# PlatoGL: Effective and Scalable Deep Graph Learning System for Graph-enhanced Real-Time Recommendation

Dandan Lin
Shijie Sun
danieslin@tencent.com
cedricsun@tencent.com
WeChat, Tencent Inc.
Shenzhen, China

Jingtao Ding*
Department of Electronic Engineering, Tsinghua University
Beijing, China
dingjt15@tsinghua.org.cn

Xuehan Ke
Hao Gu
elviske@tencent.com
nickgu@tencent.com
WeChat, Tencent Inc.
Shenzhen, China

Xing Huang
Chonggang Song
healyhuang@tencent.com
jerrycgsong@tencent.com
WeChat, Tencent Inc.
Shenzhen, China

Xuri Zhang
Lingling Yi
ninjazhang@tencent.com
chrisyi@tencent.com
WeChat, Tencent Inc.
Shenzhen, China

Jie Wen
Chuan Chen
welkinwen@tencent.com
chuanchen@tencent.com
WeChat, Tencent Inc.
Shenzhen, China

## ABSTRACT

Recently, graph neural network (GNN) approaches have received huge interests in recommendation tasks due to their ability of learning more effective user and item representations. However, existing GNN-based recommendation models cannot support *real-time* recommendation where the model keeps its freshness by continuously training the streaming data that users produced, leading to negative impact on recommendation performance. To fully support graph-enhanced large-scale recommendation in real-time scenarios, a deep graph learning system is required to dynamically store the streaming data as a graph structure and enable the development of any GNN model incorporated with the capabilities of real-time training and online inference. However, such requirements rule out existing deep graph learning solutions. In this paper, we propose a new deep graph learning system called *PlatoGL*, where (1) an effective block-based graph storage is designed with non-trivial insertion/deletion mechanism for updating the graph topology in-milliseconds, (2) a non-trivial multi-blocks neighbour sampling method is proposed for efficient graph query, and (3) a cache technique is exploited to improve the storage stability. We have deployed PlatoGL in Wechat, and leveraged its capability in various content recommendation scenarios including live-streaming, article and micro-video. Comprehensive experiments on both deployment performance and benchmark performance (*w.r.t.* its key features) demonstrate its effectiveness and scalability. One real-time GNN-based model, developed with PlatoGL, now serves the major online traffic in WeChat live-streaming recommendation scenario.

*Jingtao Ding is the corresponding author. Work done while at Tencent.


*CIKM '22, October 17–21, 2022, Atlanta, GA, USA.*

ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557084

## CCS CONCEPTS

• **Information systems** → **Recommender systems**; *Network data models*; *Collaborative filtering*; *Personalization.*

## KEYWORDS

Graph Neural Network; Real-time Recommendation; Deep Graph Learning System

**ACM Reference Format:**
Dandan Lin, Shijie Sun, Jingtao Ding, Xuehan Ke, Hao Gu, Xing Huang, Chonggang Song, Xuri Zhang, Lingling Yi, Jie Wen, and Chuan Chen. 2022. PlatoGL: Effective and Scalable Deep Graph Learning System for Graph-enhanced Real-Time Recommendation. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511808.3557084

## 1 INTRODUCTION

As one of the most successful and well-known applications of machine learning, personalized recommendation system has become an indispensable part in wide areas of user-oriented web services, including e-commerce [29, 38], video streaming [5, 21, 39], live streaming [25], news delivery [24, 41], etc. By accurately modeling user preferences from their historical interactions (*e.g.*, click, watch) and other side information (*e.g.*, user-user social relation and item-attribute), the recommendation system can help users finding their interested items from massive amount of candidates, which greatly alleviates the so-called *information overload* problem. To achieve this goal, it is essential for a recommendation model to learn effective user/item representations from collected interactions and rich side information. Since both user-item interactions and other side information have graph structure, graph neural network (GNN) algorithms [12, 28, 32] have been considered as promising solutions for user/item representation learning in recommendation systems [9, 14, 30, 31]. In industry practices, GNN based recommendation systems have been successfully deployed in a wide variety of scenarios, *e.g.*, product recommendation in e-commerce [7, 16] and content recommendation in social media [37, 43].

Although GNN-based recommendation models currently show their effective representation learning, their recommendation power is still underestimated since existing deployed GNN models are *static* by fixing the graph topology and cannot support *real-time recommendation*. Note that supporting real-time recommendation is a crucial capability for content recommendation scenarios where user interest is highly dynamic and non-stationary [3, 15, 33]. For example, a micro-video user may first watch sports related feeds that are out of her long-term interest and then quickly switch to a trending news feed due to her rising attention on current affairs. Above issue, known as *concept drift* [8], calls for a power recommendation model that can learn user preference in real-time. Specifically, it should fulfill the following two tasks: (1) continuously training with the newly coming data (termed the *streaming* data) to capture users' instant interests and demands, termed the *real-time training phase*, and (2) quickly inferring a user's up-to-the-minute preferences *in-millisecond*, termed the *online inference phase*. Currently, real-time DNN-based recommendation systems have been widely adopted in industrial content recommendation scenarios (including Youtube [36], Kuaishou [33] and etc.) to improve user engagement and platform profit. Thus, it is equally important for GNN-based recommendation models to capture user interests in real-time.

Unlike traditional DNN models, the development of GNN models requires a deep graph learning system that provides the distributed graph storage for large-scale graphs and computation capabilities to facilitate GNN training [40, 44]. However, existing deep graph learning solutions [17, 18, 22, 40, 44] cannot incorporate GNN-based recommendation models with the capabilities of real-time training and online inference due to the following challenges:

- **How to enable the deployment of a constantly-training GNN model?** Unlike traditional GNN model training that keeps the graph data unchanged, the real-time training phase of a GNN model requires a dynamically-updating graph whose topology is continuously-updated immediately users' latest interactions (*i.e.*, interests) are captured. However, the deployment of such models is non-trivial. Firstly, to guarantee freshness of graph information, a graph learning system need to support constant and high-frequency graph updates in real-time where the updates should be finished in milliseconds for large-scale graphs with billion nodes. Secondly, as each model is constantly updated and their memory cost largely depends on graph scale, it requires a maintenance mechanism to avoid unlimited memory increasing in the circumstance of multi-model simultaneous training.
- **How to meet the stringent requirements on supporting GNN model inference on the fly?** Compared with the real-time training, the online inference phase of a GNN model is far more challenging to be implemented for a graph learning system in terms of *query efficiency* and *storage stability*. Firstly, the model online serving has a stringent latency requirement so as to provide the recommendation responses to users in milliseconds. To achieve the in-milliseconds online inference, a deep graph learning system should provide the capability of efficient graph query, *i.e.*, efficiently retrieving neighborhood of a specific node. It is because each GNN model involves a necessary operation, *i.e.*, *message passing* which aggregates the messages from neighbours of a source node to get the representation of this source node [12]. Secondly, the stability of the graph storage has a significant impact on the online serving quality of a model. Specifically, if the graph storage is crashed and not available at some time point, the online inference of a GNN model must be delayed, leading to a poor user experience. To bypass above challenges, an easy solution for GNN-based recommendation models is to complete the *message passing* operations *offline* and use the static user/item representations for online prediction [16, 37], making the recommendation results insensitive to user real-time preferences.

