PlatoD2GL: An Efficient Dynamic Deep Graph Learning System for Graph Neural Network Training on Billion-Scale Graphs

Xing Huang† Dandan Lin* Weiyi Huang† Shijie Sun† Jie Wen† Chuan Chen†
WeChat, Tencent Inc.† Shenzhen Institute of Computing Sciences*
{healyhuang, viniehuang, cedricsun, welkinwen, chuanchen}@tencent.com† lindandan@sics.ac.cn*

***Abstract*—Recently, huge interests in both academic and industry have been posed to Graph Neural Network due to its power on revealing the topological information inside the data. To support the real-world applications, most of (if not all) which contain large-scale graphs with billions of edges, a number of graph-based deep learning systems have been proposed and implemented. However, all of them fail to efficiently process the dynamic graphs in terms of both memory and time cost. The state-of-the-art suffers from two issues: (1) *expensive memory consumption* due to the huge indexing overhead of numerous key-value pairs in traditional key-value topology storage and (2) *inefficient dynamic updating* due to the heavy updates on indexing structures for weighted neighbor sampling. In this paper, we proposed a Dynamic Graph-based Learning System *PlatoD2GL* to address above two issues. Specifically, we design a novel and effective non-key-value data structure, termed *samtree*, for dynamic topology storage, which largely reduces the memory cost. In addition, we propose an efficient sampling indexing structure *FSTable* by utilizing Fenwick tree, guaranteeing the efficiency of both dynamic updates and weighted sampling. Comprehensive experiments have demonstrated that, for dynamic updating the graph topology, the efficiency of our system *by up to 79.8%* and *by up to 6.3 times* in terms of memory and time cost, respectively. Now, our system serves the major traffic in WeChat Platform for training various GNN models.**

I. Introduction

By taking the graph data (*i.e.,* vertices and edges) as input, graph neural network (GNN) models update the model parameters by considering both topology information and attributes on vertices or edges. Due to its power, GNN models have been widely applied in various real-world applications in many critical fields, including recommendation systems [24], [31], spam review detection [22], fraud detection [5], [29], [6], biomolecule classification [28], natural language processing [35] and so forth. Figure 1 plots an example of training phase of GNN models in reality where the graph is stored in a distributed way. Specifically, a GNN approach updates the embedding of a vertex by iteratively aggregating the information from its neighbors. For example, to update the embedding of vertex 1, it firstly *samples* a proportion of it neighbors, *i.e.,* vertex 2 and vertex 3, then *aggregates* the information of sampled neighbors, and finally, *combines* the previous information of vertex 1 as the neighbor information as the final embedding. Note that sampling a proportion of neighbours is widely used in real world applications since aggregating all neighbours of a vertex is infeasible for large-scale graphs [13], [3], [14], [38], [12].

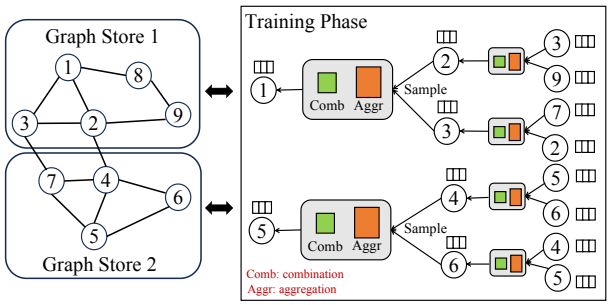


Fig. 1: A running example of GNN training phase.

In general, to support the training of GNN models, a deep graph learning system is important by providing the following two abilities: (1) a *distributed graph storage* for storing the large scale graphs (which is common in reality); (2) an *efficient sampling strategy* for sampling K-hop neighbors of a vertex; For example, for the live-streaming recommendation scenarios in WeChat (the largest social media APP in China), the user-item interaction graph could be with *nearly 63.3 billion edges*, which cannot be stored in a single machines. However, due to the explosion of GNN models in real-world applications, supporting the GNN training on dynamic graphs is also an indispensable requirement for a deep graph learning system. For instance, for the online recommendation services, the user interest is highly dynamic and non-stationary [4], [16], [33], [24], leading to the frequently-evolving user-item graphs. If a GNN-based recommendation model cannot capture the *instant* user interest, the user might not be interested in the recommended items inferred by the GNN model, making the loss of user in the platform [9].

For training GNN on large scale graphs, a lot of distributed deep graph learning systems have been proposed, *e.g.*, *AliGraph* [38], *Euler* [17], *Plato* [18], *DistDGL* [37], *ByteGNN* [36], and *PlatoGL* [24]. Most of existing deep graph learning systems, *e.g.,* Euler, Plato, DistDGL and ByteGNN provide a *static* graph storage, *i.e.,* directly storing the graph in a cluster of physical machines (termed *graph servers*) by using the graph partition methods (like METIS [19]). However, these systems fail to support dynamic graphs since the graph needs to be re-partitioned and re-deployed *from scratch* in

graph servers when an edge is inserted/deleted in the graph.

Among existing systems, PlatoGL [24] is the state-of-the-art to support a dynamic graph storage by utilizing the block-based *key-value* store. That is, edges in a graph are stored as tuples of ⟨key, value⟩, where the key is a vertex s in the graph with some extra information and the value is a block that stores a proportion of neighbors of vertex s. In addition, for the efficient weighted neighbor sampli, PlatoGL [24] proposed a block-based sampling method by exploiting the Inverse Transform Sampling (ITS) method [34]. However, PlatoGL suffers from the following two limitations.

- **the expensive memory consumption issue.** Since a source vertex s might have multiple blocks, each key designed by PlatoGL consist of various information except the unique identifier (ID) of vertex s for *uniquely mapping* to a specific block. Thus, it has to maintain large number of indexings for mapping the keys with their corresponding values (which is the limitation of the key-value format), leading to expensive memory overhead.
- **the inefficient dynamic graph updates issue.** It needs to update *cumulative sum table* (CSTable) (which is used for fast weighted sampling) for each source vertex. To be specific, when a new neighbor of a source vertex s is inserted, the CSTable of s should be re-computed from scratch since each element v in CSTable requires to sum up all the weights of vertices before v. In worst case, it takes $O(n_{\mathbb{L}})$ time cost where $n_{\mathbb{L}}$ is the number of elements (*i.e.,* out-neighbors) in CSTable. The case would be worse in reality since the graph is frequently evolving in real-world applications.

Contributions. This paper tackles these questions. We introduce *PlatoD2GL*, a distributed dynamic deep graph learning system that supports efficient dynamic graph updates without taking much memory cost and fast sampling strategy. The following shows our contributions.

(1) We propose a dynamic graph storage layer inside PlatoD2GL to store multiple GNN-related data. Specifically, we design a novel and effective *non-key-value* topology storage, termed *samtree*, which largely reduces the memory cost for the dynamic updates by avoiding maintaining the indexings for key-value mapping. Besides, we propose efficient but non-trivial insertion and deletion mechanisms, and theoretically proved that the average time cost of insertion and deltion is *linear* to the number of vertices in the samtree.
(2) We propose an efficient sampling indexing structure *FSTable* by utilizing Fenwick tree [8]. To our best knowledge, we are the first to exploit Fenwick tree for dynamic graphs. In addition, we theoretically prove that the time complexity of updating a FSTable with $n_{\mathbb{L}}$ elements is $O(\log n_{\mathbb{L}})$, which is more efficient than existing CSTable used in PlatoGL.
(3) We also design a novel weighted sampling method *FTS* by incorporating our FSTable with traditional ITS method.
(4) We propose two novel optimization techniques inside our PlatoD2GL system: (1) a compression technique by compressing our *samtree* for further reducing the memory cost and (2) a batch-based latch-free concurrent mechanism by avoiding the race conditions for accelerating the dynamic updating process.
(5) We experimentally verified that by conducting comprehensive experiments on two public datasets and one production dataset with up to 2.1 billion nodes and 63.9 billion edges. Comprehensive experiments showed that PlatoD2GL takes less memory cost than state-of-the-art by **up to 79.8%** on billion-scale graphs. Besides, PlatoD2GL is faster than state-of-the-art by **up to 6.3 times** and **up to 10.1 times** in terms of dynamic updates and sampling.

Symbol	Description
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	The directed graph G with vertices set V, edges set E
$w_{u,v}$	The weight of an edge $e(u, v)$
w_u	The sum of weights of all edges that linked from u
$\mathcal{N}_s$	The set of out-neighbours of vertex s
n_s	The number of out-neighbours of vertex s
$\mathcal{T}_s$	The samtree of vertex s for storing its out-neighbors
c, H	The node capacity and height of a smatree $\mathcal{T}$, respectively
r_i	The node in the i-th level of a smatree $\mathcal{T}$
α	The slackness for α-Split algorithm
$n_{\mathbb{L}}$	The number of vertices in a leaf node of a samtree
$\mathbb{C}$	The cumulative sum table (CSTable) for ITS method
$\mathbb{F}$	The FSTable for FTS method
$S_{\mathbb{L}}$	The sum of weights in a leaf node

TABLE I: Frequently-used notations.

II. Preliminaries

In this section, we firstly introduce background of the problem studied in this paper, and next presents a widely-used weighted sampling method *Inverse Transform Sampling* (ITS), which is the building stone in our proposed system. Table I lists the notations used in this paper.

A. Background

We start with a simple directed weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{W})$ where $\mathcal{V}$ and $\mathcal{E}$ represents the set of vertices and edges,respectively; and $\mathcal{W}$: $\mathcal{E} \rightarrow \mathbb{R}^+$ is a function that assigns a weight $w_{u,v}$ to an edge $e(u, v)$ linking from vertex u to vertex v. For simplicity, we denote a weighted edge as $e(u, v, w_{u,v})$. Let $\mathcal{N}_u$ denote the set of out-neighbours of vertex u. In the sequel, we use "out-neighbor" and "neighbor" interchangeably if it is clear in the context. In this paper, we consider the heterogeneous graph with multiple types of vertices and/or edges, which is common in real-world applications. Besides, in the sequel, we focus on the dynamic graphs. Given a time interval t, a *dynamic graph* can be considered as a series of graphs $\{\mathcal{G}^{(t)}|t \in [1, T]\}$ where $\mathcal{G}^{(t)}$ is a heterogeneous graph at timestamp t and T is the largest timestamp.

