SSD-enabled Graph Neural Network Training on a Single Machine via In-memory Caching

Yuwei XU
yuweixu@link.cuhk.edu.cn
The Chinese University of Hong Kong, shenzhen

***Abstract*—Graph Neural Networks (GNNs) have emerged as a powerful tool for various graph-structured inference tasks, but their training scalability remains a significant challenge as graph sizes continue to grow. Distributed training is a common solution, scaling across multiple CPU nodes, but disk-based methods for single-machine systems are less explored, despite offering a cost-effective alternative. High-performance storage devices like NVMe SSDs can be leveraged to improve single-node scalability, though data movement between memory and disk presents a bottleneck. Traditional GNN training pipelines often fail to address this issue, leading to inefficiencies. In this work, we introduce three methods that improve the SSD-based GNN training system to process billion-scale graphs on a single machine. Inspired by the inspector-executor execution model in compiler optimization, Ginex restructures the GNN training pipeline by separating sample and gather stages, therefore reducing I/O overhead using Belady's optimal caching algorithm. The experiments show that Ginex achieves up to 2.67× higher throughput than existing SSD-extended GNN systems. Building on Ginex, we present OUTRE and TIGER. Although Ginex accomplishes larger percentages of data requests through caches, the enormous overall requested data volume is unchanged. To address this, we present OUTRE, focusing on reducing the overall requested data volume. OUTRE is an out-of-core GNN training framework that reduces redundancy and minimizes memory usage by dynamically loading only essential graph data. It efficiently manages graph storage on disk, enabling scalable training on large graphs without overwhelming system memory, and the training time is halved again compared to the Ginex solution. Additionally, we introduce TIGER, a GNN framework specifically designed for inductive learning on large-scale knowledge graphs (KGs). TIGER employs a novel, efficient streaming procedure that facilitates rapid subgraph slicing and dynamic subgraph caching to minimize the cost of subgraph extraction. We propose a novel two-stage algorithm SiGMa to solve the optimal subgraph slicing problem practically. By decoupling the complicated problem into two classical ones, SiGMa simultaneously achieves low computational complexity and high slice reuse. we demonstrate that TIGER significantly reduces the running time of subgraph extraction, achieving up to 7.9× speedup relative to the basic training procedure**

***Index Terms*—Article submission, IEEE, IEEEtran, journal, LaTeX, paper, template, typesetting.**

I. Introduction

RECENTLY, the success of Deep Neural Networks (DNNs) has broadened their application beyond images and texts to include graphs. Graph Neural Networks (GNNs), a new class of DNNs, have emerged as a powerful alternative to traditional graph analytics for tasks [1]–[3] like node classification [4], recommendation [5], [6], and link prediction [7]. Leveraging their expressive power, GNNs effectively capture the relational information among input nodes, enabling strong generalization performance.

The GNN training process introduces unique challenges, particularly in data preparation. Unlike traditional DNNs, where data samples (e.g., images) are independent, nodes in a graph are interconnected. For a single iteration of mini-batch training, GNNs require feature vectors not just for target nodes but also for their neighbors [4], [8]. This necessitates retrieving the L-hop neighborhood of each node, where L corresponds to the model depth, resulting in significant memory overhead when dealing with large-scale graphs.

The size of graph datasets has recently surged to several hundred GBs and even beyond one TB [9], [10]. While sampling-based GNNs can reduce memory usage by adjusting batch and sample sizes, fetching neighborhood information still requires the entire graph to reside in memory. A common solution to address memory limitations is to distribute graph storage across multiple machines and train GNNs in a distributed manner. However, previous analysis [11] reveals that the primary bottleneck in sampling-based GNN training lies in the data preparation stage, which generates a high volume of data requests. As shown in Fig. 1, subgraph extraction for large-scale Knowledge Graphs can dominate training time, accounting for 75% to 90% across three datasets. For instance, a 3-hop subgraph can include up to one million triples, leading to significant delays. Meanwhile, GPUs and other computing devices often remain underutilized during training, making multi-machine scaling inefficient. Moreover, the additional communication overhead can further exacerbate this underutilization problem.

With the rapid advancement of storage technologies, Solid State Drives (SSDs) now offer sequential bandwidths in the range of multiple GBs, and their prices have dropped significantly in recent years. As a result, researchers have started exploring the potential of incorporating SSDs into sampling-based GNN training. Disk-based GNN training presents a promising alternative, as modern NVMe SSDs provide enough capacity to store the entire input graph. However, SSDs remain several orders of magnitude slower than host memory, particularly for random access operations. Consequently, simply fetching L-hop neighborhoods from external storage can exacerbate the already time-consuming data preparation process, ultimately degrading training performance.

This paper was produced by the IEEE Publication Technology Group. They are in Piscataway, NJ.
Manuscript received April 19, 2021; revised August 16, 2021.

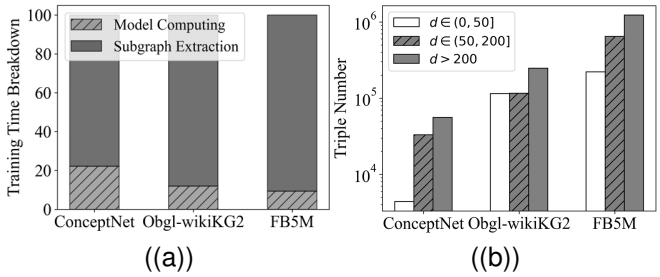


Fig. 1: (a)Training time breakdown on large graphs, (b)3-hop subgraph size of central entities with different degrees.

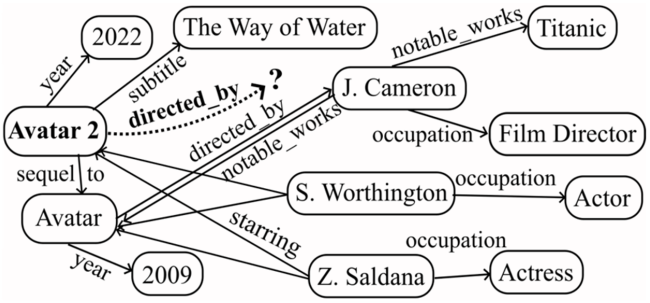


Fig. 2: Example of graph with text

Thus, we propose Ginex [12] (Graph inspector-executor), the first GNN training system based on high-performance NVMe SSDs. Ginex optimizes memory usage by implementing an in-memory caching technique, particularly enhancing the gather operation, which is the most I/O-intensive task in GNN training. Inspired by the inspector-executor model in compiler optimization [13], [14], Ginex divides the training process into two phases: Inspector Phase: Samples a sufficient number of batches to prepare an optimal caching mechanism for gathering operations. Executor Phase: Completes batch processing, utilizing the cache informed by the first phase.

However, the main focus of this dual-cache approach is to explore how to accomplish more data requests through the caches in host memory, while the overall requested data volume is unchanged. How to reduce it remains to be explored. Through quantitative analysis, we identify two redundancies in out-of-core sampling for GNN training: Neighborhood Redundancy and Temporal Redundancy. To address these, we propose OUTRE [15], an out-of-core de-redundancy framework designed to minimize data requests. OUTRE incorporates three innovations: 1) To reduce Neighborhood Redundancy, we propose constructing training batches with min-cut graph partitions [16] to decrease links between different batches' training nodes. 2) Motivated by previous work [17], [18], we approximate node embeddings with their histories to reduce Temporal Redundancy. 3) To alleviate exhausting tuning efforts, we present an automatic cache space management module to help OUTRE adapt to various configurations.

Moreover, these GNN training systems are unable to address the efficiency challenges of subgraph extraction in large-scale KGs [19]–[21]. Subgraph extraction in the large-scale inductive KG reasoning system poses three challenges: Subgraph Data Completeness, Heterogeneous Graph Structure, and SSD Storage and Access. They render other system-level acceleration techniques nearly ineffective. To address these challenges, we present TIGER [22], which supports efficiently training state-of-the-art GNN-based reasoning models on a single machine. Specifically, TIGER has four innovations. 1) To accelerate subgraph extraction while maintaining subgraph completeness, TIGER employs a novel 'Slice & Cache' procedure to precompute subgraphs in a streaming way. 2) Considering the heterogeneous graph structure, we propose a specific atom-based subgraph slicing problem. Regarding slicing quality and reusability, we prove that the atom-based slicing problem is NP-hard. 3) We design a novel two-stage subgraph slicing algorithm, SiGMa, to solve it.

This paper makes the following contributions:

- We propose Ginex [12] (Graph Inspector-Executor), the first GNN training system that leverages high-performance NVMe SSDs. We decompose the training process into two stages to optimize data handling and improve efficiency.
- We introduce OUTRE [15], an out-of-core de-redundancy framework that analyzes the minimum cut graph and employs point embedding approximation to reduce redundancies in out-of-core sampling during GNN training.
- We present TIGER [22], a framework for heterogeneous graph, especially KG reasoning tasks. We prove that the cache slice strategy, which optimizes both quality and reusability, is NP-hard. Then we propose the SiGMa algorithm to efficiently address the problem.