To solve these challenges, we consider **four key requirements** that a real-time graph learning system should meet: *(1) In-milliseconds Dynamically-updating.* The dynamic updating of graph should satisfy a latency of milliseconds in a billion-scale graph. *(2) Low Memory Consumption.* From an economical cost perspective, one has to keep a low memory consumption for the simultaneous training of multiple GNN models. *(3) Ultra-high Query Efficiency.* As a key procedure in the real-time training and online inference phases, graph query should be of high efficiency, like finding the neighbourhood of a node. *(4) High Storage Stability.* The graph storage should be stable with high tolerance to guarantee online recommendation service. However, all of existing deep graph learning systems fail to satisfy above four key requirements simultaneously (more discussions could be found in Section 2.2). In this paper, we propose the *first* industrial deep graph learning system *PlatoGL* that satisfies above four key requirements simultaneously, setting the stage for a large-scale GNN-based real-time recommendation system. PlatoGL has been deployed in Wechat[1], the largest social networking service in China, and supports various content recommendation scenarios. The following shows our **contributions**.

- We propose a new storage layer called *MKVGraph* inside PlatoGL to store multiple GNN-related data. Specifically, in *MKVGraph*, we design an effective *block-based neighborhood storage* with non-trivial insertion and deletion mechanisms to dynamically update the graph topology data in a latency of milliseconds.
- We design a novel and non-trivial *multi-block neighbour sampling* method with indexing structures to efficiently sample the neighborhood of nodes in the graph, satisfying the ultra-high query efficiency requirement.
- We apply a cache strategy to reduce the number of concurrent graph queries to the graph storage, which keeps the online graph storage safe under huge number of queries per second, and thus, improves the storage stability.
- We have developed a *real-time* GNN recommendation model with our PlatoGL system in WeChat live-streaming recommendation scenario and conducted online A/B test to show its superiority over a *static* GNN model without real-time capability. Now, the *real-time* GNN model serves in the major online traffic in WeChat.
- Comprehensive experiments showed that PlatoGL satisfies above four key requirements.

## 2 BACKGROUND AND EXISTING SOLUTIONS

In this section, we firstly introduce background of the problem studied in this paper, and next present existing large-scale deep graph learning frameworks along with their limitations.

[1] https://www.wechat.com/en

## 2.1 Background

We start with a simple directed weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{W})$ where $\mathcal{V}$ and $\mathcal{E}$ represent the set of nodes and edges,respectively; and $\mathcal{W}$: $\mathcal{E} \rightarrow \mathbb{R}^+$ is a function that assigns a weight $w_{uv}$ to an edge $e_{uv}$ linking from node $u$ to node $v$. For each node $u$, let $\mathcal{N}_u$ denote the set of $u$'s neighbours. In this paper, we consider the *heterogeneous* graph with multiple types of nodes and/or edges, which is common in real-world recommendation scenarios. Besides, to support the real-time recommendation, the graphs evolve with time, which is formally defined as follows. 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 $t$.

**General recommendation problem.** A common paradigm for general recommendation models is to reconstruct users' historical interactions by the representations of users and items. Formally, we formulate the problem as follows:

Definition 1 (Recommendation Problem). *Given a heterogeneous graph $\mathcal{G}$ and the user-item interaction matrix $\mathcal{Y} = \{y_{ui} : u \in \mathcal{V}_U, i \in \mathcal{V}_I, e_{ui} \in \mathcal{E}\}$ where $\mathcal{V}_U$ and $\mathcal{V}_I$ are the node sets of users and items, respectively. The recommendation problem aims to predict the **unknown rating** between user $u$ and item $i$.*

**GNN-based recommendation method.** GNN approaches could be basically formulated as *message passing* [10, 12, 28, 31, 40], where nodes in the graph propagate their messages to other neighbours and compute their own representations (*i.e.*, in form of embedding) by aggregating the received messages from their neighbours as well. Given a heterogeneous graph $\mathcal{G}$, we denote the feature vector of user node $u$ as $\mathbf{h}_u^{(0)}$ whose value is usually assigned as its attribute vector $\mathbf{f}_u$. To get the representation of node $u$ at layer $l$, a GNN approach performs the computations as follows:

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

where $f(\cdot)$, $\bigoplus(\cdot)$, and $g(\cdot)$ are customized functions (*i.e.*, neural network modules) for *calculating* messages from each neighbour of node $u$, *aggregating* the messages of all neighbours of node $u$, and *updating* the representation of node $u$, respectively. In the literature [2, 11–13, 44], as aggregating all neighbours of node $u$ is infeasible in 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. Similarly, the representation of item node $i$ is obtained as follows:

$$\mathbf{h}_i^{(l+1)} = g(\mathbf{h}_i^{(l)}, \bigoplus_{u \in \mathcal{N}_i} f(\mathbf{h}_i^{(l)}, \mathbf{h}_u^{(l)})). \tag{2}$$

Generally, a GNN-based recommendation model denoted as $\mathcal{M}$ can be divided into three parts: $L$-layered user representations $\{\mathbf{h}_u^{(l)}\}$, $L$-layered item representations $\{\mathbf{h}_i^{(l)}\}$ and scoring function $\mathcal{F}$ that takes $\{\mathbf{h}_u^{(l)}\}$ and $\{\mathbf{h}_i^{(l)}\}$ as inputs to compute the $u$-$i$ preference score $\hat{y}_{ui}$ by a customized neural network module.

**GNN-based recommendation in real-time scenarios.** In real-time recommendation 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 up-to-the-minute user preference from $\mathcal{G}^{(t)}$ that receives constant updates from streaming data. That is, at the timestamp $t$, they exploit the topological information in the graph $\mathcal{G}^{(t)}$ for the sampling, aggregating and updating operations.

## 2.2 Existing Deep Graph Learning Frameworks

In industries, there are several deep graph learning systems tailored for GNN-based recommendation, *e.g.*, AliGraph [44], Euler [17], Plato [18], and DistDGL [40]. However, all of them fail to satisfy the four key requirements simultaneously, and thus cannot support the real-time recommendation scenarios. Firstly, all these systems directly store the graph in a cluster of physical machines (termed *graph servers*) by using the graph partition methods (like METIS [19]). However, such graph storage cannot support the in-milliseconds dynamic updates 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. To our best knowledge, no efficient solution for dynamic graph partition is involved in these systems. Secondly, most solutions, like AliGraph and Plato, build independent graph storage for different recommendation models even the models exploit the same graph data. It takes a huge memory cost since there are possibly over hundreds of models running in a specific industrial scenario. Thirdly, for neighbourhood sampling, both AliGraph and Plato need to read all neighbours of a node from different graph serves into memory for computations, leading to a fairly high time cost for queries, and they also use a memory-expensive sampling method which brings a heavy burden of the resource usage. Finally, for these systems, the graph storage is far away from the high stability since they cannot leverage the light-weight storage replication mechanism to keep a safe storage under extreme scenarios. Comparatively, our PlatoGL system can avoid those issues with novel and non-trivial designs in terms of both graph storage and neighbour sampling.

# 3 SYSTEM OVERVIEW

In this section, we first introduce the architecture of PlatoGL and then present how to build a real-time graph-enhanced recommendation system with PlatoGL.

