
# OUTRE: An OUT-of-core De-REdundancy GNN Training Framework for Massive Graphs within A Single Machine

Zeang Sheng*
Peking University
shengzeang18@pku.edu.cn

Wentao Zhang◇
Peking University
wentao.zhang@pku.edu.cn

Yangyu Tao
Tencent Inc
brucetao@tencent.com

Bin Cui*†
Peking University
bin.cui@pku.edu.cn

## ABSTRACT

Sampling-based Graph Neural Networks (GNNs) have become the de facto standard for handling various graph learning tasks on large-scale graphs. As the graph size grows larger and even exceeds the standard host memory size of a single machine, out-of-core sampling-based GNN training has gained attention from the community. For out-of-core sampling-based GNN training, the performance bottleneck is the data preparation process that includes sampling neighbor lists and gathering node features from external storage. Based on this observation, existing out-of-core GNN training frameworks try to accomplish larger percentages of data requests without inquiring the external storage by designing better in-memory caches. However, the enormous overall requested data volume is unchanged under this approach. In this paper, we present a new perspective on *reducing the overall requested data volume*. Through a quantitative analysis, we find that *Neighborhood Redundancy* and *Temporal Redundancy* exist in out-of-core sampling-based GNN training. To reduce these two kinds of data redundancies, we propose OUTRE, an OUT-of-core de-REdundancy GNN training framework. OUTRE incorporates two new designs, *partition-based batch construction* and *historical embedding cache*, to reduce the corresponding data redundancies. Moreover, we propose *automatic cache space management* to automatically organize available memory for different caches. Evaluation results on four public large-scale graph datasets show that OUTRE achieves 1.52× to 3.51× speedup against the SOTA framework.

**PVLDB Reference Format:**
Zeang Sheng, Wentao Zhang, Yangyu Tao, Bin Cui. OUTRE: An OUT-of-core De-REdundancy GNN Training Framework for Massive Graphs within A Single Machine. PVLDB, 17(11): 2960-2973, 2024.
doi:10.14778/3681954.3681976

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/PKU-DAIR/OUTRE.

*School of CS & Key Lab of High Confidence Software Technologies, Peking University
◇Center for Machine Learning Research, Peking University
†Institute of Computational Social Science, Peking University (Qingdao)

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681976

## 1 INTRODUCTION

Graph Neural Networks (GNNs) are becoming increasingly popular in academia and industry, and are widely used on various graph learning tasks, including node classification [25, 28, 33, 44, 52, 58], link prediction [4, 50, 57], recommendation [1, 11, 18, 30, 51, 53], graph clustering [8, 14, 59], and drug discovery [17, 24, 36, 49]. During training, GNNs typically fetch the $L$-hop neighborhood for each node to complete its computation, where $L$ is the model depth. These fetching operations incur extremely high memory cost when training on large-scale graphs. To address this issue, researchers propose sampling-based GNNs [6, 9, 16, 55] that randomly sample a subset of neighbors for each node at each layer instead of utilizing the entire neighborhood. Evaluations show that sampling-based GNNs have significantly higher efficiency and smaller memory cost while maintaining similar convergence accuracy with GNNs trained on full neighborhoods [16].

Recently, the size of graph datasets have grown to several hundred GBs and even exceeded one TB [20, 27]. Although the memory footprint of sampling-based GNNs can be small by tuning the batch size and the sample size, naïvely fetching neighborhood information still requires the in-memory existence of the entire graph. A straightforward solution to the memory scarcity issue is to extend the graph storage to multiple machines and train GNNs distributely. However, previous work [32] found that the performance bottleneck of sampling-based GNN training is the data preparation process that issues a great number of data requests. On the other hand, the computing devices (e.g., GPUs) are often underutilized during training. Thus, scaling sampling-based GNN training to multiple machines is not economical. Besides, the additionally introduced communication cost may further exacerbate the underutilization issue of computing devices.

With the fast development of storage technologies, Solid State Drivers (SSDs) can now achieve multiple GBps of sequential bandwidth and their prices have dropped significantly recently. Therefore, some researcher have begun to explore the possibility of incorporating SSD into sampling-based GNN training. We refer to the attempts that extend the memory space to external storage (e.g., disk, SSD) as "out-of-core" executions. However, SSDs are still several magnitudes slower than host memory, especially for random accesses. Thus, naïvely fetching $L$-hop neighborhoods from external storage will further slow down the already dominating data preparation process and deteriorate the final training performance. Targeting reducing the percentage of data requests that inquire the

external storage, an existing framework [39] uses the host memory to cache important graph data. Specifically, it adopts a dual-cache approach that utilizes a neighbor cache to cache neighbor lists and a feature cache to cache node features. This dual-cache approach has achieved significant speedup against naïve baselines.

The main focus of this dual-cache approach is to explore how to accomplish more data requests through the caches in host memory. However, the overall requested data volume is unchanged, and how to reduce it remains to be explored. In this paper, we present a new and a more fundamental perspective: *reducing the overall requested data volume*. Through a quantitative analysis, we find that two kinds of data redundancies exist in out-of-core sampling-based GNN training. 1) *Neighborhood Redundancy:* Plenty of nodes are redundantly included in the sampled $L$-hop neighborhoods of many different training batches as one node can be connected to many different batches' training nodes. 2) *Temporal Redundancy:* Calculating the exact node embeddings at every training iteration incurs many unnecessary data requests as most embeddings only experience modest changes across most training iterations.

We propose OUTRE, an OUT-of-core de-REdundancy GNN training framework that aims to *reduce the overall requested data volume* by reducing two kinds of data redundancies. OUTRE is built on the dual-cache approach and presents three new designs: 1) To reduce *Neighborhood Redundancy*, we propose constructing training batches with min-cut graph partitions. By decreasing links between different batches' training nodes, the overlapped neighborhood size can be reduced. We make profound performance optimizations to one streaming graph partitioning algorithm [43] and achieve efficient out-of-core graph partition. 2) Motivated by previous work [13, 21], we approximate node embeddings with their histories to reduce *Temporal Redundancy*. We only cache second-layer node embeddings to strike a balance among memory cost, training efficiency and accuracy. 3) To alleviate exhausting tuning efforts, we present an automatic cache space management module to help OUTRE adapt to various datasets and hardware configurations. Specifically, we add a profiling epoch ahead of training and collect the approximated I/O savings for different caches. A low-cost search process then finds the optimal memory split scheme.

Our contributions can be summarized as follows:

- (1) *New Findings.* We present a new design perspective, *reducing the overall requested data volume*, for out-of-core sampling-based GNN training and locate two kinds of data redundancies, *Neighborhood Redundancy* and *Temporal Redundancy*, through a quantitative analysis.
- (2) *New Framework.* We propose a new out-of-core sampling-based GNN training framework, OUTRE. Inside the framework, we present *partition-based batch construction* and *historical embedding cache* to reduce the corresponding two data redundancies. We further propose *automatic cache space management* to automatically generate decent memory split schemes for the caches with real-world profiling.
- (3) *SOTA Performance.* We compare OUTRE with the SOTA out-of-core sampling-based GNN training framework on four public large-scale graph datasets. The evaluation results illustrate that OUTRE achieves 1.52× to 3.51× speedup and maintains comparable convergence accuracy.

# 2 BACKGROUND

## 2.1 Graph Neural Networks

Graph Neural Networks (GNNs) are neural networks operating on graph-structured data where each node is associated with a feature vector. GNNs can capture valuable knowledge within each node's neighborhood other than only the node itself. Thus, GNNs outperform traditional deep learning methods (e.g., MLP) that cannot utilize neighborhood information on various graph learning tasks.

An $L$-layer GNN is constituted by $L$ consecutive GNN layers. The operations on node $v$ of the $l$-th ($1 \le l \le L$) GNN layer can be formulated as follows:

$$\mathbf{h}_v^l = update\left(\mathbf{h}_v^{l-1},\ aggregate\left(\left\{\mathbf{h}_u^{l-1} \mid u \in \mathcal{N}_v\right\}\right)\right), \tag{1}$$

where $\mathcal{N}_v$ is the neighbor set of node $v$. The *aggregate* function aggregates knowledge from node $v$'s neighborhood. The aggregated knowledge and node $v$'s own feature are fed into the *update* function. Regarding real-world applications, each node in the graph might have dozens or even hundreds of neighbors. Thus, fetching the collective $L$-hop neighborhood for even a small number of nodes would incur extensive time and memory cost when $L$ is large, which is called the "neighbor explosion" problem [16]. Many researches [6, 9, 16, 55] propose sampling-based GNNs that sample only a subset of the whole neighborhood to alleviate this problem. Sampling-based GNNs are considerably more scalable and efficient on large-scale graphs than GNNs that train on full neighborhoods.

## 2.2 Out-of-core Sampling-based GNN Training Stage Decomposition

As the graphs used for GNN training grow larger [20, 27], using sampling-based GNNs has become the de facto standard for training GNNs on large-scale graphs. The training pipeline of sampling-based GNNs under the typical CPU-GPU hybrid system can be decomposed into four stages: *Sample*, *Gather*, *Transfer* and *Compute*. (1) *Sample*: for a batch of nodes (denoted by $\mathbf{N}_b$), CPU extracts their collective $L$-hop neighborhood (denoted by $\widetilde{\mathbf{N}}_b$) recursively from the graph's adjacency matrix. (2) *Gather*: CPU gathers the node features of $\widetilde{\mathbf{N}}_b$ from the feature matrix and put them into a contiguous memory area. (3) *Transfer*: CPU transfers the $L$-hop neighborhood connections and the gathered features from host memory to GPU. (4) *Compute*: GPU executes the forward and backward operations according to specific GNNs. The *Sample* and the *Gather* stage together are often denoted as *Data Preparation* since they prepare data required by model execution.