II. BACKGROUND

A. Graph Tasks

Node Classification is one of the fundamental tasks in Graph Neural Networks (GNNs), where the goal is to predict a label for each node based on its features and the graph's structure. In node classification, nodes are typically associated with a specific category or class, and the objective is to learn a model that can predict these labels for unseen nodes. This task is widely applicable in domains such as social networks, where the goal might be to predict the interests or demographics of users based on their connections and interactions. In citation networks, node classification can be used to categorize academic papers into different research topics. GNNs excel at this task by effectively propagating information from neighboring nodes through the graph, capturing the structural dependencies between nodes and improving classification accuracy by leveraging both node attributes and graph connectivity.

Link Prediction is another crucial task for GNNs, focusing on predicting the likelihood of an edge forming between two nodes in a graph. This task is often framed as a binary classification problem, where the objective is to predict whether a connection will appear between two nodes based on their current interactions or features. Link prediction is particularly useful in social network analysis, where it can be applied

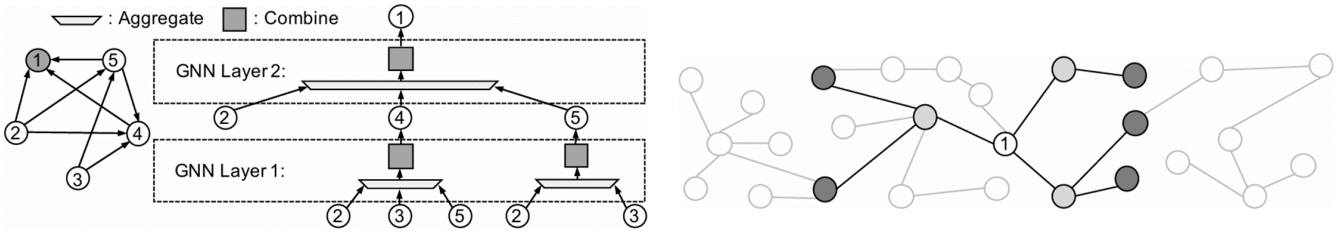


Fig. 3: (a) 2-layer GNN training on Node 1, (b) Sampling for a 2-layer GNN (sampling size = (3,2), batch size = 1).

to recommend new friendships or connections by identifying potential links based on existing social interactions. Similarly, in knowledge graph completion, link prediction is used to infer missing relationships between entities, improving the graph's comprehensiveness. By learning latent patterns of node connectivity and the structural properties of the graph, GNNs can effectively predict missing links, making them a powerful tool for dynamic graph analysis.

Graph Reasoning involves tasks that require deeper reasoning over the entire graph structure, often requiring the model to infer latent relationships or perform logical operations over the graph. This task is crucial in the context of knowledge graph reasoning, where GNNs are applied to predict relationships between entities or answer complex queries about a graph. For instance, link prediction in knowledge graphs requires the model to understand and infer logical relationships between entities, such as "is-a" or "part-of" relationships. Inductive graph reasoning, which focuses on reasoning about unseen parts of the graph, is particularly important for real-world applications such as question answering, where the graph represents knowledge that must be inferred. GNNs enable efficient reasoning by propagating information across the graph, enabling the model to generate logical inferences and extract meaningful insights from large-scale graph-structured data.

B. Graph Neural Network

GNN Training operates on graph-structured data, where each node has its own feature vector. GNNs aim to produce a quality embedding for each node in the graph capturing its neighborhood information on top of its own feature. These embeddings may be used for various downstream tasks such as node classification and link prediction. To obtain the embedding of a node, GNN takes the feature vectors of not only the target node for embedding computation, which is called seed node, but also its L-hop in-neighbors as input. Each layer in GNN is responsible for synthesizing feature information of the nodes at each hop, which means that L-layer GNN is able to reflect up to L-hop in-neighbors [4], [8].

Each layer of GNN consists of two main steps: Aggregate and Combine. The embedding of node v after the ith layer, denoted as h_v^i, is computed as the following:

$$h_v^i = \text{Combine}(\text{Aggregate}(\{h_u^{i-1} | u \in N(v)\}))$$

$N(v)$ denotes the neighbor set of node v. In Aggregate step, the features of the incoming nodes are aggregated into a single vector. While popular options for aggregation functions are simple operations like mean, max and sum, more sophisticated aggregation functions are also drawing attention [41]. The aggregated feature then goes through Combine step which is essentially a fully connected (FC) layer with a non-linear function. Fig.3(a) illustrates this process with an example of a 2-layer GNN training on Node 1.

Neighborhood Sampling. The inter-node dependence of training data poses a unique challenge to GNN training. Even if we use a small batch size, the training cost for each batch can still be quite high because collecting L-hop in-neighbors leads to exponential growth of memory footprint. Neighborhood sampling is a popular technique for this neighborhood explosion problem. Instead of sampling L-hop in-neighbors of seed nodes, sampling algorithms select only a subset of them. One representative work is GraphSAGE, which randomly samples only a predefined number of in-neighbors at each aggregation step. Fig.3(b) shows an example with a 2-hop computational graph for Node 1 being sampled. The sampling size in this example is (3,2) which means that it selects (at most) three among the neighbor nodes connected to the target node (Node 1) and (at most) two are selected for each of the previously selected nodes. Its variants differ in several aspects of sampling function design like the granularity of sampling operation or the choice of probability distribution for sampling [23], [24]. In practice, it is not usual to go beyond three layers, and popular choices of sampling size for GraphSAGE are (25, 10), (10, 10, 10), and (15, 10, 5) [25].

C. GNN Training System

The state-of-the-art DNN frameworks [26], [27] employ a mixed CPU-GPU training system, where CPU stores the graph data and is in charge of data preparation, whereas GPU executes the core GNN operations, i.e., aggregate and combine. GPU memory capacity is often fairly limited to store the graph data, while the massive parallelism of GPU is key to accelerating GNN computations. Fig.4 visualizes a typical process of mixed CPU-GPU training of GNN which consists of four steps: (1) sample, (2) gather, (3) transfer, and (4) compute. At every iteration, seed nodes for a single batch as well as their neighbors are extracted by traversing the graph structure (i.e., adjacency matrix) (sample). The adjacency matrix is usually stored in the compressed sparse column (CSC) format as it allows fast access to in-neighbors of each

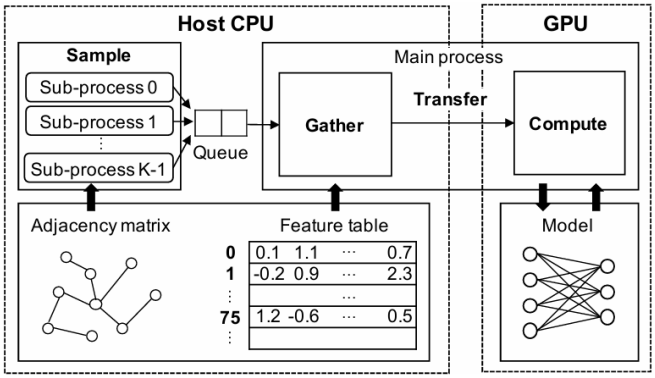


Fig. 4: Overview of conventional GNN training system

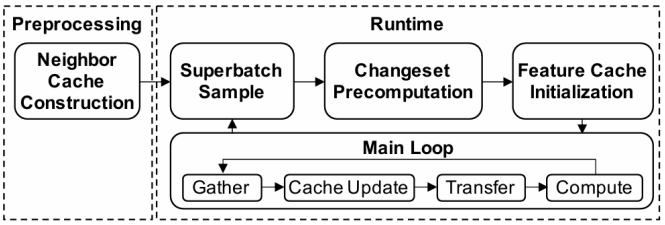


Fig. 5: Ginex training pipeline overview

node. Then, the sparsely located feature vectors of the sampled nodes are collected into a contiguous buffer (gather), which is transferred to GPU over PCIe interface (transfer). Lastly, GPU performs forward/backward propagation to compute gradients and updates the parameters (compute). Since sampling operation is often memory-intensive, it is common to spawn multiple sub-processes to increase the sampling throughput. Each sub-process performs a sampling job for a batch, and puts the result into a shared queue. The main process fetches the sampling result from the shared queue and executes the remaining jobs, i.e., gather, transfer, and compute.

III. GINEX

A. Main Design

Ginex is a system for efficient training of a very large GNN dataset by using SSD as a memory extension. Fig. 5 depicts a high-level overview of Ginex's training pipeline. After a short preprocessing procedure, Ginex starts training by iterating the following four stages: superbatch sample, changeset precomputation, feature cache initialization, and main loop. In the superbatch sample stage, Ginex performs sampling for a predefined number of batches, which we call superbatch, all at once. With the sampling results, Ginex finds all the information necessary to manage the feature cache for the following gather operations in the changeset precomputation stage. Specifically, in this stage, Ginex determines (i) which feature vectors to prefetch into the feature cache at initialization, and (ii) which feature vectors to insert and which ones to evict from the feature cache at each iteration. After a short transition stage for the feature cache initialization, Ginex completes the remaining tasks including gather in the main loop stage.