## 3.1 Architecture of PlatoGL

Based on the GNN general architecture described in Section 2, we construct the system architecture of the *PlatoGL* platform, as shown in Figure 1. On the whole, it consists of two layers: (i) *the graph storage layer* called *MKVGraph*, which stores graph topology, attributes information of nodes or edges, indexing structures for fast samplings and a cache for caching frequently-used attributes or edges. Note that our storage layer is based on online cloud storage system [42] and decoupled with concrete GNN models, and so, multiple simultaneously trained GNN models using the same training data can interacts with a single graph storage, avoiding the redundant graph storage. (ii) *the TF-based operators layer*, which designs several basic operators of GNN algorithms inside the general Tensorflow (TF) computation framework, namely *Sampling*, *Aggregating* and *Updating*.

## 3.2 Real-time Recommendation with PlatoGL

In this part, we demonstrate how a real-time GNN-based recommendation system with PlatoGL work in terms of real-time training

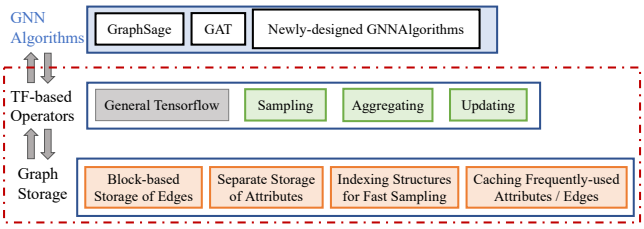



**Figure 1: Overview of PlaotGL (the red dashed rectangle).**

and online reference, with a workflow shown in Figure 2. At the very bottom in Figure 2 is the logging service that captures users behaviours (*e.g.*, clicking a list of items) in the recommendation scenarios. Above logging, the green part is our PlatoGL system where the graph storage MKVGraph works round the clock, *i.e.,* (1) dynamically updating the graph topology with the real-time streaming data, and (2) efficiently answering the graph queries from GNN models. Thus, due to such two abilities of PlatoGL system, the real-time training and online inference become feasible.

**Real-time Training.** The top-left yellow part in Figure 2 shows the workflow of real-time training process with the real-time data stream. The streaming data provides labeled instances (*i.e.*, $\langle user, item \rangle$ pairs) for training a real-time GNN model (along with the corresponding features of users/items). Specifically, it performs the following three steps subsequently: (i) calling the TF-based operators to sample neighbourhood of the users (or items) that model requires; (ii) calling the TF-based operators to aggregate and update the user (or item) embeddings; and (iii) starting the loss computations and model parameter updating process. In the meantime, the constantly-updating model is deployed into services with very low latency by exploiting *Ekko* [27].

**Online Inference.** the top-right yellow part in Figure 2 shows the online inference process. Whenever a user request is posted, the recommendation system firstly retrieves corresponding features of the user and candidate items, and then sends ids of the user and items along with their features to the deployed GNN recommendation model for prediction. Note that online prediction of GNN model would again involve the graph queries for retrieving the neighbourhood or attributes. Since a recommendation model could be used in different stages, *i.e.*, *recall* stage or *ranking* stage, the prediction output can be either an embedding vector that can be used for nearest-neighbor-search in *recall*, or a scalar score that can be directly used for *ranking* candidate items.

## 4 DETAILS OF PLATOGL SYSTEM

In this section, we elaborate the design of our PlatoGL system. Specifically, Section 4.1 elaborates how PlatoGL stores the graph-relevant data. Section 4.2 introduces how PlatoGL efficiently samples the data from the graph storage. Section 4.3 presents a caching technique in PlatoGL. All these techniques make PlatoGL system satisfy the four requirements for the real-time recommendation tasks simultaneously.

### 4.1 Graph Storage: MKVGraph

*4.1.1 Overview of MKVGraph.* Existing deep graph learning systems adopt the graph partition methods to split the input graph to multiple partitions with a minimal number of edges across partitions. However, it is difficult to do the dynamic graph update in a latency of milliseconds since they need the graph partition from scratch. To meet the in-milliseconds dynamically-updating requirement, we propose a straightforward but effective storage *MKVGraph* which abandons the graph partition method for distributing billion-scale graphs in different servers. To be specific, *MKVGraph* stores the graph topology in the format of *key-value pairs* based on a large-scale in-use cloud-based online-data storage system (termed *PaxosStore*) in WeChat [42]. Note, the *key-value pairs* stores the data as a tuple of ⟨ key, value ⟩ where key is globally unique and value is the result stored in the key. Such design possesses three advantages: (i) PaxosStore system helps to put the key-value pair data to different graph servers without considering the graph partition; (ii) PaxosStore is equipped with a *light-weight storage replication mechanism* for improving the storage stability; and (iii) PaxosStore is cloud-based so that all the graph data be available online. Besides, in *MKVGraph*, we also design a new *block-based storage* as well as non-trivial methods for how to efficiently insert/delete the data into/from the storage to guarantee efficient dynamic graph updating (see Section 4.1.2). Finally, *MKVGraph* provides different storage for different types of information to reduce the space cost (see Section 4.1.3).

*4.1.2 Topology Storage.* MKVGraph stores the graph topology in the form of edges where a dangling node without neighbour can be regarded as a special case. In literature, there are two solutions to store edges as key-value pairs: either (1) one key-value pair stores one edge (*i.e.*, one neighbour of a source node), or (2) one key-value pair store all neighbours of a source node. However, both solutions have severe limitations. The first one suffers from huge memory cost while the second one suffers from the *query perspiration* and *insert amplification* issues when a hub node (with a large amount of neighbours) is queried. Because it needs to visit all neighbours of a node even if some neighbours are not concerned. To tackle these issues, we design a *block-based* key-value storage. Its basic idea is to partition the neighbours of node $s$ into *multiple blocks of fixed-size* where each block contains a subset of neighbours of node $s$. Hence, for hub nodes, its neighbours are in multiple blocks while for small nodes (with a few neighbours) in a single block. Although this method seems to be straightforward, how to efficiently process the dynamic updates is non-trivial since we need to balance the number of neighbours in each block (to be introduced later).

Figure 3 shows the block-based storage. The key is a tuple with multiple-bytes. Specifically, for any node $s \in \mathcal{V}$ and its neighbours set $\mathcal{N}_s$, the key is $\mathcal{K} = \langle s, KType, f_E, b \rangle$ where $s$ is the unique identifier (ID) of source node, $KType$ is the key type, $f_E$ is the edge type in heterogeneous graphs, and $b$ is the ID of block. For easy reference, $b$ is assigned by the largest ID of neighbour in the block. Note that for edge storage, the key type is *'edge'* for distinguishing from the attribute storage or indexing storage. Besides, the size of a block is application-specified. In our implementation for WeChat recommendation services, we store at most 256 neighbours in one block. The corresponding value is a block which contains two parts: *header* which stores the overal information in this block, and *neighbour units* which store each neighbour. To be specific, the header stores the information of this block and it is a tuple with

