Graph Neural Network Caching in Large Scale Temporal Graphs

Zimeng Zhou

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Graph neural networks are frequently being trained and used on large scale graphs, often in the magnitude of billions of nodes and edges. This creates the need to design more efficient machine learning systems that can handle the large amount of graph data. A popular strategy is to use distributed training by scaling out the amount of CPU or GPU resources used and designing efficient ways of communicating between these resources, in order to store and perform computations on large graphs. Moreover, GNN systems have recently been adding support for using SSDs as a part of the training pipeline. In order for efficient data movement to take place between the GPU, CPU, and SSD, GNN frameworks often create GNN specific caches to cache graph components.

In this thesis we explore feature caching for large-scale temporal graphs. We first analyze the temporal and topological properties of 4 temporal graphs, and hypothesize how they affect the cache hit ratio. We propose various metrics to quantify these properties of the graphs. With these properties in mind, we propose and evaluate different caching strategies, and moreover explore how using temporal GNN models affect our cache hit ratio. We also explore how both cache construction and graph properties can affect the hit ratio. We find that different temporal graphs may exhibit a diverse range of sampling patterns, and find that static out-degree caching, a popular strategy used in various GNN systems[18][31] perform the worst on the 4 temporal graphs investigated in this thesis, while adaptive caching strategies perform the best.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Zimeng Zhou*)

Acknowledgements

I would like to thank my supervisor Luo Mai for his invaluable guidance and expertise, without which I wouldn't have been able to develop my interests in graph neural networks. I would also like to thank his PhD student Congjie He for his thoughtful suggestions and ideas, which has made the honours project a great experience. Lastly, I would like to thank my friends and family for supporting me throughout the year.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Graphs exist everywhere in the world. They are an important abstract data type that has helped us define and describe an object by its connection with other objects. Graphs may represent a large variety of different networks; we can use them to describe physical networks, such as a street network or electrical wiring, and also virtual ones, such as social media networks or even the internet, which is a large network that connects computers all over the world. Given the versatility and power of what a graph can express, there has always been an interest in developing neural networks which take graph data as input. In the recent couple of years there has been an increase in the amount of research done on graph neural networks (GNNs), whether this is from an algorithmic perspective or a systems perspective. These recent advances have also come coupled with the adoption of GNNs in the industry, whether that is in recommendation systems, anomaly detection in cybersecurity, or traffic prediction.

Moreover, given the general increase in model size in machine learning systems and the increase in the amount of data used as model input, there has been a growing need to design more efficient frameworks and machine learning systems to process the growing amount of data[24]. This is also the trend in graph neural networks, where we notice the graphs used in industry becoming increasingly big. For example, Pintrest's recommendation system uses a GNN model whose graph has over 2 billion nodes and 17 billion edges[29]. These large scale graphs create the need for GNN systems that can process the data effectively. When processing the data, we will often need to utilize both the CPU, GPU, and SSD to store the graph data. A large bottleneck is the bandwidth of data transfer between the different devices. A solution to this is to use a GNN specific cache to cache graph data for each device, so that the quantity of data transferred is reduced.

With these problems in mind, in the thesis we have 2 main objectives:

1. Given that GNN feature caching is shown to be effective on static models and graphs[17], we want to investigate whether caching is also effective on temporal GNN models and graphs.

2. We want to evaluate which caching strategies work best for temporal graphs, and also what factors affect their efficiency.

1.2 Contributions

A summary of our contributions are given here:

- Explored the temporal and topological properties of 4 temporal graphs and hypothesized about their effects on the caching locality. Designed metrics to quantify the sampling pattern of different graphs using the temporal and topological properties.
- Based on our evaluation of the properties of the temporal graphs, we propose various caching strategies that can efficient capture the sampling locality.
- Evaluated different caching strategies on a temporal sampler, and analysed both cache and graph properties that affect cache hit rate. Previous literature[19] on feature caching only evaluates performance on static GNN/samplers.
- Investigated and evaluated the feasibility of static caching on temporal graphs. Static out-degree based caching was evaluated in depth, given its adoption in GNN frameworks and in literature. We also explored other forms of centrality metrics for static caching.

1.3 Thesis Structure

In Chapter 2, we present a brief introduction to graph neural networks, and discuss the current challenges faced when dealing with large graphs. We also give a discussion on the related work in tackling these challenges, as well as how these challenges also apply to temporal graphs.

In Chapter 3, we introduce the datasets used, and use them to explore the temporal and topological properties of the graphs, and evaluate how these properties affect the temporal locality that can be found in the datasets. We propose metrics to quantify the temporal and topological properties of the graphs.

In Chapter 4, we propose and give an overview of the caching strategies we will be exploring, and hypothesize which graph properties will result in a particular caching strategy to be effective.

In Chapter 5, we first present how we designed our experiments for evaluating the different caching strategies. This includes outlining the temporal sampler that was used. We then evaluate how the cache, graph, and sampler properties affect the hit ratio, and discuss how the metrics found in Chapter 3 correlate with our results.

We conclude our thesis in Chapter 6. We provide a summary of our results, then an evaluation of the importance and difficulties that were faced. We finish by giving suggestions on future work that can be done.

Chapter 2

Background

Our project targets the problem of making graph neural network model training and inference more efficient when used on large scale graphs. In order to understand the challenges of this problem, let us first given an outline of how graph neural networks are trained and used.

2.1 Graph Neural Networks

We define a graph as a pair $G = (V, E)$, where V is the set of graph vertices or nodes, and E is the set of vertex pairs or edges. These edges can either be directed or undirected. Directed edges mean for an edge between a source and destination node, information can only flow from the source to the destination. Graphs with undirected edges can have information flow in any direction between two nodes that share an edge. Finally, for GNNs in addition to the graph structure, each node or edge can store additional information/features. This information is usually in the form of low-dimensional vector embeddings.

The graph structure, or adjacency matrix can be stored in multiple formats, some of which allow for faster traversal and more compact storage. The adjacency matrix is a graph representation where the matrix row and columns represent graph vertices. That is, the matrix value at (i, j) is 1 if vertices i and j share an edge, otherwise it is 0. We usually use a compressed sparse row (CSR) or compressed sparse column (CSC) format to store our graph structure for efficiency of computation. This format compactly stores the structure in 3 1-dimensional arrays.

As some broad intuition, we can regard GNNs as a generalization of the input data to prior neural network models such as convolutional or recurrent neural networks[26]. CNNs primarily operate on data structured as a grid, such as images, which can be thought of as a graph with a grid like structure. Similarly RNNs operate on sequences of data, which can also be represented as a linear directed graph.

2.1.1 Graph Learning Applications

Generally GNNs are used to solve 3 types of prediction tasks: node level, edge level, and graph level tasks.

Node level tasks: These tasks involve labelling or making predictions for each individual node. This includes tasks such as node embedding, where we can use GNNs to learn a low dimensional vector representation for each node. This vector representation can then be used in other machine learning tasks. Another task is node classification, where we can learn embeddings to classify/label nodes in the graph. Node classification also includes other similar sub-problems such as node clustering, where we try to cluster related nodes together, or node anomaly detection, where try to identify nodes with a particular behaviour.

Link level tasks: The general goal is to predict/label individual edges in the graph. This includes tasks such as edge classification and link prediction. For edge classification, we use GNNs to learn edge or node embeddings, and then use these embeddings to make predictions. For link prediction we try to predict whether an edge exists between two nodes. This could be done by finding node embeddings for each node, then to predict whether an edge exists between two nodes, we can pass the two node embeddings into a binary MLP classifier.

Graph level tasks: The general goal is to predict properties that belong to the entire graph. The input to these problems are usually a large collection of graphs. An application for this type of prediction task could be to predict properties of chemical molecules, where we represent each molecule as an individual graph.

While these 3 different tasks focus on predicting and labelling different parts of the graph, the general process of creating embeddings remains the same for all the tasks.

2.1.2 Architectures

Having described some problems GNNs can solve, let us delve into how GNNs work. In the previous sections we mentioned that GNNs will generate embeddings, which can later be used in application specific tasks. Taking node embeddings as an example, our goal is to create embeddings for each node such that they capture both the node's neighboring graph structure as well as its neighboring information, such as other node and edge features. This is because in our new node embedding space, we want "similar" nodes to be close to each other.

GNNs use an idea called message passing[7] or neighbor aggregation to learn embeddings for its graph attributes. This is a method to propagate information across the graph so that nodes have a way to update their own embedding by using information from their neighbors. We can summarise message passing in 3 main components with the formula below:

$$\begin{aligned} \mathbf{x}_i^0 &= \mathbf{x}_i^0 \\ \mathbf{x}_i^{k+1} &= \gamma\left(\mathbf{x}_i^k, \bigoplus_{j \in N(i)} \phi\left(\mathbf{x}_i^k, \mathbf{x}_j^k, \mathbf{e}_{j,i}\right)\right) \end{aligned} \tag{2.1}$$

where $\mathbf{x}^0$ is the original node feature and $\mathbf{x}^k$ is the updated node feature embedding for node i for the k^{th} layer. Let's break the equation down:

1. Message function: ϕ is our message function. This function can take in the node feature embedding $\mathbf{x}_i^k$ at layer k, its neighboring feature embedding $\mathbf{x}_j^k$, or the edge embedding $\mathbf{e}_{j,i}^k$. For simple GNN models the message passing function usually only involves the neighbor feature embedding, such that $\phi(\mathbf{x}_j^k) = W\mathbf{x}_j^k$, where W is a weight matrix.
2. Aggregation function: $\bigoplus_{j \in N(i)}$ is our aggregation function. Its purpose is to aggregate all the messages from the neighbors of node i, represented as $N(i)$. This function could be any order invariant function, such as an average or max function. This is because the order in which we evaluate the neighbors of a node should not matter.
3. Update function: γ is our update function. This applies a final transformation to the aggregated messages. This function is often the sigmoid function $\sigma()$.

Let us illustrate message passing through an example graph.

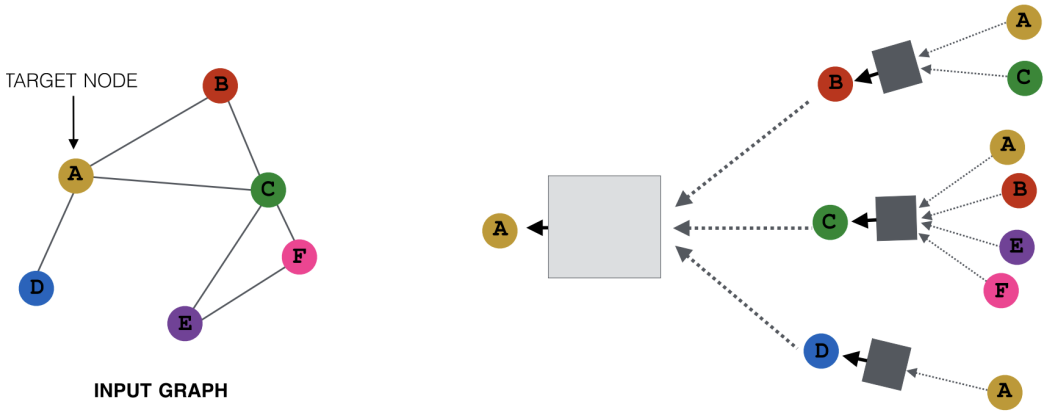

Figure 2.1: Message passing example. Taken from Stanford CS224w course notes[16]

Figure 2.1 shows a graph on the left, and its computational graph on the right. We can use a computational graph to visualize the operations that are done in equation 2.1. In the equation if we assume $k = 1$, we are using a GNN to do one layer of sampling. Thus from the target node A we would be performing sampling on A's neighbor nodes, which would be nodes B, C, and D. For each neighbor node we use them to create a message, then in the light grey box on the right we apply our aggregation function. We finally use this to update the original node A. If we have more than 1 layer, we repeat this process first for the neighbors of A's neighbors.

Note that the message passing weights and aggregation weights are shared for each layer for every computational graph generated. This means we can use the learned weights to find an embedding for unseen graphs. To see message passing in practice,

we will briefly discuss two classic GNN models: Graph Convolutional Networks[11] (GCN) and GraphSAGE[8]

- For GCN the embedding of a node i at the k^{th} layer is defined as:

$$h_i^{k+1} = \sigma(\sum_{j \in N(i)} W_k \frac{h_j^k}{|N(i)|})$$

  Note the message function multiplies the neighbor embedding h_j^k by W_k and normalizes it by the number of neighbors of node i.