When adapting this four-stage training pipeline to out-of-core settings, the training framework needs to access the adjacency matrix and the feature matrix from external storage during *Data Preparation*. The *Transfer* and the *Compute* stage remain unchanged. To find the performance bottleneck of the training process, we extend the popular graph learning library PyG [12] to out-of-core settings by letting it read the memory-mapped feature matrix and indices of the CSR-formed adjacency matrix from SSDs. We keep the indptr of the CSR-formed adjacency matrix in host memory as it requires only $O(\mathbf{N})$ space, where $\mathbf{N}$ is the number of nodes in the graph. The training time decomposition of a 3-layer GraphSAGE [16] with hidden size of 256 and sample size of (10,10,10) on

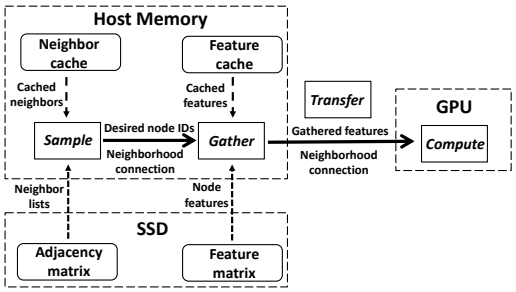



**Figure 1: Overview of a typical dual-cache out-of-core sampling-based GNN training framework.**

two OGB datasets [20] is shown in Table 1. To simulate the out-of-core environment, we limit the available host memory to 4GB and 64GB for ogbn-products and ogbn-papers100M, respectively. The batch size is set to 1,000. Table 1 illustrates that more than 95% of the training time is spent on *Data Preparation*, meaning that *Data Preparation* heavily bounds the GNN training performance.

## 2.3 Existing Out-of-core Sampling-based GNN Training Frameworks

Observed that *Data Preparation* is the performance bottleneck, existing out-of-core sampling-based GNN training frameworks [38, 39, 42, 47] proposes various kinds of designs to accelerate *Data Preparation*. There are mainly three branches of existing out-of-core sampling-based GNN training frameworks. The first branch of work [22, 47] follows previous attempts of out-of-core graph processing systems [31, 45, 46, 61]. They split graphs to partitions and execute only on the in-memory partitions. This design can bring considerable performance improvement yet suffers from accuracy loss when the graph size is significantly larger than the host memory size. The second branch of work [38, 42] explores how to efficiently train GNNs on large-scale graphs when GPU threads can directly access data on SSDs [40]. They remove CPU from the I/O stack and achieve high I/O bandwidth. The third branch of work [39] adopts a dual-cache approach, whose overview is depicted in Figure 1. The neighbor cache and the feature cache stores important neighbor lists and node features in host memory, respectively. They focus on designing better cache policies to accomplish more data requests without inquiring external storage.

We built our framework on the third branch of work. Instead of only trying to increase the percentage of data requests saved from inquiring external storage, we propose a new and more fundamental perspective that explores how to *reduce the overall requested data volume*. Unlike the other two branches, the first branch of work does not utilize full graph data during training. For fairness, we do not compare with it in this paper. The second branch of work is orthogonal to the third branch, and these two branches can be easily combined. Therefore, we compare our framework mainly with the third branch of work in this paper.

# 3 OBSERVATIONS

In this section, we firstly propose a new metric to quantify *the overall requested data volume* for out-of-core sampling-based GNN training. Then, we conduct a quantitative analysis and try to locate two kinds of data redundancies in out-of-core sampling-based GNN training. Finally, we propose two corresponding insights based on the analysis results, which may help to boost training performance.

**Table 1: Training time decomposition on two OGB datasets under out-of-core environments.**

| Training Stages | *Sample* | *Gather* | *Transfer* | *Compute* |
|---|---|---|---|---|
| ogbn-products (4G mem) | 374.61s | 14.57s | 9.49s | 4.98s |
| ogbn-papers100M (64G mem) | 121.03s | 3542.62s | 75.21s | 20.94s |

**Table 2: *Redundancy Ratio* on two OGB datasets.**

| Datasets | #Nodes | #Training | $\sum_{b=1}^{B} \lvert\widetilde{\mathbf{N}}_b\rvert$ | *RR* |
|---|---|---|---|---|
| ogbn-products | 2,449,029 | 196,615 | 93,527,142 | **475.69** |
| ogbn-papers100M | 111,059,956 | 1,207,179 | 142,040,373 | **117.66** |

## 3.1 *Redundancy Ratio*

As mentioned in Section 2.2, the training framework needs to access the adjacency matrix and the feature matrix from external storage during *Data Preparation*. Here, we empirically analyze *the overall requested data volume* during *Data Preparation*. To conduct the analysis quantitatively, we propose a new metric, "*Redundancy Ratio*" (abbreviated as *RR*). *RR* is defined as the sum of the collective $L$-hop neighborhood sizes of all training batches versus the number of training nodes:

$$RR = \frac{\sum_{b=1}^{B} |\widetilde{\mathbf{N}}_b|}{\sum_{b=1}^{B} |\mathbf{N}_b|}, \tag{2}$$

where $\widetilde{\mathbf{N}}_b$ and $\mathbf{N}_b$ is the collective $L$-hop neighborhood set and the training node set for the $b$-th training batch, respectively. $\sum_{b=1}^{B} |\widetilde{\mathbf{N}}_b|$ is positively correlated with the number of sampled neighbors during the *Sample* stage and the number of gathered node features during the *Gather* stage. Thus, $\sum_{b=1}^{B} |\widetilde{\mathbf{N}}_b|$ is a decent indicator for *the overall requested data volume* during *Data Preparation*. Moreover, out-of-core sampling-based GNN training is bounded by *Data Preparation* as shown in Table 1. Therefore, *RR* is also an appropriate metric to reflect the training performance.

To demonstrate how *RR* performs, we train the same GraphSAGE model as in Section 2.2 and report their *RR*s in Table 2. Table 2 shows that the sum of the collective $L$-hop neighborhood sizes ($\sum_{b=1}^{B} |\widetilde{\mathbf{N}}_b|$) would be more than one hundred times greater than the number of training nodes ($\sum_{b=1}^{B} |\mathbf{N}_b|$) and even exceed the graph size.

## 3.2 Neighborhood Redundancy

Most existing GNN training frameworks construct each training batch by selecting training nodes randomly and then sample their collective $L$-hop neighborhood. Randomly selecting training nodes equips the model with higher convergence accuracy. However, the sampled collective $L$-hop neighborhoods of different training batches are massively overlapped since one node may be connected to many different training batches' training nodes. We name this phenomenon that the sampled neighborhoods of different training batches are massively overlapped as "*Neighborhood Redundancy*". Decreasing the overlapped neighborhood size between different

**Table 3: *Redundancy Ratio* reduction brought by partition-based selection on two OGB datasets.**

| Datasets | Random selection | Partition-based selection |
|---|---|---|
| ogbn-products | 475.69 | 321.53 (**-32.41%**) |
| ogbn-papers100M | 117.66 | 86.36 (**-26.60%**) |

**Table 4: *Redundancy Ratio* reduction brought by reusing historical embeddings on two OGB datasets.**

| Datasets | reuse 0% | reuse 10% | reuse 20% |
|---|---|---|---|
| ogbn-products | 475.69 | 365.48 (**-23.17%**) | 322.63 (**-32.18%**) |
| ogbn-papers100M | 117.66 | 103.35 (**-12.16%**) | 85.91 (**-26.98%**) |

training batches is vital to reduce *Neighborhood Redundancy*. Looking at it reversely, this target is equivalent to **increasing the number of edges between the training nodes within each training batch** because the number of edges in the graph is fixed.

To validate this hypothesis, we adopt the min-cut graph partitioning algorithm METIS [26] to partition the graph into 10,000 parts where the number of intra-partition edges is maximized. Then we randomly select several partitions every time, shuffle the training nodes within them, and construct training batches according to the pre-defined batch size value. We change the training node selection approach from the original random selection to the partition-based selection mentioned above and repeat the experiments in Section 3.1. Evaluation results in Table 3 illustrate that the partition-based selection helps reduce *Redundancy Ratio* by over 25% on both datasets. The significant reduction implies that **increasing the number of edges between the training nodes within each training batch** helps to *reduce the overall requested data volume* and thus accelerates out-of-core sampling-based GNN training.

## 3.3 Temporal Redundancy

Motivated by the observation that most embeddings only experience modest changes across the majority of training iterations, existing work [7, 13, 21, 54] tries to approximate node embeddings with histories generated in previous iterations. The optimization space here is that deep learning models are insensitive to small errors. Thus, cautiously approximating the newest results with histories would not harm the model's effectiveness.

Existing work reusing histories targets in-memory GNN training and mainly aims at reducing the computation redundancy. Under the out-of-core setting, **reusing historical node embeddings** helps to reduce also the data redundancy that incurs unnecessary data requests to external storage. We denote this kind of data redundancy by "*Temporal Redundancy*" as it describes a kind of time-level redundancy. For those embeddings that their histories would substitute, the data request volume induced by their sampling and gathering operations is eliminated.

Considering that the host memory space is limited, we implement a historical embedding cache that only reuses embeddings of the second layer. Moreover, we select high-degree nodes as reuse candidates to explore how much requested data volume can be reduced. The evaluation results of the same 3-layer GraphSAGE are provided in Table 4. Table 4 shows that the *Redundancy Ratio* decreases significantly when 20% nodes reuse their historical embeddings. Thus, we hypothesize that **reusing historical node embeddings** can bring decent performance gain to out-of-core sampling-based GNN training.

# 4 FRAMEWORK DESIGN

In this section, we introduce our proposed out-of-core sampling-based GNN training framework OUTRE in detail. Firstly, Section 4.1 provides an overview of OUTRE, where we describe the modifications we make to the conventional four-stage training pipeline and briefly introduce the three main designs of OUTRE: *partition-based batch construction*, *historical embedding cache*, and *automatic cache space management*. Then, we explain these three designs one by one in detail in Section 4.2, 4.3 and 4.4, respectively.

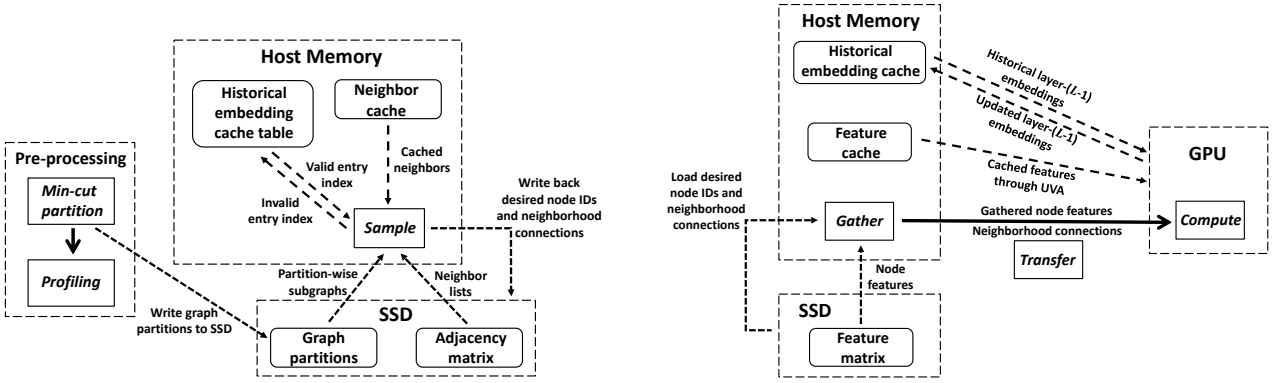
## 4.1 Framework Overview

*4.1.1 Training Pipeline.* We provide a workflow overview of our proposed framework, OUTRE, in Figure 2. OUTRE is built on the existing dual-cache framework (i.e., OUTRE also has the neighbor cache and the feature cache), and it makes two essential modifications to the conventional four-stage training pipeline.

