ETC: Efficient Training of Temporal Graph Neural Networks over Large-scale Dynamic Graphs

Shihong Gao
The Hong Kong University of Science and Technology
sgaoar@connect.ust.hk

Yiming Li
The Hong Kong University of Science and Technology
yliix@cse.ust.hk

Yanyan Shen*
Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Yingxia Shao*
Beijing University of Posts and Telecommunications
shaoyx@bupt.edu.cn

Lei Chen
The Hong Kong University of Science and Technology (GZ)
leichen@cse.ust.hk

ABSTRACT

Dynamic graphs play a crucial role in various real-world applications, such as link prediction and node classification on social media and e-commerce platforms. Temporal Graph Neural Networks (T-GNNs) have emerged as a leading approach for handling dynamic graphs, using temporal message passing to compute temporal node embeddings. However, training existing T-GNNs on large-scale dynamic graphs is prohibitively expensive due to the ill-suited batching scheme and significant data access overhead. In this paper, we introduce ETC, a generic framework designed specifically for efficient T-GNN training at scale. ETC incorporates a novel data batching scheme that enables large training batches improving model computation efficiency, while preserving model effectiveness by restricting information loss in each training batch. To reduce data access overhead, ETC employs a three-step data access policy that leverages the data access pattern in T-GNN training, significantly reducing redundant data access volume. Additionally, ETC utilizes an inter-batch pipeline mechanism, decoupling data access from model computation and further reducing data access costs. Extensive experimental results demonstrate the effectiveness of ETC, showcasing its ability to achieve significant training speedups compared to state-of-the-art training frameworks for T-GNNs on real-world dynamic graphs with millions of interactions. ETC provides a training speedup ranging from 1.6× to 62.4×, highlighting its potential for efficient training on large-scale dynamic graphs.

PVLDB Reference Format:
Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. ETC: Efficient Training of Temporal Graph Neural Networks over Large-scale Dynamic Graphs. PVLDB, 17(5): 1060 - 1072, 2024.
doi:10.14778/3641204.3641215

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eddiegaoo/ETC.

*Yanyan Shen and Yingxia Shao are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641215

1 INTRODUCTION

Dynamic graphs, which are constantly updated with new nodes and edges, play a crucial role in many real-world applications. For instance, users on social platforms engage with each other over time by commenting on posts or sending messages. Likewise, on e-commerce platforms, users buy a diverse range of items at different time. To support various downstream tasks such as node classification and link prediction, it is necessary to learn representations of nodes based on new interactions. Temporal Graph Neural Networks (T-GNNs) [15, 27, 31–33] are at the forefront of this endeavor, having achieved state-of-the-art performance in learning representations on dynamic graphs. T-GNNs employ recursive temporal message passing to compute the temporal embedding of a target node at a specific timestamp. This process involves time-dependent neighbor sampling and time-encoded neighborhood aggregation, which enables T-GNNs to capture the propagation process on evolving graphs more effectively. Recent studies have demonstrated that T-GNNs can significantly outperform static GNNs [11, 14, 29] and snapshot-based GNNs [8, 10, 23] by a considerable margin in terms of predictive performance.

Generally, the training of T-GNNs is in an offline and chronological fashion [27, 33]. Figure 1 provides an illustration of the general T-GNN training workflow. Given an input dynamic graph stored on CPU, it is split into multiple training batches in the *preprocessing* stage. Each training batch contains a number of interactions, of which the timestamps are contiguous in an increasing order. After the preprocessing step, the T-GNN starts to process the generated batches on GPU in a sequential fashion to preserve the intrinsic temporal dependency of the input dynamic graphs. The processing of a batch consists of *temporal neighbor sampling*, *input data access*, and *model computation*. Given the target interactions in a batch, it is firstly required to conduct temporal neighbor sampling for the nodes included in the target interactions. Then based on the computation graph generated in the sampling phase, the corresponding data such as node state vectors, which summarize the past interactive information of nodes, are accessed and fed to the T-GNN model for computation.