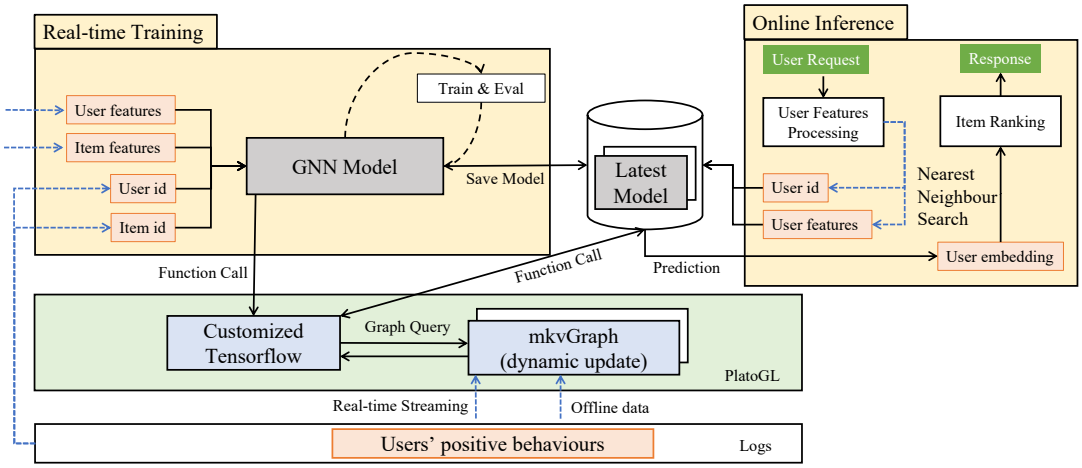



**Figure 2: Workflow of a PlatoGL-aided real-time recommendation system, where "Online Inference" shows the *recall* stage.**

multiple-bytes, namely, $\langle C_{unit}, C_w, p_b \rangle$ where $C_{unit}$ is the number of neighbour units in this block, $C_w$ is the sum of weight held by each neighbour in this block, and $p_b$ is a probabilistic value to sample this block (to be introduced in Section 4.2). For each neighbour unit in the block, its form is designed as a tuple $\langle t, p_t \rangle$ where $t$ is a neighbour $t \in \mathcal{N}_s$ of source node $s$ and $p_t$ is a probabilistic value for sampling node $t$ (to be elaborated in Section 4.2). For a dangling node without neighbour, then its corresponding value is a block containing the header only.

**Insert operation.** When a block to be inserted is not full, the insertion is easy. However, the case becomes complicated when the block is full since it requires to balance the trade-off between insertion efficiency and graph query efficiency. If we directly add a new block and insert the new neighbour there, the number of neighbours in each block might be extremely imbalance where some blocks are full while others have only few. Besides, we also consider not to increase unlimitedly the number of blocks of a source node. To tackle above issues, we propose an insertion mechanism by splitting the blocks of a source node in a balanced format. Its basic idea is that each block has the expected number of neighbours after splitting, which avoids multiple block-splitting operations in the future. Algorithm 1 describes the pseudocode of this mechanism where *high* and *low* denotes the largest and smallest number of neighbours allowed in the block, respectively. Each newly-produced block contains only $\frac{high+low}{2}$ neighbours except the last block whose number of neighbours might be smaller than $\frac{high+low}{2}$.

**Delete operation.** In general, the delete is also easy. However, when the number of neighbours in a block reaches the allowed smallest value, the deletion is challenging since it needs to keep all blocks balanced. To achieve this goal, we propose a deletion mechanism by merging the current block with one of the nearest blocks which contains fewer neighbours. Note that a block has at most two nearest blocks. Algorithm 2 describes the pseudocode. After merging, if the size of a new block exceeds the largest value, then a splitting operation on this block will be performed.

*4.1.3 Separate Attribute Storage.* A heterogeneous graph in real-world scenarios usually carries the attribute information on both

**Algorithm 1:** Insertion-Split

**Input:** *high*, *low*, current block $B$, the function $Cnt()$ to compute the number of neighbours in a block
**Output:** A list of split blocks *res*;

$N \leftarrow (high + low)/2$; // the expected number of neighbours in a block
$i \leftarrow 0$; // the index of block
$res \leftarrow \emptyset$; // a vector of split blocks
$B_{split} \leftarrow B$; // the block after removing neighbours.
**while** $(Cnt(B_{split}) - i \cdot N) > high$ **do**
  $begin \leftarrow i \cdot N$; $end \leftarrow (i+1) \cdot N$;
  $res[i] \leftarrow$ the neighbours in $B$ with index from *begin* to *end*;
  $i \leftarrow i + 1$;
  Update $B_{split}$ by removing the neighbours in $res[i]$;
**if** $Cnt(B_{split}) > i \cdot N$ **then**
  $res[i] \leftarrow$ the remaining neighbours in $B$;
**return** *res*;

**Algorithm 2:** Deletion-Merge

**Input:** *high*, current block $B$, the function $Cnt()$ to compute the number of neighbours in a block, the function $GetSmallerNeiblock()$ to find the block which is nearest to $B$ and contains fewer neighbours
**Output:** The new block $B_{new}$;

$B_{old} \leftarrow$ GetSmallerNeiblock($B$);
$B_{new} \leftarrow$ the neighbours in both $B_{old}$ and $B$;
**if** $Cnt(B_{new}) > high$ **then**
  **return** Insertion-Split($B_{new}$);
**return** $B_{new}$;

nodes and edges. Since the attributes on different nodes/edges always overlap, we store the attribute information separately from the graph topology. The attributes of nodes/edges in GNN algorithms can be classified into two types: (1) the sparse features, *i.e.*, the corresponding vectors have only one non-zero value; (2) the dense features, *i.e.*, their vectors contain many non-zero values. However, if the sparse features are stored in the same format as the dense features, the resources must be wasted largely. Thus, we design different storage format for the sparse and dense features. Figure 4a shows the storage of sparse attribute for a node $s$, which

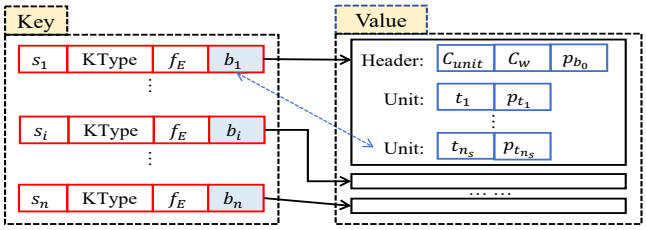
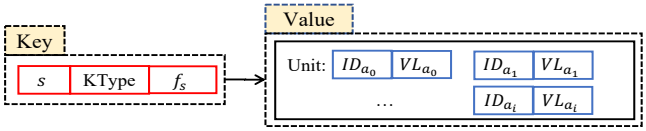
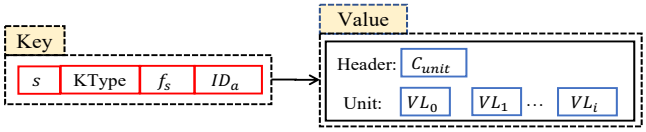



**Figure 3: The edges storage for each node $s_i$ in the graph where $i \in [1, n]$ and $n_s$ is the number of neighbours of node $s$.**




**(a) The storage of sparse attributes for a node $s$ where $ID_{a_i}$ and and $VL_a$ are the identifier and the non-zero value of attribute $a$.**




**(b) The storage of dense attributes for a node $s$ where $C_{unit}$ is the number of non-zero values.**

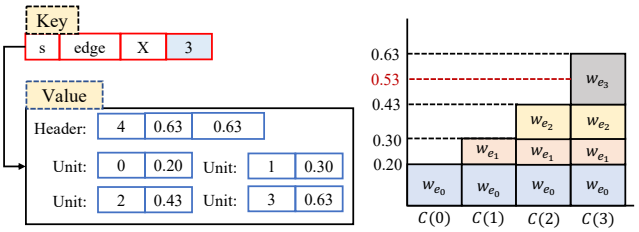
**Figure 4: The storage of node attributes in MKVGraph.**

saves space cost by removing the zero values. Figure 4b plots the storage of dense attributes for a node $s$ by storing all the non-zero values of an attribute in one key-value format. In terms of edges, the format is similar to that of nodes except that it contains the information of edge type and target node. Due to space limit, we skip the details here.