1) A pre-processing stage is added ahead of the conventional pipeline. During pre-processing, OUTRE executes an out-of-core min-cut graph partitioning algorithm to partition the graph into many small parts that are later used to construct training batches. Then, OUTRE runs a profiling epoch that includes only the *Sample* and the *Gather* stage to collect necessary information for the *automatic cache space management* module to determine the optimal memory split scheme for different caches.

2) OUTRE decouples the *Sample* stage from the later three stages following [39]. The conventional four-stage training pipeline executes all the stages consecutively for each training batch. Thus, the neighbor cache (required by the *Sample* stage) and the feature cache (required by the *Gather* stage) have to reside in host memory simultaneously. In contrast, decoupling the *Sample* stage from the pipeline makes it possible for these two caches to occupy the host memory exclusively. Therefore, more graph data can be cached, which leads to potentially higher training performance. To implement the decoupling, OUTRE writes the sampled collective $L$-hop neighborhoods back to external storage after the *Sample* stage. Before the *Gather* stage, OUTRE reads each training batch's sampled results from external storage and executes the same as the conventional pipeline afterwards.

*4.1.2 Main Designs.* The three main designs of OUTRE are (1) *partition-based batch construction*, (2) *historical embedding cache*, and (3) *automatic cache space management*. Firstly, *partition-based batch construction* is proposed to reduce *Neighborhood Redundancy* mentioned in Section 3.2. Each training batch in OUTRE is constructed by several randomly chosen graph partitions generated by the min-cut graph partitioning algorithm during pre-processing. The size of overlapped collective $L$-hop neighborhoods of different training batches can be significantly decreased by increasing intra-batch connections. Therefore, *the overall requested data volume* of the *Sample* and the *Gather* stage is considerably reduced, which accelerates out-of-core sampling-based GNN training.




(a) Pre-processing and *Sample* stage

(b) *Gather*, *Transfer*, and *Compute* stage

**Figure 2: The workflow overview of OUTRE.**

Secondly, aiming to reduce *Temporal Redundancy* observed in Section 3.3, we implement a *historical embedding cache* that is involved in both the *Sample* and the *Compute* stage. During the *Sample* stage, *historical embedding cache* tells the main process of what nodes can be skipped according to the states of the encountered nodes. Then, during the *Compute* stage, GPU pulls corresponding historical node embeddings from *historical embedding cache* and pushes the updated node embeddings to it according to the information inside the *historical embedding cache* table.

Thirdly, we propose *automatic cache space management* to automatically manage the sizes of different caches in OUTRE. This way, OUTRE can achieve robust performance across various datasets and hardware configurations without exhausting tuning efforts. Specifically, OUTRE utilizes the collected information from the profiling epoch during pre-processing to automatically determine the optimal memory split scheme for the feature cache and the *historical embedding cache* in a low-cost manner. The remainder of this section will detail the three main designs of OUTRE.

## 4.2 Partition-based Batch Construction

The conventional random selection approach leads to massive overlap between the collective $L$-hop neighborhoods of different training batches, which we referred to as *Neighborhood Redundancy* in Section 3.2. In OUTRE, we propose constructing training batches according to min-cut graph partitions. Specifically, we first partition the graph into many small parts (e.g., 10,000 parts) using min-cut graph partitioning algorithms. The partitioned subgraphs, including partition-wise adjacency matrices and the full/train/validation/test node IDs within the partition, are written to external storage. OUTRE randomly chooses several min-cut partitions during the *Sample* stage as a *macro batch*. Then, it shuffles the training nodes within these partitions and constructs training batches (*micro batches*) according to the pre-defined batch size value. OUTRE further loads corresponding partition-wise adjacency matrices into host memory as the *batch-specific dynamic neighbor cache*. This way, OUTRE reads from external storage only when the desired neighbor lists cannot be found in both the global neighbor cache and the *batch-specific dynamic neighbor cache*.

Although there is no intrinsic limitation on the choices of the min-cut graph partitioning algorithm in OUTRE, some popular choices like METIS [26] are inappropriate since they are in-memory algorithms and would incur unacceptable memory cost when applied to large-scale graphs (see experimental comparison in Section 6.3). Their high memory requirement disobeys the out-of-core setting where host memory is limited. Therefore, we implement one streaming graph partitioning algorithm, FENNEL [43], in OUTRE. We enable FENNEL to execute in the out-of-core manner by making it read the indices of the CSR-formed adjacency matrix from external storage. We make profound performance optimizations to the original FENNEL algorithm to accelerate its execution on large-scale graphs. More implementation details of the modified FENNEL algorithm in OUTRE can be found in Section 5.2.

Previous researches [32, 39, 60] intuitively cache the neighbor lists of high-degree nodes in the neighbor cache since they are more likely to be included in the collective $L$-hop neighborhoods during the *Sample* stage. However, node degree is no longer an appropriate metric for the neighbor cache in OUTRE. OUTRE tries to maximize the connections within each training batch and additionally treats partition-wise adjacency matrices as the *batch-specific dynamic neighbor cache*. As a result, those high-degree nodes that have many in-batch neighbors are no longer proper candidates for the neighbor cache. Thus, we cache the neighbor lists of nodes with large numbers of *cross-partition neighbors* in OUTRE. This way, the counts for in-batch neighbors are eliminated, leading to more accurate approximations for the requested data volume of the *Sample* stage. The number of *cross-partition neighbors* for each node can be collected during partitioning with little cost.

## 4.3 Historical Embedding Cache

Previous work [7, 13, 21, 54] observes that most embeddings only experience modest changes across most training iterations. Based on this observation, they propose to reuse historical node embeddings as approximations for the newest iteration. Besides the computation redundancy pointed out by previous work, evaluation results in Section 3.3 illustrate that reusing histories can also reduce *Temporal Redundancy* and possibly accelerate out-of-core GNN training.

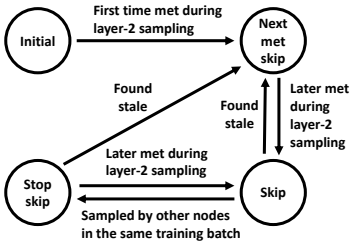



**Figure 3: The FSM for state changes of cache candidates.**

However, previous work caches embeddings for all the layers and all the nodes. These designs induce huge memory cost that is inappropriate for the out-of-core environment. Moreover, most previous work do not proactively evict low-quality historical embeddings, leading to accumulated errors [21]. To suit the characteristics of the out-of-core environment, we present three nontrivial modifications to the original history reusing mechanism [13]. This altered mechanism in OUTRE is denoted by "*historical embedding cache*".

1) OUTRE only caches the historical node embeddings of *the second layer*. Each cached second-layer node can potentially prune an ($L$-1)-depth computation graph. Although caching for the first layer can possibly prune $L$-depth computation graphs, doing so in OUTRE brings little benefit. Due to *partition-based batch construction*, training nodes are highly probable to be sampled by nodes in the same training batch at the first layer. Under such scenario, the originally pruned $L$-depth computation graph shrinks to a 1-hop computation graph as this node's second-layer embedding is required to calculate exact first-layer embeddings for certain nodes. Moreover, the 1-hop sampling and gathering cost has already been greatly reduced by *partition-based batch construction*. Thus, we choose to cache for *the second layer* to strike a balance between the size of possibly pruned computation graphs and the probability of being invalidated by nodes in the same training batch.

2) OUTRE only caches histories for a certain number of *important* nodes. This design makes it possible to constrain the largest possible cache size, which is desirable for managing the limited host memory under the out-of-core environment. In this paper, we propose a new node-wise importance metric that directly records the node-wise ($L$-1)-hop neighborhood size while excluding those cached in host memory. The node-wise ($L$-1)-hop neighborhood size information is collected in the profiling epoch during pre-processing. The details about this new node-wise importance metric and the profiling epoch will be introduced later in Section 4.4.

3) OUTRE proactively invalidates *staled* embeddings in *historical embedding cache* to reduce the accumulated approximation errors during training. A previously cached embedding is considered as *staled* if the current iteration number minus the iteration number it was most recently updated exceeds a pre-set threshold called *staleness*. The idea of using *staleness* to control the quality of cached embeddings comes from [21].

To implement the historical embedding cache in OUTRE, we need to insert new operations during the *Sample* and the *Compute* stage. For the *Sample* stage, during sampling for the second layer, OUTRE checks the node states inside the *historical embedding cache* table and determines whether to execute sampling for certain cache candidates. The Finite State Machine (FSM) illustrating cache candidates' state transformations is shown in Figure 3. OUTRE skips sampling for a cache candidate only when its state is "Skip". For the *Compute* stage, before executing the model's $L$-th layer, OUTRE checks the *historical embedding cache* table and generates two node ID lists indicating what historical embeddings should be pulled from the cache and what updated ones should be pushed to the cache, respectively. Then, OUTRE executes the corresponding pull and push operations and feeds the full input that consists of embeddings from ($L$-1)-th layer's computation and histories pulled from *historical embedding cache* to the model's $L$-th layer. The detailed training pipeline of OUTRE is shown in Algorithm 1 and 2.