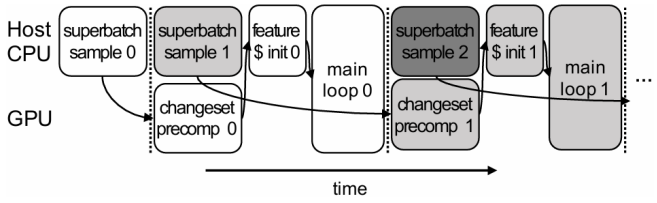


Fig. 6: Superbatch-level pipeline of Ginex

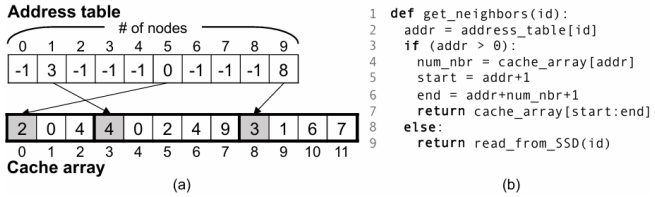


Fig. 7: (a) Ginex neighbor cache structure and (b) pseudo-code for its operation

Inspector-Executor Execution Model. The inspector-executor execution model [13], [14] is originally introduced to enable runtime parallelization and scheduling optimization of loops. An inspector procedure runs ahead of the executor to collect information that is available only at runtime, such as data dependencies among array elements. The executor is an optimized version of the original application that utilizes this runtime information to optimize data layout, iteration schedule, and so on. Ginex embraces this execution paradigm to improve the efficiency of in-memory caching for GNN training. In particular, the first two runtime stages, superbatch sample and changeset precomputation, correspond to the inspector, and the main loop stage to the executor. By running ahead the sample operation for the entire superbatch, Ginex collects complete information about the nodes to be accessed later in the gather stage, thus enabling optimal management of the feature cache.

Neighbor Cache Construction. Fig. 6 does not show this process as unlike the feature cache which dynamically manages its data, Ginex uses a static neighbor cache for the whole training process. Therefore, Ginex constructs the neighbor cache with a given size during offline preprocessing time. To make the neighbor cache, Ginex examines the graph structure (i.e., adjacency matrix) and picks out important nodes whose list of in-neighbors would be cached. After finishing this construction, Ginex saves the neighbor cache by dumping it to SSD, which would be loaded at the beginning of each of the following superbatch sample stages. This avoids the repeated cost of constructing the neighbor cache, which may include a large number of random reads of which sizes are usually only a few tens or hundreds of bytes.

B. Superbatch-level Process

Superbatch Sample. As shown in Fig. 6, superbatch sample takes up the first stage of Ginex runtime. In this stage, Ginex first loads the neighbor cache which has been constructed and stored in SSD during preprocessing. Basically, all memory

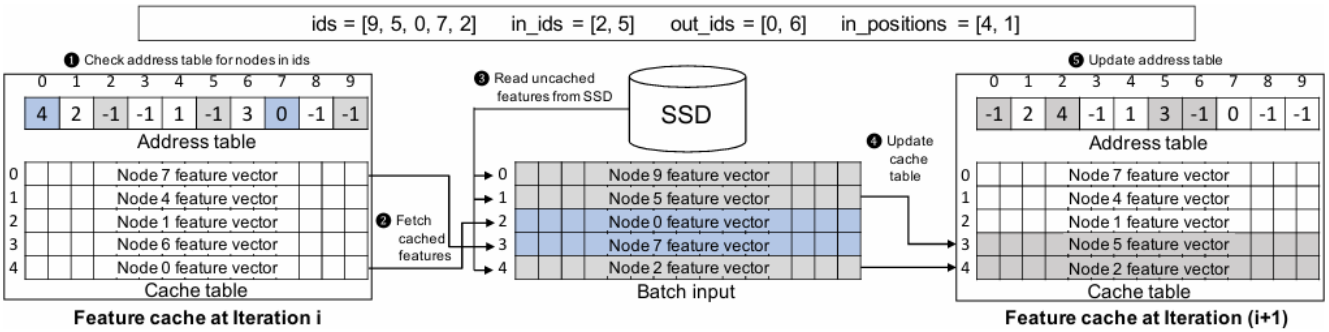


Fig. 8: Example of cache update followed by gather

space except the working buffer for sampling processes can be used for the neighbor cache. After that, multiple sub-processes are launched and sampling is started for as many batches as the superbatch size, S. When accessing the neighbor information during the sampling process, Ginex first looks up the cache and only reads the data from SSD when it is not present in the cache. The sampling results of a superbatch are then written to SSD. Usually, a sampling for each batch results in two types of data. One is ids which is an 1-D list of all the sampled nodes' IDs. The other is adj, a data structure that describes the connectivity among the sampled nodes. Ginex stores these two data in separate files annotated with the batch index. In total, $2 \times S$ files (ids_0,ids_1, ... ,$ids\_(S-1)$, adj_0, adj_1, ... , $adj\_(S-1)$) are generated. The size of each file varies depending on the sampling size, the batch size, and the characteristics of the dataset, but usually ranges from several hundred KB to a few MB.

Changeset Precomputation. Changeset precomputation is the second stage of Ginex runtime. Instead of computing a changeset (i.e., which features to insert into and evict from the feature cache) every time Ginex performs gather in main loop stage, Ginex precomputes all the changesets beforehand by examining the list of sampled nodes ids files). This is to accelerate the changeset computation in batch on GPU. It is difficult to allocate enough memory and computation resources of GPU for the changeset computation in main loop stage as it involves GPU computation. As the total size of ids files may exceed the GPU memory capacity, each ids file is first loaded on the CPU memory and then streamed into GPU when needed. The results of the changeset precomputation are sent back to CPU, and are stored in SSD also by streaming. Besides the changesets, a list of the feature vectors to prefetch into the cache at initialization is also obtained at this stage. Specifically, S + 1 files are generated in this step including one for cache initialization $init$ and the others for cache update for every S iterations ($update\_0$, $update\_1$, ... , $update\_(S-1)$).

Feature Cache Initialization. This process is performed after superbatch sample and changeset calculations are completed. In this step, Ginex reads the previously created $init$ file from SSD and constructs the feature cache. This process includes reading feature vectors of the nodes specified in $init$ file as well as building an address table that will be used for cache look-up.

Main Loop. This stage is where all the remaining GNN training operations for each batch (gather, transfer, and compute) are performed iteratively. In addition, Ginex performs one more operation, cache update, in between. At each iteration, Ginex reads a set of ids, adj and $update$ files from SSD in the order of the batch index. Then, Ginex makes batch input by gathering feature vectors according to the ids file from either the cache or SSD, and updates the cache as indicated in the $update$ file. Lastly, the batch input and adj are transferred to GPU in order to perform forward and backward pass as well as model update in the same way as the conventional GNN training system.

Superbatch-level Pipeline. While the four runtime stages for the same superbatch should be serialized, the jobs from different superbatches can be pipelined. Taking this opportunity, Ginex performs the jobs for different superbatches in a pipelined manner in order to improve end-to-end performance. Specifically, changeset precomputation for each superbatch is executed in parallel with the superbatch sample of the next superbatch. $superbatchsample$ runs on CPU, while changeset precomputation mainly consumes GPU resources except the I/O overhead of streaming ids files and the changeset precomputation results. This makes these two stages apposite candidates of parallel execution. Figure 9 visualizes Ginex's superbatch-level pipeline. While the storage overhead of runtime files is doubled as a result of pipelining, it successfully hides most of the changeset precomputation overhead.

Implications on Training Quality. The new training schedule of Ginex has no impact on training quality, as it only changes the execution order of operations which do not have any dependence with each other. It does not require any change in sampling algorithm or GNN model computation.

IV. OUTRE

A. Redundancy Analysis

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.

During Data Preparation, the training framework accesses the adjacency and feature matrices from external storage. We

TABLE I: 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 II: Redundancy Ratio on two OGB datasets.

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 III: 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 IV: 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%**)

analyze the total data volume requested and introduce a new metric, "Redundancy Ratio" (RR), defined as:

$$RR = \frac{\Sigma_{b=1}^{B}|\tilde{N}_b|}{\Sigma_{b=1}^{B}|N_b|}.$$

Here, $\tilde{N}_b$ represents the collective L-hop neighborhood size, and N_b is the training node set for the b-th batch. The sum $\Sigma_{b=1}^{B}|\tilde{N}_b|$ indicates the overall data volume requested during preparation. Since out-of-core sampling-based GNN training is limited by this phase (as shown in Table I), RR is a relevant metric for assessing training performance. To illustrate RR's effectiveness, we train a GraphSAGE model and present the results in Table II, which shows that the total collective L-hop neighborhood sizes can exceed one hundred times the number of training nodes and even surpass the graph size.

Neighborhood Redundancy. Most GNN training frameworks create batches by randomly selecting training nodes and sampling L-hop neighborhoods. This random selection leads to the significant overlap in the sampled neighborhoods across different batches, termed "Neighborhood Redundancy". Reducing this overlap is crucial for minimizing redundancy, which can be achieved by increasing the number of edges among training nodes within each batch.