Basic operations in GNN models. GNN approaches could be basically formulated as *message passing* [37], [10], [32], [13], [30], where nodes in the graph propagate their messages to their neighbours and updates their representations (*i.e.*, in form of embedding) by aggregating the received messages from their neighbours. Given a heterogeneous graph $\mathcal{G}$, the embedding $\mathbf{e}_u^{(0)}$ of each node u is initially assigned as its features vec-

tor $\mathbf{f}_u$. To get the representation of node u at layer l, denoted by $\mathbf{e}_u^{(l)}$, a GNN approach performs the computations as follows:

$$\mathbf{e}_u^{(l+1)} = g(\mathbf{e}_u^{(l)}, \bigoplus_{i \in \mathcal{N}_u} f(\mathbf{e}_u^{(l)}, \mathbf{e}_i^{(l)})), \quad (1)$$

where $f(\cdot)$, $\bigoplus(\cdot)$, and $g(\cdot)$ are customized functions (*i.e.*, neural network modules) for (i) *calculating* messages from each neighbour of node u, (ii) *aggregating* the messages of all neighbours of node u, and (iii) *combining* self-message of node u with the neighbor information to update representation of node u, respectively. In the literature [13], [3], [14], [38], [12], as aggregating all neighbours of node u is infeasible for large-scale graphs, a *sampling* method to sample a proportion of neighbours is adopted to enhance the efficiency of the GNN algorithms without sacrificing much accuracy.

Dynamic GNN models. In dynamic scenarios, a GNN model $\mathcal{M}^{(t)}$ works on a dynamic graph $\mathcal{G}^{(t)}$ during both training and inference periods, as $\mathcal{M}^{(t)}$ is expected to learn user preference from $\mathcal{G}^{(t)}$ that receives updates. That is, at the timestamp t, they exploit the topological information in the graph $\mathcal{G}^{(t)}$ for the sampling, aggregation and combination operations.

B. Existing Weighted Sampling Method and Its Limitations

As stated above, one of the key operations in most GNN approaches is to select a subset of vertices from a node s's out-neighbors, where $s \in \mathcal{V}$, with *either* unweighted random sampling *or* weighted sampling. Specifically, the unweighted random sampling is to sample a vertex from s's out-neighbors uniformly at random, *i.e.,* for each out-neighbor of s, the probability to be sampled is $\frac{1}{n_s}$. While the weighted sampling is to sample an out-neighbor according to the edge weight. In other words, given a set $\mathcal{N}_s$ of s's neighbors, as well as the edge weight $w_{s,u}$ for each $u \in \mathcal{N}_s$, it is to select an out-neighbor u with the probability $\frac{w_{s,u}}{w_s}$ where w_s is the sum of weights of edges linked from s, which is not always equal to 1 in reality. Thus, compared with the unweighted random sampling, the weighted sampling is more complicated and challenging to be computed for GNN approaches.

The Inverse Transform Sampling (ITS) method [34] is a widely-used approach for answering the weighted neighbor sampling. In the following, we present how the ITS method works. Suppose a vertex s has n_s neighbours, *i.e.,* $[v_0, v_1, ..., v_{|n_s-1|}]$. Instead of directly storing the edge weights, the ITS method computes a *cumulative sum table* (*CSTable*) $\mathbb{C}$ that stores the sampling probability of each out-neighbor. Specifically, for the i-th neighbor, the sampling probability is computed as the *prefix sum* (*i.e.,* cumulative sum) of weights located before i, as shown as follows:

$$\mathbb{C}(i) = \sum_{j=0}^{i} w_{s,v_j} \quad (2)$$

Obviously, the ITS method takes the same memory cost as storing the edge weights, avoiding extra memory cost.

Next, to sample an out-neighbor, it firstly generates a random number $\mathcal{R}$ in $[0, \mathbb{C}(n_s-1))$ and finds *the smallest* i where $\mathbb{C}(i) > \mathcal{R}$ using binary search, producing v_i as the sampled neighbor. The time complexity to sample the neighbourhood of a source node s is $O(\log n_s)$ since only binary search is used.

Although ITS method is memory-friendly and efficient for weighted sampling, it is computational expensive for the dynamic graphs due to the maintenance overhead of CSTables. Specifically, when a new neighbor u is inserted, *or* a neighbor is deleted, *or* the weight of a neighbor is updated, the CSTable $\mathbb{C}$ requires to be updated, taking $O(n_s)$ time cost since it has to update all the values located after u. The maintenance overhead becomes more expensive in real-world since the graphs are frequently changed. In this paper, we propose a sampling method that is efficient in terms of both memory cost and times cost for dynamic updates (to be introduced in Section V).

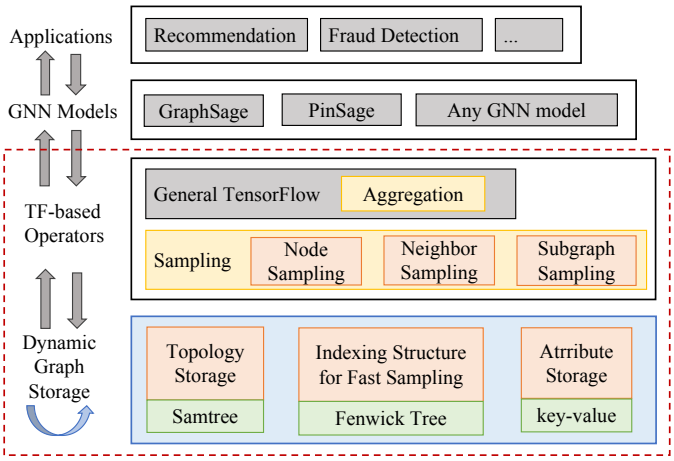


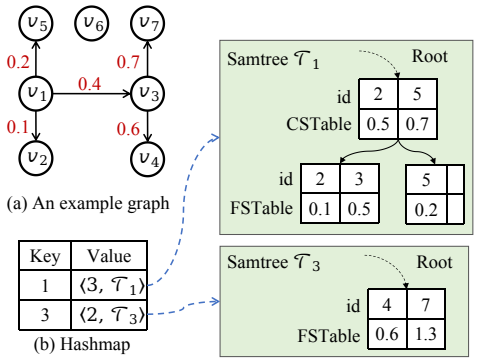
Fig. 2: Overview of PlaotD2GL (the red dashed rectangle).

III. System Overview

In this section, we give an overview of our proposed *PlatoD2GL* system, whose framework is plot in Figure 2.

On the whole, it mainly consists of two layers. From top to bottom, the first layer is *the TF-based operators layer* which provides TensorFlow-based operators to support the aggregation and sampling operations in GNN models. In particular, for sampling operation in GNN models, three sampling methods are proposed for dynamic graphs, namely, *node sampling* (which samples a set of nodes from a whole graph), *neighbor sampling* (which samples a fixed-number of neighbors of a given node in the graph), *subgraph sampling* (which samples a subgraph pivoted at a given node in the graph). The second layer is *the dynamic graph storage layer*, which stores dynamic graph topology, indexing structures for fast samplings, and attributes information of nodes or edges. To be specific, for storing dynamic graph topology, we propose a novel memory-efficient *non-key-value* data structure **Samtree** with non-trivial but efficient insertion and deletion mechanisms. In addition, to support sampling operations in dynamic graphs, we propose an indexing structure *Fenwick-tree*. As for the attribute storage, the key-value store is used.

In the sequel, Section IV elaborates the dynamic graph storage and Section V presents the efficient sampling strategy inside the system *PlatoD2GL*.



(a) An example graph

(b) Hashmap

Fig. 3: A running example of graph topology storage.

IV. Dynamic Graph Storage

In this section, we introduce our dynamic graph storage with efficient insertion and deletion mechanisms.

Challenges. Most of existing deep graph learning systems, *e.g.*, DGL [37], AliGraph [38], and Plato [18], provide a *static* graph storage, which is computationally expensive since such static graph storage needs to be reconstructed from scratch when the graph is updated. Among existing systems, PlatoGL [24] can support a dynamic graph storage by utilizing the block-based *key-value* store. Specifically, inside PlatoGL, edges in a graph are stored as tuples of $\langle$ key, value $\rangle$, where the key is a node u in the graph and the value is a set of blocks, each of which stores a proportion of neighbors of node u. However, the dynamic graph storage of PlatoGL suffers from *expensive memory consumption* since it has to maintain large number of indexings for mapping a key with its corresponding value (which is the limitation of the key-value store). Besides, each key in PlatoGL consist of various information except the unique identifier (ID) of source node, increasing the memory consumption. To address this issue, we propose a *memory-efficient* dynamic graph storage by designing a *non-key-value* data structure *Sample Tree* (**Samtree**) for storing the graph topology, which largely reduces the memory cost. Note that our *Samtree* also helps efficient graph queries, especially sampling a subset of neighbors for a given node (to be introduced in Section V). In this section, we first introduce the data structure *Samtree*, and then describe its insertion and deletion mechanisms for storing edges in the graph.

A. Samtree Structure

In this paper, we define a samtree with node capacity c in Definition 1.

Definition 1. *A samtree with node capacity c is a tree that satisfies the following properties: (1) each node [1] has at most c children; (2) every internal node has at least $\lceil \frac{c}{2} \rceil$ children; (3) the root node has at least two children unless it is a leaf; and (4) all leaves appear on the same level.*

[1]For clarity of presentation, in the sequel, we use the word "node" for a tree structure, which can store multiple information, to distinguish from the vertices in a graph.