**Algorithm 1** The *Sample* stage in OUTRE.

**Input:**
number of partitions $n_{part}$, number of partitions per macro batch $n_{per\_part}$, number of training nodes in each batch *batch_size*, number of layers $L$, *historical embedding cache*

**Output:** number of iterations $n_{iter}$, sampling results stored on external storage

```
1:  % Sample stage
2:  n_iter = 0;
3:  Generate macro_batches according to n_part and n_per_part;
4:  for each macro_batch do
5:    Generate micro_batches according to batch_size;
6:    for each micro_batch do
7:      Get initial train_nid;
8:      for l ∈ {1, 2, ..., L} do
9:        Initiate new_train_nid as empty;
10:       for node in train_nid do
11:         % maintain historical embedding cache
12:         if l >= 2 && node is cache candidate then
13:           Change node state according to FSM in Figure 3 in
              historical embedding cache table;
14:           If state is Skip, then skip sampling for node;
15:         end if
16:         if node hit the two neighbor caches then
17:           Copy neighbors from host memory;
18:         else
19:           Read neighbors from external storage;
20:         end if
21:         If neighbor new, add to new_train_nid;
22:       end for
23:       train_nid = new_train_nid;
24:     end for
25:     n_iter += 1;
26:     Write sampling results to external storage;
27:   end for
28: end for
```

## 4.4 Automatic Cache Space Management

As there are three caches in OUTRE (i.e., the neighbor cache, the feature cache and *historical embedding cache*), determining their

**Algorithm 2** The *Gather*, *Transfer*, and *Compute* stage in OUTRE.

**Input:**
number of layers $L$, number of iterations $n_{iter}$,
*feature cache*, *historical embedding cache*

**for** each batch out of $n_{iter}$ batches **do**
  **% *Gather* stage**
  Read corresponding sampling results from external storage;
  **for** *node* in *train_nid* **do**
    **if** *node* hit *feature cache* **then**
      Record hit information for GPU;
    **else**
      Read *node*'s feature from external storage;
    **end if**
  **end for**
  Update *feature cache* using gathered features;

  **% *Transfer* stage**
  Transfer collective $L$-hop neighborhood and *feature cache* hit information from host memory to GPU;

  **% *Compute* stage**
  GPU fetches hit node features from *feature cache* via UVA;
  **for** $l \in \{1, 2, ..., L\}$ **do**
    **if** $l == L$ **then**
      Get pull and push node IDs from *historical embedding cache* table;
      Execute corresponding pull and push operations on *historical embedding cache*;
    **end if**
    Ordinary model computation;
  **end for**
**end for**

respective sizes under a given memory budget is crucial to OUTRE's performance. Suppose the available host memory budget is $B$. During the *Sample* stage, the neighbor cache can occupy the $B$ memory space exclusively. However, the feature cache and *historical embedding cache* need to reside in host memory simultaneously starting from the *Gather* stage. Determining the respective sizes of these two caches is nontrivial because the optimal memory split scheme correlates with the characteristics of different datasets and hardware configurations. Thus, no general memory split scheme suits all the setups. In OUTRE, we propose a low-cost management module to automatically generate a decent memory split scheme based on the profiling information. This module is named as "*automatic cache space management*". We denote the percentage of memory assigned to *historical embedding cache* by $\alpha \in [0, 1]$. Thus, the problem of determining the optimal memory split scheme can be transformed to determining the optimal $\alpha$.

To find the optimal $\alpha$ in a low-cost manner, we first need a proxy metric to approximate the training performance. As mentioned in Section 2.2, out-of-core sampling-based GNN training is bottlenecked by *Data Preparation*; in other words, the I/Os. Therefore, we propose a node-wise metric, $saved\_I/O$, that reflects the caching benefits of each node to the training performance. $saved\_I/O$ is measured by **the number of node features that are saved from reading from external storage to host memory**. For example, node $i$'s $saved\_I/O$ for the feature cache being 100 means that the I/O volume equivalent to 100 node features can be saved if node $i$ is cached by the feature cache. To calculate $saved\_I/O$ for the two caches, we execute a profiling epoch during pre-processing that only incorporates the *Sample* and the *Gather* stage with the feature cache and the historical embedding cache disabled.

Calculating $saved\_I/O_{feat}$ is simple because the feature cache helps to save I/O only at the *Gather* stage. Concretely, for each node feature that hits the feature cache, we add 1 to this node's corresponding position in $saved\_I/O_{feat}$.

Calculating $saved\_I/O_{hist}$ requires more effort because *historical embedding cache* helps to reduce I/O volume at both the *Sample* and the *Gather* stage. Due to that the memory access patterns of these two stages are highly different (sampling has significantly more random accesses than gathering), we use a coefficient $\beta$ to scale the I/O volume of the *Sample* stage. We approximate $\beta$ in OUTRE using real-world profiling results as follows:

$$\beta = \frac{time_{sample}}{I/O\_volume_{sample}} \Big/ \frac{time_{gather}}{I/O\_volume_{gather}}, \tag{3}$$

where $I/O\_volume_{sample}$ is the number of sampled neighbors, and $I/O\_volume_{gather}$ is the number of gathered node features.

During the *Sample* stage, we record its ($L$-1)-hop neighborhood size for each node while excluding those cached in host memory. This result is denoted by $saved\_I/O_{hist}^{sample}$. On the other hand, we collect the node-wise *unique* ($L$-1)-hop neighborhood size contribution starting from the second layer. This result is denoted by $saved\_I/O_{hist}^{gather}$. If one node is cached by *historical embedding cache*, then $saved\_I/O_{hist}^{gather}$ node features that this node uniquely requires can be saved from inquiring external storage during the *Gather* stage. The total $saved\_I/O$ brought by *historical embedding cache* can be calculated as follows:

$$saved\_I/O_{hist} = saved\_I/O_{hist}^{sample} * \beta \quad + saved\_I/O_{hist}^{gather}. \tag{4}$$

The detailed calculation procedures for $saved\_I/O_{hist}^{sample}$ and $saved\_I/O_{hist}^{gather}$ can be found in Section 5.3.

Finally, the unified caching benefit incorporating both the feature cache and *historical embedding cache* is formalized as follows:

$$\begin{aligned} benefit = \ & \mathrm{sum}\left(\mathrm{sort}(saved\_I/O_{feat})\left[:\left\lfloor \frac{B*(1-\alpha) - \mathrm{size_{tab}}}{d_{feat}} \right\rfloor\right]\right) \\ + \ & \mathrm{sum}\left(\mathrm{sort}(saved\_I/O_{hist})\left[:\left\lfloor \frac{B*\alpha - \mathrm{size_{tab}}}{d_{embed}} \right\rfloor\right]\right). \end{aligned} \tag{5}$$

For $saved\_I/O_{feat}$ and $saved\_I/O_{hist}$, we calculate the prefix sums of their descendently sorted versions in advance to accelerate the search for the optimal memory split scheme. To determine the optimal memory split scheme, we alternate $\alpha$ in equation 5 from 0 to 1 and record the highest benefit and the corresponding $\alpha$. The alternating step size is set to 0.01 by default. The search cost is negligible since a single search step costs $O(1)$.

In OUTRE, $saved\_I/O_{feat}$ and $saved\_I/O_{hist}$ in equation 5 in descendent order are treated as node-wise importance metrics to

select cache candidates for the feature cache and *historical embedding cache*, respectively. $saved\_I/O_{feat}$ and $saved\_I/O_{hist}$ reflect I/O savings of the actual training process. In comparison, conventional metrics like node degree [32, 39, 60] are intuitive and may be inconsistent with real-world conditions.

### 4.5 Additional Memory and I/O Cost Analysis

To facilitate the proposed three main designs, OUTRE requires certain amount of additional memory and I/O cost. For *partition-based batch construction*, the additional I/O cost mainly comes from the min-cut graph partition during pre-processing, where partition-wise adjacency matrices and node IDs are written to external memory. The additional writing cost here is approximately $O(\mathbf{M})$, where $\mathbf{M}$ is the number of edges in the graph. The $O(\mathbf{M})$ cost can be amortized among multiple training runs because the graph partition only executes once during pre-processing. During the *Sample* stage, OUTRE further reads partition-wise one-hop adjacency matrix into main memory. The additional reading cost here is approximately $O(\mathbf{M}+\mathbf{M})$. For *historical embedding cache*, besides the embedding cache itself, OUTRE needs to additionally maintain three $\mathbf{N}$-length arrays that record nodes' states, positions, and staleness, respectively. For *automatic cache space management*, OUTRE also maintains additionally three $\mathbf{N}$-length arrays during pre-processing, which are $saved\_I/O_{feat}$, $saved\_I/O_{hist}^{sample}$ and $saved\_I/O_{hist}^{gather}$.

To sum up, for one typical training run, the additional memory cost of OUTRE is approximately $O(\mathbf{N})$ (maintain *historical embedding cache*), and the additional I/O cost is $O(\mathbf{M})$ (read in partition-wise adjacency matrices). For example, on ogbn-papers100M, OUTRE requires 1.24GB additional memory to complete training. In return, OUTRE reads in 20.5GB less neighbor IDs during the *Sample* stage and 309.8GB less node features during the *Gather* stage than Ginex.

## 5 IMPLEMENTATION

### 5.1 Framework Implementation Overview

OUTRE is implemented based on Ginex [39], a dual-cache out-of-core sampling-based GNN training framework. Unlike Ginex, OUTRE does not bypass the system page cache when reading from files on external storage. This is because the adjacency matrix often constitutes only a small percentage of graph data and can be mostly cached in host memory by the system page cache. OUTRE adopts the neighbor cache and feature cache module in Ginex yet implements new node-wise importance metrics for cache candidate selection as mentioned in Section 4.2 and 4.4. To reduce the CPU-side load, OUTRE utilizes the *pin_memory_inplace()* and *gather_pinned_tensor_rows()* functions provided by DGL [48] to let GPU fetch node features that hit the feature cache in the zero-copy manner [34] via UVA [41]. The C++ gather function in OUTRE only gathers node features that miss the feature cache. We re-write the *NeighborSampler* class and the C++ sample function in Ginex to serve the need for *partition-based batch construction* and *historical embedding cache*. A pre-processing stage that includes min-cut graph partitioning and profiling is added ahead of the original four-stage training pipeline. The next two subsections will detail the implementation of the modified FENNEL partitioning algorithm and the calculations for *saved I/O*.