## 4.2 Sampling

For the weighted neighbour sampling, existing deep graph learning systems are not efficient since they need to retrieve all the neighbours of a source node from different graph servers into memory and directly adopt a memory-expensive sampling methods *Alias* [23, 35] which needs to construct a sampling table in memory to store the probability of each neighbour. Besides, the efficient weighted neighbour sampling becomes more challenging in our case since our block-based storage format will partition the neighbours of a node into several blocks even if all blocks are in the same graph server. To handle these challenges, we use a two-step sampling strategy as follows: (1) sampling a block w.r.t a source node, and (2) sampling a neighbour from the sampled block. To avoid the memory cost brought by *Alias* method, we adopt the *Inverse Transform Sampling* (ITS) method [35] where no extra sampling table is needed. On contrary, we can compute the probability of a node/block to be sampled when we insert the node/block in our graph storage MKVGraph, *i.e.*, $p_b$ and $p_t$ introduced in Section 4.1.2, reducing the memory cost largely.

**ITS method.** In the following, we present how to compute the sampling probability of a node. Suppose a node $s$ has $n_s$ neighbours, which constructs the set of edges as $\{e_0, e_1, ..., e_{|n_s-1|}\}$. Specifically, the sample probabilities are computed as the prefix sum based




**(a) The key-value format storing the neighbours of node $s$.** **(b) Inverse Transform Sampling for node $s$.**

**Figure 5: Running example of neighbour sampling.**

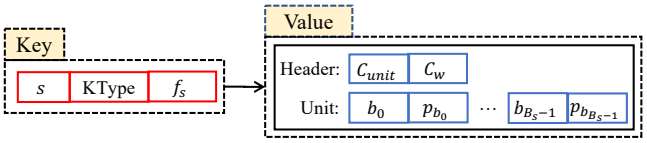



**Figure 6: Indexing structure for sampling a neighbour of a node $s$ where $B_s$ is the number of blocks for node $s$, $b_i$ id the $i$-th block of node $s$, and $p_{b_i}$ is the sampling probability of block $b_i$.**

on the weight $w_e$ on each edge $e$, *i.e.*, $C(i) = \sum_{j=0}^{i} w_{e_j}$ for the $i$-th neighbour. Next, the sampling is performed by generating a random number $r$ in $[0, C(n_s - 1))$ and finding *the smallest $i$ where* $C(i) > r$ using binary search, producing $e_i$ as the sampled edge. Figure 5 shows a running example when the neighbours of the source node $s$ are stored in only one block.

Example 1. *Suppose that a source node $s$ has four neighbours whose ID are 0, 1, 2, and 3, respectively. and the weights on edges are supposed as follows: $w_{e_0} = 0.20$, $w_{e_1} = 0.10$, $w_{e_2} = 0.13$ and $w_{e_3} = 0.20$. Figure 5a shows the storage of these four edges linked from node $s$ to each neighbour in MKVGraph. In the value, the header includes the information of 4 neighbour units, 0.63 weights, and that the probability of sampling this block is 0.63, respectively. In each neighbour unit, it contains the ID of neighbour and the sampling probability of current neighbour. For example, for node 0, its probability is 0.20, which is equal to the weight of itself; for node 1, its sample probability is the sum of $w_{e_0}$ and $w_{e_1}$, i.e., 0.30. Figure 5b shows the sampling procedure. For each neighbour, its sample probability is plotted, namely, $C(0), C(1), C(2), C(3)$. Next, a random value $r \in [0, 0.63)$ is selected. If $r = 0.53$, then node 3 is selected as the result since it is the smallest $i$ satisfying the inequality that $C(i) > r$.*

**Indexing structure.** Since node $s$ might have multiple blocks, we propose an indexing structure to accelerate the process of sampling a block for such nodes, as shown in Figure 6. Note that this indexing structure is updated immediately the graph storage finishes the dynamic insertion/deletion. Theorem 1 shows the time complexity for sampling a neighbour from multiple blocks.

Theorem 1. *The time complexity to sample the neighbourhood of a source node $s$ is $O(\log(n_s))$.*

Proof. Let $c$ be the size of a block. Sampling a block takes $\log(B_s)$ time cost and sampling a neighbour in the sampled block takes at most $\log(c)$ time by using binary search. So, the total time cost is $(\log(B_s) + \log(c))$. Since $B_s \leq n_s$ and $c \leq n_s$, the total time cost is at most $2\log(n_s)$. □

| Datasets | Relations ($S$-$T$) | #$S$ | #$T$ | #edges | #$\mathcal{A}$ |
|---|---|---|---|---|---|
| WeChat (ours) | User-Live | 340M | 2.3M | 2.5B | 11 |
| | User-Article | 535.7M | 1.8M | 3.8B | 8 |
| | User-Video | 706.3M | 17.0M | 9.0B | 8 |
| | Live-Live | 2.3M | 2.3M | 46M | 4 |
| | User-User | 1.0B | 1.0B | 8.6B | 7 |
| *OGBN* | Product-Product | 2.4M | 2.4M | 61.9M | 100 |
| *Reddit* | Post-Community | 233.0K | 233.0K | 114M | 602 |
| *Twitch* | User-Streamer | 15.5M | 465K | 124M | 2 |

**Table 1: Dataset statistics where $S$ (or $T$) stands for the source (or target) node of a relation, # denotes the set size and $\mathcal{A}$ denotes the attributes. ($K = 10^3$, $M = 10^6$, $B = 10^9$)**

### 4.3 Caching Technique

Here, we propose a caching technique in PlatoGL, which reduces the number of concurrent requests to MKVGraph, making the graph storage stable and robust. For the heterogeneous graphs, our caching technique contains two parts: (1) caching the neighbourhood of a node, and (2) caching the attributes of a node. Since both parts are similar in terms of technical contributions, we elaborate here how to cache the attributes of a node.

The intuitive idea is to store the attributes of a node that are frequently-used in a cache. For usage, if the attributes of a node are not cached, a call to *MKVGraph* is needed. Two operations are queried for caching: Cache-Set which inserts the values into the caching and Cache-Get which retrieves the values from the caching. In Cache-Set, we use two important parameters: one is *cache size*, the other is *cache expire time*. The cache size limits the number of attributes to be cached and the cache expire time indicates whether or not the attributes in the cache should be updated. In Cache-Get, for a query of attributes, if the expire time exceeds the current timestamp, it will send the request to the MKVGraph for the results. Otherwise, the attributes are directly returned.

## 5 EXPERIMENTS

In this section, we comprehensively evaluate the proposed PlatoGL system in two aspects, *i.e.*, deployment performance in a real-world recommendation system and benchmark performance (*w.r.t.* its key features) on collected datasets.