To store the graph information, we construct a samtree $\mathcal{T}_s$ for each vertex s in the graph where samtree $\mathcal{T}_s$ contains all neighbors of vertex s. For improving the efficiency of both dynamic updates and sampling queries, our proposed samtree has the following four constraints. (1) In samtree, the leaves store the graph data (*i.e.,* the neighbors of a source vertex) while the internal nodes store the aggregation information of its children in the tree. For a leaf (*resp.* non-leaf) node, a IDs list is provided for storing the IDs of vertices located in this leaf node (*resp.* the successors of this non-leaf node). (2) The IDs list in a leaf node is *unordered* for fast updating (due to the utilization of Fenwick tree) while that in a non-leaf node is *ordered* for fast search. (3) Each non-leaf node has a CSTable for efficiently sampling a leaf node in the samtree (which can be regarded as a phase of weighted neighbor sampling). (4) Each leaf node has a FSTable for fast sampling a neighbor that located in this leaf node according to the edge weights. Recall that in our system, we propose FSTable as an indexing structure for weighted neighbor sampling in dynamic graphs by utilizing Fenwick tree [8], which can accelerating the dynamic udpates of sampling probabilities of neighbors in the leaf nodes and will be elaborated in Section V.

Due to the delicate design of our samtree structure, our PlatoD2GL system can provide an efficient dynamic updates for graph storage and a fast neighbor sampling query *simultaneously*. To support the sophistication, we also propose non-trivial but efficient insertion and deletion mechanisms tailored for our samtree structure.

B. Topology Storage

In this section, we present how samtree stores the edges in the graph. Specifically, for a vertex s in the graph, we maintain a samtree $\mathcal{T}_s$ to store all the neighbors of this vertex. Besides, we used a hashmap structure to store the nodes in the graph. To be specific, the key is a vertex s and the value is a tuple of $\langle |\mathcal{N}_u|, \mathcal{T}_u \rangle$ where $\mathcal{N}_u$ is the set of out-neighbors of vertex u, and $\mathcal{T}_u$ is the sametree of u. Note that for handling concurrent updates of multiple source vertices, our implementation adapts a concurrent hashmap structure by exploiting *Cuckoohash* [7], [23]. To illustrate, an running example of graph storage in PlatoD2GL is given below, as shown in Figure 3.

Example 1. *Given a graph $G(V, E, W)$ with 7 vertices and 5 edges where $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and $E = \{e(v_1, v_2, 0.1), e(v_1, v_3, 0.4), e(v_1, v_5, 0.2), e(v_3, v_4, 0.6), e(v_3, v_7, 0.7)\}$, as shown in Figure 3(a). Figure 3(b) plots a hashmap for source vertices of these edges as follows: $\{(1 : \langle 3, \mathcal{T}_1 \rangle, (3 : \langle 2, \mathcal{T}_3 \rangle)\}$. it indicates that v_1 has 3 out-neighbors and $\mathcal{T}_1$ is the samtree of v_1 for storing its neighbors, which is present in the top-right green box. For vertex v_3, it has 2 out-neighbors and its corresponding samtree is $\mathcal{T}_3$, as shown in the bottom-right green box. As for those vertices that have no out-going edges in G, no information will be reserved in topology storage, reducing the memory cost.*

For better illustration, we set the node capacity of samtrees in this example as 2. Take vertex v_3 for example. Since it only

Algorithm 1: α-SPLIT ALGORITHM

Input: An array A to be split, the splitting position k, the slackness α
Output: The soft splitting position $\hat{k}$

$m \leftarrow$ the median position of A;
Swap the element at m with the first element in A;
$pos \leftarrow$ HOAREPARTITION(A); //position of the current pivot
if $pos \in [k-\alpha, k+\alpha]$ **then**
  $\hat{k} \leftarrow pos$;
else if $k \leq pos$ **then**
  $A' \leftarrow A[0, pos-1]$;
  $\hat{k} \leftarrow \alpha$-SPLIT$(A', k, \alpha)$;
else
  $A' \leftarrow A[pos+1, |A|-1]$
  $\hat{k} \leftarrow \alpha$-SPLIT$(A', k-pos-1, \alpha)$;
return $\hat{k}$;

Algorithm 2: INSERTION

Input: A Samtree $\mathcal{T}$, a vertex v to be inserted
Output: An updated Samtree $\mathcal{T}_{new}$

$P \leftarrow DFS(T, v)$; //a path from root to leaf for inserting v
$r_{H-1} \leftarrow$ the last node in path P;
if v has already been in r_{H-1} **then**
  Update weight of v in r_{H-1};
else
  Append v in the last position of r_{H-1};
if r_{H-1} is full **then**
  Split r_{H-1} with v by using Algorithm 1;
Update CSTables & FSTables in path P from bottom to top;
return the updated tree $\mathcal{T}_{new}$;

has 2 out-neighbors, i.e., *vertex 4 and vertex 7, the samtree $\mathcal{T}_3$ can store the neighbors in one leaf node. Inside the leaf node, it contains two information: (1) a ID list of neighbors in this leaf node and (2) a* FSTable *which is computed by considering the weights. For example, in FSTable, the first element is 0.6, which is the weight of neighbor 4, and the second element is the sum of the weights of vertices 4 and 7, which is 1.3.*

Next, take vertex v_1 for example. Since it has 3 out-neighbors, larger than the maximum node capacity of a samtree, its samtree $\mathcal{T}_1$ contains one non-leaf node and two leaf nodes (the details of such tree construction will be introduced in Section IV-C). Inside $\mathcal{T}_1$, the non-leaf node contains two types of information: (1) a ID list where the i-th element is the smallest ID in its i-th child; and (2) a CSTable, e.g.,*, the first element is the sum of weights in its first child, which is 0.5, and the second element is the sum of weights in both the first and second child,* i.e., $0.5 + 0.2$*, which is 0.7. For the leaf nodes in $\mathcal{T}_1$, the results are similar to $\mathcal{T}_3$ by carrying the FSTables. Although the computation of FSTable looks same as that of CSTable in this example, our FSTable is totally different from CSTable when the nod capacity is larger than 2 (see more details in Section V).*

C. Insertion Mechanism

In this section, we elaborate how to insert a new edge $e(u, v, w)$ into the topology storage. That is, how to insert the new neighbor v with edge weight w to the samtree $\mathcal{T}_u$ of vertex u. The basic idea of insertion is to firstly find the specific leaf node that new neighbor v should be located in $\mathcal{T}_u$, and then insert vertex v in the leaf node.

Challenges. Although the idea is simple and straightforward, the case becomes challenging when the leaf node is full since it requires to balance the trade-off between insertion efficiency and graph query efficiency. If we increase unlimitedly the node capacity of a samtree, the memory cost is expensive and it violates the definition of samtree in Definition 1. Besides, as stated in above, the leaf node in a samtree has an unordered constraint where the ID list is unordered but the elements in non-leaf nodes are in increasing order. Thus, the leaf node cannot be split into two nodes by *dividing the elements evenly* since such operation *fails* to maintain the ordered constraint of non-leaf nodes. To maintain such two constraints in samtree, a greedy splitting method is to firstly sort the elements in r_{H-1} in increasing order, and then split it into two equal parts. However, this method is *computationally expensive* since it takes $O(n_{\mathbb{L}} \log n_{\mathbb{L}})$ time cost for sorting elements where $n_{\mathbb{L}}$ is the number of elements in this leaf node. Furthermore, the time cost will be worse for the dynamic graphs since the samtree structures are frequently changed to insert new neighbors, and thus, cannot satisfy the efficient dynamic-updating requirement in reality. To tackle above issue, we propose a α-relaxed split (termed α-Split) algorithm by *omitting the strict orders* of elements in the leaf node r_{H-1}.

In this section, we firstly elaborate our α-Split algorithm, and then present the complete implementation of our insertion algorithm for samtree structure.

α-Split algorithm. In particular, our α-Split algorithm aims to find the pivot k in r_{H-1} such that the following two constraints can be satisfied: (1) for any j such that $j < k$, $v_j < v_k$; and (2) for any j such that $j > k$, $v_j > v_k$, where v_i is the i-th element in r_{H-1}. Thus, based on pivot k, the leaf node r_{H-1} can be easily split into two parts *without strictly sorting*. Besides, to improve the splitting efficiency, our algorithm selects an α-*approximate* pivot $\hat{k}$ such that the approximate pivot $\hat{k}$ satisfied the following α-relaxed inequality:

$$k - \alpha \leq \hat{k} \leq k + \alpha, \tag{3}$$

where α is the user-specified slackness parameter. Obviously, the larger the slackness α, the faster the splitting speed, less balanced the nodes after splitting, and vice versa.

To achieve this goal, our α-Split algorithm finds the α-approximate pivot $\hat{k}$ in a recursive manner by exploiting the traditional *Hoare Partition* algorithm [15]. In each round, it selects the value in the median position as the candidate pivot p, and puts this candidate pivot at the correct position k' in the list by using *Hoare Partition* algorithm such that all the values before k' is smaller than this pivot while the values after k' is larger. After that, it checks whether this candidate pivot position satisfies the α-relaxed inequality. The selection terminates when such a position that satisfies the α-relaxed inequality is found.

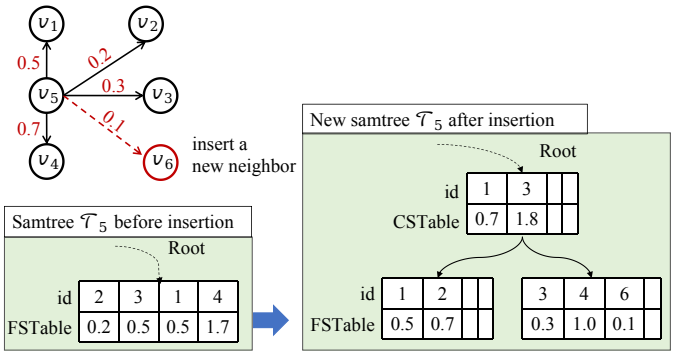


Fig. 4: A running example of samtree insertion.