- For GraphSAGE the embedding of a node i at the k^{th} layer is defined as:

$$h_i^{k+1} = \sigma(W^k \cdot CONCAT(AGG(h_j^k, \forall j \in N(i)), h_i^k))$$

  GraphSAGE has a two aggregation steps. First we aggregate all the neighbors of node i. The aggregation function could be the mean, max pooling, or an LSTM. Lastly it concatenates the aggregated neighbors with the embedding of node i from the previous layer so that the target embedding does not vanish.

Both of these models use node-wise sampling[20], where for each node we sample a finite amount of neighbors.

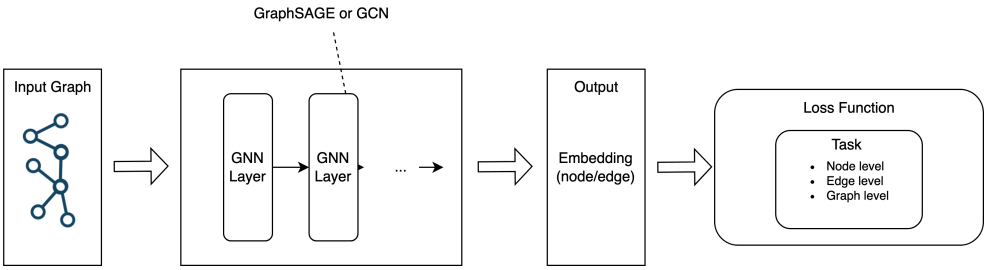

Figure 2.2: Whole pipeline overview

Figure 2.2 show the whole training/inference pipeline for GNNs. All the components of message passing are differentiable, which means we can apply standard machine learning training through backpropagation through all the GNN layers. Note GNNs often only have 2 to 4 fully connected layers, as each layer brings an exponential increase to the number of neighbors sampled, as well as the problem of over-smoothing, where the learned representations for node all become similar.

2.2 GNN serving

Now that we have a theoretical understanding on how GNNs work, let us now examine how they are implemented in practice and some of the challenges that are faced when working with large graphs. Compared to typical deep neural network model training, GNN model training/inference is unique as we must collect the input data by sampling the graph and collecting the node feature data.

For sampling based GNN training there are broadly two main approaches, either full batch or mini-batch training. Given how GNN training usually occurs on the GPU[15], due to its ability to perform large-scale parallel computations, full batch training is often infeasible for large scale graphs, since it requires the entire graph loaded into memory. GPU memory is often much smaller than the graph data structure. GCN is an example of a model used in full batch training.

Mini-batch sampling-based GNN training has been used as a workaround to this problem. An example of this is GraphSAGE, where instead of considering all the nodes in the graph, it selects n target nodes to sample, where n is much smaller than the total amount of nodes in the graph. Moreover, for each layer we sample a fixed number of neighbors. All the selected target nodes and its sampled neighbors form a subgraph/sample, which can be then used to train the model.

Popular GNN frameworks such as Pytorch Geometric[5] (PyG), and Deep Graph Library[28] (DGL) both adopt a mini-batch training approach. The general approach is shown in Figure 2.3.

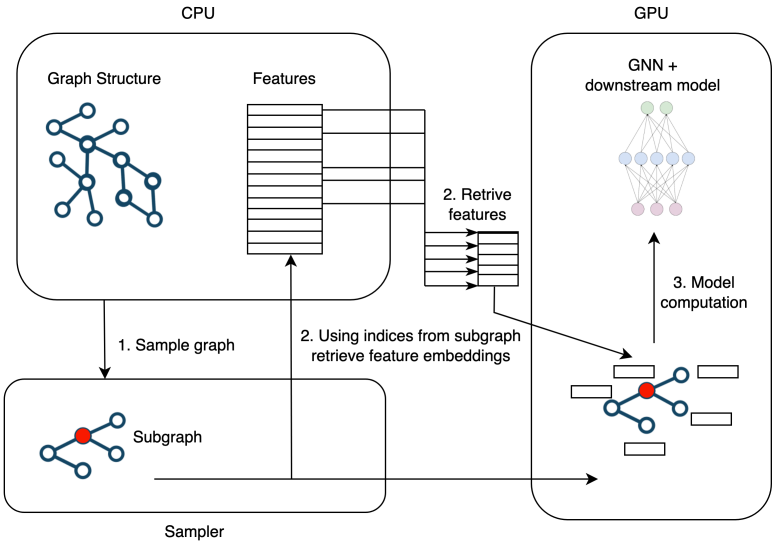

Figure 2.3: GNN sampling process. Diagram adapted from BGL[19]

Before we describe the training/inference process, note that we are storing the graph in CPU memory. Moreover, we are using two data structures to store the graph data. We first store the adjacency matrix, or the graph structure. This is usually in the compressed sparse row/column (CSR/CSC) format, for fast accesses. We store the feature vectors for each node separately, in a feature table.

For each iteration in training, we perform the following operations:

1. Sample: We first choose a set of target nodes to sample, then we find their neighbors indices from the adjacency matrix through sampling.
2. Gather: Secondly, having found the all the node indices, we then retrieve the feature embeddings from the feature table.

3. Transfer: Having sampled the indices and gathered the features for the mini-batch on the CPU, we now transfer it into GPU memory.
4. Computation: Using this data, we perform model computation.

2.2.1 Challenges

Given this process, there are several challenges facing this pipeline when working with large graphs. One of the main challenges is the overhead for transferring the data (from sample and gather) to the GPU. Moreover, if main memory cannot fit the entire graph, we would resort to storing the graph on SSD, which would further add the overhead of transferring data from SSD to main memory, which is extremely slow.

Compared to other DNN models such as recurrent neural networks or convolutional neural networks, GNN models are several of magnitudes smaller, given how they typically only use a few layers. This means GNN model computation is fast, and thus unlike for some CNN or RNNs, it is hard to overlap the compute time with the data transfer for the next iteration, without increasing the PCIe bandwidth. In fact, according to a recent paper[10], when comparing GraphSAGE to a CNN ResNet50 model, we would need 3 orders of magnitude of more bandwidth to overlap the data transfer time with model computation.

This problem could be solved by either scaling up the number of GPUs or CPUs used, so that the entire graph can fit inside either device. This is not a permanent solution, as the real world datasets continue to grow larger, some of which are over 100TB in size [19].

We instead focus on the approach of decreasing the amount of data transferred between devices by caching the node feature embeddings. By caching feature embeddings that are frequently accessed in the GPU, we do not need to use additional bandwidth to transfer this feature embedding to the GPU. If we are using SSD as storage, this logic also applies to caching feature embeddings in main memory. In fact, if both are used we could try implementing a two layer cache system where we cache the embedding in both a main memory and GPU cache. This could be a potential future work.

2.3 Prior Literature On Caching

There have been several works recently that have touched upon and implemented a version of feature embedding caching. Let us briefly summarise their approaches.

PaGraph[18] proposes to use a static feature embedding cache. Static means that the cache never be updated during run time, and that we fill in the cache pre-computation. PaGraph selects embedding features to cache based on a heuristic estimating how popular they are.

AliGraph[31] adopts the least recently used approach (LRU) for the feature embedding cache, and also creates a cache for the adjacency matrix, by using a similar metric to PaGraph.

BGL[19] focuses on GNN training, and proposes using a FIFO cache in addition to using a *"proximity-aware ordering"* for choosing nodes when creating mini-batches, so that neighbor sampling will have a higher probability of requesting the same nodes between mini-batches. They show this method has better hit ratio than dynamic caching strategies such as LRU or LFU.

Ginex[23] also focuses on GNN training, and proposes a "provably optimum" feature cache, by grouping several batches into a "superbatch", and calculates the optimal feature embeddings to evict from the cache so that the next batch will have a minimal amount of cache misses.

In our work, instead of focusing on optimizing the cache efficiency through clever ways of re-ordering the training data or pre-computing the cache replacements, we focus on GNN inference/serving, where we cannot or do not necessarily have enough future data to implement similar techniques. We instead focus on the caching strategies that don't require a large amount of future data, and see if we can find strategies that are effective for temporal graphs. Let us give a description of temporal graphs and how they are used in GNNs in the next section.

2.4 Temporal Graphs

Until recently, a majority portion of research in GNNs have been focused on static graphs, which are graphs that do not change over time where the nodes and edges are fixed. However, many real world graphs are dynamic, where for example the number of edges a node has could change over time. This is the case in social networks, where if users represented nodes, and edges represented interactions between two users, then we would expect constant adding or deletion of edges, or even nodes.

2.4.1 Temporal Graph Neural Networks

In order to capture these new types of interactions and make predictions on them, temporal graph neural networks have been introduced. The overall goal of temporal GNNs (TGNNs) is the same as static models; by using a temporal graph we aim to create node embeddings which can then be used in other downstream tasks such as link prediction.

We can broadly classify TGNN models into two frameworks[6]:

1. Time-and-graphs/snapshot TGNNs: As a natural way of static graphs to represent dynamic ones, this approach focuses on creating multiple consecutive static "snapshots" of the dynamic/temporal graph, and using these sequence of snapshots to capture the evolution of node embeddings over time in order to find a temporal node embedding. Some current state-of-the-art snapshot TGNNs include Evolve-GCN[22].

2. Time-then-graphs: In this framework, we first represent all the edges in a temporal graph with a timestamp representing when it was created, which then can be

thought of as an edge feature for a static graph. This results in the possibility of multiple edges existing between two nodes, but at different time stamps.

An example of time-then-graphs include TGN[25]. In the TGN model the temporal graph is modelled as a sequence of ”interaction events”, where an interaction event is a time-stamped temporal edge that occurs between nodes i and j. Moreover, interaction events are also the inputs to this model; for each interaction event we want to predict the likelihood of this event existing. TGN extends the message passing framework by adding a memory module, where each node has a memory state which compactly represents the node's past interactions. This memory state of a node is updated whenever an interaction event involves the node, usually by using an RNN such as a LSTM or GRU. The memory module thus allows TGN to learn the temporal dependencies of each node.

2.4.2 Temporal Graph Sampling

In the previous sections, we gave a brief description of how sampling is done for static graphs. For node-wise sampling, we would specify a set number of neighbors we would sample for each neighbor. For temporal graphs we must modify this strategy in order to apply it to TGNNs. TGNNs require training to be done chronologically in order to preserve the temporal dependencies. Moreover, the chronological ordering also applies to sampling. This is intuitive, since given a graph that includes edge interactions from timestep 0 to t_n, if we sample the graph at timestep t_k where $t_k < t_n$, we should only be able to sample edges that occur before time t_k.

Implementing temporal samplers that can generalize to different TGNN models is not a trivial problem, as new data structures for storing temporal neighbors will have to be designed. In fact, most popular GNN frameworks such as PyG and DGL do not support edge-based temporal sampling yet. Fortunately, a recent paper[30] has implemented temporal sampling, even though the framework is still in development. We will discuss this more in Chapter 5.

In our work we focus on data fetching, that is the sample and gather operations mentioned in section 2.2, and thus we don't focus on the specific model used for computation. Nonetheless we can assume the problem we are dealing with is a link prediction problem, where given an edge, we want to predict the strength of that connection. Our input data is thus the timestamp ordered interaction events (edge creations), where for each interaction we want to predict its strength. Given this timestamp ordered list of events, we believe there exists temporal locality when sampling consecutive interaction events, and therefore making temporal graphs suitable for caching.

Chapter 3

Locality in Temporal Graphs

Given the challenges addressed in the previous chapter, let us explore the characteristics of temporal graphs and how we may exploit them to create more efficient caching strategies. In this chapter we introduce the temporal datasets used, and analyse both the graph topology as well as the temporal and topological sampling patterns present in the data.

3.1 Datasets

Given how we want to study caching in temporal graphs, and especially how they can solve problems that arise in large-scale graphs, we will use 2 large temporal graphs and 2 smaller scale temporal graphs. While in general there is a lack of large-scale open-source temporal graphs available in academia[15], we chose the Stack Overflow[1] and Taobao[2] datasets, as they were two of the largest temporal graphs freely available online. The two smaller temporal graphs are Reddit and Wikipedia, which are chosen as they often appear in temporal graph literature[13][25][30].