### 5.1 Experimental Setup

**Evaluation Platform.** We evaluate PlatoGL over a cluster with 20 servers, among which 8 are used for training a GNN recommendation model and the rest for graph data storage. Servers used in model training are equipped with 256GB DRAM and two 2.50GHz Intel(R) Xeon(R) Platinum 8255C CPUs, each of which has 48 cores. If not specified, each server is equipped with 6 training worker processes and 3 parameter server process. As for the storage servers, each is equipped with 192GB DRAM and one 2.60GHz AMD EPYC 7K62 90-core CPU. Note the replication factor is set as 3, thus there are actually another 24 storage servers for backup.

**Datasets.** To evaluate the benchmark performance of PlatoGL, we use one production dataset (termed *WeChat*) and three public datasets, *i.e.*, *OGBN* [1], *Reddit* [12] and *Twitch* [25], all of which are large-scale. Table 1 shows the statistics. Notice that our production dataset *WeChat* is retrieved from three content recommendation scenarios in Wechat App (article, micro-video and live-streaming), which contains a large-scale heterogeneous graph with *1.2 billion nodes and 23.9 billion edges in total*. Specifically, *WeChat* dataset contains five kinds of relations (*i.e.,* edges): (i) *User-Live* that a user interacted with a live room in the live-streaming service; (ii) *User-Article* that a user read or clicked an article in the official-account service; (iii) *User-Video* that a user watched a video in the video-sharing platform; (iv) *Live-Live* which is the relationship between two live-rooms in the live-streaming service; and (v) *User-User* which is the friendship between two users in WeChat.

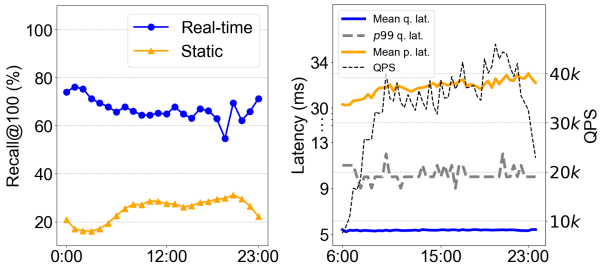
**Baselines of Live-streaming Recommendation.** To evaluate the deployment performance of PlatoGL in a real-world real-time recommendation scenario, we adopt the live-streaming recommendation service in WeChat which recommends a list of live-rooms to users. We have two baselines: (1) a *real-time two-tower DNN* model (termed *DSSM*) [36], which computes the user embeddings and items embeddings separately; and (2) a *Static GNN* model without real-time capability (termed *MvDGAE*) [43], which is effective in the cold-start recommendation task where the new user has little item interactions by regarding the cold-start as a missing data problem. Both base models have been already deployed in WeChat live-streaming recommendation service. Besides, the hyper-parameters of both base models are best-tuned. The GNN architecture could be found in [43]. In the following, we introduce how to deploy the *static* GNN model *MvDGAE* in live-streaming recommendation scenario. This *static* model is incrementally trained and updated in a daily manner. Specifically, for the model serving on *day*-$T$, the training process uses the interaction logs on *day*-$(T-1)$ and *Wechat* graph data on *day*-$(T-2)$ (to avoid information leaking). Comparatively, after deploying our PlatoGL system, we can train a *real-time GNN model* whose architecture is the same as the *static* GNN model. Differently, *real-time* model is constantly trained by learning user behavior from the up-to-the-minute data stream, as well as the dynamically updated *Wechat* graph data stored in PlatoGL.

**Model settings.** For fair comparison, we use the same model architecture (*i.e.*, MvDGAE) for both static and real-time GNN models. Besides, both of them are set up with the same hyper-parameters. Firstly, the embedding dimension of MvDGAE is set to 128, the dropout rate is 0.2, the learning rate is 0.05, and the batch size is 1024. Secondly, the heterogeneous graph we used is *WeChat* dataset in Table 1 where five relations are listed. We also leverage the meta-paths extracted from the heterogeneous graphs to guide the GNN to obtain aggregated embedding of users/items. From the perspective of user, we used the following meta-paths: (1) User-Live, (2) User-Video, (3) User-Article, and (4) User-User-Live; From the perspective of item (*i.e.*, Live in our scenario), we used the following meta-paths: (1) Live-Live and (2) Live-User-Live. For each hop in above meta-paths, we sample 50 neighbours for a node.

**Metrics for Recommendation.** To show the effectiveness of real-time recommendation on our live-streaming services, three real-world metrics are used: (1) user-click-through-rate (*uctr*), *i.e.,* $\frac{\#users-who-clicked}{\#user-who-viewed}$, which is the ratio of users who clicked at least one recommended item over the total number of users that have been recommended a list of items ; (2) pageview-click-through-rate (*pctr*), *i.e.,* $\frac{\#clicks-on-items}{\#views-of-items}$, which is the ratio of the times that users clicked on an item over the total number of times that

| Model | uctr | pctr | watch-time |
|---|---|---|---|
| real-time DNN | +0.00% | +0.00% | +0.00% |
| Static GNN | +0.305% | +0.603% | +1.108% |
| Real-time GNN | **+0.496%** | **+1.369%** | **+2.582%** |

**Table 2: Online A/B test recommendation results.**




**(a) Accuracy** **(b) Latency and throughput**

**Figure 7: Performance results of PlatoGL-aided real-time GNN recommendation ("q."/"p." denotes query/prediction).**

the recommended items have been viewed to the users; (3) watch-time [25], which records the dwelling time that users watched on recommended items. For all metrics, the larger the values, the better the recommendation model. Besides, we also adopt the commonly used Recall@$K$ as an evaluation metrics and set $K$ as 100.

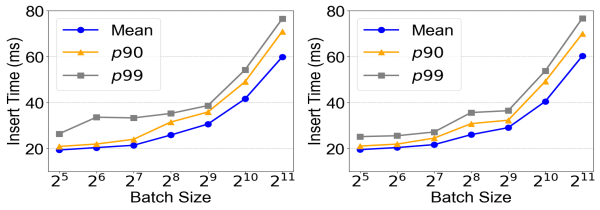
**Metrics for Efficiency.** Three metrics are used to evaluate graph updating and query efficiency of our PlatoGL system: (1) *mean* time, which is the average computational time cost among 1000 operations; (2) *p90* and (3) *p99*, which are the time cost at the 90-th and 99-th percentile over 1000 operations, respectively.

## 5.2 Real-time Recommendation

In this section, we evaluate the effectiveness of newly deployed *real-time* GNN-based recommendation model by comparing with the in-use real-time DNN model and static GNN model. We conduct an online A/B test over one week during April 2022, where nearly 15.5 million users are involved.

**User Engagement.** Due to commercial confidentiality and secrecy agreement, we use the results of the real-time DNN model as the base and only show the relative improvement rate of the static GNN model and the real-time GNN model, respectively, as shown in Table 2. The results demonstrate the effectiveness of training and serving GNN model in real-time. Specifically, the *real-time* GNN model achieves the best performance among three models in terms of uctr, pctr and watch-time. Compared with the real-time DNN model, the real-time GNN model has better performance **by 2.582%** in terms of watch-time. Meanwhile, compared with the *static* GNN model, the *real-time* one also achieve high watch-time **by up to 1.474%**. Notice that in industrial recommendation scenarios, gain of 1% is a substantial improvement [4, 34, 39]. Aided by PlatoGL that dynamically updates the graph data and enables online learning of GNN, the *real-time* model can use the newly graph topology for training and take into account the user's instant interests, *i.e.*, solving concept-drift problem. Now, the real-time GNN model serves the major online traffic in live-streaming recommendation.