Algorithm 1 presents the pseudocode of our α-Split algorithm. Given an ID array (list) A in a leaf node to be split, the splitting position k, and the slackness α, it returns the final soft spliting position $\hat{k}$ as result. To call this algorithm, the splitting position k is initialized as $\frac{|A|}{2}$ where $|A|$ is the size of array A, since the best splitting position is the median for guaranteeing the tree balance. Firstly, it swaps the element $A[m]$ in the median position m of A with the last element in A (Lines 1-2). Secondly, it puts $A[m]$ in the correct position pos by using HoarePartition algorithm and returns pos as the candidate pivot in this round(Line 3). Then, it checks whether pos satisfies the α-relaxed inequality (Lines 4-11). If yes, it directly sets $\hat{k}$ as pos, and the process terminates (Line 5). Otherwise, it chooses a child for pivot selection in next round by comparing the current position k with pos (Line 6). If $k \leq pos$, it indicates that the target position is in the left part of array A, and so, it narrows the array A by focusing the elements whose indices are in the range of $[0, pos-1]$, and then calls the α-Split algorithm by taking as input a new array A', k and α (Lines 7-8). Otherwise, the target position is in the right part of array A. Hence, it sets a new array A' by focusing the elements whose indices are in the range of $[pos+1, |A|-1]$, and next calls the α-Split algorithm by taking as input a new array A', $(k-pos-1)$ and α (Lines 9-11).

Unlike the *QuickSelect* algorithm that finds the exact pivot k that split an array into two equal parts, our α-Split algorithm selects an *approximate* pivot $\hat{k}$ with α. Thus, the *QuickSelect* algorithm can be regarded as a special case of our α-Split algorithm by setting $alpha$ as 0. Theorem 1 proves that the average time cost of Algorithm 1 is linear to the number of neighbors in the leaf node. Thus, our splitting algorithm reduces the time cost from $O(n_{\mathbb{L}} \log n_{\mathbb{L}})$ to $O(n_{\mathbb{L}})$, which is efficient. Due to the space limite, the proof could be found in Appedic C.

Theorem 1. *The average time cost of Algorithm 1 is* $O(n_{\mathbb{L}})$.

Implementation of the insertion algorithm. The pseudocode for the insertion machanism of *Samtree* is presented in Algorithm 2. Given a samtree $\mathcal{T}$ and the vertex v to be inserted, Algorithm 2 returnes the new samtree $\mathcal{T}_{new}$ after insertion. Firstly, it searches a path P that vertex v should be located by using DFS search on the samtree $\mathcal{T}_u$ (Line 1), *i.e.,* $P = \{r_0, r_1, ..., r_{H-1}\}$ where H is the height of samtree $\mathcal{T}$. During DFS search, in each non-leaf node where the elements in ID list is in *increasing order*, it compares the ID of vertex v with the ID list for maintaining the ordered constraint in non-leaf node. Specifically, it finds the largest j where the j-th element in ID list is larger than the ID of v and goes for the j-th child. Secondly, it checks the leaf node r_{H-1} in path P to see how to insert v in r_{H-1} (Lines 2-8). If vertex v has already been in r_{H-1}, it updates the weight of v accordingly. Otherwise, it appends vertex v to the leaf r_{H-1}. After insertion, if the number of elements in r_{H-1} is greater than the specified threshold r_{max} (*i.e.,* node capacity), an insertion-split operation will be triggered in r_{H-1} to split the node r_{H-1} by using Algorithm 1 to find the soft splitting position. Finally, it updates the CSTable or FSTable of each node in path P from the leaf r_{H-1} to the root r_0 since the probabilities for sampling have been changed due to the insertion of v (Line 9). How to update the CSTable and FSTable will be introduced in Section V. Besides, when the splitting on leaf node triggers the non-leaf nodes in path P to be split, our method is much simple since the internal node stores the elements in increasing order. Specifically, for each internal node r_i where $i \in [0, H-2]$, it can directly split r_i into two parts with equal number of elements, *i.e.,* $\{v_{i,0}, v_{i,1}, ..., v_{i,m}\}$ and $\{v_{i,(m+1)}, ..., v_{i,n_{\mathbb{L}}-1}\}$, where m are the median position of r_i. In total, for each internal node, it takes $O(n_{\mathbb{L}})$ time cost to split since it takes $O(1)$ time cost to find the median position and $O(n_{\mathbb{L}})$ time cost to copy the elements in internal node. For illustration, a running example of insertion is given below.

Example 2. *Suppose the neighborhoods of a vertex* v_5 *is as shown in Figure 4, where it has 4 out-neighbors,* i.e., v_1, v_2, v_3 *and* v_4. *The bottom-left green box is its samtree with node capacity* 4 *before insertion. When a new neighbor* v_6 *is inserted. The algorithm firstly finds the leaf node to be inserted. Since the number of elements in this node exceeds the node capacity, the* α*-Split Algorithm is triggered to split the leaf node as two leaves:* i.e., *(1)* {*1,2*} *and (2)* {*3, 4, 6*}, *each of wich has a FSTable. Next, it creates a new non-leaf node with a new ID list and a CSTable. Specifically, in the ID list, the first element is the smallest ID in the left child and the second element is the smallest ID in the right child. In CSTable, the first element is the sum of weights in the left child,* i.e., *0.7, and the second element is the sum of weights of both left and right children,* i.e., *1.8. Besides, we can find that in the FSTable of the right child, the third element is the weight of vertex* v_6, *which is related to our FSTable construction (to be introduced later).*

In addition, we theoretically prove the time complexity of the insertion algorithm for samtree structure is $O(H \cdot n_{\mathbb{L}})$, as shown in Theorem 2. For the lack of space, the proof of this theorem could be found in Appendix D.

Theorem 2. *The average time cost of algorithm 2 is* $O(H \cdot n_{\mathbb{L}})$.

Remark. Due to our α-Split algorithm, each leaf (*resp.* non-leaf) node in a samtree contains at least $\frac{c}{2} - \alpha$ neighbor vertices

Method	Updates of Indexing Structure			Sampling
	New insertion	In-place update	Deletion	
ITS	$O(1)$	$O(n_{\mathbb{L}})$	$O(n_{\mathbb{L}})$	$O(\log n_{\mathbb{L}})$
FTS **(ours)**	$O(\log n_{\mathbb{L}})$	$O(\log n_{\mathbb{L}})$	$O(\log n_{\mathbb{L}})$	$O(\log n_{\mathbb{L}})$

TABLE II: Time complexity of FTS and ITS for dynamic graphs in terms of updates on indexing structures and sampling a vertex in a leaf node of samtree.

(*resp.* children).

D. Deletion Mechanism

In general, the deletion operation is easy. Specifically, for internal nodes r_i, if an element is deleted at position m in r_i, then the elements behind $v_{i,m}$ are moved forward by one step, and the size CSTable and FSTable are updated accordingly. For leaf node r_{H-1}, it firstly swap the element with the last element in r_{H-1} and then delete the last element directly (since the elements in leaf node are unordered). However, after deleting, if the number of elements in r_{H-1} is below the allowed threshold r_{max}, the deletion is challenging since it needs to keep all nodes in samtree balanced. To achieve this goal, we propose a deletion mechanism by merging this leaf node with its nearest sibling node. Due to its simplicity, we omit the pseudocode of the merge operation.

V. Sampling

In this section, we present our sampling method FTS with an efficiently-updated indexing structure for dynamic graphs.

Challenges. For the weighted neighbour sampling, existing deep graph learning systems are *inefficient* since they need to retrieve all the neighbours of a source node from different graph servers into memory. Besides, most of them directly adopt a *memory-expensive* sampling methods *Alias* [34], [25] which needs to construct an extra sampling table in memory to store the probability of each neighbour of a given source node. Although existing system *PlatoGL* avoids loading an extra sampling table into memory by utilizing the *Inverse Transform Sampling* (ITS) method [34], it suffers from the heavy time cost for updating the sampling probabilities of a given source, *i.e.,* the CSTable as introduced in Section II-B. Table II shows the time complexity of ITS for dynamic graphs in terms of updates on the CSTable for a leaf node in the samtree. There exists three cases for updates on the CSTable: (1) new insertion where a neighbor is newly inserted; (2) in-place update where the weight of an existing neighbor is updated; and (3) deletion where a neighbor is deleted. In particular, the time cost of a new insertion, the time cost of updating CSTable is $O(1)$ since the vertices in a leaf node in our samtree is *unordered*, making a new neighbor directly append at the last position. However, the time cost for the in-place update, *e.g.,* a vertex v is updated with w' it takes $O(n_{\mathbb{L}})$ to update the CSTable since it has to update all sampling probabilities after v in the CSTable. Similar for the deletion operation. It is very time-consuming in reality since the in-place and deletion operations happen frequently in real-world applications. To efficiently update the sampling probabilities, we proposed a *Fenwick Tree-based sampling* (FTS) method, by incorporating the Fenwick tree structure [8] with the ITS sampling method, reducing the updating time cost for each case to $O(\log n_{\mathbb{L}})$, as shown in Table II. In the following, we firstly elaborate a Fenwick tree-based Sum Table (*FSTable*) in Section V-A, and then present the FTS method by utilizing the FSTable in Section V-B.

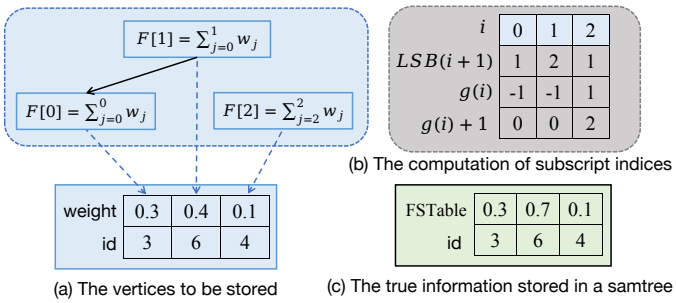


Fig. 5: A running example of FSTable computation.