Dataset	# Nodes	# Edges/interactions	Timespan
Stack Overflow	2,601,977	63,497,050	2774 days
Taobao	5,150,018	100,150,807	9 days
Reddit	11,000	672,447	30 days
Wikipedia	9,227	157,474	30 days

Table 3.1: Dataset Properties

Table 3.1 shows some basic properties of the graphs. Reddit, Wikipedia, and Taobao are bipartite interaction graphs, where the nodes represent either a user or an item (Reddit page, Wikipedia article, shopping item respectively). This means there exists edge links between users and items, however no interactions between users or items themselves. The Stack Overflow dataset represents a non-bipartite user-user interaction graph, where each node represents a user, and edges between nodes represent an interaction (u_i, u_j) represents an interaction between users u_i and u_j. In all our datasets, we notice that there is an order of magnitude more edges than nodes. Furthermore, every edge has a

timestamp which corresponds to when the edge interaction took place. In Table 3.1 the time span corresponds to the difference between the earliest and latest edge interactions.

Before the datasets could be used we did some pre-processing. For the bipartite graphs the edges are directed from the user to the item. We transform these edges to be undirected, so that we can also traverse from an item to a user. This is done so that we may perform multi-layer message passing. Additional pre-processing was done to the datasets Taobao and Stack Overflow, as they were found online as .csv files, where each row corresponded to a single edge interaction event. The source and destination node indices were initially random integers, thus we first transformed the indices so that the total range of the node indices corresponded to the total number of unique nodes, i.e. we assign indices from $[0, n-1]$, where n is the total number of unique nodes. This makes our data easier to iterate through in the future, as there are no gaps in the indices. Lastly we sort the edges by timestamp, and save the dataset as a PyG Data object.

In our datasets each edge corresponds to an interaction, but we can also interpret each edge as a new request for us to handle during serving. Before we start examining what kind of temporal and spacial localities exist in these requests in our graph, let us first visualize the frequency of the requests to understand our data better.

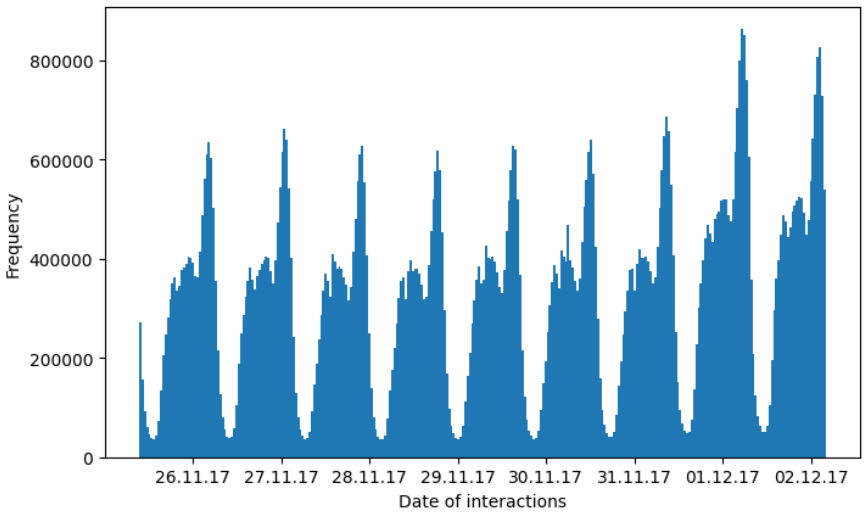

Figure 3.1: Taobao dataset's edge interaction frequency with respect to time

In Figure 3.1, we notice a cyclic pattern in the number of edge interactions in the dataset with respect to time. Since the timespan from which the data is collected is 9 days, and the dataset is Taobao, a popular online shopping platform, it is clear that each of the 9 peaks correspond to when the shopping activity was highest each day. Likewise, the troughs are likely during the night when activity is the lowest. While we won't be analyzing how edge interaction frequency affects caching strategies, it is clear that our datasets contain rich and large amounts of patterns to explore. For example, at different times of the day we may expect to find different temporal or topological patterns in the edge interactions. In the following sections we explore the datasets as a whole.

3.2 Temporal Properties

Temporal locality generally refers to the tendency of data that has been used recently to be reused again in the near future. In our graphs, the temporal property stems from the fact that we perform sampling on edges based on when that edge interaction occurred. Thus we hypothesize temporal locality exists in our data assuming recently accessed nodes will be accessed again in the near future. If so, this provides us with an incentive to use caching strategies that can capture temporal locality.

3.2.1 Consecutive Node Accesses

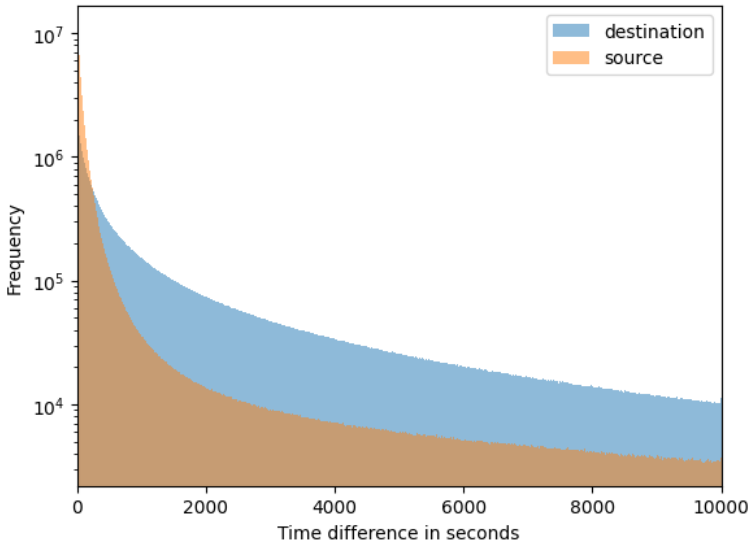

Figure 3.2: Time difference between consecutive interactions of the same destination/source node in Taobao dataset

Figure 3.2 is a logarithmic graph which shows the time difference between consecutive interactions in either a source or destination node. We only plot the time difference for the Taobao dataset, and summarize the statistics for the other datasets in Table 3.2. The other datasets follow a similar pattern. We note that the majority of time difference values are small, with the median time difference for source nodes being 75 seconds. Given the decreasing shape of the graph, we can conclude the time difference between when the same node gets interacted with is usually small. Generally there seems to exist temporal locality in our data - nodes that have been recently accessed have a higher probability of being accessed again than after a long time.

In Table 3.2, we note unlike Taobao, the graphs Overflow and Reddit have a lower median time difference for destination nodes. This could be explained through the nature of the dataset; Overflow and Reddit datasets have a much larger proportion of unique source nodes compared to destination nodes, which makes an interaction with a destination node relatively more likely. In contrast the Taobao dataset has a larger proportion of destination nodes. While our caching implementation does not differentiate between source and destination nodes, if there is a large discrepancy in the

access patterns between the two, it may be interesting to explore using different caching strategies for the different nodes.

Metrics	Overflow	Taobao	Reddit	Wikipedia
median src time diff	2375	75	4617	311
median dst time diff	1252	1259	229	333
coefficient of variation	2.01	1.69	1.68	2.2
avg. requests/min	16	7680	15	3.5

Table 3.2: Temporal Properties

3.2.2 Graph Burstiness

Since all of our datasets follow the pattern visualized in Figure 3.2, let us find a more precise way of comparing how much temporal locality different graphs contain. We expect graphs which exhibit a ”bursty”[14] behavior to have more temporal locality. This fits our intuition; suppose there is a post that goes viral on social media, then in a short time span there will be a large number of queries related to the post. These types of sudden and large bursts of traffic to specific content is what makes data temporal.

Let us further illustrate with an example. Suppose we have node A who has two different access patterns. The first access pattern requests node A every 5 seconds. The second pattern requests node A 12 times in the span of a second, however it only does this one every minute. The second pattern exhibits more temporal locality, as the access pattern is more ”bursty”; after the node has been requested there is a very high probability it will be requested again. Suppose in a scenario where during an interval of 5 seconds, the number of unique nodes being requested is larger than the cache’s capacity. This means that in our first access pattern, node A will be evicted from the cache before its next request. In contrast, due to the burstiness of the second access pattern, node A may still be cached. Let us capture this characteristic using the coefficient of variation.

$$c_v = \frac{\sigma}{\mu}$$

σ and μ are the mean and standard deviation of the differences in time between two consecutive accesses of the same node. We can use this coefficient to measure how dispersed a probability distribution is. In our case, we want to find how dispersed the time differences between consecutively accessed nodes are.

In our discussion above about node A, our first data access pattern would correspond to a constant random variable, as requests happen at a fixed rate. This produces a coefficient of 0, since the standard deviation is 0. Suppose the requests are modelled by a Poisson distribution, that is the likelihood for the next request is equal for any interval of time. The time between events in a Poisson process is modeled by an exponential distribution[4], where the mean is equal to its standard deviation, thus our coefficient has a value of 1. In general, a higher value for the coefficient means the more clustered/bursty our nodes are.

It is important to note that even though a dataset may have a high average burstiness coefficient, this does not necessarily mean this dataset will be easier to cache. Cache efficiency does not depend on the burstiness of a single node, but on the access pattern of all the nodes together, since it may be the case the amount of nodes that are bursty at any given time exceeds the cache capacity, or when a node is bursty this results in a large number of new unique feature nodes being requested which also exceeds the cache capacity, both of these scenarios would result in suboptimal cache performance. In Table 3.2, we calculated the average amount of requests/edges that occur per second. Using these properties, we can give a rough formula on how cache efficiency is affected:

$$CE = \frac{k_1 \cdot c_v}{k_2 \cdot t_d \cdot r_s}$$

c_v is the coefficient of variation, t_d is the time difference between the same consecutively sampled node, r_s is the amount of node requests per second, and k_1 and k_2 are constants which represent the effects of other hidden factors on cache efficiency. Generally large c_v values, small t_d and low r_s values should increase cache efficiency. The hidden constants represent various unknowns, one of which could be the effects of neighbor sampling; if sampling results in a large number of nodes not in the cache, this decreases cache efficiency.

3.3 Topological Properties

In addition to the information we can extract using timestamps of when the edges are created, let us inspect the topology of the graphs. Both of these properties affect the result of GNN sampling. Let us explore our graphs by plotting out the out-degree of source and destination nodes, which can be interpreted as the number of ”interactions” a node has. The out-degree of a node is the number of edges that node has that are directed to other nodes.

Metrics	Overflow	Taobao	Reddit	Wikipedia
avg. dst out-degree	5	24	683	157
avg. src out-degree	3	101	67	19
avg. sampled distance	3.125	3.6	2.95	3.59

Table 3.3: Topological Properties

3.3.1 Node Out-degree Importance

In Figure 3.3 we notice destination nodes generally have much more total number of interactions than compared to source nodes. Furthermore, the maximum number of interactions for a source node does not exceed 1000. This fits our intuition, as in the Taobao dataset each source node corresponds to a user, and an interaction corresponds to a page view or a purchase during the span of 9 days when the data was collected. Destination nodes correspond to shopping items, and thus have many more items that are generally more interacted with. When caching we could expect a large number of

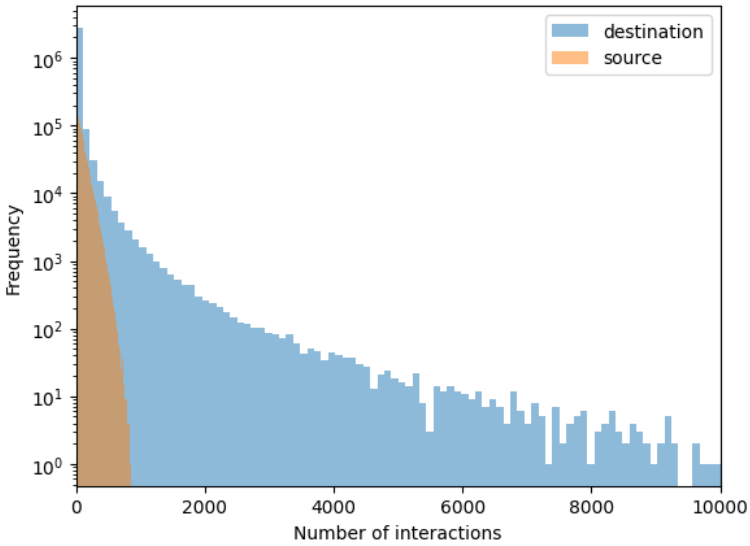

Figure 3.3: Number of total interactions for the same destination/source node for Taobao dataset

node features to be destination nodes. That being said, from Table 2 we note that the average out-degree of destination nodes is lower than source nodes, with values 24 and 101 respectively. This could be because many shopping items are rarely interacted with.