**Table 5: Overview of the four large-scale graph datasets.**

| Datasets | #Nodes | #Edges | #Features | #Train nodes | Dataset size |
|---|---|---|---|---|---|
| ogbn-papers100M | 111,059,956 | 3,231,371,744 | 128 | 1,207,179 | 70GB |
| mag240M-cite | 121,751,666 | 2,595,497,852 | 768 | 1,112,392 | 385GB |
| IGB-medium | 10,000,000 | 120,077,694 | 1024 | 6,000,000 | 40.8GB |
| IGB-large | 100,000,000 | 1,223,571,364 | 1024 | 60,000,000 | 401.8GB |

### 5.2 Modified FENNEL Partitioning Algorithm

FENNEL [43] is a streaming graph partitioning algorithm that processes nodes in a stream and inserts each node to the partition that scores the highest in its cost function. The cost function of FENNEL is the weighted sum of the partition size and the number of common neighbors in the partition. To adapt FENNEL to the GNN training scenario, we add an additional penalizing term for the number of training nodes within each partition in the cost function.

The most performance-critical part of FENNEL is calculating the number of common neighbors between each node's neighborhood and each existing partition. We adopt Bloom Filters [3] to approximate the calculations for intersection cardinality. Maintaining one Bloom Filter for each partition requires enormous memory when the number of partitions is huge. Thus, we re-implement FENNEL as a two-level graph partitioning algorithm to alleviate this memory issue. For example, when the desired number of partitions is 10,000, we partition the original graph into 100 partitions during first-level partitioning and further partition each first-level partition into 100 small partitions. Moreover, having observed that calculating the cost function of different partitions in FENNEL is fully independent, we parallelize FENNEL on the partition level by OpenMP [10] to exploit thread-level parallelism.

### 5.3 Calculations for *saved I/O*

Calculations for *saved I/O* consists of calculations for three arrays: $saved\_I/O_{feat}$, $saved\_I/O_{hist}^{sample}$ and $saved\_I/O_{hist}^{gather}$. We implement the calculation for these three arrays in the *NeighborSampler* class, and these arrays are shared among all the dataloader processes. The updates to these shared arrays are marked as critical sections and serialized by a shared lock. $saved\_I/O_{feat}$ records the node-wise access frequency during the *Gather* stage. For each training batch, we increase $saved\_I/O_{feat}$'s corresponding positions of all the nodes in its sampled collective $L$-hop neighborhood by 1.

$saved\_I/O_{hist}^{sample}$ records node-wise $(L-1)$-hop neighborhood size while excluding those cached in host memory. $saved\_I/O_{hist}^{gather}$ records node-wise *unique* contribution to the collective $(L\text{-}1)$-hop neighborhood for the training nodes and their 1-hop neighbors. To calculate these two arrays, we write a new profiling-specific sampling function, in which OUTRE generates two new sets of adjacency matrices starting from the second layer sampling. One set of adjacency matrices only records edges not cached in host memory for $saved\_I/O_{hist}^{sample}$. The other set only records edges that connect to *unique* nodes that are new to the sampled collective neighborhood for $saved\_I/O_{hist}^{gather}$. After sampling for a training batch, OUTRE sets the weight of all the nodes in the sampled collective $L$-hop neighborhood to 1. Then, OUTRE multiplies this

**Table 6: Overall training performance comparison on four datasets.**

| Configurations | | | ogbn-papers100M | | | | | mag240M-cite | | | | | IGB-medium | | | | | IGB-large | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *Sa.* | *Ga.* | *Tr.* | *Co.* | Total | *Sa.* | *Ga.* | *Tr.* | *Co.* | Total | *Sa.* | *Ga.* | *Tr.* | *Co.* | Total | *Sa.* | *Ga.* | *Tr.* | *Co.* | Total |
| GraphSAGE | $L$=2 | PyG+mmap | **8.5s** | 181.4s | 8.5s | 7.1s | 213.8s | **9.4s** | 360.9s | 30.0s | **5.9s** | 433.3s | **11.2s** | 411.9s | 256.0s | 28.7s | 878.7s | **141.1s** | 46,437.9s | 1,320.1s | 960.7s | 50,148.1s |
| | | Ginex | 143.7s | 116.5s | 9.3s | **5.2s** | 275.7s | 148.1s | 231.9s | 49.3s | 6.1s | 435.4s | 109.3s | 258.9s | 326.8s | **23.9s** | 723.6s | 746.4s | 5,020.1s | 2,256.9s | **309.7s** | 7,606.1s |
| | | $\text{Ginex}_{\text{mod}}$ | 118.2s | 93.1s | **3.0s** | 17.3s | 227.6s | 121.3s | 186.2s | **6.8s** | 19.3s | 349.3s | 40.1s | 226.8s | **163.2s** | 68.4s | 498.7s | 646.2s | 4,291.1s | **993.5s** | 825.8s | 5,173.2s |
| | | OUTRE | 83.9s | **52.6s** | 3.2s | 13.1s | **178.1s** | 88.6s | **79.8s** | 8.2s | 13.3s | **204.7s** | 17.8s | **174.4s** | 204.7s | 54.5s | **428.3s** | 385.4s | **3,792.6s** | 1183.1s | 763.6s | **4,359.8s** |
| | $L$=3 | PyG+mmap | 121.0s | 3,542.6s | 75.2s | 20.9s | 3,822.8s | **61.9s** | 4,571.5s | 146.6s | 14.3s | 5,139.6s | **30.4s** | 6,412.4s | 1,494.8s | 53.7s | 9,302.2s | - | - | - | - | Out of time |
| | | Ginex | 292.1s | 284.1s | 90.9s | **8.1s** | 605.4s | 197.1s | 513.1s | 377.8s | **7.3s** | 970.4s | 233.3s | 1,946.7s | 2,292.8s | **41.9s** | 3,773.6s | 2,905.7s | 38,475.3s | 16,668.7s | **334.0s** | 42,220.0s |
| | | $\text{Ginex}_{\text{mod}}$ | 155.4s | 152.8s | **6.0s** | 16.6s | 338.0s | 157.2s | 484.0s | **34.6s** | 71.8s | 720.9s | 151.4s | 1,788.4s | **713.3s** | 484.1s | 2,707.8s | 2,491.2s | 27,835.7s | **12,974.6s** | 4,510.3s | 34,340.3s |
| | | OUTRE | **106.6s** | **77.0s** | 8.6s | 14.2s | **235.4s** | 111.7s | **177.2s** | 47.4s | 45.6s | **349.1s** | 62.9s | **444.3s** | 828.4s | 384.4s | **1,706.5s** | **480.3s** | **21,721.8s** | 13,227.4s | 3,340.3s | **27,755.3s** |
| GAT | $L$=2 | PyG+mmap | **9.9s** | 172.8s | 10.9s | 13.8s | 218.6s | **9.3s** | 376.4s | 30.4s | 13.3s | 460.0s | **11.9s** | 415.1s | 256.3s | 73.7s | 947.2s | **141.7s** | 45,983.9s | 1,314.6s | 1,207.3s | 50,007.4s |
| | | Ginex | 136.6s | 134.1s | 9.9s | **10.3s** | 288.8s | 137.9s | 242.4s | 44.4s | **13.0s** | 439.4s | 110.1s | 261.3s | 329.2s | **63.8s** | 781.8s | 752.2s | 4,618.1s | 2,264.9s | **679.1s** | 7,638.9s |
| | | $\text{Ginex}_{\text{mod}}$ | 121.4s | 103.2s | **3.1s** | 23.2s | 245.1s | 119.7s | 197.6s | **7.1s** | 28.3s | 367.3s | 37.6s | 231.7s | **171.6s** | 113.2s | 554.3s | 638.6s | 4,041.7s | **1,033.2s** | 1,283.6s | 5,482.2s |
| | | OUTRE | 84.1s | **55.6s** | 3.7s | 16.2s | **180.2s** | 87.9s | **78.2s** | 7.6s | 20.9s | **202.5s** | 16.4s | **171.5s** | 213.4s | 90.4s | **461.1s** | 394.6s | **3,476.1s** | 1,039.2s | 1,146.4s | **4,571.6s** |
| | $L$=3 | PyG+mmap | 98.8s | 2,376.9s | 63.2s | **9.3s** | 2,535.6s | **65.9s** | 3,943.7s | 224.9s | **31.8s** | 4,427.9s | **22.4s** | 4,108.4s | 1,482.5s | **47.2s** | 6,991.3s | - | - | - | - | Out of time |
| | | Ginex | 256.6s | 265.6s | 81.8s | 32.1s | 654.6s | 195.2s | 468.5s | 412.0s | 39.8s | 1,092.4s | 225.3s | 1,917.9s | 2,363.2s | 198.5s | 4,238.2s | 2,959.4s | 38,629.1s | 16,726.3s | **1,529.7s** | 44,287.2s |
| | | $\text{Ginex}_{\text{mod}}$ | 156.6s | 149.4s | **5.9s** | 27.5s | 344.2s | 153.9s | 443.8s | **31.5s** | 89.7s | 680.6s | 145.3s | 1,749.7s | **725.2s** | 619.7s | 2,912.3s | 2,476.5s | 27,979.2s | **12,628.4s** | 4,962.3s | 34,859.7s |
| | | OUTRE | **94.5s** | **73.1s** | 8.8s | 26.4s | **226.7s** | 98.8s | **124.7s** | 41.4s | 55.1s | **310.6s** | 53.2s | **537.8s** | 813.9s | 497.8s | **1,975.5s** | **495.6s** | **22,137.9s** | 13,036.1s | 3,792.7s | **28,564.0s** |
| GCN | $L$=2 | PyG+mmap | **8.2s** | 167.0s | 8.4s | 7.3s | 199.3s | **8.6s** | 374.1s | 29.9s | 7.0s | 446.9s | **11.8s** | 408.6s | 255.1s | 29.4s | 873.5s | **143.0s** | 46,243.3s | 1,323.2s | 873.2s | 49,868.4s |
| | | Ginex | 140.2s | 122.6s | 9.9s | **4.4s** | 277.0s | 138.9s | 230.2s | 49.9s | **5.8s** | 423.2s | 112.7s | 259.4s | 333.2s | **22.7s** | 730.3s | 738.5s | 4,900.8s | 2,253.9s | **284.4s** | 7,566.6s |
| | | $\text{Ginex}_{\text{mod}}$ | 116.5s | 97.3s | **3.7s** | 18.6s | 231.7s | 113.6s | 185.5s | **6.3s** | 19.1s | 342.3s | 38.3s | 225.4s | **164.6s** | 67.5s | 495.6s | 642.7s | 4,383.6s | **957.8s** | 813.7s | 5,135.8s |
| | | OUTRE | 82.4s | **53.0s** | 4.4s | 13.3s | **174.3s** | 79.5s | **80.2s** | 8.1s | 14.3s | **192.9s** | 18.2s | **178.1s** | 215.7s | 53.3s | **441.7s** | 390.4s | **3,810.8s** | 1,096.7s | 743.5s | **4,369.8s** |
| | $L$=3 | PyG+mmap | **94.8s** | 2,457.7s | 63.5s | 13.1s | 2,616.8s | **66.2s** | 4,019.6s | 366.1s | 11.5s | 4,620.2s | **23.3s** | 3,946.9s | 1,484.1s | 47.4s | 6,790.4s | - | - | - | - | Out of time |
| | | Ginex | 271.4s | 255.6s | 83.8s | **7.6s** | 611.9s | 204.3s | 450.5s | 405.6s | **9.8s** | 1,032.6s | 233.8s | 1,900.8s | 2,349.6s | **31.8s** | 3,846.1s | 5,092.9s | 25,843.4s | 27,971.6s | **578.2s** | 47,524.3s |
| | | $\text{Ginex}_{\text{mod}}$ | 165.9s | 173.4s | **6.8s** | 18.7s | 357.2s | 164.4s | 424.8s | **37.6s** | 73.2s | 674.2s | 143.6s | 1,673.6s | **773.2s** | 457.6s | 2,617.2s | 2,537.6s | 27,075.4s | **12,495.6s** | 4,367.4s | 31,836.3s |
| | | OUTRE | 99.7s | **61.9s** | 9.5s | 15.9s | **202.2s** | 91.8s | **136.4s** | 49.4s | 40.3s | **305.4s** | 52.2s | **433.4s** | 845.9s | 377.5s | **1,795.3s** | **485.3s** | **21,965.9s** | 13,106.7s | 3,127.8s | **26,959.2s** |