To validate this approach, we employ the min-cut graph partitioning algorithm METIS [28] to divide the graph into 10,000 parts, maximizing intra-partition edges. We then randomly select partitions, shuffle nodes, and construct batches. By shifting from random selection to partition-based selection, we observe a reduction in Redundancy Ratio of over 25% across both datasets, as shown in Table III. This significant decrease indicates that enhancing intra-batch connectivity effectively lowers the overall data volume requested, thereby accelerating out-of-core sampling-based GNN training.

Temporal Redundancy. Existing research [17], [18], [29] suggests that most node embeddings undergo only modest changes across training iterations, leading to efforts to approximate them using historical data. This strategy leverages the insensitivity of deep learning models to small errors. In out-of-core settings, reusing historical embeddings reduces both computational redundancy and "Temporal Redundancy", which refers to unnecessary data requests from external storage. By substituting current embeddings with historical ones, we eliminate redundant data requests.

Given the limitations of host memory, we implement a cache for historical embeddings, focusing on high-degree nodes for reuse. Evaluation results in Table IV for a 3-layer GraphSAGE model indicate a significant reduction in Redundancy Ratio when 20% of nodes reuse their historical embeddings suggesting improved performance in out-of-core sampling-based GNN training.

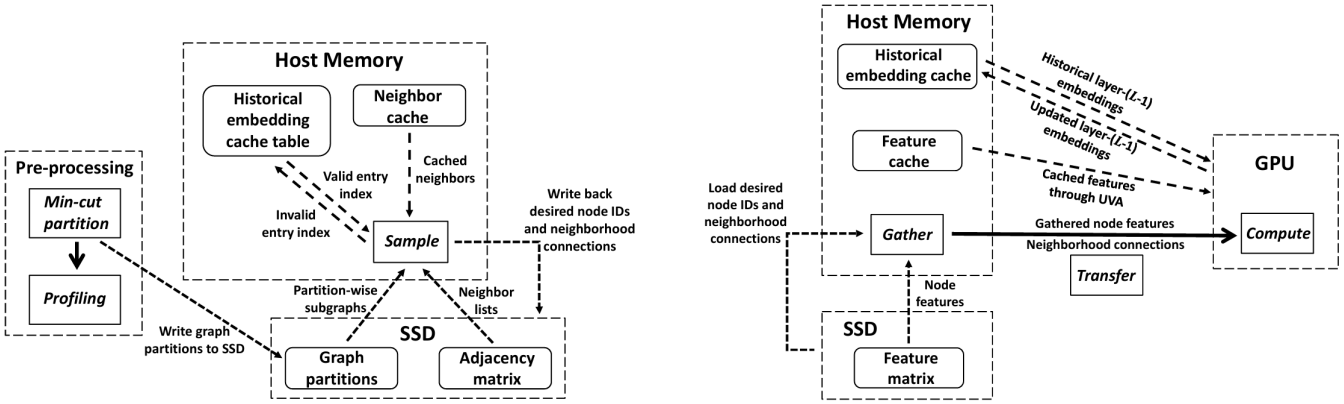
B. OUTRE Design

In this section, we introduce our proposed out-of-core sampling-based GNN training framework OUTRE in detail. Firstly, we provide an overview of OUTRE, where we describe the modifications we make to the conventional four-stage training pipeline. Then we introduce the three main designs of OUTRE: partition-based batch construction, historical embedding cache, and automatic cache space management, respectively.

1) Training Pipeline Overview: Figure 9 illustrates the workflow of our proposed framework, OUTRE, which is built upon the existing dual-cache framework, incorporating both a neighbor cache and a feature cache. OUTRE introduces two key modifications to the conventional four-stage training pipeline:

Pre-processing Stage Addition. OUTRE adds a pre-processing stage before the conventional pipeline. In this stage, it executes an out-of-core min-cut graph partitioning algorithm to divide the graph into smaller parts, which are later utilized to construct training batches. Following this, OUTRE conducts a profiling epoch that includes only the Sample and Gather stages. This profiling collects essential information for the automatic cache space management module, enabling it to determine the optimal memory allocation for different caches.

Decoupling of the Sample Stage. OUTRE decouples the Sample stage from the subsequent three stages, as suggested by **Ginex** [12]. In the conventional four-stage pipeline, all stages are executed consecutively for each training batch, requiring both the neighbor cache (needed for the Sample stage) and the feature cache (needed for the Gather stage) to reside in host memory simultaneously. By decoupling the Sample stage, OUTRE allows these two caches to occupy host memory exclusively at different times. This change enables a larger amount of graph data to be cached, which can enhance training performance. To implement this decoupling, OUTRE writes the sampled collective L-hop neighborhoods back to external storage after the Sample stage. Before the Gather stage, OUTRE retrieves the sampled results for each training batch from external storage and proceeds with the remaining stages as in the conventional pipeline.



(a) Pre-processing and *Sample* stage

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

Fig. 9: The workflow overview of OUTRE

2) Main Designs: OUTRE incorporates three main designs that enhance its efficiency and performance in GNN training: (1) partition-based batch construction, (2) historical embedding cache, and (3) automatic cache space management. Each of these designs addresses specific types of redundancy and optimizes the overall training process.

Partition-Based Batch Construction. This design aims to reduce Neighborhood Redundancy. In OUTRE, each training batch is constructed from several randomly selected graph partitions created by the min-cut graph partitioning algorithm during the pre-processing stage. By increasing intra-batch connections, the overlap in collective L-hop neighborhoods among different training batches is significantly minimized. This reduction leads to a considerable decrease in the overall data volume requested during the Sample and Gather stages, thereby accelerating out-of-core sampling-based GNN training.

Historical Embedding Cache. To mitigate Temporal Redundancy, OUTRE implements a historical embedding cache that is utilized during both the Sample and Compute stages. During the Sample stage, this cache informs the main process about which nodes can be skipped based on the states of the encountered nodes. This selective sampling reduces unnecessary computations. In the Compute stage, the GPU retrieves the relevant historical node embeddings from the cache and updates them based on the information stored in the historical embedding cache table. This dual-stage involvement of the historical embedding cache enhances efficiency by leveraging previously computed embeddings, thereby reducing redundant calculations.

Automatic Cache Space Management. OUTRE features an automatic cache space management system designed to optimize the sizes of different caches dynamically. This mechanism enables OUTRE to maintain robust performance across various datasets and hardware configurations without requiring extensive manual tuning. Specifically, OUTRE employs the data collected during the profiling epoch in the pre-processing stage to automatically determine the optimal memory allocation for the feature cache and the historical embedding cache. This low-cost approach ensures that resources are utilized effectively, adapting to the specific needs of the training process.

V. TIGER

To implement the Slice & Cache procedure for highly efficient subgraph extraction, we design a novel training framework, TIGER, tailored for large-scale inductive KG reasoning. We first overview the TIGER training pipeline. Then we describe two core components of TIGER, atom-level subgraph slicing, and subgraph caching.

A. TIGER Training Pipeline

Fig10 illustrates the training pipeline of TIGER. Given the KG triple data stored on the SSDs, TIGER initially undertakes Atom Cache Construction, loading a portion of atom triples (1-hop subgraphs) into the main memory to accelerate the subsequent subgraph extraction. Following this, TIGER starts model training by iterating the Super-batch Loop, which is specifically designed to pre-compute input data of multiple batches before mini-batch training, thus reducing repetitive calculations and facilitating the upcoming cache mechanism. A super-batch loop starts with three precomputation stages: Query Sampling, Subgraph Slicing, and Subgraph Caching, where all required subgraphs are reconstructed into uniformly sized slices and stored in either the Slice Cache or SSDs. Subsequently, the super-batch loop trains multiple batches of sampled queries through continuous Mini-batch Loops, leveraging the precomputed slices to minimize subgraph extraction costs. As the super-batch loops progress, the precomputation time decreases due to the dwindling number of unsliced subgraphs. Notably, such subgraph precomputation ensures efficient training without affecting models, thus there is no sacrifice for effectiveness.

AtomCache Construction. The Atom Cache, established at the start and unchanging in subsequent super-batch loops, aims to reduce the cost of reading atoms (e.g. 1-hop query subgraphs) from SSDs. It contains two cache structures within a predefined size: one as a 1-D array for storing neighbor entity IDs and another for atom triples of high-degree entities. This

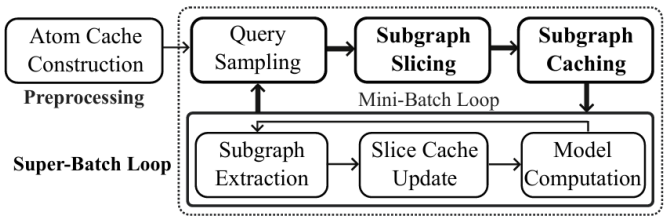


Fig. 10: The training pipeline overview of TIGER

cache employs direct addressing, recording each entity's array index and data length to achieve swift O(1) cache lookups.

Query Sampling. Given the hyper-parameter $superbatch\_size$ determining the number of mini-batches, this stage samples all batches of training queries together. Unlike the basic pipeline samples within each mini-batch loop, TIGER preemptively identifies the specific subgraphs each batch will access prior to mini-batch training. This foresight enables the implementation of an efficient caching mechanism for subgraph extraction.