Algorithm 3: FSTable Update Algorithm

Input: The current FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements, weight w, updated index i
Output: The new FSTable $\mathbb{F}'$
1: **while** $i < n_{\mathbb{L}}$ **do**
2: $\quad \mathbb{F}[i] \leftarrow \mathbb{F}[i] + w$;
3: $\quad i \leftarrow i + LSB(i+1)$;
4: **Return** $\mathbb{F}$;

A. Fenwick Tree-based Sum Table

A traditional Fenwick tree is a data structure that can efficiently calculate prefix sums in an array of values by exploiting the binary representation of indices in array. For an array A of $n_{\mathbb{L}}$ values (*i.e.,* weights), a Fenwick tree is an array $\mathbb{F}$ with $n_{\mathbb{L}}$ elements, each of which is the sum of *some values* of A with continuous indices. However, traditional Fenwick tree structure cannot be directly used in our case due to the following two limitations. (1) the *indices-inconsistency* issue. For dynamic updates where a vertex u is newly inserted (or deleted), the indices of all elements after u will be changed, leading to reconstruction of the fenwick tree, which generates a significant overhead. (2) the *prefix-sums-inconsistency* issue. Each element in a traditional Fenwick tree is calculated as the prefix sum of values in *a subset of* A with continuous indices, and thus, the traditional Fenwick tree cannot be directly used for weighted sampling. To tackle these limitations, we adapt Fenwick tree to construct a Fenwick-tree Sum Table (FSTable) by letting the elements in the leaf node as unordered (one of samtree constraints in Section IV) and designing efficiently mechanisms for insertion, deletion and in-place updates on FSTable. After that, we propose a new sampling method *FTS* based on FSTable to avoid the prefix-sum inconsistency issue. Before we go into the details of *FTS*, we firstly present how we calculate the prefix sums by adapting the traditional Fenwick tree in below.

Algorithm 4: FSTABLE INSERTION ALGORITHM

Input: The current FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements, weight w
Output: The new FSTable $\mathbb{F}'$
1: $i \leftarrow n_{\mathbb{L}}$; $s \leftarrow w$; $k \leftarrow 0$;
2: **while** $2^k < i$ **do**
3: $\quad x \leftarrow i - 2^k$;
4: $\quad$ **if** $(x+1)\& - (x+1) = 2^k$ **then**
5: $\quad\quad s \leftarrow s + \mathbb{F}[x]$;
6: $\quad k \leftarrow k + 1$;
7: $\mathbb{F}' \leftarrow$ Append s in $\mathbb{F}$;
8: **Return** $\mathbb{F}'$;

1) FSTable: Given a source vertex s and its samtree $\mathcal{T}_s$, and a leaf node in a samtree where the weight array A of $n_{\mathbb{L}}$ elements is $A = \{w_{s,1}, w_{s,2}, ..., w_{s,n_{\mathbb{L}}}\}$, our FSTable $\mathbb{F}$ is an array of $n_{\mathbb{L}}$ elements where the i-th element is as follows:

$$\mathbb{F}[i] = \sum_{j=g(i)+1}^{i} w_{s,j}, \text{ where } g(i) = i - LSB(i+1). \quad (4)$$

Notice that $LSB(i+1)$ is a function that clears the *last set bit* (LSB) of the binary representation of $(i+1)$, *i.e.,* $LSB(i) = 2^k$ where k is the number of trailing zeros in the binary representation of i. For example, When $i = 6$, $LSB(6) = LSB(110_2) = 2^1 = 2$ since the binary representation of 6 has only 1 trailing zero. From Equation (4), FSTable has a property that each element in FSTable is the sum of values in A whose indices is in a pre-defined range of $[g(i)+1, i]$. In this paper, we call this phenomenon as the *soft prefix sums*.

Besides, FSTable inherits the *tree-property* from Fenwick-tree structure, *i.e.,* elements in FSTable can be regarded as a tree where the i-th element $\mathbb{F}[i]$ is equal to the prefix sum of all its children. Hence, if the children can be found, then $\mathbb{F}[i]$ can be calculated easily. Following previous work [8], given the i-th element $\mathbb{F}[i]$, an element $\mathbb{F}[x]$ with index x that satisfies the following equations simultaneously can be considered as a child of $\mathbb{F}[i]$:

$$\begin{cases} x + 2^{k'} = i, \text{ if } 2^{k'} < i \\ (x+1)\& - (x+1) = 2^{k'}, \end{cases} \quad (5)$$

where k' is the number of trailing zeros in the binary representation of x. Thus, the computation of $\mathbb{F}[i]$ becomes as below:

$$\mathbb{F}[i] = w_i + \sum\nolimits_{x \in \mathbb{X}_i} \mathbb{F}[x], \quad (6)$$

where $\mathbb{X}_i$ is the set of children of $\mathbb{F}_i$. An runing example of our FSTable calculation is presented below.

Example 3. *Suppose there are 3 vertices in a leaf node of a samtree whose weights are as shown in Figure 5,* i.e., *the weight array is* $A = \{0.3, 0.4, 0.1\}$. *For example, when* $i = 0$, *its value is equal to* $\sum_{j=0}^{0} w_j$ *since* $g(0) = -1$ *and* $g(0)+1 = 0$. *Thus, the value is exactly* w_0, i.e., $\mathbb{F}[0] = 0.3$. *When* $i = 1$, *its value is computed as* $\sum_{j=0}^{1} w_j$, *which is the sum of the first two elements in A,* i.e., $\mathbb{F}[1]$ *=0.7. Besides,* $\mathbb{F}[0]$ *is a child of* $\mathbb{F}[1]$ *since index* 0 *satisfies Equation (5). When* $i = 2$, *its value is computed as* $\sum_{j=2}^{2} w_j$ *since* $g(2) = 1$ *and* $g(2)+1 = 2$. *So,* $\mathbb{F}[2] = w_2 = 0.1$, *and it has no child.*

2) Implementation: Next, we show how our FSTable can be updated in $O(\log n_{\mathbb{L}})$ time cost for dynamic updates in in terms of in-place update, new insertion and deletion.

In-place Update. When a vertex in a leaf node is updated with a new weight w, it is simple to update FSTable $\mathbb{F}$ by updating all the parents with w. Algorithm 3 gives the pseudocode of our update algorithm. It takes as input the current FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements, the undpated index i, and the weight w to be updated. and outputs the new FSTable after updating. It recursively updates the weight of a parent by w (Line 2). After that it gets the index of another parent as Equation(4) (Line 3). The process terminates until the index is out of range of FSTable. Obviously, the time cost of Algorithm 3 is $O(\log n_{\mathbb{L}})$ since it visits at most $\log n_{\mathbb{L}}$ parents.

New Insertion. Given a samtree of source s, a vertex v with weight w inserted into a leaf node which has $n_{\mathbb{L}}$ vertices. Recall that in our samtree structure, the elements in the leaf node is unordered so that a vertex could be appended into a leaf node. Next, we append the weight w in FSTable by summing up the value of each children in rounds. Instead of calculating all the children, we enumerate the number k' of trailing nodes in the binary representation of the indices of its children. Algorithm 4 gives the pseudocode of our insertion algorithm on FSTable. Firstly, it initializes the current index as i, the current prefix sum s as w, and the current number k of trailing zeros as 1 (Line 1). It finds the children in rounds (Lines 2-6). In each round, it computes an index x as $(i - 2^k)$ and checks whether x satisfies Equation(5) (Line 3). If yes, it increases s by $\mathbb{F}[x]$ (Lines 4-5). Then, it increases the value of k by 1 (Line 6). The computation terminates when 2^k is smaller than the current index i. Finally, it appends s to the FSTable, and return the new FSTable (Lines 7-8). Theorem 3 proves that the time complexity of Algorithm 4 is $O(\log n_{\mathbb{L}})$.

Theorem 3. *When a vertex with weight is newly insereted, the time complexity of Algorithm 4 is* $O(\log n_{\mathbb{L}})$.

Proof. Obviously, at most $O(\log n_{\mathbb{L}})$ children are computed since it checks at most $O(\log n_{\mathbb{L}})$ indices. □

Deletion. When a vertex v with index i is deleted from a leaf node, the operation is complex since its deletion has an influence on other elements in FSTable $\mathbb{F}$. To avoid updating all the elements in FSTable, our deletion mechanism is to firstly swap the element $\mathbb{F}[i]$ with the last element $\mathbb{F}[n_{\mathbb{L}}]$, then updates all the parents of $\mathbb{F}[i]$ as the in-place update algorithm does, and finally, delete the last element. Due to space limit, we omit the pseudocode of our deletion algorithm. It is easy to get that the time complexity for deletion is $O(\log n_{\mathbb{L}})$.

B. Fenwick Tree-based Sampling Method

Unlike the CSTable whose i-th element is exactly the prefix sums of all values whose index is smaller than i (*i.e., strict* prefix sums), our FSTable stores the prefix sums of a given range, *i.e., soft* prefix sums. Thus, it is challenging to use ITS method by incorporating with our FSTable since ITS requires to find the smallest index i whose prefix sum is larger than a generated random number $\mathcal{R}$.

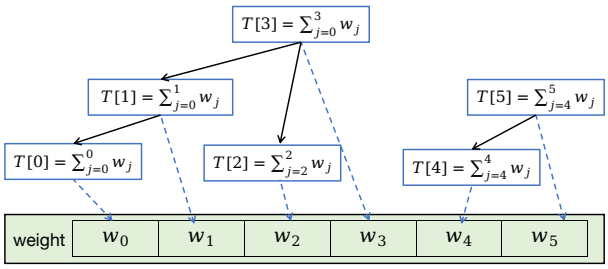


Fig. 6: An example of FSTable with 6 weights.

Algorithm 5: FTS METHOD

Input: The current FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements
Output: The index p

1: $S_{\mathbb{L}} \leftarrow$GETALLSUM($\mathbb{F}$);
2: $\mathcal{R} \leftarrow$ random number in $[0, S_{\mathbb{L}})$;
3: Find the smallest 2^m such that $2^m \geq n_{\mathbb{L}}$;
4: $left \leftarrow 0$; $right \leftarrow 2^m - 1$;
5: **while** $left < right$ **do**
6: $\quad mid \leftarrow (left + right)/2$;
7: $\quad$ **if** $mid >= n_{\mathbb{L}}$ **then**
8: $\quad\quad right \leftarrow mid$; **continue**;
9: $\quad$ **if** $\mathbb{F}[mid] > \mathcal{R}$ **then**
10: $\quad\quad right \leftarrow mid$;
11: $\quad$ **else**
12: $\quad\quad \mathcal{R} \leftarrow \mathcal{R} - \mathbb{F}[mid]$; $left \leftarrow mid + 1$;
13: **Return** $left$;

Procedure GETALLSUM
Input: The current FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements
Output: The sum $S_{\mathbb{L}}$ in this table

1: $S_{\mathbb{L}} \leftarrow 0$; $i \leftarrow n_{\mathbb{L}}$;
2: **while** $i > 0$ **do**
3: $\quad S_{\mathbb{L}} \leftarrow S_{\mathbb{L}} + \mathbb{F}[i-1]$;
4: $\quad i \leftarrow i - LSB(i)$;
5: **Return** $S_{\mathbb{L}}$;

Our FTS method is based on a *sub-tree sum* property in the FSTable that for an integer k that $k > 0$, the $(2^k - 1)$-th element is equal to the prefix sum of all the elements before $(2^k - 1)$, as shown in Theorem 4 (whose proof could be found in Appendix E). This theorem indicates a set $\mathbb{S}$ of indices i such that $\mathbb{F}[i]$ is the prefix sum of the nodes before i.