weight vector with the two sets of adjacency matrices in the reserve order to generate respective updates to the two shared arrays.

# 6 EVALUATION

In this section, we compare the training performance of OUTRE with other out-of-core sampling-based GNN training frameworks on four public large-scale graph datasets.

## 6.1 Experiment Setup

**Hardware and software configurations.** All the evaluations are conducted on a Linux server with two Intel(R) Xeon(R) Platinum 8255C CPUs, a single NVIDIA Tesla V100 with 32GB GPU memory and PCIe GEN3 NVMe SSDs. 64GB host memory is locked for evaluation unless otherwise stated. As for software versions, we use Python 3.9, PyTorch 1.12.1, and CUDA 11.6.

**Datasets.** We conduct the evaluations on four public large-scale graph datasets: ogbn-papers100M, mag240M-cite, IGB-medium, and IGB-large. Ogbn-papers100M and mag240M-cite are from Open Graph Benchmark (OGB) [20]. Mag240M-cite is a homogeneous graph containing only "paper" nodes and "paper-cite-paper" edges of the original mag240M graph. IGB-medium and IGB-large are from Illinois Graph Benchmark (IGB) [27]. The two IGB datasets have significantly larger training sets than the two OGB datasets. The dataset statistics are briefly summarized in Table 5.

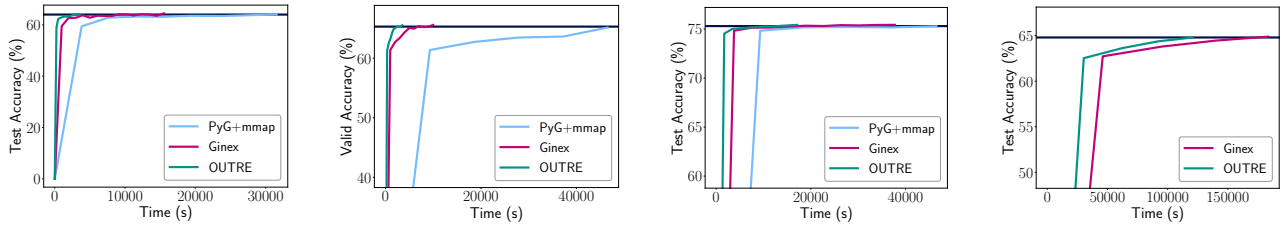
**Compared baselines.** The two compared baseline frameworks are PyG [12]+mmap and Ginex [39]. The PyG version in the evaluation is 2.3.1. PyG+mmap is extended from PyG by letting it read memory-mapped adjacency matrices and node features from external storage with NumPy's *memmap* method. We also compare to a baseline called "$\text{Ginex}_{\text{mod}}$". $\text{Ginex}_{\text{mod}}$ uses the same read flag as OUTRE that makes system page cache to cache graph data. Moreover, $\text{Ginex}_{\text{mod}}$ enables GPU to directly access cached node features in host memory via UVA, same as OUTRE. For OUTRE, we set the number of partitions in *partition-based batch construction* to 10,000, and every training batch is constructed with 20 randomly chosen partitions. In *historical embedding cache*, we set *staleness* threshold to 100, and only cache the second layer's historical embeddings.

**Evaluation workloads.** We adopt training a 3-layer GraphSAGE [16] as the main comparison workload. The hidden size and the sample size are set to 256 and (10,10,10), respectively. The default batch size is set to 1,000. We further extend the comparison to training 2-layer and 3-layer GAT [44] and GCN [28] in the main experiment. We set the neighbor cache size in Ginex and OUTRE to 1GB for IGB-medium and 10GB for other datasets. The feature cache size in Ginex is set to 10GB for ogbn-papers100M and IGB-medium and 30GB for mag240M-cite and IGB-large. The aggregated size of the feature cache and the historical embedding cache in OUTRE is set to be the same as the feature cache size in Ginex.

## 6.2 Overall Performance

We report the per-epoch time of the four compared frameworks on the four datasets in Table 6. The evaluation workloads here are training 2-layer and 3-layer GraphSAGE, GAT and GCN. The per-epoch time is decomposed following the four-stage pipeline (i.e., *Sa.* for *Sample*, *Ga.* for *Gather*, *Tr.* for *Transfer*, and *Co.* for *Compute*). The *Gather* stage is pipelined with the two later training stages in Ginex, $\text{Ginex}_{\text{mod}}$ and OUTRE. Thus, their per-epoch time is less than the sum of the four stages. We put this Ginex's and $\text{Ginex}_{\text{mod}}$'s pre-computation time for the optimal cache policy into their *Gather* time since it aims to accelerate the *Gather* stage.

Table 6 shows that OUTRE consistently outperforms all the baselines across different model configurations and datasets. Equipped with techniques in OUTRE that target improving data transfer efficiency, $\text{Ginex}_{\text{mod}}$ outperforms Ginex on all the evaluation workloads. We observe that GNN option has little effect on the overall training performance, which conforms to the fact that out-of-core sampling-based GNN training is bounded by *Data Preparation*. Intuitively, the number of model layers is positively correlated with the amount of data redundancies. Reflected in evaluation results, we find that OUTRE shows larger average speedup against Ginex on 3-layer workloads than 2-layer workloads (2.45× vs. 1.78×). To note that, Ginex is shown to have larger per-epoch time than PyG+mmap on training 2-layer GNNs on ogbn-papers100M due to its overhead. We also find that PyG+mmap has the highest sampling




(a) ogbn-papers100M (b) mag240M-cite (c) IGB-medium (d) IGB-large

**Figure 4: Time-to-accuracy comparison of PyG+mmap, Ginex and OUTRE.**

**Table 7: Impacts of different partition methods.**

| Partition Methods | ogbn-papers100M | | IGB-medium | |
|---|---|---|---|---|
| | per-epoch time | acc | per-epoch time | acc |
| no partition | 389.9s | 64.28 | 2,089.4s | 75.39 |
| METIS | 213.6s | 64.18 | 1,640.7s | 75.31 |
| modified FENNEL | 235.4s | 64.15 | 1,706.5s | 75.33 |

**Table 8: Ablation study of performance optimizations on FENNEL's partition performance.**

| Partition Methods | ogbn-papers100M | | IGB-medium | |
|---|---|---|---|---|
| | partition time | peak mem | partition time | peak mem |
| METIS | 17,854.6s | 477.3G | 498.8s | 18.9G |
| original FENNEL | >12h | 28.1G | >12h | 1.7G |
| + OpenMP | >12h | 28.1G | 2675.1s | 1.7G |
| + Bloom Filter | >12h | 333.6G | 2106.4s | 29.1G |
| + two-level partition | 9,710.2s | 34.2G | 794.6s | 2.4G |
| - Bloom Filter | >12h | 29.3G | 5751.4s | 1.9G |

performance among all the frameworks although it does not cache important neighbor lists. We attribute this phenomenon to the fact that the other three frameworks all need to write sampling results to external storage, which drags the sampling execution.

We also draw the time-to-accuracy curves of the 3-layer GraphSAGE on all the four datasets for PyG+mmap, Ginex and OUTRE in Figure 4. We record the test accuracy (valid accuracy for mag240M-cite) and the accumulated training time of every epoch. The horizontal lines in the figures indicate the target test accuracies. Figure 4 illustrates that OUTRE achieves comparable convergence accuracy with others and has significantly lower time-to-accuracy than the two baseline frameworks.

## 6.3 Impacts of *Partition-based Batch Construction*

The *partition-based batch construction* in OUTRE constructs training batches according to min-cut graph partitions to reduce *Neighborhood Redundancy*. To achieve the target, we adopt a streaming graph partitioning algorithm, FENNEL [43], to suit the needs of out-of-core environments. Moreover, we make profound performance optimizations to the original FENNEL algorithm, which accelerate its execution significantly. However, the partition quality of FENNEL cannot compete with METIS [26], and some of our performance optimizations introduce approximation errors.

**Table 9: Impacts of the choices for cached layers in *historical embedding cache*.**

| Cached layer | ogbn-papers100M | | IGB-medium | |
|---|---|---|---|---|
| | per-epoch time | acc | per-epoch time | acc |
| only 1st | 238.4s | 64.13 | 1,796.7s | 75.36 |
| only 2nd | 235.4s | 64.15 | 1,706.5s | 75.33 |
| all | 253.7s | 64.08 | 2,297.5s | 75.28 |

To explore the impacts of partition qualities, we provide OUTRE's per-epoch time and final accuracy on ogbn-papers100M and IGB-medium when using different graph partitioning algorithms in Table 7. For METIS, we use the version implemented in DGL [48]. "Modified FENNEL" denotes the variant with all our performance optimizations, and "no partition" denotes the variant that disables *partition-based batch construction*. Table 7 illustrates that the "no partition" variant requires considerably higher per-epoch time than others, which shows the effectiveness of *partition-based batch construction*. Moreover, the per-epoch time and final accuracy of "modified FENNEL" is close to "METIS". This observation demonstrates that the slightly lower partition quality of modified FENNEL compared to METIS does not influence training performance much.

