) to 22.2× (AM). Further, as shown in Figures 11(b) - (d), Dense, One-element, and Sparse group representations, on average, reduce the memory by 323.67×, 21.51×, 6.41×, respectively. Overall, AM graph enjoys the most savings because this group contains the highest ratio of dense groups (72.7%); see Figure 11(e). It is worth noting that our GA optimization resolves the out-of-memory issue faced by the BS design for TW.



(a) Insertion **(b)** Deletion **(c)** Mixed

Figure 12. Impacts of batched updates optimization.

Batched updates. Figure 12 exhibits the performance impacts of our batched graph update optimization. In this study, we ingest ten 100K "insertion", "deletion", and "mixed" graph updates to Bingo in streamed vs. batched designs. Overall, our batched update designs are, respectively, 1006.1x, 1119.1x, 992.5x, faster than streaming update on "insertion", "deletion", and "mixed" update cases because (i) we can parallelize all the updates in batched updates, and (ii) we only perform one rebuild for all the updates. Further, "delete" enjoys the best speedup thanks to our 2-phase parallel delete-and-swap optimization. "Mixed" experiences the smallest speedup because one has to invoke both "insert" and "delete" kernels to perform the updates.

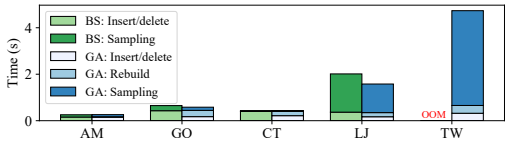


Figure 13. Time consumption of BS vs GA, where BS and GA are defined in Figure 11.

Table 4. Group conversion ratio in LJ graph.

Group Type	Dense	Regular	Sparse	One element
Dense	—	0.02%	0.01%	0.47%
Regular	0.01%	—	<0.01%	0.02%
Sparse	<0.01%	<0.01%	—	0.14%
One element	0.05%	0.03%	0.01%	—

Impact on time cost. Figure 13 presents the time consumption breakdown of with vs without GA optimization. Surprisingly, our GA optimization is, on average, 1.09× faster than BS in addition to the dramatic memory saving presented in Figure 11. This speedup is offered from three aspects: (i) The sampling on GA optimization is 1.05x - 1.61x faster. (ii) Our insert/delete is faster than BS because updating one-element, sparse, and dense groups is faster, and (iii) as shown in TABLE 4, while rebuild introduces extra overheads compared to BS, the conversion from one group type to the other is very low for LJ graph. Particularly, the highest conversion rate is less than 0.47%. This leads the time of GA insert/delete step together with the rebuild step to be merely 8% slower than the BS insert/delete step for the worst case (AM).

6.4 MicroBenchmarks

This section uses DeepWalk with 100K mixed graph updates for microbenchmark evaluations.

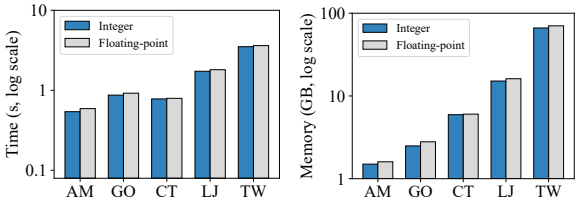
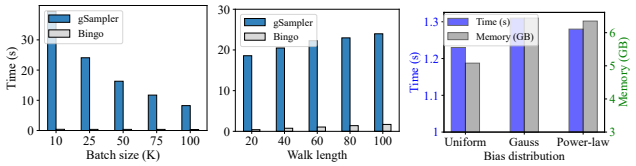


Figure 14. BINGO: Integer bias vs. floating-point bias.

Floating-point vs. integer bias. Figure 14 presents the performance of BINGO on the same dataset with different bias data types. For the fair comparison, the floating-point bias is the integer bias added with a random floating-point value between 0−1.00. Overall, the floating-point bias merely consumes, on average, 1.02× longer and 1.08× more memory than the integer counterpart, which is acceptable.



(a) Batch size. **(b)** Walk length. **(c)** Bias distribution.

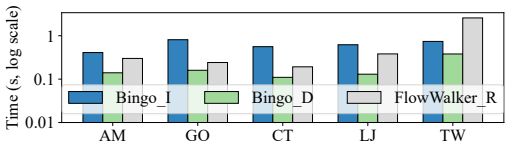
Figure 15. Varying evaluation configurations.