Subgraph Slicing. This stage is crucial for optimizing subgraph extraction. It extracts unsliced subgraphs accelerated by the Atom Cache, and segments each of them into uniformly sized slices. The results are recorded in a mapping dictionary that links query entity IDs to their corresponding slice IDs, enabling direct retrieval from sequential slice data and significantly reducing random SSD access. To improve efficiency, we introduce atom-level subgraph slicing, which transforms subgraphs into disjoint atoms. This approach reduces both time complexity and memory usage during the slicing process.

Subgraph Caching. After slicing, the next stage involves storing the generated slices on SSDs. We propose an atom-based slice structure that reduces SSD storage requirements by a factor of three. To decrease SSD access frequency, some slices are cached in main memory, referred to as the Slice Cache, organized in a 2-D array using direct addressing. Additionally, we implement a robust caching mechanism that precomputes the next-access slices for each mini-batch, facilitating dynamic cache updates during mini-batch training and enhancing slice data loading efficiency.

Mini-batch Loop. In this stage, given a batch of queries, the associated slice IDs are retrieved from the slice mapping dictionary. The corresponding slice data is first extracted from the Slice Cache and then from the SSDs, which is used to construct the batch subgraph. After that, the Slice Cache is updated with the precomputed changeset for this batch to enhance the cache hit ratio in subsequent mini-batch loops. Finally, TIGER transfers batch queries and complete subgraphs into the GPU to compute the GNN-based model.

B. Atom-Level Subgraph Slicing Problem

Problem Formulation. To minimize the time complexity and memory usage of the subgraph slicing process, we transform the triple-level subgraph slicing problem into an atom-level representation. An atom-based subgraph is defined as $G_q = \{(e_a, w_a)|e_a \in N_q^{L-1}, w_a = |\mathcal{G}_{E_a}^1|\}$, where $|G_q| = \Sigma_{a\in G_q} w_a$ and w_a denotes the atom weight (i.e., the number of atom triples). The formal definition of the atom-level subgraph slicing problem is as follows:

Definition 1: (Atom-level Subgraph Slicing). Given a predetermined slice size h and a collection of query entities Q, the corresponding atom-based subgraphs are $G = G_q|q \in Q$ and the collection of atoms appearing in G are denoted as $A = \{a|w_a < h\}$. The atom-level subgraph slicing problem is to construct a collection of slices S satisfying that: each slice $S_i \in S$ consists of multiple distinct atoms in A and the total weight of atoms $|S_i| = \Sigma_{a\in S_i} w_a \leq h$; any subgraph $G_q \in G$ can be composed by a set of non-overlapping slices $S-q \subseteq S$, i.e., $G_q = \cup S_q$.

Optimization Target. The optimization objective for atom-level subgraph slicing is to load all subgraph triples while minimizing the number of slices accessed. Given the sizes of subgraphs and slices, the required number of slices depends on two key factors: $SliceRedundancy$ and $SliceUtilization$. Thus, we define the optimization target, referred to as the $SlicingScore$, as follows:

$$f(\mathbf{G},\mathbf{S},h) = \frac{|\mathbf{S}|}{\sum_{G_q\in\mathbf{G}}\lceil|G_q|/h\rceil} = \underbrace{\frac{\sum_{G_q\in\mathbf{G}}|\mathbf{S}_q|}{\sum_{G_q\in\mathbf{G}}\lceil|G_q|/h\rceil}}_{\text{Slice Redundancy Rate } \delta_R} \times \underbrace{\frac{|\mathbf{S}|}{\sum_{G_q\in\mathbf{G}}|\mathbf{S}_q|}}_{\text{Slice Utilization Rate } \delta_U}, \quad (1)$$

where $|\mathcal{G}_q|$ and $|\mathbf{S}_q|$ represent the number of subgraph triples and slices, respectively. The $slicingscore$ is defined as the ratio of distinct slices in $\mathbf{S}$ to the minimum number of slices required for each subgraph, based on the goals of atom-level slicing. According to Equation1, this score can be broken down into two metrics: $SliceRedundancyRate$ (δ_R) and **Slice Utilization Rate** (δ_U). δ_R measures the ratio of actual loaded slices to the minimum required, with lower values indicating reduced redundancy. δ_U assesses slice reuse by comparing the number of distinct slices to the total required. Therefore, subgraph slicing is considered optimal when the slice set $\mathbf{S}$ minimizes the slicing score, which is expressed as $\delta_R \times \delta_U$.

THEOREM 1: Minimizing $\delta_R \times \delta_U$ to obtain an optimal atom-based subgraph slicing is NP-hard.

PROOF 1: We establish the NP-hardness of our target problem, atom-based subgraph slicing, by reducing the classical Bin Packing problem to it. Given multiple items/atoms with different weights, Bin Packing is to assign each atom to a bin of size h such that the total number of bins used is minimized. It is clear that this Bin Packing problem is equivalent to the special case of our target problem where we are slicing only one query subgraph G_q. In this case, the slice utilization rate δ_U is fixed at 1 and the optimization target transforms to minimizing the slice redundancy rate δ_R. If a polynomial-time algorithm solves our problem, it implies one exists for the NP-hard Bin Packing problem. Consequently, atom-based subgraph slicing must also be NP-hard.

THEOREM 2: Given the subgraph set $\mathbf{G}$ and atom set $\mathbf{A}$, the metric δ_R of a slicing result $\mathbf{S}$ has a lower bound:

$$\delta_R \left\lceil \left(\sum\nolimits_{a\in\mathbf{A}} w_a\right)/h \right\rceil / \left(\delta_U \cdot \sum\nolimits_{G_q\in\mathbf{G}} \lceil|G_q|/h\rceil\right) \quad (2)$$

PROOF 2: According to the definition of the slicing score, $\delta_R \times \delta_U = f(\mathbf{G},\mathbf{S},h) = |\mathbf{S}|/(\sum_{G_q\in\mathbf{G}}\lceil|G_q|/h\rceil)$. Meanwhile, the total slice number $|\mathbf{S}| \left\lceil \left(\sum_{a\in\mathbf{A}} w_a\right)/h \right\rceil$. Because $\delta_U 1$ and the other terms are constants, the lower bound holds.

C. Subgraph Slicing Algorithm

To address slice redundancy and utilization, we propose a two-stage algorithm, SiGMa, for atom-based subgraph slicing. The algorithm first reuses existing slices and then assigns unmatched atoms into multiple slices. We note the complex relationship between slice quality and the functions for slice generation and matching. The slice-generating process directly influences the redundancy rate δ_R, while slice matching affects the reuse of existing slices (δ_U). Additionally, generated slices impact reuse frequency, and effective matching can improve δ_R by avoiding high-redundancy slices. Thus, we develop two specific algorithms for slice generation and matching, focusing on both δ_R and δ_U while ensuring efficiency.

Slice Generating Algorithm. The goal of slice generation is to create a few h-length slices $\mathbf{S}_{gen}$ that together form the input atom set $G'_q = \{a_1, a_2, \cdots, a_m\}$ (where $m = |G'_q|$). These atoms are sourced from the same atom-based subgraph G_q, with each atom's weight w_a not exceeding h. According to Theorem 2, to minimize δ_R while reducing δ_U, the payload of each slice should be as close to h as possible, thereby minimizing the number of slices $|\mathbf{S}_{gen}|$. This minimization is akin to the NP-Hard Bin Packing problem, where items of varying sizes must fit into the fewest bins of fixed capacity. We will first explore feasible solutions to this classical problem: Here, we first list some feasible solutions to this classical problem: leftmargin=0.3cm

- *Naive Solution (Next-Fit Algorithm, NF)*
- *Greedy Solution (First-Fit Decreasing Algorithm, FFD)*
- *Optimal Solution*

Although NF and FFD algorithms can produce good solutions in a reasonable timeframe, they do not consider the optimization of the slice utilization rate δ_U. A clique in which atoms are connected densely with others usually appears in more subgraphs than a random combination of atoms. To this end, we design a slice-generating algorithm that indirectly controls the slice utilization by adjusting the atom processing order, as shown in Algorithm 1. Specifically, different from the FFD algorithm uses the decreasing order of atom weights, we visit the atom list with the Postorder Depth-First Search thereby decreasing the distances among atoms in a local window. Because the input atom list G'_q is usually not a complete subgraph, starting from only one atom may not cover the whole list. Therefore, we traverse the k-hop neighbors of q in G'_q as multiple root nodes (Line 2). For each neighbor atom u, the algorithm checks if u is in G'_q, pushing u onto the atom stack L_{stack} (Lines 3-5). While L_{stack} is not empty, it retrieves the top atom a. If a is unvisited, mark it as visited, gather its unvisited neighbors, sort by atom weights, and finally push them onto the stack (Lines 6-11). If a is visited, it is popped from L_{stack} and inserted into one slice in $\mathbf{S}_{new}$ via the FFD algorithm (Lines 12-14). On Lines 15-16, after searching the whole branch of one neighbor, the generated slices are filtered by the capacity threshold $alpha$ to ensure a low δ_R. The atoms in low-capacity slices would be re-packed in the next iteration. Finally, on Line 17, the rest atoms would be packed via the FFD algorithm.