We report the partition time and the peak memory footprint of METIS and FENNEL, and also the effects of our performance optimizations on ogbn-papers100M and IGB-medium in Table 8. All the methods partition the graph into 10,000 partitions. Compared to METIS, all the FENNEL variants except "+ Bloom Filter" have significantly lower memory footprint, which satisfies the needs of the out-of-core environment. Furthermore, the results also show that all our performance optimizations contribute to the partition performance as expected. The significant longer running time of "- Bloom Filter" shows that even with two-level partition, the usage of Bloom Filter is necessary to reduce the time complexity of single search operation to a constant. To note that, OUTRE only executes the graph partition algorithm once for a specific graph. Thus, its time cost can be amortized among multiple training runs.

## 6.4 Impacts of *Historical Embedding Cache*

We propose to only cache the historical embeddings of *the second layer* in OUTRE to achieve the highest performance-cost ratio. To validate this design, we report the per-epoch training time and final accuracy of caching only the second layer, only the first layer and all the layers on ogbn-papers100M and IGB-medium in Table 9. We

**Table 10: Impacts of number of model layers to *historical embedding cache* on ogbn-papers100M.**

| #Layers | only cache 2nd | cache all the layers |
|---|---|---|
| | per-epoch time | per-epoch time |
| 2 | 178.1s | 191.6s |
| 3 | 235.4s | 263.7s |
| 4 | 298.7s | 340.5s |

**Table 11: Impacts of different staleness threshold values in *historical embedding cache.***

| *Staleness* | ogbn-papers100M | | IGB-medium | |
|---|---|---|---|---|
| | per-epoch time | acc | per-epoch time | acc |
| 1 | 271.6s | 64.32 | 1,922.2s | 75.44 |
| 50 | 248.4s | 64.21 | 1,712.1s | 75.37 |
| 100 | 235.4s | 64.15 | 1,706.5s | 75.33 |
| 500 | 211.7s | 63.74 | 1,121.4s | 75.14 |
| 1,000 | 200.8s | 63.64 | 1,047.8s | 74.83 |

keep the size of *historical embedding cache* fixed among these three variants. We also report the training performance of only caching the second layer and caching all the layers when the number of model layers vary in Table 10. Table 9 illustrates that only caching the second layer performs significantly better than only caching the first layer in OUTRE. This phenomenon can be attributed to *partition-based batch construction* that greatly reduces I/O cost of fetching 1-hop neighborhoods. On the other hand, we also observe that only caching the second layer outperforms caching all the layers. As mentioned in Section 4.3, caching one node of the second layer can possibly prune an ($L$-1)-depth computation graph which has the highest performance-cost ratio in OUTRE. Thus, under the same cache size budget, caching only the second layer can prune more redundant computation subgraphs, leading to higher performance. This performance priority remains when the number of model layers grows to more than 3, as shown in Table 10.

As mentioned in Section 3.3 and 4.3, *historical embedding cache* in OUTRE can reduce *Temporal Redundancy* and accelerate out-of-core GNN training. However, these benefits come at the cost of introducing approximation errors. The error volume can be controlled by the pre-set threshold *staleness*. For example, setting *Staleness* to 1 is equivalent to disabling *historical embedding cache*. On the other hand, setting *Staleness* to a large value risks producing unsatisfactory accuracy. In Table 11, we report the per-epoch time and the final accuracy of OUTRE on ogbn-papers100M and IGB-medium when *Staleness* varies from 1 to 1,000. Table 11 shows that the per-epoch time gradually increases and the final accuracy decreases as expected when *Staleness* varies from 1 to 1,000. Compared to disabling *historical embedding cache* (i.e., *Staleness* being 1), setting *Staleness* to 100 helps to accelerate training by 15.4% and 19.6% on ogbn-papers100M and IGB-medium, respectively. When setting *Staleness* to 1,000, although the per-epoch time continues to decrease, the convergence accuracy is 0.68% and 0.61% lower than when *historical embedding cache* disabled, respectively. The accuracy loss over 0.5% might not be acceptable under many applications. Thus, the default *Staleness* is set to 100 in OUTRE for the balance between training performance and final accuracy.

**Table 12: Impacts of different node-wise importance metrics on ogbn-papers100M.**

| #Layers | Metrics | *Sample* | *Gather* | Total |
|---|---|---|---|---|
| $L$=2 | random | 85.4s | 58.1s | 187.6s |
| | degree | 84.3s | 54.7s | 185.4s |
| | *saved_I/O* | **83.9s** | **52.6s** | **178.1s** |
| $L$=3 | random | 157.4s | 122.6s | 336.6s |
| | degree | 122.5s | 99.9s | 262.6s |
| | *saved_I/O* | **106.6s** | **77.0s** | **235.4s** |
| $L$=4 | random | 171.6s | 151.7s | 405.4s |
| | degree | 141.2s | 128.7s | 354.9s |
| | *saved_I/O* | **109.3s** | **103.3s** | **298.7s** |

## 6.5 Impacts of *Automatic Cache Space Management*

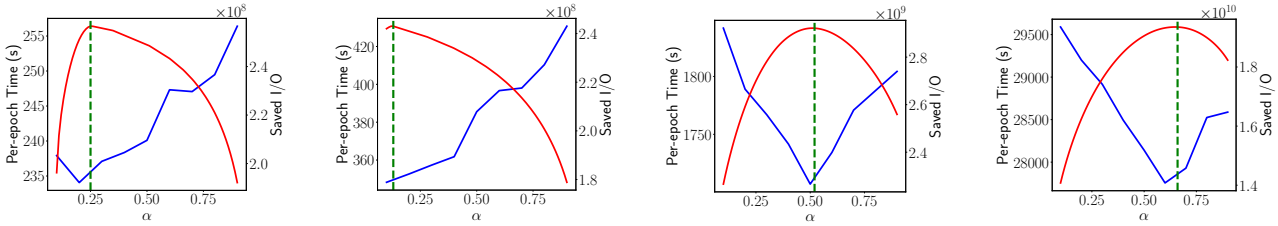
The *automatic cache space management* module in OUTRE helps to automatically generate a decent memory split scheme for the feature cache and *historical embedding cache*. In this subsection, we validate the effectiveness of this module. We manually vary the memory split ratio $\alpha$ from 0.1 to 0.9 with step size 0.1 and evaluate each configuration's corresponding per-epoch time. The evaluation results on all the four datasets are shown in Figure 5. The approximated I/O saving (i.e., *benefit* in Equation 5) returned by the *automatic cache space management* module is also drawn in the same figures. The blue lines and the red lines denote the per-epoch time and the approximated I/O savings, respectively.

Figure 5 illustrates that the real-world performance and the approximated I/O saving calculated by OUTRE show consistent trends. For example, the approximated I/O saving peaks at 0.25 and 0.52 (shown in green dashed lines) on ogbn-papers100M and IGB-medium, respectively. And the real-world per-epoch time also achieves its lowest in the nearby $\alpha$ range. We also observe that the optimal memory split ratios of different datasets differ. The reason might be that the feature dimension of IGB-series datasets is significantly larger than that of ogbn-papers100M (1024 vs. 128). Thus, assigning more memory to the feature cache for IGB-series datasets is reasonable. This observation further shows the importance of the *automatic cache space management* module.

As mentioned in Section 4.4, OUTRE uses $saved\_I/O_{feat}$ and $saved\_I/O_{hist}$ returned from real-world profiling to select cache candidates for the feature cache and *historical embedding cache*, respectively. Here, we report OUTRE's per-epoch runtime of using different node-wise importance metrics in Table 12. Evaluation results show that our proposed new node-wise importance metrics for selecting cache candidates outperforms

# 7 RELATED WORK

**In-memory GNN Training Frameworks.** As GNNs show significant performance superiority over traditional deep learning methods on graph learning tasks, how to accelerate and scale GNN training on large-scale graphs starts to attract the community's attention. PaGraph [32] observes that many nodes are redundantly transferred to GPU and proposes to cache some of the high-degree nodes in GPU memory. ROC [23] formalizes the problem of finding optimal intermediate data to cache in GPU memory and solves it by dynamic programming. Moreover, it uses a runtime prediction




(a) ogbn-papers100M (b) mag240M-cite (c) IGB-medium (d) IGB-large

**Figure 5: Effectiveness of the search process inside *automatic cache space management*.**

model to achieve balanced graph partitions. PyTorch-Direct [34] adopts the Unified Virtual Addressing (UVA) [41] technique to let GPU directly access the graph data within the host memory without CPU's involvement. $P^3$ [15] exploits the potential of applying model parallelism to GNN training. DGCL [5] notices the heterogeneous data links in real-world systems and presents an algorithm to solve the communication planning problem. There are also some attempts on GNN-specific accelerators [29, 35, 56].

**Out-of-core GNN Training Frameworks.** The graph size used for GNN training grows larger [20, 27], and the sizes of some public graph datasets have exceeded the typical host memory size of a single machine. Thus, how to incorporate external storage into current GNN training frameworks becomes a hot topic. There are mainly three branches of existing work. The first branch of work [22, 47] proactively moves in and out partitioned graph data to carry out out-of-core GNN training. However, they only execute GNN training on in-memory partitions, and graph data on external storage is simply neglected, which leads to lower accuracy. HierBatching [22] additionally pins some high-degree nodes in the host memory to compensate for the lost neighbors. In contrast, OUTRE would go to external storage to find the requested graph data that has not been stored in host memory. Thus, *partition-based batch construction* in OUTRE causes no accuracy loss to sampling-based GNN training.

The second branch of work [38, 42] is built on a previous system [40] that enables GPU threads to make on-demand and fine-grained data requests to NVMe SSDs and improves I/O efficiency significantly. GIDS [38] applies this technique to GNN training. Helios [42] further decouples the originally synchronous I/O stack to an asynchronous one and achieves decent performance gain.

The third branch of work [39] adopts the dual-cache framework that caches both neighbor lists and node features. This branch focuses on how to increase the percent of requested data volume that can be saved from accessing external storage. Ginex [39] implements the optimal caching policy [2] for the feature cache with the data access order collected during the *Sample* stage. We build OUTRE on this branch, and propose a new and more fundamental design perspective on *reducing the overall requested data volume*.