Given our discussion on graph out-degree nodes, let us use a toy graph example to show how can we take advantage of spacial locality that occurs during graph sampling. Suppose we chose n nodes to be sampled on a random basis and that we sample all the neighbors of the chosen node, then as n tends to infinity we should expect the ranked frequency of nodes sampled to be equal to the ranking of nodes with the highest out-degree neighbors. To see this, let us examine Figure 3.4.

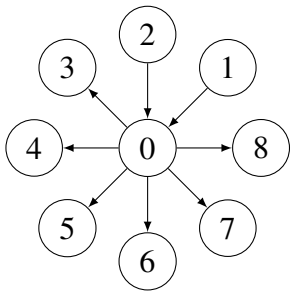

Figure 3.4: Graph out-degree example

In this graph, we notice node 0 has an out-degree of 6 and an in-degree of 2. Nodes 3 to 8 have an in-degree of 1, and nodes 1 and 2 have an out-degree of 1. Suppose we performed sampling on node 6. Since it only has one in-degree neighbor, it only samples node 0. If we pick n random nodes to sample in this graph, it is clear that node 0 will be sampled the most, since it has the highest out-degree. Various GNN systems[31][18] have used this observation to design caching strategies. Let us evaluate if this strategy may be used on our temporal graphs.

3.3.2 Node degree and Access Probability Correlation

Given that we know there should be a correlation between our ranking for the most sampled nodes and the nodes with the largest out-degrees, let us evaluate to what extent this is true on our datasets. In order to evaluate the ordering/ranking of the most sampled feature nodes, let us use the Spearman Rank Correlation coefficient, which finds how correlated two variables are. The coefficient is defined as:

$$r_s = 1 - \frac{6\sum(R(X_i) - R(Y_i))^2}{n(n^2-1)}$$

where n is the number of feature nodes, $R(X_i)$ and $R(Y_i)$ are the ranking of nodes X_i and Y_i, i.e. the node that is sampled the most will have a rank of 1. Spearman's coefficient ranges from $[-1, 1]$, where 1 represents a perfect positive correlation, whereas 0 represents no correlation. If the out-degree and access frequency ranking is the same, we would expect a correlation of 1.

In addition to finding out the coefficient for each of the datasets, let us also examine how the number of neighbor nodes sampled affects the coefficient. We will plot how the number of layers and the number of nodes sampled per layer will affect the correlation coefficient. In practice, if we want to sample k neighbors per layer, this means at each layer we pick k different edges and pick the nodes that are on the other side of the edge.

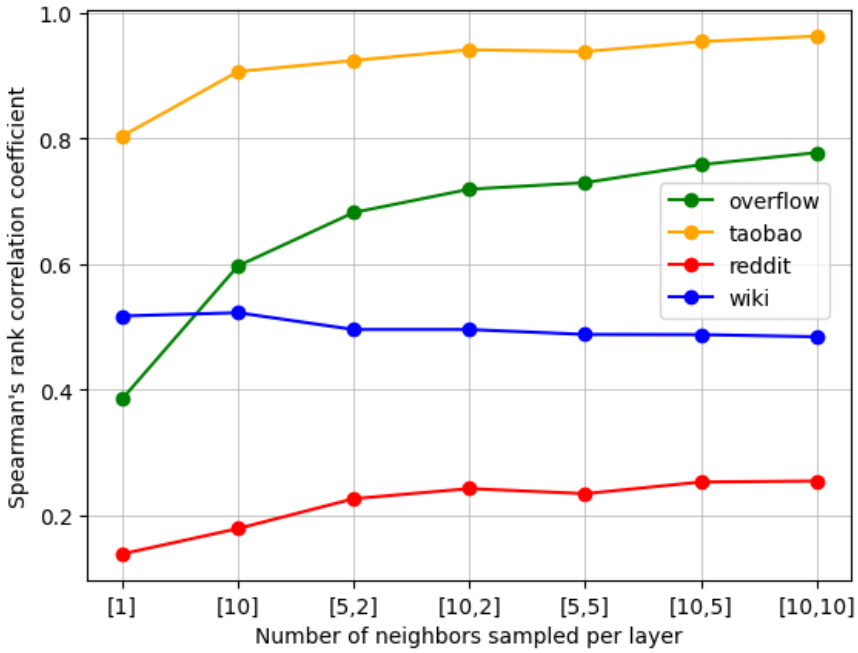

Figure 3.5: Correlation between node out-degree and sampled node frequency rank

In Figure 3.5 we notice Overflow, Taobao, and Reddit datasets have a general positive correlation between the coefficient and the number of neighbors sampled. This fits our intuition, as when more neighbors are sampled nodes with higher out-degrees will be sampled more, as shown in Figure 3.4. However, the Wikipedia dataset maintains a

similar coefficient even as we increase the number of neighbors sampled. This could be due to the nature of temporal graphs, as between two nodes there may be multiple edges that appear at different timestamps. Therefore, even as we increase the number of neighbors sampled (edges we pick) we may still be sampling the same nodes, albeit through different edges. This results in similar ranking coefficients.

Moreover, Reddit and Wikipedia have low coefficients, on average they have a coefficient of 0.23 and 0.52 respectively. This shows that there is not a strong correlation between the ranking of out-degree and sampled nodes in these two datasets, and suggests the access pattern does not favour popular nodes. In comparison, Taobao and Overflow have a coefficient of at least 0.6 for nearly any configuration of the number of neighbors or layers. For the Overflow dataset, when we only sample one neighbor, there is low correlation, however the coefficient increases as we increase the number of neighbors sampled. This suggests the access frequency of nodes for these datasets are similar to their out-degree ranking. This shows that the access pattern for these datasets is usually characterized by sampling high out-degree nodes.

3.3.3 Average distance between sampled nodes

In the previous section we discussed temporal locality and focused on how long it took for the same node to get sampled again. Apart from that metric, it is also useful to measure how ”far apart” consecutively sampled edges are. We can define how far apart two nodes are in a graph by the shortest path between them, which is the minimum number of edges to get from one node to the other.

Calculating this metric provides us with two benefits. Firstly, it gives us a new way of quantifying how much temporal locality a graph has, by using the topology of the graph. Intuitively, the closer consecutively sampled edges are, the higher the probability the nodes sampled from the next edge are already cached in our feature cache. Assuming neighbor sampling is performed on nodes n_1 and n_2, and these nodes are close, the benefit comes from the fact that these nodes may share sampled nodes. Secondly, assuming we find the shortest path between n_1 and n_2 to be of length $l1$, then $\lceil \frac{l1}{2} \rceil$ is a lower bound for the number of layers needed to be sampled so that n_1 and n_2 will share a common node in their computational graph. This is illustrated in the figure below.

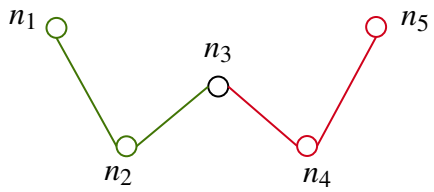

Figure 3.6: Neighbor sampling example

In Figure 3.6, the shortest path distance between n_1 and n_2 is 4. We expect at least $\lceil \frac{4}{2} \rceil = 2$ layers needed in order for n_1 and n_2 to share a common node. If our GNN samples two layers, shown in green and red, we expect the two nodes to share n_3. This is a lower bound since during neighbor sampling, we do not sample all the neighbors of a node, thus the shortest path route may not be included in the nodes that were sampled.

We can therefore use this metric as a locality estimate in graphs. As an extreme example, if the distance between consecutively sampled nodes is 30, we can assume there will be no overlap in the nodes sampled from the two original nodes. For two consecutive edges, to calculate this metric we apply the following formula:

$$v(e1,e2) = \frac{p(e1_{src},e2_{src}) + p(e1_{src},e2_{dst}) + p(e1_{dst},e2_{src}) + p(e1_{dst},e2_{dst})}{4}$$

$e1$ and $e2$ are the edges that are consecutively sampled, and $p()$ calculates the shortest path between two nodes. Since each edge has two nodes in the equation above we permute through all the combinations of two nodes and take an average. The results for our datasets are summarized in Table 3.3. We note the datasets have similar values, with Reddit having the lowest average shortest path at 2.95 whereas Taobao having a metric of 3.6. However, while Reddit and Taobao had 0.5% and 0% of node pairings that didn't have a path between them respectively, for Stack Overflow and Wikipedia 7.7% and 3.9% of node pairings didn't have a path respectively. When using this metric these values should also be taken into consideration.

Chapter 4

Caching Techniques

Having examined the temporal and topological properties of our datasets, let us now analyze different caching techniques, and hypothesize how well they may perform on temporal graphs.

4.1 Static Caching

In various GNN systems and in literature, such as in PaGraph, a static caching method has been proposed. We define static caching as a caching strategy where a cache is pre-populated with a set of nodes, and when the cache gets queried it never updates itself even if there is a cache miss.

4.1.1 Static Out-degree Caching

Based on our analysis in section 3.3, we noticed in some datasets there exists a correlation between the sampling node access ranking and the node out-degree. This suggests caching the top k out-degree nodes could be a viable strategy. We initially populate the cache with the highest out-degree nodes of the current graph, and do not modify it after construction. When there is a cache miss, that is when the requested node is not in the cache, we do not update the cache, in contrast to dynamic caches, such as LRU or LFU. Note in later sections out-degree caching will be referred to as static caching.

An advantage of static caching is that cache requests will have low latency, as the cache does not need to be updated. That being said, there are several disadvantages. Firstly, by definition a static cache is very poor in capturing temporal locality in data. For example, if there is a sudden increase in activity on nodes with a low out-degree that are not in the cache, these will always be cache misses. Moreover, when performing serving/inference on temporal graphs, each new edge that we perform inference on will be added onto the graph. This means the graph's node out-degree that is calculated beforehand may not reflect the current out-degree ranking after inference has been running for some time. This suggests the performance may degrade over time, if we do not periodically manually update the static cache. We further explore static out-degree

caching in Chapter 5, and discuss the idea of periodically manually updating the cache ranking more in depth.

4.1.2 Graph Centrality

Using the graph's out-degree is one way of pre-populating the static cache. Moreover it is also one of the simplest metrics used to calculate the centrality of a graph. Centrality is an indicator which assigns an importance ranking to each node of a graph. Apart from a graph's node out-degree, there exists many other ways of assessing graph centrality. Let us evaluate these other metrics, and evaluate how feasible they are when used as a way of pre-populating static caches.

Closeness centrality: Unlike degree centrality which only considers the node's immediate neighbors, closeness centrality measures the average shortest path length from the node to all other nodes in the graph:

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}$$

$n-1$ is the number of nodes that can be reached from node u, and $d(v,u)$ is the length of the shortest path from v to u. Since we need to find the shortest path for all pairs of nodes, this algorithm has $O(n^3)$ complexity [2]. This may be too computationally expensive, especially when evaluating large graphs with billions of nodes. If the dataset we are using is not too large, we could apply this algorithm, and treat the computation of the centrality ranking as a pre-computation one-time cost before initializing the cache and running the model. However, as we discussed in the previous section, for temporal graphs our ranking list may have to be updated in order to reflect the new graph structure that is created as edges are being requested.

In the next chapter, we will discuss other alternative ways to approximate this centrality metric so that large scale graphs can use this metric effectively.