Algorithm 1: Slice Generating

Input : Atom list $G'_q = \{(e_a, w_a)\}$, slice capacity h, hop number k, capacity threshold α.
Output: Generated slices $\mathbf{S}_{gen}$.

1 Initialize the slice set $\mathbf{S}_{dfs}$;
2 **foreach** *k-hop neighbor atom u of q* **do**
3 **if** $u \notin G'_q$ **then continue**;
4 Initialize a collection $\mathbf{S}_{new}$ and an atom stack L_{stack};
5 Push the atom u to L_{stack};
6 **while** *L_{stack} is not empty* **do**
7 $a \leftarrow$ top atom of L_{stack};
8 **if** *a is not visited* **then**
9 Mark a as visited;
10 Gather unvisited neighbor atoms w of a in G'_q;
11 Push all w to L_{stack} sorted by weight;
12 **else**
13 $a \leftarrow$ pop L_{stack};
14 Bin packing $\mathbf{S}_{new} \leftarrow FFD(\mathbf{S}_{new}, \{a\})$;
15 $\mathbf{S}_{fill} \leftarrow \{S_i \in \mathbf{S}_{new} | size(S_i) \geqslant \alpha\}$;
16 $\mathbf{S}_{dfs} \leftarrow \mathbf{S}_{dfs} \cup \mathbf{S}_{fill}$, $G'_q \leftarrow G'_q - \bigcup \mathbf{S}_{fill}$;
17 Bin packing the rest atoms: $\mathbf{S}_{gen} \leftarrow \mathbf{S}_{dfs} \cup FFD(\varnothing, G'_q)$;

Slice Matching Algorithm. Slice matching aims to align an atom-based subgraph G_q with existing slices from the slice data $\mathbf{S}$. The goal is to select disjoint slices $\mathbf{S}_{mat} \subseteq \mathbf{S}$ that are proper subsets of G_q while maximizing their union. This problem resembles the NP-hard Set Cover problem but requires each selected slice to be non-overlapping, minimizing the Slice Redundancy Rate (δ_R). Although fuzzy matching can improve slice utilization, it increases computation and introduces redundant triples.

The slice matching algorithm consists of two stages. In the first stage, we gather the k-hop neighbors N_q^k of query q and collect slices $\mathcal{S}_N$ relevant to these neighbors. These slices are sorted by historical utilization and matched preferentially. This approach significantly reduces the size of the atom set G'_q. In the second stage, we match slices containing atoms from the remaining G'_q, excluding those that intersect with it. We sort candidate slices by capacity, matching those with more atoms first and ensuring that slices with a payload below a threshold α are removed. The ratio of slice matching $|G'_q|/|G_q|$ is influenced by various factors, but the total capacity of matched slices is guaranteed to be at least α.

VI. EXPERIMENT

A. Experimental Setup

System Configurations. All the methods discussed in this article utilize caching to enhance GNN training, resulting in minimal performance disparity between the CPU and GPU used. Each method is compared against a baseline in the experiments. Notably, the CPUs employed in the three methods all feature a minimum of 16 cores. For GPU resources, both Ginex and OUTRE utilize NVIDIA V100 GPUs, with Ginex equipped with 16GB of video memory and OUTRE with 32GB. Additionally, both OUTRE and TIGER allocate 64GB

TABLE V: Summary of Datasets

Dataset	Nodes	Edges	Dataset Size
ogbn-papers100M	444.24M	14.24B	569GB
ogbn-products	220.41M	20.24B	388GB
com-friendster	262.43M	15.48B	393GB
twitter-2010	208.26M	14.05B	326GB
mag240M-cite	121.75M	2.60B	385GB
IGB-medium	10.00M	120.08M	40.8GB
IGB-large	100.00M	1.22B	401.8GB
Knowledge Dataset	**Nodes**	**Edges**	**Relations**
OgblKG2	2.50M	17.13M	535
FB5M	3.98M	17.87M	7523
ConceptNet	28.37M	34.07M	50
Freebase	86.05M	338.67M	14851

of memory. All three methods leverage SSDs with substantial storage capacity.

Datasets. This survey evaluates several key datasets used for large-scale graph neural networks (GNNs) and knowledge graph (KG) reasoning, and the statistics of these datasets are given in Table2.: Graph Datasets: ogbn-papers100M(papers) [30], ogbn-products(products) [30], com-friendster [31], and twitter-2010(Twitter) [31] are scaled following the methodology in [32]. Specifically, we use a graph expansion technique, which adapts Kronecker graph theory [33] to preserve innate distributions of recipe graphs like power-law degree distribution and community structure. Ogbn-papers100M and mag240M-cite are from Open Graph Benchmark(OGB) [30].Mag240M-cite is a homogeneous graph containing only "paper" nodes and "paper-cite-paper" edges of the original mag240M graph. Knowledge Graph Datasets: IGB-medium and IGB-large are from Illinois Graph Benchmark(IGB) [10]. The two IGB datasets have significantly larger training sets than the two OGB datasets. For Knowledge Graph, OgblwikiKG2 is derived from Wikidata [30], while FB5M is a Freebase subset [34].

Models In the evaluation of Ginex, a 3-layer GraphSAGE [35] and a 2-layer GCN [36] are employed, both configured with a hidden dimension of 256, and the sampling size for GraphSAGE is set to (10, 10, 10). For OUTRE, a 3-layer GraphSAGE [35] serves as the primary comparison model, also with a hidden size and sampling size of 256 and (10, 10, 10), respectively, while the default batch size is set to 1,000. This method further extends its comparisons to include 2-layer and 3-layer GAT and GCN models in the main experiment. TIGER utilizes six recent GNN-based models for inductive KG reasoning, specifically REDGNN [37], NBFNet [38], AdaProp [39], A*Net [40], GraPE [41], and RUNGNN [42], with default settings of a hidden dimension of 32, a layer number L of 3, and a batch size of 16. The models are trained by minimizing multi-class cross-entropy loss, and the triples in the graph G are augmented with reverse and identity relations, employing two evaluation metrics for the KG reasoning task.

Comparison Baselines. In evaluating Ginex, we compare it with two baselines: PyG+ and Ali+PG. PyG+ is a modified PyG [26] framework that supports disk-based GNN training using a memory-mapped graph dataset, created with NumPy's memmap function for efficient I/O operations. Ali+PG enhances PyG+ with an in-memory caching mechanism, utilizing Aligraph-style [43] and PaGraph-style caches for neighbor

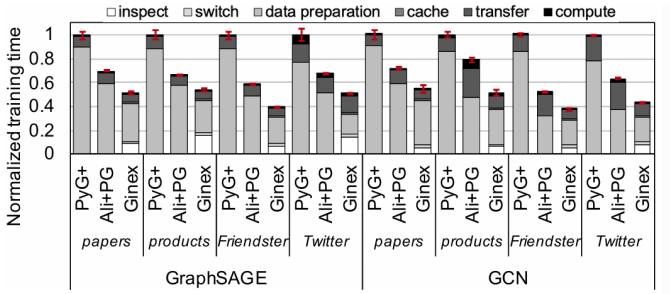


Fig. 11: Normalized training time breakdown of PyG+, Ali+PG, and Ginex. Smaller is better.

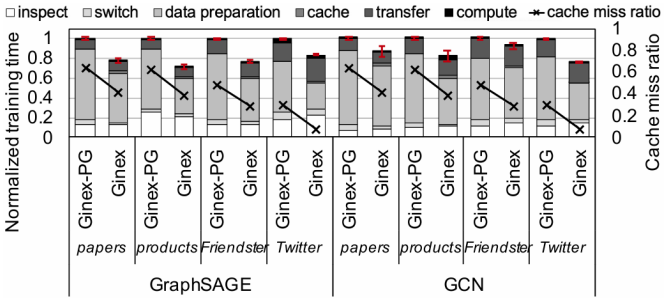


Fig. 12: Normalized training time breakdown of Ginex-PG and Ginex

and feature caching, respectively. For OUTRE, we compare PyG+mmap and Ginex, with the former allowing the reading of memory-mapped adjacency matrices and node features. We also introduce "Ginexmod," which caches graph data and enables GPU access to cached node features in host memory via UVA. In the TIGER evaluation, we compare with three baselines: Basic, Atom, and Ginex. The Basic baseline represents the standard training pipeline for inductive GNN models, while Atom uses AtomCache to load subgraph atoms, addressing the bottleneck of SSD random access. Ginex, noted for its robust caching mechanism, divides subgraph triples into equal-length slices without reuse, which can create a bottleneck due to the volume of slices. Our focus is specifically on optimizing the subgraph extraction process, excluding other aspects of GNN training from our experiments.

B. Overall Performance

Ginex. We measure the training time breakdown of PyG+, Ali+PG, and Ginex across four datasets. For GraphSAGE, the Ginex superbatch sizes are set to 3300, 2100, 3600, and 6400 for $papers$, $products$, $Friendster$, and $Twitter$, respectively. For GCN, the superbatch sizes are 2500, 300, 900, and 900. These values are derived from the offline profiling heuristic in Section 3.5. The actual size of the runtime files is within 3% of the 100 GB target.