**Reusing Historical Node Embeddings.** Reusing historical node embeddings is first proposed by [7] and is later generalized by GAS [13]. GraphFM [54] proposes the Feature Momentum technique that applies momentum steps on historical node embeddings and outperforms GAS. The above work stores historical embeddings for all the nodes and all the model layers, incurring enormous memory costs. ReFresh [21] proposes one staleness-based and one gradient-based metric to control both the size and the quality of the stored historical embeddings. OUTRE's *historical embedding cache* mainly follows Refresh's design, yet we make nontrivial modifications to suit the characteristics of out-of-core environments. Specifically, we only cache node embeddings of the second layer to achieve high performance-cost ratio. Moreover, we pre-define the cache candidate to limit the cache size strictly. Finally, we propose a new node importance metric for cache candidate selection, which reflects real-world training performance.

**Graph Partitioning in Out-of-core Graph Processing Systems.** Existing out-of-core graph processing systems [45, 46, 61] target conventional graph processing workloads (e.g., PageRank [37], connected components [19]) and execute iteratively. These graph processing system cannot execute GNN training workloads without profound modifications. They usually use heuristic graph partitioning algorithms and focus on how to reduce the I/O operations at the level of the whole iterative process (e.g., skip loading certain graph partitions, reduce the number of iterations). In contrast, OUTRE targets GNN training and utilizes min-cut graph partitioning algorithms to reduce the I/O operations in just one round of traversal for all the training nodes, which is different from the design objective of out-of-core graph processing systems.

## 8 CONCLUSION

In this paper, we propose a new out-of-core sampling-based GNN training framework, OUTRE. Unlike existing work that tries to reduce the percentage of data attempts that inquire external storage, we present a new design perspective that explores how to *reduce the overall requested data volume*. We conduct a quantitative analysis on out-of-core sampling-based GNN training and find two kinds of data redundancies exist in its requested data volume. Then we propose three corresponding new designs, *partition-based batch construction*, *historical embedding cache* and *automatic cache space management*, to reduce the data redundancies and accelerate training. Evaluation results on four public large-scale graph datasets show that OUTRE achieves 1.52× to 3.51× speedup against the SOTA framework.

## ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (U23B2048 and U22B2037), research grant No. SH-2024JK29, PKU-Tencent joint research Lab and High-performance Computing Platform of Peking University. Bin Cui is the corresponding author.

# REFERENCES


[1] Bushra Alhijawi and Ghazi Al-Naymat. 2022. Novel Positive Multi-Layer Graph Based Method for Collaborative Filtering Recommender Systems. *Journal of Computer Science and Technology* 37, 4 (2022), 975–990.

[2] Laszlo A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Lei Cai, Jundong Li, Jie Wang, and Shuiwang Ji. 2021. Line graph neural networks for link prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 9 (2021), 5103–5113.

[5] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[7] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning*. PMLR, 942–950.

[8] Man-Sheng Chen, Jia-Qi Lin, Xiang-Long Li, Bao-Yu Liu, Chang-Dong Wang, Dong Huang, and Jian-Huang Lai. 2022. Representation learning in multi-view clustering: A literature review. *Data Science and Engineering* 7, 3 (2022), 225–241.

[9] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 257–266.

[10] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[11] Shaojie Dai, Yanwei Yu, Hao Fan, and Junyu Dong. 2022. Spatio-temporal representation learning with social tie for personalized POI recommendation. *Data Science and Engineering* 7, 1 (2022), 44–56.

[12] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[13] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International conference on machine learning*. PMLR, 3294–3304.

[14] Yasuhiro Fujiwara, Yasutoshi Ida, Atsutoshi Kumagai, Masahiro Nakano, Akisato Kimura, and Naonori Ueda. 2023. Efficient Network representation learning via cluster similarity. *Data Science and Engineering* 8, 3 (2023), 279–291.

[15] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 551–568.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NIPS*. 1024–1034.

[17] Haohuai He, Guanxing Chen, and Calvin Yu-Chian Chen. 2024. Integrating sequence and graph information for enhanced drug-target affinity prediction. *Science China Information Sciences* 67, 2 (2024), 1–2.

[18] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 639–648.

[19] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. 1979. Computing connected components on parallel computers. *Commun. ACM* 22, 8 (1979), 461–464.

[20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[21] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. 2023. ReFresh: Reducing Memory Access from Exploiting Stable Historical Embeddings for Graph Neural Network Training. *arXiv preprint arXiv:2301.07482* (2023).

[22] Tianhao Huang, Xuhao Chen, Muhua Xu, Arvind Arvind, and Jie Chen. 2023. HierBatching: Locality-Aware Out-of-Core Training of Graph Neural Networks. https://openreview.net/forum?id=WWD_2DKUqdJ

[23] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[24] Dejun Jiang, Zhenxing Wu, Chang-Yu Hsieh, Guangyong Chen, Ben Liao, Zhe Wang, Chao Shen, Dongsheng Cao, Jian Wu, and Tingjun Hou. 2021. Could graph neural networks learn better molecular representation for drug discovery? A comparison study of descriptor-based and graph-based models. *Journal of cheminformatics* 13, 1 (2021), 1–23.

[25] Taisong Jin, Huaqiang Dai, Liujuan Cao, Baochang Zhang, Feiyue Huang, Yue Gao, and Rongrong Ji. 2022. Deepwalk-aware graph convolutional networks. *Science China Information Sciences* 65, 5 (2022), 152104.

[26] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[27] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. *arXiv preprint arXiv:2302.13522* (2023).

[28] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJU4ayYgl

[29] Jia-Jun Li, Ke Wang, Hao Zheng, and Ahmed Louri. 2023. GShuttle: Optimizing Memory Access Efficiency for Graph Convolutional Neural Network Accelerators. *Journal of computer science and technology* 38, 1 (2023), 115–127.

[30] Zhi-Yuan Li, Man-Sheng Chen, Yuefang Gao, and Chang-Dong Wang. 2023. Signal contrastive enhanced graph collaborative filtering for recommendation. *Data Science and Engineering* 8, 3 (2023), 318–328.

[31] Xiao-Fei Liao, Wen-Ju Zhao, Hai Jin, Peng-Cheng Yao, Yu Huang, Qing-Gang Wang, Jin Zhao, Long Zheng, Yu Zhang, and Zhi-Yuan Shao. 2024. Towards High-Performance Graph Processing: From a Hardware/Software Co-Design Perspective. *Journal of Computer Science and Technology* 39, 2 (2024), 245–266.

[32] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.

[33] Xiaojun Ma, Ziyao Li, Guojie Song, and Chuan Shi. 2023. Learning discrete adaptive receptive fields for graph convolutional networks. *Science China Information Sciences* 66, 12 (2023), 222101.

[34] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with GPU-oriented data communication architecture. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2087–2100.

[35] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, Ramprasath S, and Sachin S Sapatnekar. 2022. GNNIE: GNN inference engine with load-balancing and graph-specific caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 565–570.

[36] Thin Nguyen, Hang Le, Thomas P Quinn, Tri Nguyen, Thuc Duy Le, and Svetha Venkatesh. 2021. GraphDTA: Predicting drug–target binding affinity with graph neural networks. *Bioinformatics* 37, 8 (2021), 1140–1147.

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, et al. 1999. The pagerank citation ranking: Bringing order to the web. (1999).

[38] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2023. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *arXiv preprint arXiv:2306.16384* (2023).

[39] Yeonhong Park, Sunhong Min, and Jae W Lee. 2022. 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 (2022), 2626–2639.

[40] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[41] Tim C Schroeder. 2011. Peer-to-peer & unified virtual addressing. In *GPU Technology Conference, NVIDIA*.

[42] Jie Sun, Mo Sun, Zheng Zhang, Jun Xie, Zuocheng Shi, Zihan Yang, Jie Zhang, Fei Wu, and Zeke Wang. 2023. Helios: An Efficient Out-of-core GNN Training System on Terabyte-scale Graphs with In-memory Performance. *arXiv preprint arXiv:2310.00837* (2023).

[43] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 333–342.

[44] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[45] Keval Vora. 2019. {LUMOS}:{Dependency-Driven} Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 429–442.

[46] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic {I/O} Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 507–522.

[47] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 144–161.

[48] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint*

*arXiv:1909.01315* (2019).

[49] Qianwen Wang, Kexin Huang, Payal Chandak, Marinka Zitnik, and Nils Gehlenborg. 2022. Extending the nested model for user-centric XAI: A design study on GNN-based drug repurposing. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2022), 1266–1276.

[50] Haixia Wu, Chunyao Song, Yao Ge, and Tingjian Ge. 2022. Link prediction on complex networks: An experimental survey. *Data Science and Engineering* 7, 3 (2022), 253–278.

[51] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *Comput. Surveys* 55, 5 (2022), 1–37.

[52] Riting Xia, Chunxu Zhang, Yan Zhang, Xueyan Liu, and Bo Yang. 2024. A novel graph oversampling framework for node classification in class-imbalanced graphs. *Science China Information Sciences* 67, 6 (2024), 1–16.

[53] Shuo Xiao, Dongqing Zhu, Chaogang Tang, and Zhenzhen Huang. 2023. Combining Graph Contrastive Embedding and Multi-head Cross-Attention Transfer for Cross-Domain Recommendation. *Data Science and Engineering* 8, 3 (2023), 247–262.

[54] Haiyang Yu, Limei Wang, Bokun Wang, Meng Liu, Tianbao Yang, and Shuiwang Ji. 2022. GraphFM: Improving large-scale GNN training via feature momentum. In *International Conference on Machine Learning*. PMLR, 25684–25701.

[55] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[56] Bingyi Zhang and Viktor Prasanna. 2023. Dynasparse: Accelerating gnn inference through dynamic sparsity exploitation. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 233–244.

[57] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).

[58] Wentao Zhang, Ziqi Yin, Zeang Sheng, Yang Li, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Graph attention multi-layer perceptron. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4560–4570.

[59] Zhao-Bo Zhang, Zhi-Man Zhong, Ping-Peng Yuan, and Hai Jin. 2023. Improving entity linking in Chinese domain by sense embedding based on graph clustering. *Journal of Computer Science and Technology* 38, 1 (2023), 196–210.

[60] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[61] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. {GridGraph}:{Large-Scale} Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.