Betweenness centrality: For a given node in the graph, this centrality metric measures how many times node v is passed through when calculating the shortest path between all possible pairs of nodes that are not node v:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

For all set of nodes (s,t) that don't contain v in our graph, we find the ratio of the number of shortest paths that contain v from s to t over the number of shortest paths from s to t. This metric suffers problems similar to closeness centrality, as it has $O(n^2 \cdot logn)$ complexity [2]. This metric could be useful when analyzing temporal graphs, as we can interpret the metric as which nodes regulate the "flow" around the graph the most. These are not necessarily nodes with high out-degree, but rather nodes that are positioned at the "bridges" of the graph.

Eigenvector centrality: The intuitive idea for this metric is that a node is deemed important if it connects to other important nodes. Therefore, unlike degree centrality, even though a node may have lots of connections, this does not imply it has a high eigenvector centrality. This is a relatively popular centrality measure. In fact, Google's PageRank is a form of eigenvector centrality. We compute this metric using the power iteration method[12], which has a complexity of $O(n \cdot m)$, where m is the number of edges.

We hypothesize that these metrics may perform better than using out-degree, as they capture global properties of the graph, rather than just the properties of a single node. This is evaluated in the next chapter.

4.2 Simple Replacement Policies

4.2.1 Least Recently Used (LRU)

LRU is a dynamic caching strategy which evicts the least recently accessed node. This is an ideal caching strategy if our data exhibits a lot of temporal locality, that is if a node has been sampled recently it has a high likelihood of being sampled again. It can be implemented straightforwardly by using a hash table and a doubly linked list, where the hash table holds the keys for $O(1)$ lookup and the doubly linked list used for $O(1)$ insertion.

The disadvantages of LRU are that the cache can be "flushed" easily. In the extreme case, suppose for a cache of size k, and k new unique nodes that aren't in the cache are queried. All the previous nodes in the cache will be cleared out and filled with the k new nodes. If these k new nodes happen to only be queried this one time then may have lost the previously cached values which may have been more useful. Furthermore, even if there exist some nodes that are often queried periodically, if there are more nodes that can fill up the cache and evict the existing node before the next consecutive node query, then LRU will perform very poorly. This was briefly touched upon when discussing node burstiness.

4.2.2 Least Frequently Used (LFU)

LFU is also a dynamic caching strategy which evicts the least frequently accessed node. Each element in the cache has a counter for how many times it has been accessed. This can be implemented with a hash table and two doubly linked lists, resulting in $O(1)$ for lookup and insertion. This may be an ideal caching strategy if the graph sampling access pattern samples a set of popular nodes frequently. Unlike LRU, it does not flush the entire cache when a consecutive number of unseen nodes are queried. LFU has disadvantages similar to the static cache, for example if there is a change in access pattern such that a previously popular node is now rarely queried, this node will stay in the cache, due to its high frequency count.

4.2.3 Custom Rankings

In addition to the two classic caching strategies mentioned above, we also designed two new custom caching strategies to test.

1. As a natural extension to static out-degree caching, we modify LRU such that we evict the node with the lowest out degree ranking. While this means we need to keep track of the out-degree ranking of all the nodes, the memory used for this will be negligible, as we only need to remember the indices. We expect this strategy to partially solve the problem of not capturing increased activity from nodes that are not in the cache. On the other hand, even if a low out-degree node becomes popular, it will quickly be evicted since if it has the lowest out-degree in the cache it will be the first to be evicted.
2. We explore the idea of using the number of neighbors that are sampled for each unique node as a ranking. We find how many neighbors are sampled by first specifying how many neighbors we want to sample at each layer. Suppose in the first layer we choose to sample 50 neighbors, and in the second layer we sample 25. For nodes that are well-connected we may be able to sample the maximum number of neighbors, that is for each 50 neighbors we can also sample 25 of their neighbors, resulting in a sample size of $50 \cdot 25 = 1250$ nodes. For nodes with only a few neighbors, after sampling they will have a total sample size which is much less than 1250 nodes. Similar to the first strategy, we also use an eviction policy of removing the node with the lowest number of sampled neighbors.

4.3 Adaptive Caching

Given how LRU caching can exploit temporal locality and LFU caching is able to adapt to the frequency of nodes sampled/topology of the graph, let us find a way to combine the two caching strategies. Since we are performing inference on the edges being requested, we would like to be able to use the caching strategy that best suits the current data access pattern present in the graph. In practice this could mean we want to change the importance we assign to the two different caching strategies dynamically.

A solution to this is the Adaptive Replacement Cache[27][3] (ARC). ARC is a caching algorithm that keeps track of both the most recently and most frequently used nodes. Let us demonstrate ARC with an example.

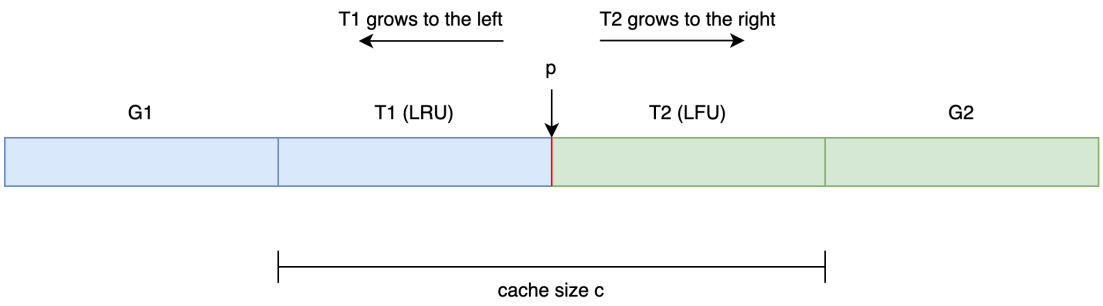

Figure 4.1: Adaptive Replacement Cache (ARC) illustration

In Figure 4.1, initially the cache of size c is split evenly into two lists T1 and T2. While both T1 and T2 use an LRU eviction policy, T1 is designed to capture ”recency” access patterns whereas T2 captures ”frequency” patterns. T1 will contain the node feature embeddings that are requested exactly once, whereas T2 will contain all the embeddings that have been requested more than once. As an example, when a node is first queried, and it doesn't exist in the cache, it is pushed into T1. Later, if this node is still in T1 and is queried again it will be moved onto the top of T2.

This caching strategy is adaptive since the size of T1 and T2 can change. We use a number p to denote the split for how much space we should allocate to T1 and T2. In the figure above, p initially splits T1 and T2 evenly, represented by the red line. However, if our node query pattern suggests that using an LRU caching pattern is more beneficial, p would conceptually shift to the right, so that T1 gets allocated a bigger space than T2. Note that either T1 or T2 can be at maximum of total cache size c.

ARC determines how to adaptively calculate p/the size of T1 and T2 by using the auxiliary queues (FIFO) G1 and G2. These two lists can be thought of as ”ghost/history” lists, as they contain only the node indices that were recently evicted from the lists T1 and T2. G1 and G2 are placed at the end of T1 and T2. When the last element in T1 or T2 is evicted, it becomes the first element in G1 and G2 respectively. The idea is that when we query a new node that results in a cache miss in T1 but a hit in G1, this signifies that our T1 size is too small, and should allocate more space to T1. If more space was allocated to T1 the node that was just missed may have still been in T1 (instead of being pushed into G1). Likewise is there is a miss in T2 but a hit in H2 this means T2 should be bigger.

Equipped with some intuition, let us describe briefly how we change the size of T1 and T2. In actual implementation, p represents the target size of T1. When there is a cache miss, p is used to determine whether a node should be evicted from T1 or T2. There are two scenarios:

1. There is a cache miss in T1 and T2, but a hit in either G1 or G2: If $|T1| > p$, we evict from T1, else T2. We add the new node that was missed into T2, and shift p in the appropriate direction.
2. There is a cache miss in T1, T2, G1 and G2: If $|T1| > p$, we evict from T1, else T2. We add the new node that was missed into T1.

ARC has several advantages. Firstly, it can adapt to handle the current access pattern. We thus hypothesize that ARC will perform at least as good as LRU and LFU. Secondly, unlike LRU it is resistant to the cache getting flushed. While T1 may be cleared when a sequence of nodes not in the cache get requested, T2 stays intact. Lastly, it has roughly the same runtime as the LRU cache, as all the lists follow the LRU eviction policy.

Chapter 5

Cache Evaluation

In this section we evaluate the proposed caching strategies in the previous chapter. We first give an overview of our methodology used when evaluating the caching strategies, and then evaluate the various factors that affect the cache hit ratio. We first evaluate cache size and number of neighbors sampled. We then focus on static caching, and explore how the ranking of out-degree nodes are constructed and what centrality metrics are used affect caching. Lastly we evaluate common sampling strategies that are specific to temporal graph neural networks, and how they affect the cache hit ratio.

5.1 Methodology

In order to test our caching strategies on temporal graphs, we must use an appropriate neighbor sampler. We initially tried to use the popular python library Pytorch Geometric to perform sampling, however we found out PyG does not currently support edge timestamp based sampling. Fortunately, TGL[30], a framework implemented by AWS, proposes a temporal sampler that can be used on various TGNN models, including both snapshot-based TGNNs and time-then-graph TGNNs. They implement the sampler by introducing a new data structure to store the graph structure and edge timestamps, which they call T-CSR. Let us illustrate how it works with an example.

In Figure 5.1, we have a root node A that has 4 neighbors, B, C, D, E. Each edge has a corresponding timestamp of when this edge appears. We can compactly represent this graph structure using the CSR format. Our index pointer array consists of all the nodes in the graph. Each node in the index pointer array points to the starting position in the indices array where the neighbors of the node are kept in consecutive indices, sorted by timestamp.

Building on top of CSR, T-CSR adds a timestamp array, which stores the times when the edges occur. In addition, assuming the TGNN model uses k snapshots, for each index pointer we also store $k+1$ pointers which separate the timestamp array into k different sections. In Figure 5.1, we have two snapshots S_0 and S_1. These are separated by 3 pointers, pt_0 which points to the start of S_0, pt_1 to the end, and pt_2 to the end of S_1. Using these pointers it is possible to sample neighbors from a snapshot in $O(1)$

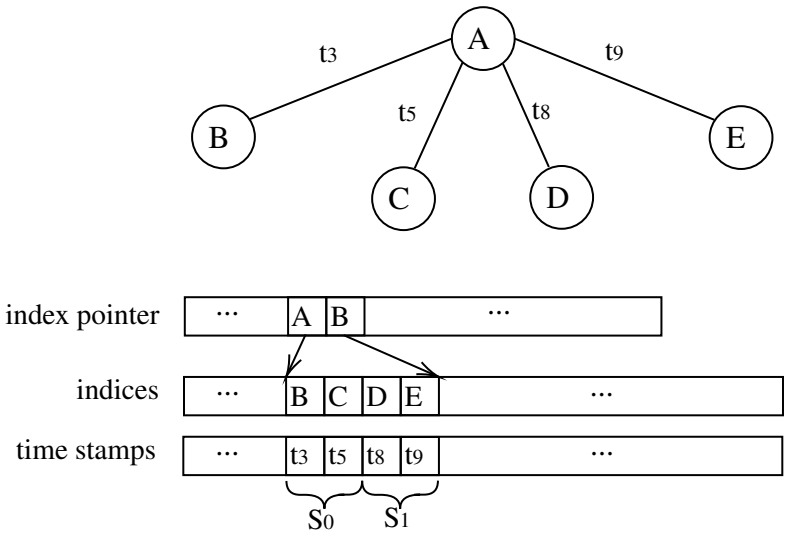

Figure 5.1: T-CSR example, adapted from TGL paper

time. The total space complexity is $O(2|E| + (n+2)|V|)$, since we have the indices and timestamp array of size $|E|$, $n+1$ pointers for all V nodes, as well as the index pointer array of size $|V|$. Let us briefly examine how the temporal sampler works through pseudocode.