Apart from the outstanding performance of T-GNNs, researchers in DB community [18, 19, 39] recently have discovered that the training of T-GNNs is prohibitively expensive on large-scale dynamic graphs, such that merely one training epoch for existing

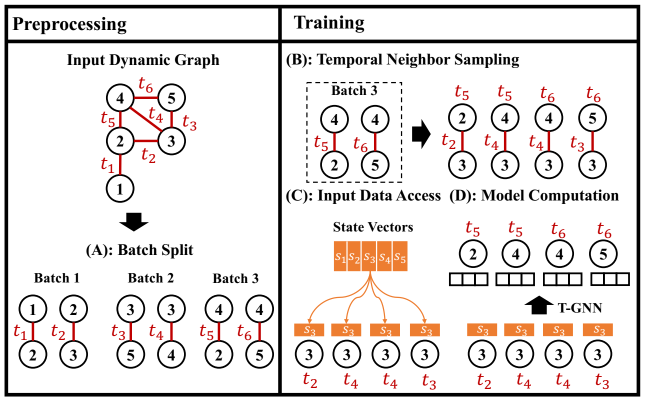


Figure 1: A toy example of T-GNN training. The input dynamic graph contains 6 interactions with different timestamps. The batch size is 2 (2 target interactions), the number of sampled neighbor is 1.

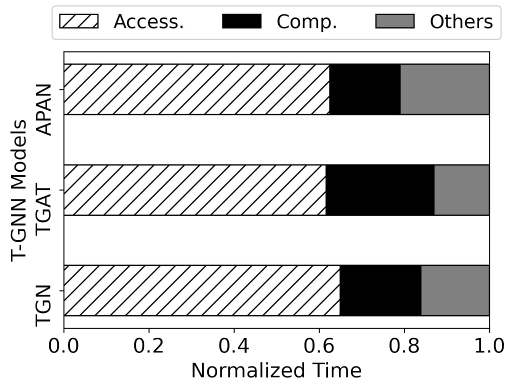


Figure 2: Comparison of the normalized time among the input data access (Access.), model computation (Comp.), and the other data preparation operations (Others) when training different T-GNN models on GDELT using TGL.

T-GNNs over large-scale dynamic graphs would take hours to complete [19]. To tackle the overwhelming T-GNN training costs, Orca [18] and Zebra [19] focus on improving the model computation efficiency of T-GNNs. Orca utilizes historical embeddings to avoid certain computational workload. Zebra proposes to modify the aggregation operation in T-GNN by a temporal personalized PageRank mechanism, which only aggregates the most influential neighbors for target nodes to reduce computation costs. However, their techniques can only be generalized to synchronous T-GNNs [15, 27, 33]. By contrast, TGL [39] is a generic training framework that supports various types of T-GNNs [15, 27, 28, 31, 33]. Nonetheless, it only focuses on accelerating the sampling stage but overlooks the other stages in the general T-GNN training workflow. In this paper, we identify that: **(1)** the typical small batch size setting in the preprocessing stage can heavily hinder the model computation efficiency, and **(2)** the input data access stage occupies a large proportion of total training time. These two bottlenecks limit the scalability of training T-GNNs on large-scale dynamic graphs.

Bottleneck I: Small Batch Size Hinders the Efficiency of Model Computation. In the preprocessing stage, existing works [15, 18, 19, 27, 31–33] usually employ a small batch size to mitigate the intra-batch information loss issue [15, 27, 31]. In T-GNN training, in order to process the interactions in a given batch in parallel by GPU, the intra-batch dependencies among target nodes in the same batch have to be deserted causing information loss [27]. The larger batch size is, the more intra-batch dependencies can be lost, which will degrade model performance [27, 31]. However, when training over large-scale dynamic graphs with over millions of interactions, utilizing a small batch size to alleviate information loss results in unsatisfactory training efficiency. Since a small batch size setting not only produces numerous training batches for T-GNNs to process the whole graph, but also results in limited exploitation of GPU parallelism [15], which largely impedes the model computation efficiency of T-GNNs over large-scale dynamic graphs. As shown in [18, 19], with batch size set as 200, it takes over 12 hours for existing T-GNNs to finish just one epoch of model computation on the large-scale Wiki-Talk [3] dataset.

Bottleneck II: High Input Data Access Costs Dominate the Training Process. When tackling the large-scale dynamic graphs, we find that the input data access is de facto the dominant part of the overall training process. As shown in Figure 2, the input data access phase generally takes up over 60% of the total training time, which vastly overshadows the costs of model computation and the other data preparation operations. Such overwhelming cost originates from frequent data transfer between CPU and GPU. Since the input data such as node features are stored on the CPU side, the required input data for each batch are constantly selected on CPU and transferred to GPU in the entire training loop.