Varying evaluation configurations. Figure 15(a) shows the execution time of gSampler vs BINGO under different updating batch sizes for 1 million updates on the LiveJournal dataset. We observe that the runtime of gSampler and BINGO decrease while the batch size increases. This is because larger batch size provides more parallel execution opportunities and helps reduce the rebuild time.

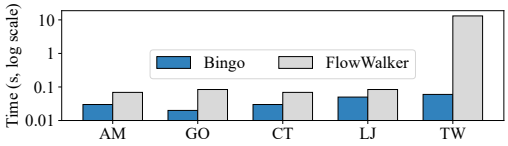
Figure 15(b) studies the performance of gSampler vs BINGO under different walk lengths. With the increase in walk lengths, indicating the increase in workloads, both gSampler and BINGO experience longer runtime. With the help of BINGO accelerated sampling, the gap between gSampler and BINGO also widens from 18.17 s to 22.27 s.

Figure 15(c) evaluates the performance of BINGO under different bias distributions. BINGO on workload with a Uniform bias distribution consumes the least memory and time. This is because uniform bias distribution results in more dense groups and a lower rejection rate. In contrast, the other two distributions have relatively skewed bias distribution, requiring more memory and time to process.

Piecewise breakdown. Figure 16 studies the time consumption of 1 million insertions vs. 1 million deletions vs. 1 million sampling in BINGO and FlowWalker on various graph



(a) Updating time. BINGO_I and BINGO_D denote the time consumption of BINGO performing 1 million insertions, and 1 million deletions, respectively. FlowWalker_R means FlowWalker reloads the graph after applying both the 1 million insertions and 1 million deletions.



(b) Sampling time.

Figure 16. Piecewise breakdown: BINGO vs. FlowWalker on updating and sampling time.

datasets. This study reveals the dominant factors affecting the overall performance. On average, BINGO's insertion is 3.96× slower than deletion. BINGO's sampling is around 2 orders of magnitude faster than deletion and insertion, thanks to $O(1)$ sampling time complexity.

Figure 16(a) also compares the updating time of BINGO vs FlowWalker. FlowWalker achieves slightly faster (2.35X) updating performance than BINGO because it does not require maintaining auxiliary sampling structures. Instead, it simply reloads the new graph after updates. In contrast, BINGO must maintain additional sampling-related data structures, leading to marginally higher update times.

Figure 16(b) further compares their sampling time. Both BINGO and FlowWalker achieve efficient sampling, but BINGO consistently outperforms FlowWalker across all datasets. Notably, in large-scale graphs like TW, BINGO maintains low sampling latency, while FlowWalker suffers a significant performance drop, with BINGO achieving a remarkable 218.7× speedup. This gap stems from FlowWalker's reliance on reservoir sampling, which has an $O(d)$ complexity. As the graph size and vertex degrees increase, this complexity becomes a bottleneck, significantly impacting sampling efficiency.

7 Related Work

7.1 Random Walk and Sampling Systems

CPU-based systems. KnightKing [73] pioneered the effort of building a dedicated graph engine for random walk applications. It introduces a walker-centric programming interface, divides the large biases into two smaller ones in pursuit of a lower rejection rate, and adopts 1-D graph partitioning to enable distributed random walk. GraphWalker

[62], on the other hand, builds a single machine-based random walk framework with a focus on I/O-efficient sub-graph loading strategy. ThunderRW [52] notices that random walks severely under-utilize memory bandwidth. Thus, it breaks random walks into four steps and uses step interleaving to over-subscribe memory access requests. This design triggers aggressive software prefetching to improve memory bandwidth utilization. FlashMob [72] simply re-arranges the memory operations of various sampling tasks to improve memory throughput during random walks. Most recently, TEA [22] implements a hybrid sampling algorithm, which combines ITS and alias methods, for fast and memory-efficient sampling on a new type of graph, i.e., temporal graphs.

Accelerator-based systems. C-SAW [44] leads the effort of building a general sampling framework on GPUs. This paper mainly optimizes the ITS method for first-order sampling. Inspired by KnightKing, NextDoor [24] applies rejection sampling to GPUs and introduces transit parallelism for better load balancing and caching. Skywalker [60] implements parallel construction of alias tables on GPUs for improved load balancing, with Skywalker+ [61] extending Skywalker to multiple GPUs. Recently, gSampler [15] introduced a set of expressive matrix-centric Application Programming Interface (APIs) to enhance the generality and efficiency of graph sampling for graph learning. We also noticed that LightRW [54] has extended ThunderRW to FPGAs.