**Model Accuracy.** Except the overall improvement in online evaluation, we also investigate the model accuracy of two models. Specifically, we firstly split the offline daily test data into each hour, and




**(a) Reddit.** **(b) Wechat.**

**Figure 8: Time cost of dynamic graph update vs batch size.**

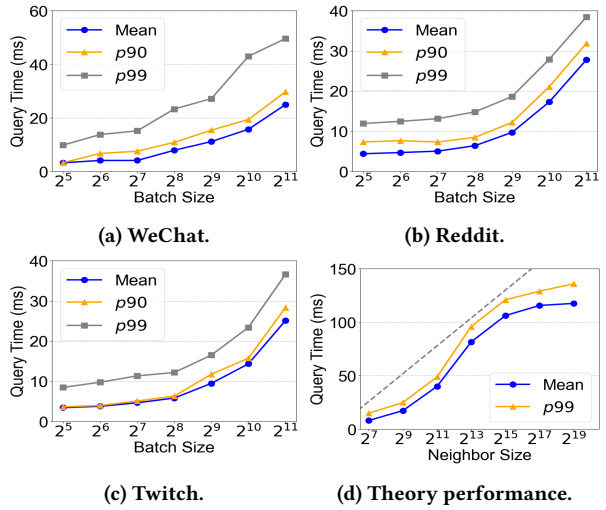
then use the available models before a specific hour $h$ for prediction and evaluation on the test data of *hour-h*, *i.e.*, the *static* model trained on data of the last day and the *real-time* model updated until *hour-h*. As observed from Figure 7a that shows the recommendation results in one day *w.r.t.* Recall@100 metric, the *real-time* GNN model achieves better performance **by up to 4 times**. Note that the offline improvement is much higher than that in the online A/B test due to two reasons: (1) the online evaluation contains a *ranking* stage after the *recall* stage where the ranking stage would alter the prediction score of items (which is widely used in industry); and (2) the online evaluation used different metrics, *i.e.*, uctr and pctr are more sophisticated than Recall@100. From the results, we can observe that performance of the *real-time* model is rather steady across the whole day (blue), while that of the *static* model is relatively higher in peak hours of evening (orange). As most of training data is generated during these hours, the *static* model might be biased towards users' behaviors in these periods, while the *real-time* model can always capture up-to-the-minute user preference during a whole day.

**System Performance.** Figure 7b demonstrates PlatoGL's system throughput QPS (*i.e.*, the number of inference requests) and the mean and tail latency ($p99$) over time, respectively. For comparison, we provide the results of graph query latency (blue/grey) and model prediction latency that contains query-related latency (orange). As illustrated by the curve of QPS, during the correspond peak in the evening, model prediction latency increases from 30$ms$ to 34$ms$, while graph query latency still maintains at a steady and low level, around 5$ms$. This indicates that the deployed PlatoGL successfully handles the peak traffic and helps the whole recommendation system respond quickly to user requests.

## 5.3 Evaluation of Key Features in PlatoGL

*5.3.1 Dynamic Updates.* To evaluate the performance of inserting the new edges to the graph storage, we test the insert time cost by varying the batch size which is the number of edges to be inserted each time. For each batch-level insertion, we conduct 1000 times. From Figure 8, we can observe that even with 2048 edges, the graph could be updated by taking less than 80 millisecond, satisfying the in-millisecond dynamically-updating requirement. Besides, on the largest graph WeChat, when 256 edges are newly-inserted, it takes only **24 milliseconds**. It is due to the power of our efficient insertion mechanism and block-based key-value store (which updates only several blocks of a source node instead of all neighbours).

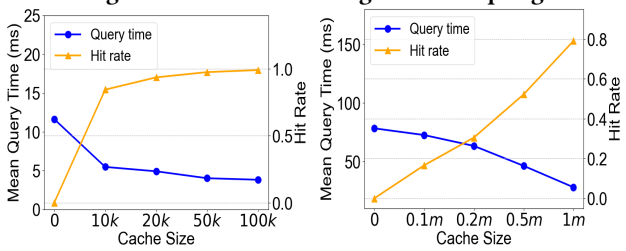
*5.3.2 Sampling.* To evaluate the performance of neighbour sampling, we test the query time cost by varying the batch size. For each node in the batch, we sampled 50 neighbours. Figure 9 shows




(a) WeChat. (b) Reddit.

(c) Twitch. (d) Theory performance.

**Figure 9: Time cost of neighbour sampling.**

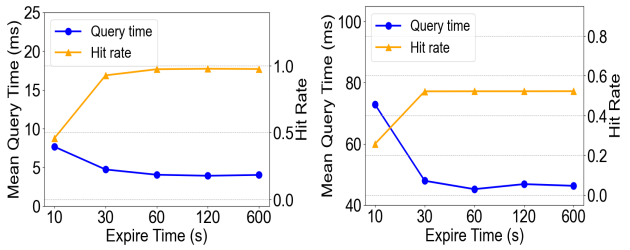



(a) WeChat. (b) OGBN.

**Figure 10: Effect of caching technique: Hit Rate / Mean Query Time vs Cache Size where expire time is fixed as 600s.**

the query time cost of sampling the neighbours for a source node on different datasets. The results show that even with 2048 source nodes, our system can answer the queries by taking *less than 40 milliseconds* in terms of mean query time on all datasets. Specifically, in WeChat dataset, in terms of mean query time, PlatoGL takes only **10 and 22 milliseconds** to sample the neighbours of 1024 and 2048 source nodes, respectively, which is very efficient for real-time training and online inference. It is because our multi-block neighbour sampling strategy can do the hierarchical sampling on the neighbours of a node without touching all blocks in the graph storage. In addition, to verify that our experimental results are consistent with the theoretical time complexity (*i.e.*, Theorem 1), we conduct an experiment by testing the sampling time cost w.r.t the neighbour size of a source node (which are synthetic). The neighbour size is varied from 128 to 524,288. Figure 9d shows the results where the grey dotted line plots the theory result (*i.e.*, linear to x-axis), and the blue and orange ones show the experimental results. We can see that the experimental results are always smaller than the theoretical one, which is consistent with theory.

*5.3.3 Caching.* Here, we evaluate the effectiveness of the caching technique. We cache the attributes of nodes in the graph, and consider a request to return the attribute of a node with caches. We randomly sample 1000 source nodes from graph. Two metrics are used: (1) *hit rate*, which is the ratio of the requests that can be found in cache over the total number of requests and (2) *mean query time*, which is the average time cost.




(a) WeChat. (b) OGBN.

**Figure 11: Effect of caching technique: Hit Rate / Mean Query Time vs Cache Expire Time where cache size is fixed as 500K.**

Figure 10 plots the results by varying the cache size on WeChat and OGBN datasets. The results demonstrate that the hit rate increases as the cache size increases. For the largest dataset (*i.e.*, WeChat), the hit rate is nearly 100% even though the cache cached 20,000 nodes only. Thus, our caching technique can consume less memory cost without sacrificing the accuracy. Figure 11 shows the results by varying the cache expire time. When the expire time increases from 10 seconds to 30 seconds, the hit rate (query time resp.) increases (decreases resp.) as the expire time increases. It is because the longer the expire time, the more information could be found in cache. However, the hit rate and query time become steady when the expire time set over 30 seconds. It indicates that we could set a proper expire time for caching.