Theorem 4. *Given a FSTable $\mathbb{F}$ and an integer k that $k > 0$, $\mathbb{F}[2^k - 1] = \sum_{j=0}^{2^k-1} w_{s,j}$.*

Take Figure 6 for example where 6 elements in FSTable $\mathbb{F}$ are presented. When $k = 1$, $\mathbb{F}[2^k - 1] = \mathbb{F}[1] = \mathbb{F}[0] + w_1$. When $k = 2$, $\mathbb{F}[2^k - 1] = \mathbb{F}[3] = \mathbb{F}[1] + w_2 + w_3$. It is easy to conclude that $\mathbb{F}[3] = \sum_{j=0}^{3} w_j$.

Implemetation of FTS method. Based on this property, we propose our FTS method by iteratively comparing the values in the FSTable with the generated random number with a *range-narrow* strategy. The major idea of our range-narrow strategy is to iteratively shorten the range to be searched from two sides of FSTable until the correct index for weighted sampling is found. Specifically, for a range of $[left, right]$, it compares the median value $\mathbb{F}[mid]$ with the generated random number $\mathcal{R}$ as follows: (i) if $\mathbb{F}[i]$ is larger than $\mathcal{R}$, it means that the correct answer is in the left side of the current range, *i.e.,* $[left, mid]$; (ii) otherwise, it indicates that the correct answer is in the right side, *i.e.,* $[mid+1, right]$, and besides, it means that the prefix sum in the right side takes only $(\mathcal{R} - \mathbb{F}[mid])$ due to the sub-tree sum property.

Algorithm 5 gives the pseudocode of our FTS method. Given a FSTable $\mathbb{F}$ of $n_{\mathbb{L}}$ elements, it returns the sampled index p for weighted sampling. Firstly, it generates a random number $\mathcal{R}$ in the range $[0, S)$ where S is the sum of all weights in FSTable (Lines 1-2). Note that S can be computed by calling *getAllSum*() function in $O(\log n_{\mathbb{L}})$ time cost. Then, it find the smallest number 2^m such that $2^m \geq n_{\mathbb{L}}$, which is consistent with ITS method for weighted sampling (Line 3). Next, it finds the exact index by using the range-narrow strategy in the range of $[0, 2^m - 1]$(Line 4). In each round, it finds the median index mid (Line 6). If this index is larger than $n_{\mathbb{L}}$ (*i.e.,* out of range), then it assigns the upper bound $right$ to this index, and goes for next round (Lines 7-8). Otherwise, it compares $\mathcal{R}$ with $\mathbb{F}[mid]$ with above range-narrow strategy (Lines 9-12). The process terminates when the range has been searched.

Theorem 5. *FTS samples a vertex in a leaf node is $O(\log n_{\mathbb{L}})$.*

Proof. From algorithm, it takes $O(\log n_{\mathbb{L}})$ to compute the weights sum in the leaf node and checks at most $O(\log n_{\mathbb{L}})$ children for comparison, and thus, the proof completes. □

C. Complete Neighbor Sampling Method

In this section, we show how to sample a neighbor of a given source node in the samtree by combining the ITS method in the non-leaf nodes and the FTS method in the leaf nodes.

Firstly, we generate a random number $\mathcal{R}$ such that $\mathcal{R} \in [0, S_s)$ where S_s is the weights sum of source node s and then perform a layer-by-layer search on the samtree $\mathcal{T}_s$ based on the random number. For each non-leaf node, it adopts ITS method with the CSTable by finding the smallest index i such that $\mathbb{C}[i] > \mathcal{R}$, and then goes to the i-th child for the next sample. When at the leaf node, it adopts our FTS method with FSTable by taking $\mathcal{R}$ as input. After that, the ID of the sampled vertex is returned as the result.

VI. OPTIMIZATION

In this section, we introduce two optimization techniques for improving the efficiency of dynamic graph building in terms of both memory and time cost. Specifically, Section VI-A elaborates our compression technqiue for further reducing the memory cost, and Section VI-B presents our concurrency mechanism for improving the time efficiency.

A. Compression Technique

To further reduce the memory cost, we propose a compression technique for samtrees of all source vertices in a graph. However, existing graph compression methods like *ZipG* [21] cannot be applied to dynamic graphs since they requires pre-processing the graph data, leading to re-computation from scratch when the graph is evolved.

To address this issue, we propose a non-trivial compression method by utilizing a *dynamic prefix compression* for the

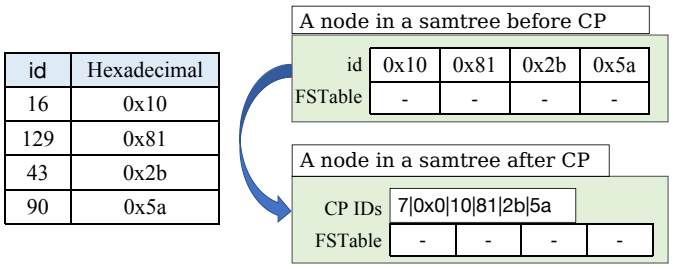


Fig. 7: An example of compression on a samtree node.

vertices in a samtree, instead of storing the complete IDs of vertices. The major idea is that, for the vertices whose first m bytes in their *hexadecimal* (*i.e.,* 64 bits) representation are the same, it stores those vertices in a string format, termed *CP-IDs*. Specifically, for a leaf node that contains $n_{\mathbb{L}}$ vertices, suppose that the first z bytes of these vertices are the same, then its CP-IDs is represented as:

$$z|prefix|suf(v_0)|suf(v_1)|...|suf(v_{n_{\mathbb{L}}}), \tag{7}$$

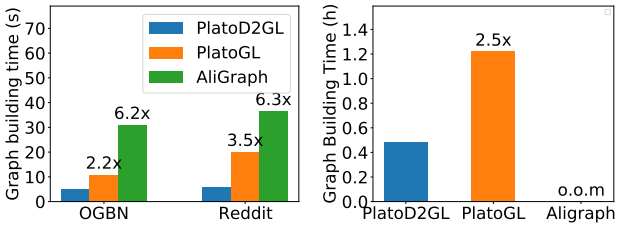
where $prefix$ is the first z bytes of their hexadecimal representation and $suf(v)$ is the suffix $(8-z)$ bytes of vertices v. Figure 7 shows an example of compression on a tree node where the first 7 bytes are used as prefix. In our settings, m is chosen from $\{0, 4, 6, 7\}$ (bytes) for fast compression. Due to the space limit, the details of updating mechanism could be found in Appendix A.

B. Concurrency Mechanism

Challenges. To handle various queries quickly, a common approach is to use thread-level parallelism which utilizes multiple threads to solve concurrent queries on samtree. An intuitive approach is to design the samtree structure as a *latch* -based tree structure, where multiple threads *asynchronously* handle various queries, and then use latches to avoid race conditions. However, this approach suffers from the following two limitations. (1) *Heavy updates.* Every time an update query is processed, the CSTables or FSTables of the nodes on the search path from the root to the leaf are updated, and so, each update query needs to latch all nodes on this path, which seriously affects performance. (2) *Expensive write operations.* Each update query generates a large number of *write* operations in memory, seriously degrading performance.
Major idea. To address the above two issues, we propose a batch-based *latch-free* concurrent method by utilizing the same idea of multiple-threads mechanism in the PALM Tree[27]. The major idea of multiple-threads mechanism is to *redistribute* the updating operations so that the updating on the same node can be processed together, avoiding huge number of idle time due to the synchronous write operations in the same node, and thus, leading to high efficiency for updating. In addition, the updating operations of a samtree are performed in a bottom-up manner, *i.e.,* from the leaf node to the root. To sum up, a samtree is modified in a *multi-threaded and latch-free* manner from bottom to top. Due to the space limit, the implementation of our concurrent machnism could be found in Appendix B.

Datasets	Relations (S-T)	#S	#T	#edges	*Density*
OGBN	Product-Product	2.4M	2.4M	61.9M	25.8
Reddit	Post-Community	233.0K	233.0K	114M	489.3
WeChat	User-Live	1.02B	1.02B	63.3B	62.06
	User-Attr	0.97B	0.97B	1.9B	1.96
	Live-Live	13.1M	13.1M	0.65B	49.62
	Live-Tag	15.1M	15.1M	30.1M	1.99

TABLE III: Dataset statistics where S (or T) stands for the source (or target) node of a relation, $\#$ denotes the set size. ($K = 10^3$, $M = 10^6$, $B = 10^9$)



(a) Public Dataset. (b) Wechat.

Fig. 8: Time cost of graph building.

VII. Experiments

In this section, we comprehensively evaluate the efficiency of our PlatoD2GL in terms of both memory and time cost.