Algorithm 1: Temporal Sampler, adapted from TGL paper

Input: T-CSR graph G, root node array n, root node timestamp array t_n, #layers L, #neighbors sampled per layer k_l, #snapshots S,
Output: Indices for sampled nodes

```
for l in [0,L] do
    for s in [0,S] do
        set n and t_n arrays to values from sampled neighbors in previous layer l − 1
        foreach timestamp t in t_n do
            if t between snapshot S_s then
                set pointer p_t to the index of t
            else
                set pointer p_t to end of S_s
            end
        end
        foreach node in n do
            sample k_l neighbors in each snapshot S_s up to p_t
        end
    end
end
```

We loop through each layer and snapshot we want to sample. In each snapshot, we sample k_l neighbors up to pointer p_t, which signifies the largest index in the snapshot which occurs at a smaller timestamp than the node being sampled. We can use this sampler for non-snapshot based TGNNs, since time-then-graph TGNNs can be viewed

as having exactly 1 snapshot which includes all the timestamps in the temporal graph. This temporal sampler was implemented in TGL with around 400 lines of C++.

To simulate how caching strategies will affect the cache hit rate, we pass in the last $k\%$ of the edges in the data as input into the sampler. While we can use this sampler to simulate GNN serving by evaluating it on previous edge interactions, we cannot perform actual serving on new edges, as we should be adding them onto our graph after the request. However, since the sampler stores the graph in a CSR format, it is there is no efficient way to add the edge without rebuilding the graph each time.

Finally, regarding static caching, unlike the dynamic caching strategies we need to first compute the out-degree ranking in order to pre-fill the cache. Assuming we start sampling from the last $x\%$ of edge interactions in the dataset, we use all the edge interactions from 0 to $x\%$ to create the out-degree ranking. That is, if we use the last 5% of edges (start sampling at 95% of data) we use edge interactions from $0-95\%$ to create the ranking. In a later section we explore how the range and time frame of edge interactions chosen to create the ranking affects the static cache hit ratio.

5.2 Factors that affect Cache Hit Rate

5.2.1 Cache Size

We vary the cache size from caching only 2.5% of all the node features to caching 80%.

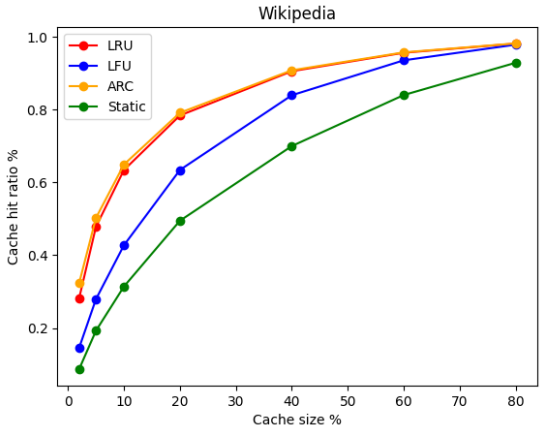

Figure 5.2: Wikipedia cache performance

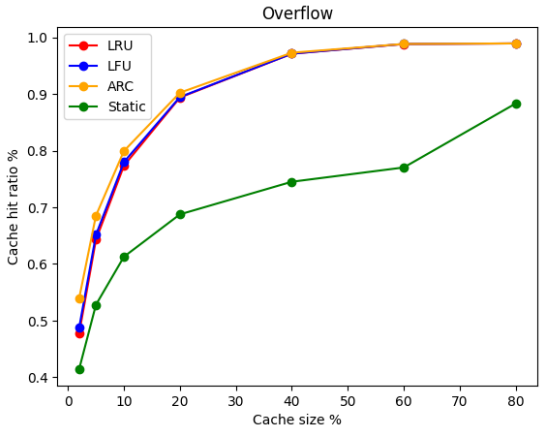

Figure 5.3: Overflow cache performance

We note the following observations from Figure 5.4 and 5.5:

- Cache hit ratio increases as the cache size increases. This is intuitive, as when the cache can store more values there is a higher probability of a cache hit.
- Static caching performs worse than the other dynamic caching strategies for each cache size. This suggests that the sampling pattern of these two temporal graphs cannot be effectively captured by caching the nodes with highest out-degree.
  - Even though the Spearman coefficient for Overflow was above 0.5, signifying a positive correlation between out-degree ranking and the ranking

of sampled nodes, compared to the other caching techniques it still performs worse. Nonetheless this coefficient may still be used to evaluate the effectiveness of static caching between different datasets. In fact, in the 4 datasets we notice an exact correlation between the coefficient ranking and the average cache hit rate for static caching.

– In Table 3.2 Wikipedia and Overflow datasets had the highest coefficient of variation, with a value of 2.2 and 2.01 respectively. This signifies the datasets having bursty behavior, which could explain why LRU performs well. This could also be an explanation to why static cache performs poorly.

- As hypothesized, ARC performs the best. ARC was successfully able to dynamically adapt its size to reflect the dominant sampling pattern in the datasets. For example, in the Wikipedia dataset it is clear to see the access pattern of the sampled edges favours LRU caching.

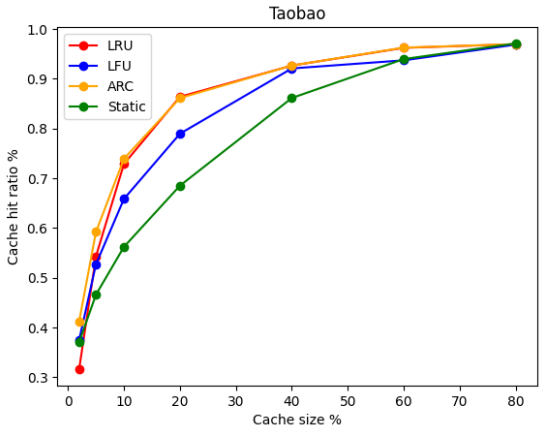

Figure 5.4: Taobao cache performance

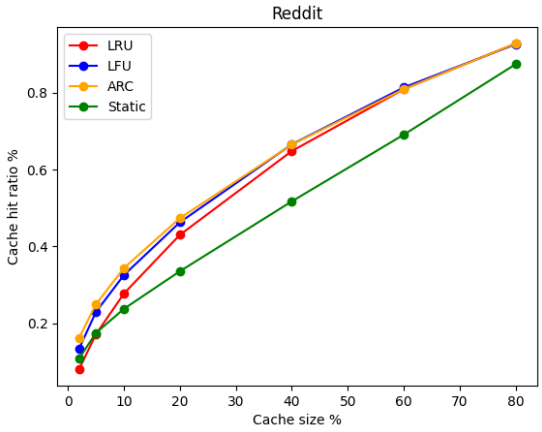

Figure 5.5: Reddit cache performance

In Figures 5.4 and 5.5 we notice a similar trend where static caching performs the worst and ARC performs the best. Reddit is the only dataset where LFU performs better than LRU. This could be explained by Reddit having the lowest coefficient of variation, the temporal metric found in section 3.2.2, which suggests that Reddit's sampling pattern is not as temporal/bursty as the other datasets.

The two custom caching ranking strategies discussed in section 4.2.3 were not included in the graphs, given the results weren't extremely significant.

- Our first strategy used the out-degree ranking as an eviction policy. This resulted in a performance nearly identical to the static cache. This implies this strategy quickly filled the cache with the same ranking as the static cache, and was not able to benefit from dynamic replacement.

- Our second strategy used the number of neighbors sampled by each node as an eviction policy. It performed poorly for small cache sizes, and for dataset Reddit and Taobao it performed slightly better than static for a cache size $\geq 40\%$. We chose to sample 50 neighbors in the first layer and 25 neighbors in the second.

The poor performance could be because of the chosen neighbor sampling sizes, which is a hyperparameter that needs to be tuned.

In our graphs ARC is the most effective caching strategy. Moreover, the popular static out-degree caching strategy is shown to be the least effective. Nevertheless, given static cache's ease of implementation and low cache query latency, it is an important strategy to consider. We later consider different centrality measures as a policy for cache rank construction in static cache's and evaluate if this results in an increase in performance. When implementing strategies it is important to benchmark the caches overhead in an end-to-end GNN inference/training scenario, as having a high cache hit ratio is not the only component in improving cache efficiency. We leave this for future work.

5.2.2 Sampling Size

We use the neighbor sampling sizes of [1], [10], [5,2], [10,2], [5,5], [10,5], [10,10], where the i^{th} element in the array represents the number of neighbors sampled in the i^{th} layer. We set the cache size to 20% for all sizes. In the figures below we notice there is not a shared general relationship between the cache hit ratio and the number of neighbors sampled between all datasets.

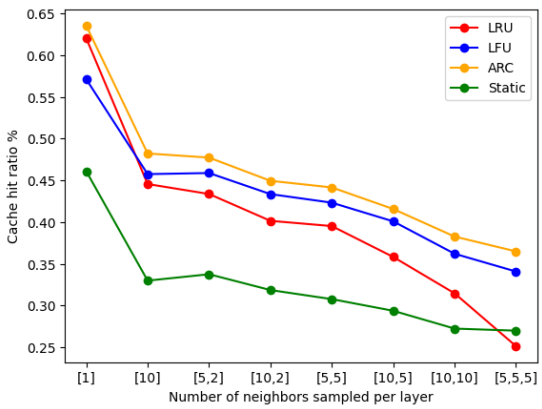

Figure 5.6: Reddit cache performance

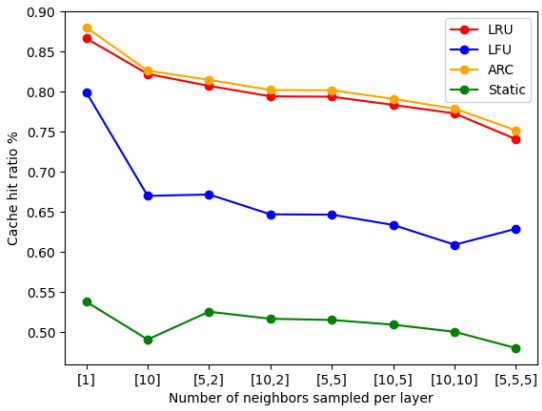

Figure 5.7: Wikipedia cache performance

For the Reddit and Wikipedia dataset there is a general negative correlation between number of neighbors sampled and cache hit ratio. Sampling more neighbors increases the probability that new nodes will be sampled in each layer. For these two datasets the negative correlation indicates the new nodes that are sampled are not creating more cache hits. We also notice static caching has the worst performance. Moreover, from these graphs we can notice similar patterns to the section on cache size, where LFU is also the second-best strategy after ARC for the Reddit dataset.

In Overflow and Taobao below we notice a different pattern. Compared to Reddit and Wikipedia both datasets have a higher average cache hit rate (both above 0.7) for static caching. This could be explained by the Spearman ranking coefficient introduced in section 3.3.2, which compared the out-degree and the node sampling ranking, as Overflow and Taobao had on average the two highest coefficients.

For Overflow there seems to be a general increase in cache hit ratio for static caching as we increase the number of neighbors sampled. This could also be explained by the Spearman coefficient graph 3.5, where we observed for Overflow the coefficient increased as the number of neighbors sampled increases. We also found the coefficient to be low and unchanging as we increased the sampled neighbors for Reddit and Wikipedia, which could partially explain their poor and decreasing cache hit performance above.

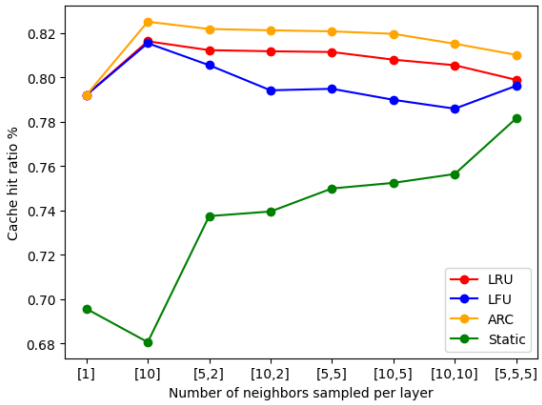

Figure 5.8: Overflow cache performance

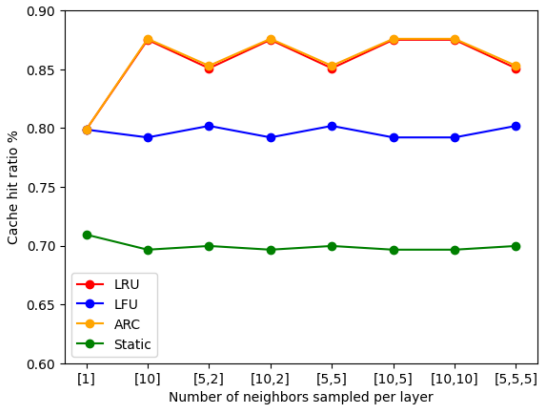

Figure 5.9: Taobao cache performance

5.2.3 Ranking construction for static cache

Since static caching is a popular strategy used in existing GNN systems, let us explore how the construction of the out-degree ranking affects the cache hit performance in temporal graphs. During the cache evaluation in the sections above, the static cache out-degree rank construction used 0 to $x\%$ of the edge interactions in the dataset, where x is the percentage of interactions from the end of the dataset used to perform sampling. However, in a realistic scenario as we continue to process new edge interactions, our out-degree ranking will gradually become ”stale”, as our ranking will not have taken into account the current edge interactions, which continuously change the graph connectivity.