## 6 OTHER RELATED WORK

In this section, we review the literature about GNN approaches for recommendation. Note that in terms of the literature about existing deep graph learning systems, we have already discussed them in Section 2.2. The GNN approches are various, some of which are designed for the homogeneous graphs while others for the heterogeneous ones. In terms of the homogeneous graphs, GCN [20] and GraphSage [12] can be considered as the basic GNN algorithms by using the convolutional operations. To be specific, GCN [20] incorporates neighbors' feature representations using convolutional operations while GraphSage [12] provides an inductive approach to combine structural information with node features. For the heterogeneous graph with multiple types of vertices and/or edges, Metapath2Vec [6] and HERec [26] formalize meta-path based random-walks to construct the heterogeneous neighborhood of a node and then leverage skip-gram models to perform node embeddings. Note that in this paper, we do not focus on in the perspective of algorithm design, *e.g.*, the GNN approaches for the dynamic graphs, but focus on the general deep graph learning system. Due to the space limit, please refer to the surveys [2, 9, 11–13, 31, 44] for more literature.

## 7 CONCLUSION

In this paper, we design the first industrial deep graph learning platform called PlatoGL that can support real-time GNN-based recommendation tasks. With sophisticated designs in terms of graph topology storage, attributes storage, neighbourhood sampling and caching, comprehensive experiments on both deployment performance and benchmark performance (*w.r.t.* its key features) demonstrate its effectiveness and scalability for real-time recommendation scenarios.

# REFERENCES


[1] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. 2016. The extreme classification repository: Multi-label datasets and code. http://manikvarma.org/downloads/XC/XMLRepository.html

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

[3] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. 2011. Streamrec: a real-time recommender system. In *SIGMOD*. 1243–1246.

[4] M. Chen, A. Beutel, P. Covington, S. Jain, F. Belletti, and E. H Chi. 2019. Top-k off-policy correction for a REINFORCE recommender system. In *WSDM*. 456–464.

[5] P. Covington, J. Adams, and E. Sargin. 2016. Deep neural networks for youtube recommendations. In *Recsys*. 191–198.

[6] Y. Dong, N. V Chawla, and A. Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *SIGKDD*. 135–144.

[7] S. Fan, J. Zhu, X. Han, C. Shi, L. Hu, B. Ma, and Y. Li. 2019. Metapath-guided heterogeneous graph neural network for intent recommendation. In *KDD*. 2478–2486.

[8] I. Gama, J.and Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[9] C. Gao, Y. Zheng, N. Li, Y. Li, Y. Qin, J. Piao, Y. Quan, J. Chang, D. Jin, X. He, et al. 2021. Graph neural networks for recommender systems: Challenges, methods, and directions. *arXiv preprint* (2021).

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

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

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

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

[14] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *SIGIR*. 639–648.

[15] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu. 2015. Tencentrec: Real-time stream recommendation in practice. In *SIGMOD*. 227–238.

[16] Z. Huang, M. Tao, and B. Zhang. 2021. Deep Inclusion Relation-aware Network for User Response Prediction at Fliggy. In *KDD*. 3059–3067.

[17] Alibaba Inc. 2020. Euler Framework for Deep Graph Learning. https://github.com/alibaba/euler.

[18] Tencent Inc. 2019. PlatoGraph Framework for Graph Algorithms. https://github.com/Tencent/plato.

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

[20] T. N Kipf and M. Welling. 2017. Semi-supervised classification with graph convolutional networks. *ICLR* (2017).

[21] C. Lei, Y. Liu, L. Zhang, G. Wang, H. Tang, H. Li, and C. Miao. 2021. SEMI: A Sequential Multi-Modal Information Transfer Network for E-Commerce Micro-Video Recommendations. In *KDD*. 3161–3171.

[22] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. 2019. Pytorch-biggraph: A large scale graph embedding system. *Proceedings of Machine Learning and Systems* 1 (2019), 120–131.

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

[24] D. Liu, J. Lian, Z. Liu, X. Wang, G. Sun, and X. Xie. 2021. Reinforced Anchor Knowledge Graph Generation for News Recommendation Reasoning. In *KDD*. 1055–1065.

[25] J. Rappaz, J. McAuley, and K. Aberer. 2021. Recommendation on Live-Streaming Platforms: Dynamic Availability and Repeat Consumption. In *Recsys*. 390–399.

[26] C. Shi, B. Hu, W. X. Zhao, and S Y. Philip. 2018. Heterogeneous information network embedding for recommendation. *TKDE* 31, 2 (2018), 357–370.

[27] C. Sima, Y. Fu, M.-K. Sit, L. Guo, X. Gong, F. Lin, J. Wu, Y. Li, H. Rong, P.-L. Aublin, et al. 2022. Ekko: A {Large-Scale} Deep Learning Recommender System with {Low-Latency} Model Update. In *OSDI*. 821–839.

[28] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. 2018. Graph Attention Networks. In *ICLR*.

[29] M. Wang, Y. Lin, G. Lin, K. Yang, and X. Wu. 2020. M2GRL: A multi-task multi-view graph representation learning framework for web-scale recommender systems. In *KDD*. 2349–2358.

[30] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua. 2019. Neural graph collaborative filtering. In *SIGIR*. 165–174.

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

[32] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. 2020. A comprehensive survey on graph neural networks. *TNNLS* 32, 1 (2020), 4–24.

[33] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. Ao, W. Xu, and J. Shu. 2020. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *SC*. 1–17.

[34] J. Yang, X. Yi, D. Cheng, L. Hong, Y. Li, S. Wang, T. Xu, and E. H Chi. 2020. Mixed negative sampling for learning two-tower neural networks in recommendations. In *Companion Proceedings of the Web Conference 2020*. 441–447.

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

[36] X. Yi, J. Yang, L. Hong, D. Z. Cheng, L. Heldt, A. Kumthekar, Z. Zhao, L. Wei, and E. H Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Recsys*. 269–277.

[37] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L Hamilton, and J. Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *KDD*. 974–983.

[38] S. Yu, Z. Jiang, D. Chen, S. Feng, D. Li, Q. Liu, and J. Yi. 2021. Leveraging Tripartite Interaction Information from Live Stream E-Commerce for Improving Product Recommendation. In *KDD*. 3886–3894.

[39] Z. Zhao, L. Hong, L. Wei, J. Chen, A. Nath, S. Andrews, A. Kumthekar, M. Sathiamoorthy, Xinyang Yi, and E. H Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Recsys*. 43–51.

[40] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *IEEE IA3*.

[41] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li. 2018. DRN: A deep reinforcement learning framework for news recommendation. In *WWW*. 167–176.

[42] J. Zheng, Q. Lin, J. Xu, C. Wei, C. Zeng, P. Yang, and Y. Zhang. 2017. PaxosStore: high-availability storage made practical in WeChat. *VLDB Endowment* (2017).

[43] J. Zheng, Q. Ma, H. Gu, and Z. Zheng. 2021. Multi-view Denoising Graph Auto-Encoders on Heterogeneous Information Networks for Cold-start Recommendation. In *KDD*. 2338–2348.

[44] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *VLDB Endowment* (2019).