A. Experimental settings

Evaluation platform. We evaluate PlatoD2GL and baselines over a cluster with 74 servers, among which 20 are used for training a GNN recommendation model and the rest for graph data storage. All machines are equipped with 110GB DRAM and two 2.50GHz Intel(R) Xeon(R) Platinum 8255C CPUs, each of which has 21 cores.
Datasets. To evaluate the efficiency and effectiveness of PlatoD2GL, we use one production dataset (termed *WeChat*) and three public datasets, *i.e.*, *OGBN* [2], and *Reddit* [13], all of which are heterogenous and of large-scale. Table III shows the statistics. Notice that our production dataset *WeChat* is retrieved from the live-streaming recommendation scenarios in Wechat App, which contains a large-scale heterogeneous graph with *2.1 billion nodes and 63.9 billion edges in total*. Specifically, *WeChat* dataset contains five kinds of relations (*i.e.,* edges): (i) *User-Live* that user interacted with a live room in the live-streaming service; (ii) *User-Attr* that a user has an attribute; (iii) *Live-Live* which is the relationship between two live-rooms in the live-streaming service; and (iv) *Live-Tag* that a live-room is associated as a tag. Note that all the datasets in our experiments are *bi-directed*.
Baselines. We compare our proposed system, *PlatoD2GL*, against two existing deep graph learning systems, *i.e., AliGraph* [38] and *PlatoGL* [24]. We do not compare against other existing deep graph learning systems since all of them are tailored for static graphs and are very time-consuming for dynamic graphs. To evaluate the effectivness of our compression technique, we also compared against our sytem without compression, termed *w/o CP*. For AliGraph, we use its default version for distributed graph storage, *i.e., hash-by-source*, so that it can be used for dynamic graphs.

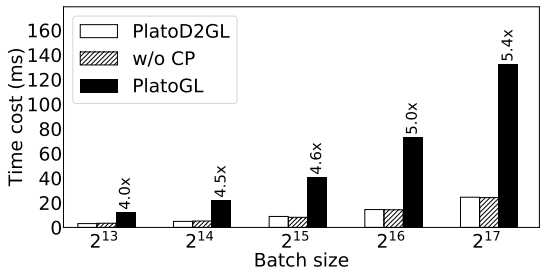


Fig. 9: Time cost of dynamic updates on WeChat dataset by varying batch size.

	OGBN	Reddit	WeChat
AliGraph	4GB	5.85GB	o.o.m
PlatoGL	4.3GB	2.2GB	4.2TB
PlatoD2GL **(ours)**	**0.81GB**	**0.73GB**	**1TB**
Improvement *(over the Second-best)*	↓ **79.8%**	↓ **66.8%**	↓ **76.2%**
w/o CP	1.13GB	1.42GB	1.22TB
Improvement *(over the w/o CP)*	↓ **28.3%**	↓ **48.6%**	↓ **18.0%**

TABLE IV: Memory cost after graph building.

Evaluation metrics. Three metrics are used to evaluate the efficiency of our PlatoD2GL system in terms of dynamic graph updating, sampling, training phase of GNN models. We reprot the average computational time cost among 1000 operations.
Default Parameters. For our system, the parameters are default as below: the node size in samtree is 256 (*i.e.,* 2^8) and the slackness α is 0. For fair comparison, the best parameters for the baselines are used.

B. Efficiency for Dynamic Graph Building

(1) Time cost of graph building. To evaluate the performance of inserting edges of a graph in a dynamic manner, we report the graph building time cost of each system on three datasets, as shown in Figure 8 where "o.o.m" means out-of-memory. The results demonstrate that our PlatoD2GL system runs faster than existing systems **by up to 6.3 times** on all datasets. Particularly, for the largest dataset WeChat, our system takes less than 0.5 hours for building, which is faster than the state-of-the-art PlatoGL **by up to 2.5 times**. It indicates the efficiency of our system for the billion-scale graphs in reality. Besides, we evaluate the time cost of dynamic updates by varying the batch size on WeChat dataset, as plotted in Figure 9. The results demonstrate that for different batch size, our proposed system is always faster than the state-of-the-art PlatoGL **by up tp 5.4 times**. Even when the batch size reaches 2^{16}, our system takes *less than 20 milliseconds* (ms) while PlatoGL requires more than 120 ms, showing the ultra-high updating efficiency of our system.
(2) Memory cost for graph storage. Table IV shows the memory cost after graph building phase where the best and second-best results of each dataset are highlighted in bold and underline, respectively. The results show that compared with the second-best, our PlatoD2GL reduces the memory cost **by up to 79.8%** on all datasets. In particular, for the largest dataset, our system reduces the memory cost from 4.2TB to 1TB, saving large number of resources in real-world. It is because our system designs a non-key-value graph storage structure *samtree* and our dynamic prefix compression technique in samtrees.

Node Capacity	64	128	256	512	1024
Leaf nodes	98.09%	99.77%	99.9%	99.95%	99.98%
Non-leaf nodes	1.91%	0.23%	0.1%	0.05%	0.02%

TABLE V: Distribution of updating operations for graph topology storage on WeChat dataset.

(3) Ablation study. To show the effectiveness of compression technique, we conducted an ablation study of dynamic insertion in terms of memory cost, as plotted in Table IV. From the results, we can see that the compression technique helps reduce the memory cost *by up to 48.6%* on all datasets.

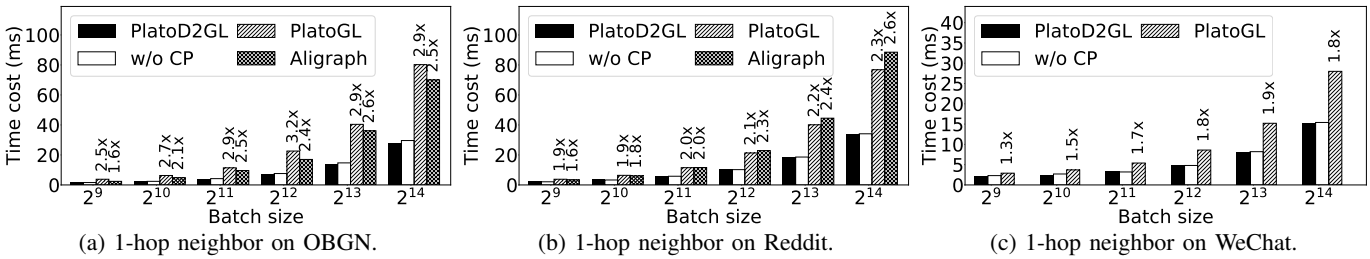
To show the effectiveness of our FSTable for dynamic updates, we reported the proportion of updating operations for graph building in terms of both leaf nodes and non-leaf nodes. WeChat dataset was used, as shown in Table V. It indicates that the dynamic updates happens most frequently on the leaf nodes, *i.e.,* larger than 98%, regardless of the node capacity of samtrees. Thus, our FSTable is more efficient to be updated than CSTable used in PlatoGL.

C. Sampling

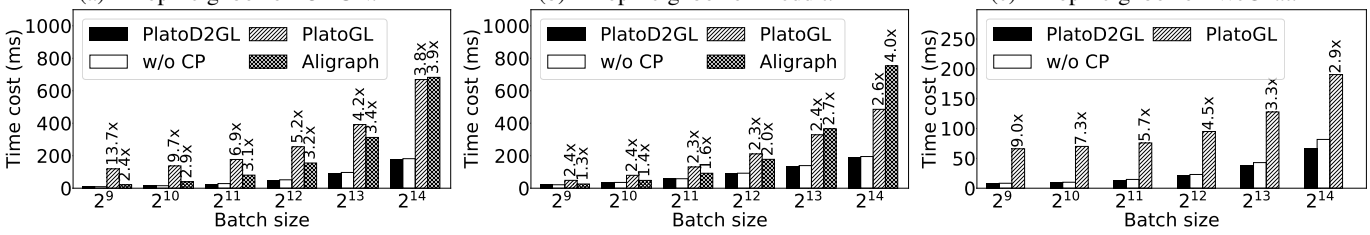
(1) Neighbor sampling. To evaluate the performance of neighbour sampling, we test the query time cost by varying the batch size on three datasets. For each node in the batch, we sampled 50 neighbours. Figure 10(a), Figure 10(b) and Figure 10(c) show the query time cost of sampling the neighbours for a source node on OBGN, Reddit and WeChat, respectively. In Figure 10(c), we do not plot the results for Aligraph since it runs out of memory. From the results, we can see that the time cost of our system increases as the batch size increases, which is consistent with theory. Besides, our system takes less time to sample neighbors than PlatoGL *by up to 2.9 times* on all datasets. Furthermore, compared with the method *w/o CP*, our system runs faster on all datasets, indicating the effectiveness of our compression techniques.
Subgraph sampling. For GNN algorithms, the multi-hops meta-paths sampling is widely-used to sample the nodes over a set of meta-paths starting from a source node. Figure 10(d), Figure 10(e) and Figure 10(f) show the query time cost of sampling 2-hop subgraphs on OBGN, Reddit and WeChat, respectively. The results show that our system achieves better performance than PlatoGL *by up to 10.1 times* on WeChat dataset. In addition, on WeChat dataset, our system still finishes sampling in less than 50 millisenconds when the batch size reaches 2^{14}, indicating the ultra-high query efficiency of our system.

D. Parameter Sensitivity

We evaluate the sensitivity of parameters in our system on the largest dataset WeChat, as shown in Figure 11. Specifically, Figure 11(a) plots the time cost of dynamic insertion by varying batch size. It shows that when the batch size increases,

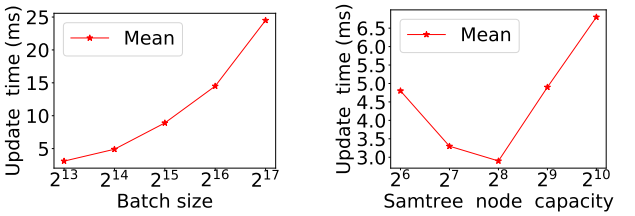


(a) 1-hop neighbor on OBGN. (b) 1-hop neighbor on Reddit. (c) 1-hop neighbor on WeChat.

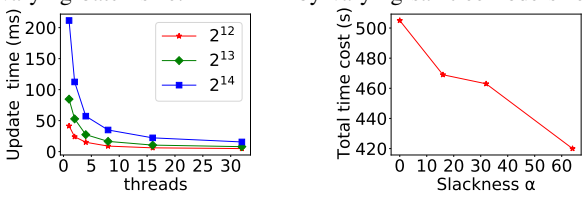


(d) 2-hop subgraph on OBGN. (e) 2-hop subgraph on Reddit. (f) 2-hop subgraph on WeChat.