Since both the start and end of when we collect edge interactions (starting at 0 or ending at x in the example above) are hyperparameters that we can set, let us explore how they affect the cache hit ratio. To do so let us answer two questions:

1. How long can we go without updating the out-degree rank, and how will it affect the cache hit ratio? Concretely, if we sample the last $x\%$ of edge interactions and calculate our rank using edges from 0 to $x-k\%$ in the dataset, how does k affect the hit ratio?

2. How far back into our history of edge interactions do we need to consider to still achieve a good hit ratio? If we sample the last $x\%$ of edge interactions and calculate our rank using edges from $x-k$ to $x\%$ how does k affect the hit ratio?

Our first question examines how stale our ranking can get, while our second question examines how much edge interaction history we need to consider.

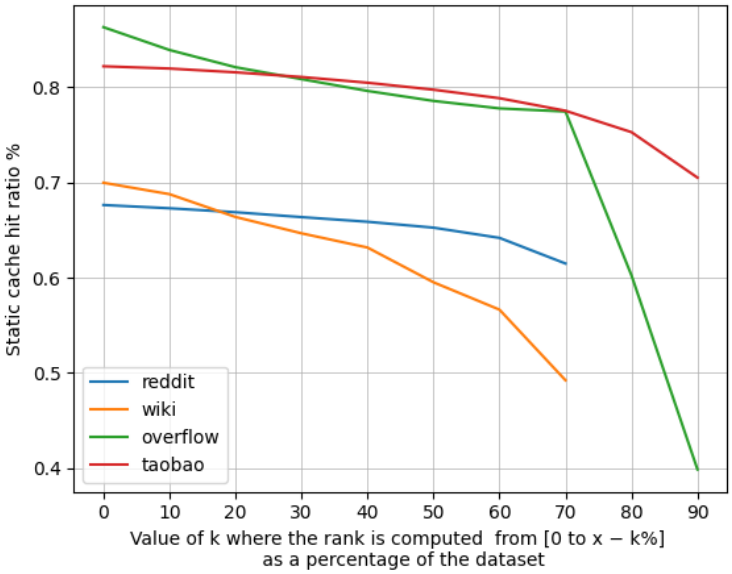

Figure 5.10: Calculate rank using interactions from 0 to $x - k\%$

Figure 5.10 shows how the cache hit ratio is affected when we start sampling from $x\%$, and calculate the ranking using edge interactions from 0 to $x - k\%$. The hit ratio is the highest when $k = 0$. This fits our intuition; the out-degree ranking is least stale when we use all the edge interactions so far. As we increase the value of k, we effectively increase the ranking staleness, and thus we see a degradation in hit ratio. We notice a gradual drop-off in hit ratio for all datasets, except Overflow, which drops off sharply after $k = 70$. This signifies the overflow ranking is more sensitive to the recent edge interactions. The lines for Reddit and Wikipedia stop at 70% since x was chosen to be 80% in order to have a large enough sample size, given the relatively small size of the datasets.

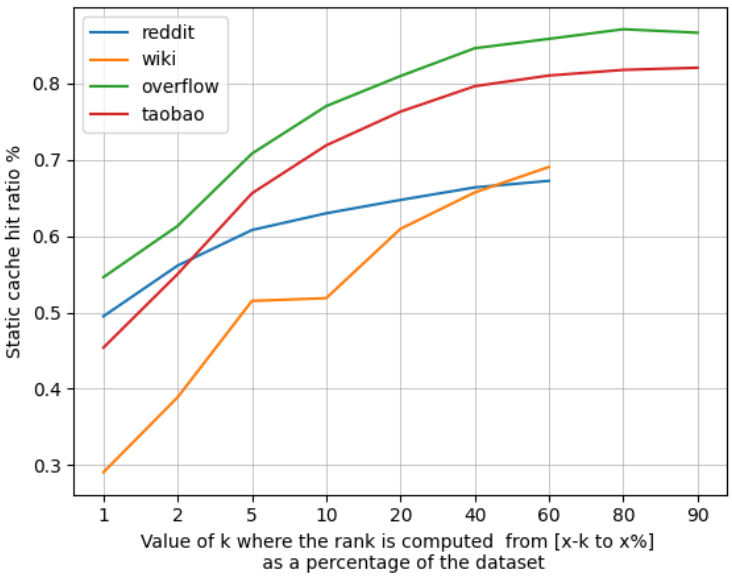

Figure 5.11: Calculate rank using interactions from $x - k\%$ to $x\%$

In Figure 5.11 the hit ratio increases the larger the value of k is. There seems to be a sharp rise as k initially increases, but gradually plateaus off when k gets sufficiently large. This can be used as an evaluation metric to how much past edge interactions we need to save until we get a satisfactory hit ratio.

These graphs answer the question of how staleness affects the hit ratio, and how far back we should start when collecting data to make the out-degree rankings. Finding out when the cache ranking should be updated and deciding how much history to save may be important properties to consider when evaluating GNN caching systems.

5.3 Static Caching Centrality Metrics

In section 4.1.2 we proposed 3 centrality metrics to compare against the default centrality metric of using the out-degree. While we were able to use the basic algorithm to calculate closeness centrality and betweenness centrality for our smaller temporal graphs (Reddit and Wikipedia) provided by NetworkX, a python package to analyze graphs, our large scale temporal graphs could not be processed in a reasonable amount of time. This was because of the large time complexity of the two original algorithms - $O(n^3)$ for closeness and $O(n^2 \cdot logn)$ for betweenness.

Fortunately, more efficient algorithms have been proposed[9] which approximate the two centrality measures. What is more, they have already been implemented on ArangoDB, an open-source graph database system. We thus imported the Overflow and Taobao datasets into a ArangoDB server and ran the two algorithms, called effective closeness and LineRank, for betweenness centrality.

- Effective closeness approximates the average length of the shortest path between a node and all other nodes by iteratively estimating how many shortest paths pass through the node.
- Linerank approximates betweenness centrality by using random walks through the graph to find the probability of visiting a specific node.

We evaluate how well each centrality metric performs when it is used to pre-populate the static cache. For the figures below we use a neighborhood sampling size of $[10, 5]$.

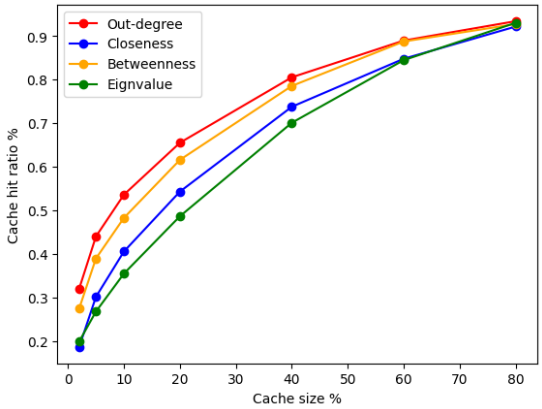

Figure 5.12: Reddit cache performance

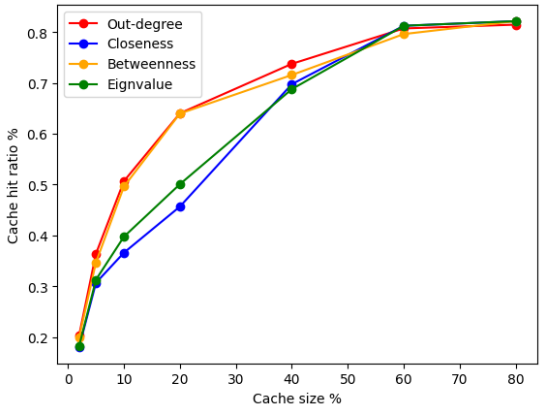

Figure 5.13: Wikipedia cache performance

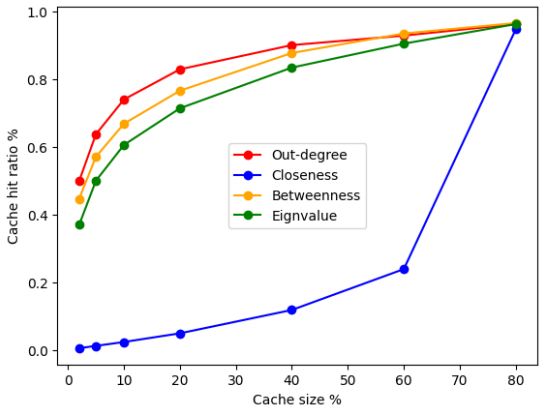

Figure 5.14: Overflow cache performance

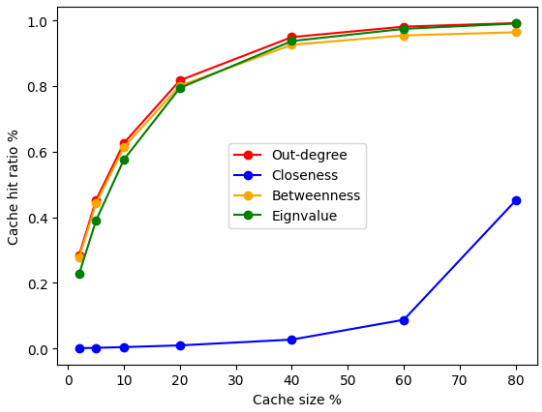

Figure 5.15: Taobao cache performance

In the figures above out-degree centrality achieved the highest hit ratio in all the datasets. The second best was betweenness centrality. For the datasets Overflow and Taobao, there seems to be an approximation error while using ArangoDB's effective closeness algorithm, leading to the poor performance. This evaluation suggests for a neighborhood sampling size of $[10, 5]$, for any cache size static cache ranking achieves the best performance in comparison to the other centrality metrics. This suggests learning the relationship between a node and all other nodes in the graph is not beneficial when performing node caching in temporal graphs.

5.4 TGNN Sampling Strategy

There are many different types of TGNN models that are used on temporal graphs, and moreover the types of sampling strategies employed by different TGNNs also vary. These design choices influence how effective our feature cache will be. Let us examine how our choice in TGNN sampling strategies will affect our cache hit ratio. We will focus on a common sampling strategy that various TGNN models choose between, which is to either sample neighbors uniformly from all the past edge interactions, or sampling the most recent neighbors using the k most recent edge interactions.

TGNN models such as TGN have found that sampling by recency rather than uniformly improves both its model accuracy as well as the time spent per training epoch. Using the T-CSR, recency sampling can be implemented straightforwardly by taking the k leftmost nodes in each snapshot in the indices array. Let us evaluate how these two sampling strategies affect the total amount of nodes sampled, as well as how it affects the number of cache misses for each of our caching strategies.

In Figure 5.16 we have plotted the number of cache misses as a ratio of the total number of nodes sampled, indicated by the first bar on the left. We used the sampling configuration that achieved the best performance in TGL's paper, where we sampled 1 layer with 10 neighbors. The values were scaled to be a fraction of the total number of neighbors sampled using the uniform sampling method, shown by the first blue bar on the left, with a value of 100.