Fig. 11 shows the results. Training time for PyG+ and Ali+PG is broken down into data preparation, transfer, and compute. Data preparation time is the CUDA stall time caused by sample or gather. Ginex's training time, on the other hand, includes additional components: inspect, switch, and cache

TABLE VI: 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**

update. Inspect time corresponds to the pipelined execution of superbatch sampling and changeset precomputation, where the changeset time is hidden by superbatch sampling. Switch time refers to the initialization of the feature cache, and Ginex's data preparation time includes gather and runtime file loading, excluding sample time.

Ginex outperforms all other workloads. For GraphSAGE, speedups range from 1.86× to 2.50× over PyG+ and 1.23× to 1.47× over Ali+PG. For GCN, speedups range from 1.83× to 2.67× over PyG+ and 1.28× to 1.57× over Ali+PG. These gains result from the efficient caching scheme for gather, which outweighs its cost, while the overhead of sample and gather serialization is minimal. The switch and cache update times contribute less than 10% to the total. The moderate storage cost of about 145 GB (including runtime files and neighbor cache) is less than half the size of the smallest dataset.

In Fig. 12, we compare Ginex with a PaGraph-based cache policy (Ginex-PG) to isolate the impact of Ginex's feature cache. Ginex consistently shows a lower cache miss ratio, leading to reduced data preparation time. While the optimal cache introduces a slight increase in inspect time due to disk I/O during changeset precomputation, this increase is limited to under 20%, which is small relative to the reduction in data preparation time.

OUTRE. We report the per-epoch time for the four compared frameworks across four datasets in Table VI. The evaluation includes training 2-layer and 3-layer GraphSAGE, GAT, and GCN. The per-epoch time is broken down into four stages: Sa (Sample), Ga (Gather), Tr (Transfer), and Co (Compute). The Gather stage is pipelined with the later training stages in Ginex, Ginexmod, and OUTRE, which results in a per-epoch time lower than the sum of all four stages. Ginex and Ginexmod's pre-computation time for the optimal cache policy is included in their Gather times since it accelerates the Gather stage.

Table 6 shows that OUTRE consistently outperforms all baselines across different model configurations and datasets. With techniques that improve data transfer efficiency, Ginexmod outperforms Ginex on all evaluation workloads. We observe that the choice of GNN model has little effect on overall training performance, confirming that out-of-core sampling-based GNN training is often bounded by data preparation. Intuitively, the number of model layers is positively correlated with data redundancy. Evaluation results reveal that OUTRE demonstrates larger average speedups over Ginex for 3-layer workloads (2.45×) compared to 2-layer workloads (1.78×).

It is important to note that Ginex shows larger per-epoch times than PyG+mmap on training 2-layer GNNs on ogbn-papers100M due to overhead. We also find that PyG+mmap achieves the highest sampling performance among all frameworks, despite not caching important neighbor lists. This is likely because the other three frameworks must write sampling results to external storage, which slows down sampling execution.

This improved performance in OUTRE can be attributed to its efficient handling of data transfer and the optimized cache policy, which reduces redundant data processing during training, similar to the gains observed with Ginex in terms of caching and gather stage efficiency.

TIGER. We present the end-to-end running time for a single training epoch of six GNN-based models in Figure 13. These models are trained using both TIGER (Our) and three baselines. The total running time is divided into three components: subgraph slicing, subgraph extraction, and model computation. The slicing time, which includes storage, is significant only during the initial training epoch. In subsequent epochs, the model can directly extract subgraph slices. As a result, we observe that TIGER, without subgraph slicing, significantly outperforms the Basic pipeline, with speedups ranging from 1.9× to 7.9×. The reduction in time becomes more pronounced as the number of epochs increases, due to the elimination of redundant extraction operations.

The Atom baseline outperforms the Basic pipeline due to preloaded atom-based subgraphs; however, extracting atoms via SSD random access still consumes considerable time. In contrast, the lower extraction time in both Ginex and TIGER highlights the effectiveness of slice-based subgraph extraction, which benefits from more efficient SSD sequential access. Notably, the subgraph slicing time in TIGER is less than 65% of that required by the Ginex baseline. While Ginex's

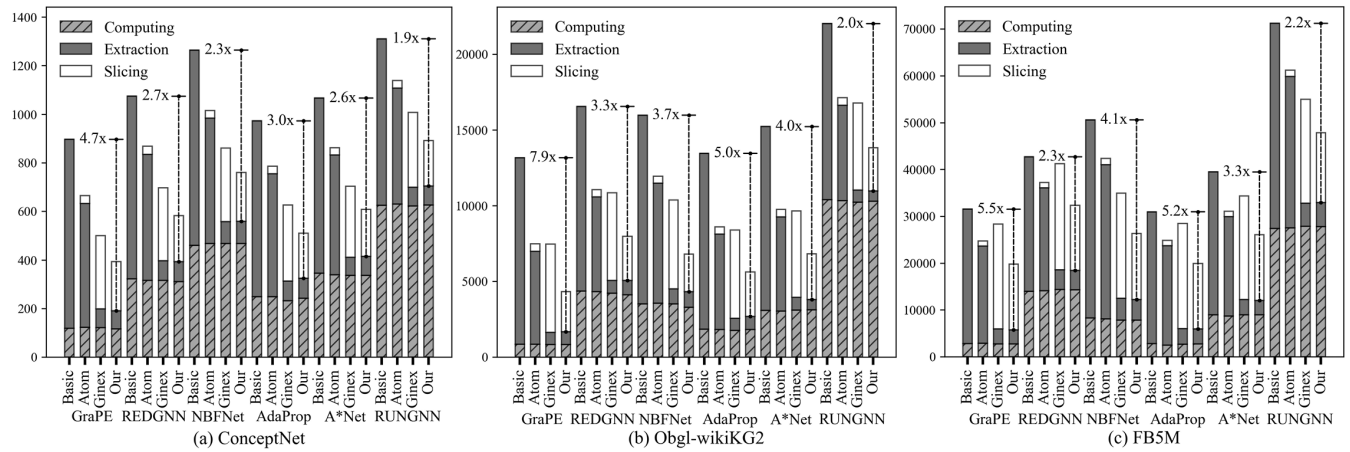


Fig. 13: Comparison of running time (s) for one training epoch on three large-scale KG datasets.

sequential slicing is more efficient than SiGMa, it results in a larger number of slices being stored, which impacts overall performance.

In comparing the six GNN models, GraPE stands out due to its notable speedup, which can be attributed to its efficient model computation achieved through path pruning. Overall, speedup correlates with the duration of the GNN model computation. Models like AdaProp and A*Net, which use path pruning techniques, show higher speedups in TIGER compared to REDGNN and NBFNet. Among these, RUNGNN—an enhanced version of REDGNN—emerges as the slowest, demonstrating the benefits of optimization strategies like path pruning in reducing computation time.

The performance gains observed in TIGER, particularly in subgraph extraction, can be attributed to its efficient slicing and sequential access, similar to the optimizations seen in Ginex. Both systems minimize redundant operations, though TIGER further optimizes this by reducing subgraph slicing time and improving data retrieval efficiency.

VII. RELATED WORK

Scalable Graph Neural Networks. To the best of our knowledge, Ginex is the first to leverage SSDs for scaling GNN training. GLIST [44] also uses SSDs to scale GNNs but focuses on inference and requires specialized hardware. In contrast, most large-scale GNN training methods adopt scale-out approaches. ROC [45] and NeuGraph [46] propose multi-GPU training systems for GNNs; however, they rely on full-batch training, which eventually hits the GPU memory capacity wall when training on very large graphs. Distributed systems utilizing multiple CPU nodes for graph storage offer more scalable solutions [43], [47]–[49]. These systems partition the graph dataset and store it in memory across a cluster, but their high system cost limits their cost-effectiveness.

Reusing Historical Node Embeddings. Reusing historical node embeddings was first proposed by [50] and later generalized by GAS [18]. GraphFM [51] introduces the Feature Momentum technique, applying momentum steps to historical embeddings, outperforming GAS. These methods store historical embeddings for all nodes and model layers, resulting in high memory costs. ReFresh [52] proposes metrics based on staleness and gradients to control both the size and quality of historical embeddings. OUTRE's historical embedding cache largely follows ReFresh's design but introduces key modifications to better suit out-of-core environments. Specifically, we cache only second-layer node embeddings to achieve a better performance-cost ratio. Additionally, we pre-define cache candidates to strictly control cache size and introduce a new node importance metric for cache candidate selection, reflecting real-world training performance.