Whether on CPU, GPU, or FPGA platforms, none of these random walk systems explore random walk on dynamically changing graphs, which is the target of Bingo.

7.2 Dynamic Graph Processing Systems

We identify a line of closely related interesting work which focuses on updating random walk results based on graph updates. These projects mainly aim to rapidly find the affected random walks that were computed previously for updating. Towards that end, Wharf [45] invents a compressed data structure that stores the random walk results with affordable memory and supports rapid results updates on dynamic graphs. FIRM [21] presents an efficient indexing scheme that can trace and update the already calculated personalized PageRank results in constant time complexity when edge insertion and deletions are involved. Similar earlier efforts in this line are FORA [63] and SpeedPPR [66].

Dynamic graph processing, which departs from traditional static graph processing systems[7, 8, 27, 33, 36, 57, 76, 77] or graph query systems[6, 13, 67], is another related direction. First, on GPU platforms, STINGER [14] and cuSTINGER [16] initialized the algorithm and data structure designs for frequent changes in dynamic graphs on GPUs. Hornet [4] further improves cuSTINGER, while faimGraph [65] designs a memory page management strategy for incremental updates on the GPU. Second, on the CPU platform, the streaming graph is a hot topic [23, 37, 38, 59]. Recent years have witnessed a surge of interest in maintaining ordered neighbors during graph updates for efficient streaming graph analytics [48], examples are hash table [1], Packed Memory Array [3, 51], Aspen [12] and Pac-tree [11]. Terrace [43] further adopts multiple data structures, including PMA and B-tree, for better performance. Most recently, LSGraph [48] advocates support for both graph updates and graph computation analytics simultaneously with ordered neighbor updates.

Bingo differs from the aforementioned two lines of work as follows: (i) Bingo is orthogonal to identifying the already calculated random walks for the update. Particularly, once the calculated random walks are identified, instead of rebuilding the sampling space from scratch, Bingo can help them rapidly update the random walks. (ii) The second line of work (i.e., dynamic graph systems) provides a foundation for Bingo. In other words, these systems provide a platform but do not directly address instance generation for sampling or random walks. In particular, we adopt Hornet to support our dynamic data structures on GPUs.

7.3 Second-Order Random Walk

Several projects have dubbed second-order random walks as dynamic graph random walks because these second-order algorithms (i.e., node2vec [17], Metapath [53], Second-order PageRank [68]) require to change the transition probability of current vertex with respect to the history of a random walk. In short, these second-order algorithms introduce a dynamic bias for an unchanged static graph.

There mainly exist two lines of efforts in this direction, i.e., algorithm innovation and system designs: (i) Regarding sampling algorithm design, KnightKing [73] introduces a novel two-step approach for second-order sampling: (1) using static sampling to select a vertex, and (2) using rejection sampling to involve the history of this random walk. On the contrary, FlowWalker [39] introduces two massively parallel sampling approaches based on Reservoir Sampling [58] to quickly sample from the newly built sampling spaces. (ii) For system design, GraSorw [31] is a disk-based system for large graphs, using a triangular bi-block scheduling strategy to convert small random I/Os into large sequential I/Os. SOWalker [69] optimizes I/O utilization by maximizing the benefit from block loading.

Bingo is orthogonal to these approaches because we work on algorithm and system designs of graphs with dynamic changing structures. We adopt Knighking's approach for handling second-order random walk applications, e.g., Node2vec.

8 Conclusion

This paper takes the initiative to build a general random walk engine for dynamically changing graphs with two key principles: (i) this system should support both low-latency streaming updates and high-throughput batched updates. (ii) This system should achieve sampling speed and memory consumption comparable to the existing Monte Carlo sampling algorithms while supporting dynamic updates. Our

system Bingo features three contributions: We first present a novel sampling algorithm that offers constant time sampling and fast updates. Furthermore, we introduce group adaptations for memory-efficient sampling space data structures. Finally, we introduce GPU-aware designs to support high-throughput batched graph updates. Our comprehensive evaluation demonstrates that Bingo outperforms the SOTA.

Acknowledgments

We thank the anonymous reviewers and shepherd Călin Iorgulescu for their helpful suggestions and feedback. This work was in part supported by the NSF CRII Award No. 2331536, CAREER Award No. 2326141, and NSF Awards 2212370, 2319880, 2328948, 2411294, and 2417750.

References

[1] Muhammad A Awad, Saman Ashkiani, Serban D Porumbescu, and John D Owens. 2020. Dynamic graphs on the GPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 739–748.