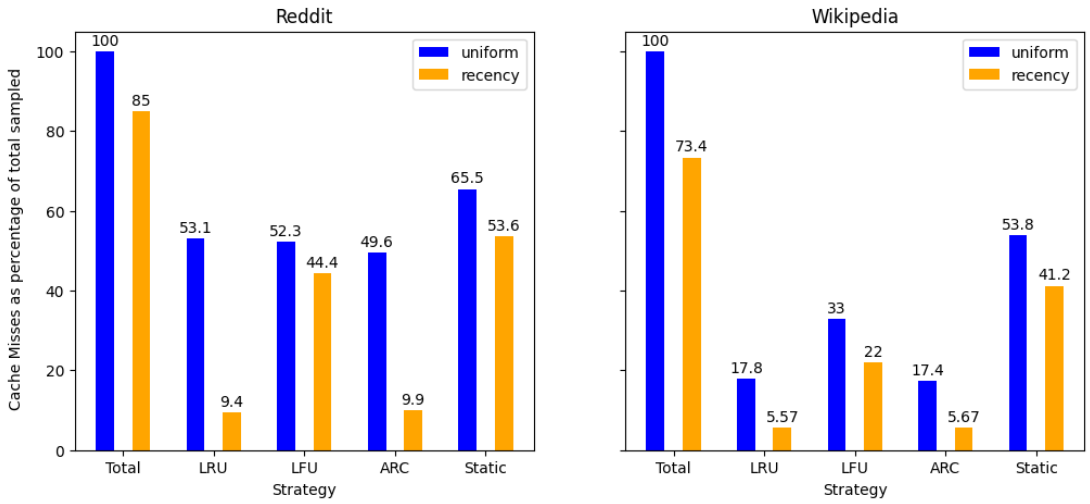

Figure 5.16: Cache miss rate comparison between uniform and recency sampling

Reddit and Wikipedia show a similar pattern when comparing the two strategies:

- The total amount of nodes sampled is greater when we use uniform sampling. In fact Reddit and Wikipedia sample 15% and 26.4% less nodes when using recency sampling, respectively.
- LRU performs the best when sampled by recency. This suggests caching strategies that effectively capture the temporal locality perform the best. Moreover, in the Reddit dataset we note that compared to uniform sampling, LRU performs significantly better, where it produces 5.6 times less cache misses.
- The adaptive nature of ARC caching strategy is also illustrated; for recency sampling ARC manages to adapt to the sampling pattern and produce a similar low number of cache misses like the LRU.
- Static caching achieves the worst miss ratio in both recency and uniform sampling.

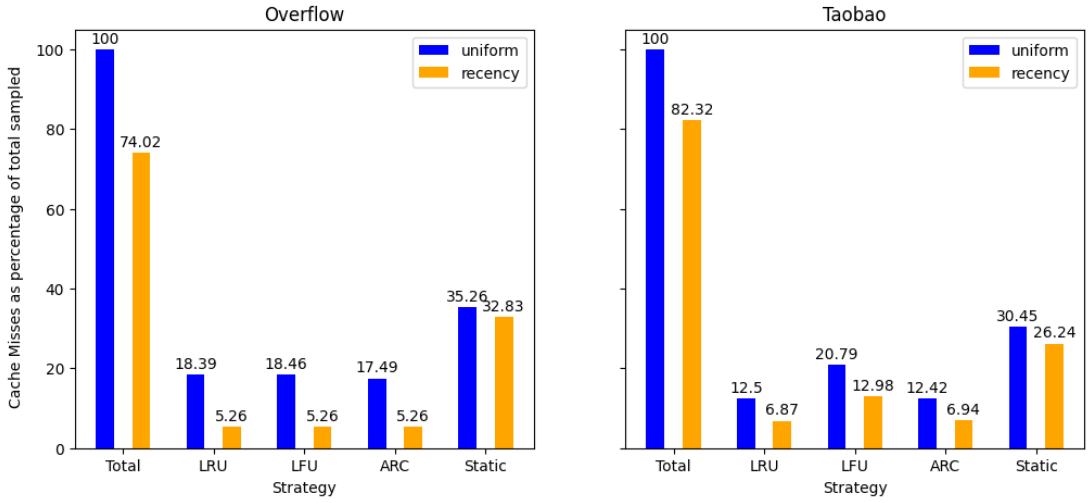

Figure 5.17: Cache miss rate comparison between uniform and recency sampling

For Overflow and Taobao datasets we notice a similar pattern to Reddit and Wikipedia. However there are two main differences.

- We notice uniform sampling in these two datasets results in a relatively low cache miss rate compared to the Reddit dataset, where uniform sampling results in poor cache performance, given more than half of the requests are cache misses.
- Switching from uniform to recency sampling does not result in as significant of a decrease in cache misses as the Reddit dataset. For Overflow and Taobao datasets switching to recency sampling results in 2.86 and 1.6 times less cache misses, compared to 5.6 for Reddit. This suggests for these datasets sampling from recent neighbors does not increase the sampling locality as much as the Reddit dataset.

Overall, we notice from a feature caching perspective, using recency sampling benefits the cache hit ratio. We observe a decrease in cache misses for all 4 datasets in all the caching strategies we evaluated when switching from uniform to recency sampling. Nevertheless, we must be aware feature caching is only one component in a GNN training system, and thus we must evaluate the trade-offs (such as in model accuracy) that any decision may potentially make.

Chapter 6

Conclusion

6.1 Results and Evaluation

We started by analyzing the temporal and topological properties of our 4 temporal graphs.

- To evaluate how temporal a graph was we proposed the burstiness metric which was found using the coefficient of variation. Using this metric, we found that Wikipedia and Reddit were the most and least bursty datasets.
- For topological properties, we evaluated correlation between the graph's out-degree and the sampling ranking through Spearman's ranking coefficient, and also proposed a metric which measures the average distance between consecutively sampled edges.

  While the average distance metric of our datasets was similar, varying from 2.95 to 3.6, and we reasoned that this metric could in general be used as a rough estimate for how much locality as graph has. Through the experiments conducted in Chapter 4 we found that the Spearman's ranking coefficient of the datasets correlated perfectly in ranking with the average cache hit ratio for static out-degree caching.

We then proposed and analyzed various caching strategies, including popular ones such as out-degree caching, LRU, and LFU. New caching strategies proposed included:

- Using other centrality measures as a metric for pre-populating the static cache.
- Extending static out-degree caching by making it dynamic; we proposed using the out-degree ranking as an eviction policy, such that we evict the node in the cache with the lowest out-degree.
- Using the number of neighbors sampled from each unique node as a ranking for the eviction policy. We hypothesized nodes with a larger number of neighbors when sampled should be more important.
- Using ARC caching. Since our graph's access patterns constantly change, we hypothesize ARC is able to continuously adapt to the graph's sampling patterns.

Lastly, we evaluated the hit ratios of the caching strategies on the temporal graphs. We found that using ARC caching resulted in the best cache hit ratio, regardless of the cache size or number of neighbors sampled. The other caching strategies that were proposed didn't perform as well. We also found that static out-degree caching, which is a popular choice for current GNN systems, performed the worst when caching using a temporal sampler. Given the large number of variables that affected the cache hit rate, it was challenging to justify and explain to what extent the graph properties we found affected the hit rate.

We concluded our evaluation with some experiments on static caching and temporal sampling parameters. We found that static caching achieved the best performance when we use the most recent edge interaction history to construct the out-degree ranking. Moreover, the more history we used the higher the cache hit ratio, however the ratio plateaus once a certain amount of history is used. We also found that in temporal sampler's, recency sampling produced less cache misses than uniform sampling.

6.2 Future Work

- In addition to considering metrics which describe the temporal and topological features of graphs separately, we can evaluate metrics that consider temporal and topological features together, such as the temporal motifs of graphs. Temporal motifs[21] are defined as a consecutive set of timestamped edges in which a specific pattern of edge interactions occur. Recognizing and quantifying these types of patterns may be useful as a heuristic when designing new caching strategies.
- When running experiments to evaluate different caching strategies, we noticed some runs took a long time to run. An interesting direction to explore would be to evaluate the latency's incurred by the different components of our system, for example the temporal sampler or specific caching strategies.
- Apart from individual components, we could also evaluate the latency of an end-to-end GNN system using our caching strategies, to see how well they work together.
- Further explore how specific TGNN models affect the cache hit ratio. We could for example evaluate how snapshot based TGNN's and time-then-graph TGNN's differ in their sampling patterns.
- Since our cache hit ratio can be improved by increasing the cache size, we could explore how to combine mixed-precision training or quantization to reduce the size of embedding features so that more vertices can be stored in the cache.
- Implement a 3-layer caching system between SSD, main memory, and GPU to analyze how different caching strategies could potentially be combined and applied for different layers of the cache.

Bibliography

[1] Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/.

[2] Taobao Behavior Data from Taobao for Recommendation. https://tianchi.aliyun.com/dataset/649.

[3] Pior Bastida. Adaptive replacement cache implementation in Python. https://gist.github.com/pior/da3b6268c40fa30c222f.

[4] John Cooper. The poisson and exponential distributions. https://neurophysics.ucsd.edu/courses/physics_171/exponential.pdf.

[5] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. https://arxiv.org/abs/1903.02428, mar 6 2019.

[6] Jianfei Gao and Bruno Ribeiro. On the Equivalence Between Temporal and Static Equivariant Graph Representations. https://proceedings.mlr.press/v162/gao22e.html, jun 28 2022.

[7] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. https://arxiv.org/abs/1704.01212, apr 4 2017.

[8] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. https://arxiv.org/abs/1706.02216, jun 7 2017.

[9] U Kang, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong. Centralities in Large Networks: Algorithms and Observations. https://www.cs.cmu.edu/˜ukang/papers/CentralitySDM2011.pdf.

[10] Taehyun Kim, Changho Hwang, and KyoungSoo . Accelerating GNN training with locality-aware partial execution. https://dl.acm.org/doi/10.1145/3476886.3477515, aug 24 2021.

[11] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. https://arxiv.org/abs/1609.02907, sep 9 2016.

[12] Qingkai Kong and Timmy Siauw. *Python Numerical Methods*. Elsevier, 2020.

[13] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. https://arxiv.org/abs/1908.01207, aug 3 2019.

[14] Renaud Lambiotte, Lionel Tabourier, and Jean-Charles Delvenne. Burstiness and spreading on temporal networks. https://arxiv.org/abs/1305.0543, may 2 2013.

[15] Yunjae Lee, Youngeun Kwon, and Minsoo Rhu. Understanding the Implication of Non-Volatile Memory for Large-Scale Graph Neural Network Training. https://ieeexplore.ieee.org/document/9530364, sep 6 2021.

[16] Jure Leskovec. Graph Neural Networks. https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/graph-neural-networks.

[17] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. Glist: Towards In-Storage Graph Learning. https://www.usenix.org/conference/atc21/presentation/li-cangyuan.

[18] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. https://dl.acm.org/doi/10.1145/3419111.3421281.

[19] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl: Gpu-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. https://arxiv.org/abs/2112.08541, dec 16 2021.

[20] Hehuan Ma, Yu Rong, and Junzhou Huang. Graph neural networks: Scalability. In Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao, editors, *Graph Neural Networks: Foundations, Frontiers, and Applications*, pages 99–119. Springer Singapore, Singapore, 2022.

[21] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in Temporal Networks. https://arxiv.org/abs/1612.09259, dec 29 2016.

[22] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs for AAAI 2020. https://research.ibm.com/publications/evolvegcn-evolving-graph-convolutional-networks-for-dynamic-graphs.

[23] Yeonhong Park, Sunhong Min, and Jae W. Lee. Ginex: Ssd-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching. *Proceedings of the VLDB Endowment*, 15(11):2626–2639, 7 2022.

[24] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. https://arxiv.org/abs/2104.07857, apr 16 2021.

[25] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal Graph Networks for Deep Learning on Dynamic Graphs. https://arxiv.org/abs/2006.10637, jun 18 2020.

[26] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A Gentle Introduction to Graph Neural Networks. *Distill*, 6(9), sep 2 2021.

[27] Debabala Swain, Bijay Paikaray, and Debabrata Swain. Awrp: Adaptive Weight Ranking Policy for Improving Cache Performance. https://arxiv.org/abs/1107.4851, jul 25 2011.

[28] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. https://arxiv.org/abs/1909.01315, sep 3 2019.

[29] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. https://arxiv.org/abs/1806.01973, jun 6 2018.

[30] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A General Framework for Temporal GNN Training on Billion-Scale Graphs. https://arxiv.org/abs/2203.14883, mar 28 2022.

[31] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A Comprehensive Graph Neural Network Platform. https://arxiv.org/abs/1902.08730, feb 23 2019.