Fig. 10: Time cost of sampling: varying batch size.



(a) Time cost of dynamic update by varying batch size. (b) Time cost of dynamic update by varying samtree node size.



(c) Time cost of concurrent dynamic update by varying threads. (d) Time cost of dynamic insertion by varying slackness α.

Fig. 11: Parameter sensitivity of PlatoD2GL on WeChat.

the time cost increases. Even when the batch size is larger than 2^{17}, the updating time cost is still less than 25 *ms*. Thus, our system can satisfy for various requirements for efficient graph building. Besides, Figure 11(b) plots the time cost of dynamic insertion by varying samtree node size. We can see that when the capacity is 2^8, it takes the least time cost. Figure 11(c) represents the time cost of concurrent dynamic updates by varying the number of threads. When the batch size is 2^{12}, the time cost decreases as the number of threads increases, which is consistent with theory. Similar results when the batch size is 2^{13} or 2^{14}. Figure 11(d) plots the total time cost of dynamic insertion by varying slackness α. The larger the value of α, the less time cost. It is because when α becomes larger, the α-split algorithm finds more soft position for splitting.

VIII. Other Related Work

In this section, we review existing literature about deep graph learning systems for graph neural networks. Note that in this paper, we focues on the in-memory graph learning system that can support efficient access to graph topology and feature data in distributed GNN training. There are several deep graph learning systems tailored for GNN-based recommendation, *e.g.*, *AliGraph* [38], *Euler* [17], *Plato* [18], *DistDGL* [37], *ByteGNN* [36], and *PlatoGL* [24] where PlatoGL is the state-of-the-art for dynamic deep graph learning system. AliGraph is an integrated platform for GNN training. For the graph storage, it integrates several graph partition methods, *e.g.,* hash by source (as PlatoGL), METIS [20] and Vertex Cut [11], in the system. However, it takes expensive memory cost for graph storage since it has to duplicate the graph topology for supporting fast sampling. Euler, Plato, DistDGL and ByteGNN also adopt different graph partition methods for storing each graph partition into a phisical machine. For Plato and ByteGNN, both of them uses a parallel mechanism for improving the efficiency of sampling neighbors, which cannot helps reduce the time cost for updating the sampling probabilities. However, all of them fail to support the dynamic graphs in reality due to *either* (1) the inefficient dynamic updates on graph data *or* (2) the expensive memory overhead, and thus, cannot support the training of GNN models in reality. Comparatively, our PlatoD2GL system can avoid expensive overhead in terms of both memory and time cost by utilizing a non-key-value storage *samtree* and a Fenwick-tree-based sampling method.

IX. Conclusions

In this paper, we design an efficient dynamic deep graph learning system *PlatoD2GL* by exploiting a tree-like structure *samtree* for storage and a Fenwick-tree-based sampling method, largely reducing the overhead in terms of both memory and time cost for dynamic updates. With sophisticated designs, comprehensive experiments on benchmark performance demonstrate its efficiency for training of graph neural networks on billion-scale graphs. Currenly, PlatoD2GL serves the major online traffic in WeChat for various recommendation scenarios.

REFERENCES

[1] Analysis of quickselect. https://www.cs.auckland.ac.nz/courses/compsci220s1c/lectures/2016S1C/CS220-Lecture12.pdf, 2016.

[2] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. The extreme classification repository: Multi-label datasets and code, 2016.

[3] H. Cai, V. W Zheng, and K. C.-C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *TKDE*, 30(9):1616–1637, 2018.

[4] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, and Mohamed F Mokbel. Streamrec: a real-time recommender system. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1243–1246, 2011.

[5] Dawei Cheng, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. Graph neural network for fraud detection via spatial-temporal attention. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3800–3813, 2020.

[6] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM international conference on information & knowledge management*, pages 315–324, 2020.

[7] Bin Fan, David G Andersen, and Michael Kaminsky. {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[8] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.

[9] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[10] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, pages 1263–1272, 2017.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[12] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[13] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, pages 1025–1035, 2017.

[14] W. L Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *arXiv.*, 2017.

[15] Charles AR Hoare. Quicksort. *The computer journal*, 5(1):10–16, 1962.

[16] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. Tencentrec: Real-time stream recommendation in practice. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 227–238, 2015.

[17] Alibaba Inc. Euler framework for deep graph learning. https://github.com/alibaba/euler, 2020.

[18] Tencent Inc. Platograph framework for graph algorithms. https://github.com/Tencent/plato, 2019.

[19] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[20] George Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

[21] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1149–1164, 2017.

[22] Ao Li, Zhou Qin, Runshi Liu, Yiqun Yang, and Dong Li. Spam review detection with graph convolutional networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2703–2711, 2019.

[23] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[24] Dandan Lin, Shijie Sun, Jingtao Ding, Xuehan Ke, Hao Gu, Xing Huang, Chonggang Song, Xuri Zhang, Lingling Yi, Jie Wen, et al. Platogl: Effective and scalable deep graph learning system for graph-enhanced real-time recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3302–3311, 2022.

[25] W. Lin. Distributed algorithms for fully personalized pagerank on large graphs. In *WWW*, pages 1084–1094, 2019.

[26] Mohammad Hashem Ryalat. A new algorithm to find the k th smallest element in an unordered list (efficient for big data). In *2022 2nd International Conference on Computing and Information Technology (ICCIT)*, pages 51–56. IEEE, 2022.

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

[28] Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. Graph convolutional networks for computational drug development and discovery. *Briefings in bioinformatics*, 21(3):919–935, 2020.

[29] Mubeena Syeda, Yan-Qing Zhang, and Yi Pan. Parallel granular neural networks for fast credit card fraud detection. In *2002 IEEE World Congress on Computational Intelligence. 2002 IEEE International Conference on Fuzzy Systems. FUZZ-IEEE'02. Proceedings (Cat. No. 02CH37291)*, volume 1, pages 572–577. IEEE, 2002.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[31] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. Kgat: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 950–958, 2019.

[32] S. Wu, F. Sun, W. Zhang, and B. Cui. Graph neural networks in recommender systems: a survey. *arXiv preprint*, 2020.

[33] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2020.

[34] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. Knightking: a fast distributed graph random walk engine. In *SOSP*, pages 524–537, 2019.

[35] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7370–7377, 2019.

[36] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15(6):1228–1242, 2022.

[37] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[38] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

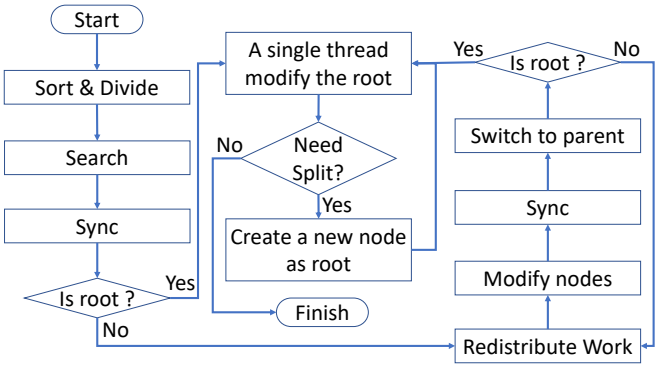


Fig. 12: The flow of multi-threaded processing on samtrees.

APPENDIX

A. Implementation of Compression Technique

Updates. When a vertex u is inserted, it firstly compares the first m bits of u with the prefix of existing CP-IDs. If the prefix of u does not exist in the existing CP-IDs, it means that node u does not exist and we directly insert u in this tree node by using the same string format. Similar for the deletion operation.

B. Implementation of Concurrent Mechanism

Implementation. Figure 12 plots the multi-threads processing flow of samtree structures. To be specific, it firstly sorts the queries according to the IDs of vertices and then evenly divides them to threads. Next, each thread searches the path in samtree from root to the leaf *w.r.t* the query, and records the path separately and synchronously. After that, if all threads finish the search task, the updating performs *in rounds*. In each rounds, if the current node is not the root of samtree, it redistributes the updating tasks to multiple threads, each of which performs the updating task in a specific leaf node. For each single thread, they starts the modification and then retrieves the updates that should be performed by its parent node. Then, a round finishes. The recursive updating operation terminates until the root is reached. When a root is reached, a single thread will perform the modification on the root, and then checks whether the root needs to be split. If yes, the α-split algorithm will be triggered. Otherwise, the whole process terminates and the updatings on this sametree are performed successfully.

C. Proof of Theorem 1

Proof. When $\alpha = 0$, our algorithm is exactly the Quick-Select algorithm whose time complexity in average-case is $O(n_{\mathbb{L}})$ [26], [1]. When $\alpha > 0$, the time cost is smaller than that when $alpha = 0$ since our algorithm does not maintain a strict increasing order. Thus, the average time cost of α-split algorithm is $O(n_{\mathbb{L}})$. The proof completes. □

D. Proof of Theorem 2

Proof. It takes $O(H)$ time to find the leaf node to be inserted. From above, we can know that the average time cost for splitting a leaf node or a non-leaf node is $O(n_{\mathbb{L}})$, thus, in the worst case where all the nodes in path P are split, the time cost is $O(H \cdot n_{\mathbb{L}})$. Besides, from Theorem 3 in Section V-A, it takes $O(n_{\mathbb{L}})$ time cost for updating the CSTable or FSTable at each level of a samtree. Hence, it takes $O(H \cdot n_{\mathbb{L}})$ time to update the CSTable and FSTable from leaf node to the root node. In total, the average time cost is $O(H \cdot n_{\mathbb{L}})$. □

E. Proof of Theorem 4

Proof. Since the equation $LSB(2^k) = 2^k$ holds for each integer k, we can derive the following equations.

$$\begin{aligned}\mathbb{F}[2^k - 1] &= \sum\nolimits_{j=g(2^k-1)+1}^{2^k-1} w_{s,j} = \sum\nolimits_{j=2^k-1-LSB(2^k)+1}^{2^k-1} w_{s,j} \\ &= \sum\nolimits_{j=2^k-1-2^k+1}^{2^k-1} w_{s,j} = \sum\nolimits_{j=0}^{2^k-1} w_{s,j}.\end{aligned}$$

The proof completes. □