[2] Saurabh Bajaj, Hui Guan, and Marco Serafini. 2024. Graph Neural Network Training Systems: A Performance Comparison of Full-Graph and Mini-Batch. *arXiv preprint arXiv:2406.00552* (2024).

[3] Michael A Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)* 32, 4 (2007), 26–es.

[4] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 442–446.

[6] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. 2023. Achieving sub-second pairwise query over evolving graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1–15.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[8] Rong Chen, Youyang Yao, Peng Wang, Kaiyuan Zhang, Zhaoguo Wang, Haibing Guan, Binyu Zang, and Haibo Chen. 2017. Replication-based fault-tolerance for large-scale graph processing. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (2017), 1621–1635.

[9] Shiyang Chen, Da Zheng, Caiwen Ding, Chengying Huan, Yuede Ji, and Hang Liu. 2023. TANGO: Re-Thinking quantization for graph neural network training on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[10] Michael Cochez, Petar Ristoski, Simone Paolo Ponzetto, and Heiko Paulheim. 2017. Biased graph walks for RDF graph embeddings. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*. 1–12.

[11] 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*. 108–121.

[12] 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*. 918–934.

[13] Mengsu Ding and Shimin Chen. 2019. Efficient partitioning and query processing of spatio-temporal graphs with trillion edges. In *IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1714–1717.

[14] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[15] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and Efficient GPU-based Graph Sampling for Graph Learning. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 562–578.

[16] Oded Green and David A Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[17] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[19] Jinquan Hang, Zheng Dong, Hongke Zhao, Xin Song, Peng Wang, and Hengshu Zhu. 2022. Outside in: Market-aware heterogeneous graph neural network for employee turnover prediction. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 353–362.

[20] Taher Haveliwala, Sepandar Kamvar, and Glen Jeh. 2003. *An analytical comparison of approaches to personalizing pagerank*. Technical Report. Stanford.

[21] Guanhao Hou, Qintian Guo, Fangyuan Zhang, Sibo Wang, and Zhewei Wei. 2023. Personalized PageRank on Evolving Graphs with an Incremental Index-Update Scheme. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[22] Chengying Huan, Shuaiwen Leon Song, Santosh Pandey, Hang Liu, Yongchao Liu, Baptiste Lepers, Changhua He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2023. Tea: a general-purpose temporal graph random walk engine. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 182–198.

[23] Wole Jaiyeoba and Kevin Skadron. 2019. Graphtinker: A high performance data structure for dynamic graph processing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1030–1041.

[24] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 311–326.

[25] Glen Jeh and Jennifer Widom. 2002. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 538–543.

[26] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research* 21, 1 (2020), 2648–2720.

[27] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[28] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350. http://dl.acm.org/citation.cfm?id=2488173

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

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation

for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[31] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs. *arXiv preprint arXiv:2203.16123* (2022).

[32] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. 2014. Random-walk domination in large graphs. In *IEEE 30th International Conference on Data Engineering*. IEEE, 736–747.

[33] Xue Li, Mingxing Zhang, Kang Chen, and Yongwei Wu. 2018. Regraph: A graph processing framework that alternately shrinks and repartitions the graph. In *Proceedings of the International Conference on Supercomputing*. 172–183.

[34] Yice Luo, Guannan Wang, Yongchao Liu, Jiaxin Yue, Weihong Cheng, and Binjie Fei. 2023. FAF: A Risk Detection Framework on Industry-Scale Graphs. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 4717–4723.

[35] Minbo Ma, Peng Xie, Fei Teng, Bin Wang, Shenggong Ji, Junbo Zhang, and Tianrui Li. 2023. Histgnn: Hierarchical spatio-temporal graph neural network for weather forecasting. *Information Sciences* 648 (2023), 119580.

[36] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[37] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-aware incremental processing of streaming graphs. In *Proceedings of the sixteenth European conference on computer systems*. 83–98.

[38] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference*. 1–16.

[39] Junyi Mei, Shixuan Sun, Chao Li, Cheng Xu, Cheng Chen, Yibo Liu, Jing Wang, Cheng Zhao, Xiaofeng Hou, Minyi Guo, et al. 2024. FlowWalker: A Memory-efficient and High-performance GPU-based Dynamic Graph Random Walk Framework. *arXiv preprint arXiv:2404.08364* (2024).

[40] Duane G Merrill and Andrew S Grimshaw. 2010. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 545–546.