Large-scale KGE Training Systems. To address the efficiency and scalability challenges with large graphs, there are some well-engineered systems for accelerating KGE training, such as DGL-KE [21], HET-KG [53], and SMORE [20]. However, these graph embedding systems mainly focus on distributed parallelism and embedding storage, which cannot be directly used for large-scale inductive KG reasoning due to the more complex nature of the GNN models [54]–[57]. Notably, many scalable frameworks work on large-scale GNN training, but they do not specifically address the issue of subgraph extraction [58]–[60]. NeuGraph [61] harmoniously combines graph computation optimizations with aspects like data partitioning, scheduling, and parallelism, within dataflow-oriented deep learning frameworks. AliGraph [**?**] refines sampling operators for distributed GNN training, and minimizes network communication by caching nodes on local systems. DistDGL [62], a distributed GNN training framework, distributes the graph and its related vector data among machines. Despite their efficiencies, the graph partitioning algorithms frequently used in these distributed GNN training systems tend to obstruct subgraph completeness. Therefore, additional data communication becomes unavoidable to compile the entire subgraph information. To address the subgraph extraction issue, some recent GNN training systems, like AliGraph [**?**] and Ginex [12], cache the neighbor information for a few central nodes.

VIII. CONCLUSION

In this paper, we propose three novel systems—Ginex, OUTRE, and TIGER—to address the challenges of scalable

GNN training with caching. Ginex leverages SSDs to support billion-scale graph datasets on a single machine, optimizing feature caching to alleviate the I/O bottleneck. This results in significant speedups, enabling GNN training on datasets far larger than what a single machine's CPU memory could handle. OUTRE introduces an out-of-core sampling-based framework that focuses on reducing overall data volume requests for external storage, rather than just minimizing the percentage of data attempts. By identifying and addressing data redundancies, OUTRE incorporates partition-based batch construction, historical embedding caching, and automatic cache space management. Finally, TIGER enhances inductive KG reasoning by accelerating subgraph extraction through a Slice&Cache procedure, which improves SSD sequential access. The SiGMa slicing algorithm balances redundancy and efficiency, offering significant computational savings. Overall, Ginex, OUTRE, and TIGER push the boundaries of scalable GNN training by optimizing data access and memory usage, setting the stage for even more efficient large-scale graph learning systems.

REFERENCES

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, p. 4–24, Jan 2021.

[2] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge & Data Engineering*, vol. 34, no. 01, pp. 249–270, jan 2022.

[3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," 2021.

[4] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10843. Springer, 2018, pp. 593–607.

[5] A. Pal, C. Eksombatchai, Y. Zhou, B. Zhao, C. Rosenberg, and J. Leskovec, *PinnerSage: Multi-Modal User Embedding Framework for Recommendations at Pinterest*. New York, NY, USA: Association for Computing Machinery, 2020, p. 2311–2320. [Online]. Available: https://doi.org/10.1145/3394486.3403280

[6] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, 2018.

[7] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, 2018.

[8] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 3844–3852.

[9] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," 2021. [Online]. Available: https://arxiv.org/abs/2005.00687

[10] A. Khatua, V. S. Mailthody, B. Taleka, T. Ma, X. Song, and W.-m. Hwu, "Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '23. ACM, Aug. 2023, p. 4284–4295. [Online]. Available: http://dx.doi.org/10.1145/3580305.3599843

[11] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:222296394

[12] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2626–2639, 2022.

[13] R. Mirchandaney, J. Saltz, and R. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 05, pp. 603–612, may 1991.

[14] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018.

[15] Z. Sheng, W. Zhang, Y. Tao, and B. Cui, "Outre: An out-of-core de-redundancy gnn training framework for massive graphs within a single machine," *Proc. VLDB Endow.*, vol. 17, no. 11, p. 2960–2973, Aug. 2024. [Online]. Available: https://doi.org/10.14778/3681954.3681976

[16] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnović, "Fennel: streaming graph partitioning for massive scale graphs," *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:9590483

[17] K. Huang, H. Jiang, M. Wang, G. Xiao, D. Wipf, X. Song, Q. Gan, Z. Huang, J. Zhai, and Z. Zhang, "Freshgnn: Reducing memory access via stable historical embeddings for graph neural network training," 2024. [Online]. Available: https://arxiv.org/abs/2301.07482

[18] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec, "Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings," 2021. [Online]. Available: https://arxiv.org/abs/2106.05609

[19] A. Kochsiek and R. Gemulla, "Parallel training of knowledge graph embedding models: A comparison of techniques," *Proc. VLDB Endow.*, vol. 15, no. 3, pp. 633–645, 2021.

[20] H. Ren, H. Dai, B. Dai, X. Chen, D. Zhou, J. Leskovec, and D. Schuurmans, "SMORE: knowledge graph completion and multi-hop reasoning in massive knowledge graphs," in *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 2022, pp. 1472–1482.

[21] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis, "DGL-KE: training knowledge graph embeddings at scale," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, 2020, pp. 739–748.

[22] K. Wang, Y. Xu, and S. Luo, "Tiger: Training inductive graph neural network for large-scale knowledge graph reasoning," *Proc. VLDB Endow.*, vol. 17, no. 10, p. 2459–2472, Aug. 2024. [Online]. Available: https://doi.org/10.14778/3675034.3675039

[23] Y. Hu, A. Levi, I. Kumar, Y. Zhang, and M. Coates, "On batch-size selection for stochastic training for graph neural networks," 2021. [Online]. Available: https://openreview.net/forum?id=HeEzgm-f4g1

[24] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018.

[25] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," *Proc. VLDB Endow.*, 2021.

[26] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[27] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[28] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:3628209

[29] W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi, "Minimal variance sampling with provable guarantees for fast training of graph neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2006.13866

[30] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[31] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[32] Y. Lee, Y. Kwon, and M. Rhu, "Understanding the implication of non-volatile memory for large-scale graph neural network training," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 118–121, 2021.

[33] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[34] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1247–1250.

[35] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., 2017.

[36] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[37] Y. Zhang and Q. Yao, "Knowledge graph reasoning with relational digraph," in *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 2022, pp. 912–924.

[38] Z. Zhu, Z. Zhang, L. A. C. Xhonneux, and J. Tang, "Neural bellman-ford networks: A general graph neural network framework for link prediction," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, 2021, pp. 29 476–29 490.

[39] Y. Zhang, Z. Zhou, Q. Yao, X. Chu, and B. Han, "Learning adaptive propagation for knowledge graph reasoning," *CoRR*, vol. abs/2205.15319, 2022.

[40] Z. Zhu, X. Yuan, M. Galkin, S. Xhonneux, M. Zhang, M. Gazeau, and J. Tang, "A*net: A scalable path-based reasoning approach for knowledge graphs," 2023.

[41] K. Wang, S. Luo, and D. Lin, "River of no return: Graph percolation embeddings for efficient knowledge graph reasoning," *arXiv preprint arXiv:2305.09974*, 2023.

[42] S. Wu, H. Wan, W. Chen, Y. Wu, J. Shen, and Y. Lin, "Towards enhancing relational rules for knowledge graph link prediction," *arXiv preprint arXiv:2310.13411*, 2023.

[43] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3292500.3340404

[44] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "GLIST: Towards in-storage graph learning," in *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, Jul. 2021, pp. 225–238.

[45] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 187–198. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf

[46] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "NeuGraph: Parallel deep neural network computation on large graphs," in *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, Jul. 2019, pp. 443–458.

[47] D. Zhang, X. Huang, Z. Liu, J. Zhou, Z. Hu, X. Song, Z. Ge, L. Wang, Z. Zhang, and Y. Qi, "Agl: A scalable system for industrial-purpose graph machine learning," *Proc. VLDB Endow.*, 2020.

[48] G. Zhao, T. Zhou, and L. Gao, "Cm-gcn: A distributed framework for graph convolutional networks using cohesive mini-batches," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 153–163.

[49] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," 2021.

[50] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," 2018. [Online]. Available: https://arxiv.org/abs/1710.10568

[51] H. Yu, L. Wang, B. Wang, M. Liu, T. Yang, and S. Ji, "Graphfm: Improving large-scale gnn training via feature momentum," 2022. [Online]. Available: https://arxiv.org/abs/2206.07161

[52] K. Huang, H. Jiang, M. Wang, G. Xiao, D. Wipf, X. Song, Q. Gan, Z. Huang, J. Zhai, and Z. Zhang, "Freshgnn: Reducing memory access via stable historical embeddings for graph neural network training," 2024. [Online]. Available: https://arxiv.org/abs/2301.07482

[53] S. Dong, X. Miao, P. Liu, X. Wang, B. Cui, and J. Li, "HET-KG: communication-efficient knowledge graph embedding training via hotness-aware cache," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 1754–1766.

[54] Z. Zhu, S. Xu, J. Tang, and M. Qu, "Graphvite: A high-performance CPU-GPU hybrid system for node embedding," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2019, pp. 2494–2504.

[55] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large scale graph embedding system," in *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[56] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, 2021, pp. 533–549.

[57] P. Fang, A. Khan, S. Luo, F. Wang, D. Feng, Z. Li, W. Yin, and Y. Cao, "Distributed graph embedding with information-oriented random walks," *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1643–1656, 2023.

[58] S. Gandhi and A. P. Iyer, "P3: distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 551–568.

[59] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: an efficient communication library for distributed GNN training," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 2021, pp. 130–144.

[60] S. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W. W. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2087–2100, 2021.

[61] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019, pp. 443–458.

[62] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," in *10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 2020, Atlanta, GA, USA, November 11, 2020*. IEEE, 2020, pp. 36–44.