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[42] Giannis Nikolentzos and Michalis Vazirgiannis. 2020. Random walk graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 16211–16222.

[43] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the international conference on management of data*. 1372–1385.

[44] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[45] Serafeim Papadias, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Space-efficient random walks on streaming graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 356–368.

[46] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[47] Sandeep Polisetty, Juelin Liu, Kobi Falus, Yi Ren Fung, Seung-Hwan Lim, Hui Guan, and Marco Serafini. 2023. GSplit: Scaling graph neural network training on large graphs via split-parallelism. *arXiv preprint arXiv:2303.13775* (2023).

[48] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: a locality-centric high-performance streaming graph engine. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 33–49.

[49] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[50] Marco Serafini and Hui Guan. 2021. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 68–76.

[51] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Technical report: Accelerating dynamic graph analytics on gpus. *arXiv preprint arXiv:1709.05061* (2017).

[52] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: an in-memory graph random walk engine. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1992–2005.

[53] Yizhou Sun and Jiawei Han. 2013. Mining heterogeneous information networks: a structural analysis approach. *ACM SIGKDD explorations newsletter* 14, 2 (2013), 20–28.

[54] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2023. LightRW: FPGA Accelerated Graph Dynamic Random Walks. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[55] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting gpus for low-latency, high-throughput gnn serving with workload awareness. *arXiv preprint arXiv:2305.10863* (2023).

[56] Pierre Terdiman. 2000. Radix Sort Revisited. (Apr. 2000). www.codercorner.com/RadixSortRevisited.htm.

[57] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. 2013. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 197–210.

[58] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[59] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[60] Pengyu Wang, Chao Li, Jing Wang, Taolei Wang, Lu Zhang, Jingwen Leng, Quan Chen, and Minyi Guo. 2021. Skywalker: Efficient alias-method-based graph sampling and random walk on GPUs. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 304–317.

[61] Pengyu Wang, Cheng Xu, Chao Li, Jing Wang, Taolei Wang, Lu Zhang, Xiaofeng Hou, and Minyi Guo. 2023. Optimizing GPU-based Graph Sampling and Random Walk for Efficiency and Scalability. *IEEE Trans. Comput.* (2023).

[62] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *USENIX Annual Technical Conference (USENIX ATC 20)*. 559–571.

[63] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient algorithms for approximate single-source personalized pagerank queries. *ACM Transactions on Database Systems (TODS)* 44, 4 (2019), 1–37.

[64] Tianyi Wang, Yang Chen, Zengbin Zhang, Tianyin Xu, Long Jin, Pan Hui, Beixing Deng, and Xing Li. 2011. Understanding graph sampling algorithms for social network analysis. In *31st international conference on distributed computing systems workshops*. IEEE, 123–128.

[65] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High performance management

of fully-dynamic graphs under tight memory constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 754–766.

[66] Hao Wu, Junhao Gan, Zhewei Wei, and Rui Zhang. 2021. Unifying the global and local approaches: An efficient power iteration with forward push. In *Proceedings of the International Conference on Management of Data*. 1996–2008.

[67] Jingqi Wu, Rong Chen, and Yubin Xia. 2021. Fast and Accurate Optimizer for Query Processing over Knowledge Graphs. In *Proceedings of the ACM Symposium on Cloud Computing*. 503–517.

[68] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember where you came from: on the second-order random walk based proximity measures. *Proceedings of the VLDB Endowment* 10, 1 (2016), 13–24.

[69] Yutong Wu, Zhan Shi, Shicai Huang, Zhipeng Tian, Pengwei Zuo, Peng Fang, and Dan Feng. 2023. SOWalker: An I/O-Optimized Out-of-Core Graph Processing System for Second-Order Random Walks. In *USENIX Annual Technical Conference (USENIX ATC 23)*. 87–100.

[70] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[71] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. 2019. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence* 4, 2 (2019), 95–107.

[72] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random walks on huge graphs at cache efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 311–326.

[73] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM symposium on operating systems principles*. 524–537.

[74] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S Yu. 2021. Consisrec: Enhancing gnn for social recommendation via consistent neighbor aggregation. In *Proceedings of the 44th international ACM SIGIR conference on Research and development in information retrieval*. 2141–2145.

[75] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient graph computation for Node2Vec. *arXiv preprint arXiv:1805.00280* (2018).

[76] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[77] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *arXiv preprint arXiv:1910.05773* (2019).