In this paper, we propose a generic framework ETC, which enables **e**fficient **t**raining over large-s**c**ale dynamic graphs for different variants of T-GNNs. It comprises several systematic designs to resolve the common efficiency bottlenecks concerning the general T-GNN training workflow.

To address **Bottleneck I**, we take a fundamentally different batching approach in the preprocessing stage. We first introduce a novel score function to quantify the impact of information loss. Then, we formulate a batch split problem that aims to minimize the number of batches used in training while controlling the extent of information loss of each batch. By reducing the number of batches, we can enlarge the batch size on average. To solve the batch split problem, we propose an efficient single-pass algorithm, which scans the input dynamic graph only once in the preprocessing stage and generates the corresponding batching scheme. We also provide a rigorous theoretical analysis to demonstrate that our algorithm can provide an optimal solution to the batch split problem.

To address **Bottleneck II**, our key observation is that: there exists plenty of redundant data access in each batch due to the temporal characteristic. For instance, the same data in the CPU storage, e.g., node state vectors that summarizes the past interactive information of nodes, can be repeatedly accessed by the nodes with the same ID but different timestamps. In practice, redundant data access can take up over 80% of the total data access volume. Based on the crucial observation, We propose a novel three-step data access policy Supra. Unlike the conventional input data access approach [18, 19, 39], which directly performs input data access based on the sampling result, Supra firstly identifies the redundant access workload according to the sampling result, then, it only performs data access for the unique data in each batch. Finally, the originally required data for a given batch are reconstructed on GPU by Supra. As Supra requires constant identification of redundant access workload throughout the training process, it is important to guarantee that the reduced data access costs by Supra are not outweighed by the cost of Supra itself. To achieve this, we manage the training workflow by proposing an inter-batch pipeline mechanism. It enables concurrent execution of the model computation for the current batch alongside Supra for the subsequent batch. This simple yet effective pipeline mechanism helps to further improve the overall efficiency gain.

In summary, we have made the following contributions:

- We present ETC, a generic framework tailored for efficient T-GNN training on large-scale dynamic graphs, which comprises systematic designs to resolve the efficiency bottlenecks in the general T-GNN training workflow.
- We formulate a batch split problem for T-GNN training over large-scale dynamic graphs with practical constraints on information loss, and propose an efficient single-pass algorithm for solving this problem, which generates large batches for higher model computation efficiency while restricting information loss for model performance preservation. Moreover, We provide a rigorous theoretical analysis demonstrating that the proposed algorithm yields an optimal solution to the batch split problem.
- ETC significantly reduces the input data access costs in training T-GNNs over large-scale graphs by a novel three-step data access policy Supra, which substantially removes data access redundancy. Besides, we propose a simple yet effective inter-batch pipeline mechanism, which reduces the cost associated with Supra and helps to further enhance the data preparation efficiency.
- Extensive experiments are conducted to showcase the effectiveness of the proposed ETC framework. The results demonstrate that ETC enjoys $1.6\times \sim 62.4\times$ training speedup compared to state-of-the-art T-GNN training frameworks.

2 BACKGROUND

In this section, we introduce the concept of Continuous-time Dynamic Graphs (CTDGs) and Temporal Graph Neural Networks (T-GNNs).

2.1 Continuous-time Dynamic Graphs

Definition 1 (Continuous-time dynamic graph (CTDG)). *A CTDG can be described as a collection of interaction events $\mathcal{G} = \{\alpha(t_1), \alpha(t_2), ...\}$, which occur over time and are ordered chronologically. Each event is represented by a tuple $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$, which represents a temporal edge in a directed graph. This tuple includes the nodes v_i and v_j which are connected by an edge, an associated feature vector $e_{ij}(t)$ which describes the edge, and a timestamp t that indicates when the interaction occurred.*

A CTDG can also be viewed as a multigraph, in which multiple edges can occur between two nodes in multiple timestamps with different edge feature vectors. In this work, we focus on Continous-Time Dynamic Graphs (CTDGs) rather than Discrete Time Dynamic Graphs (DTDGs), since CTDGs are more general and reflective in terms of dynamic evolving patterns.

Event Types Covered by CTDG. Generally, a CTDG includes not only the addition of interaction events, but also the deletion and update of interaction events, which can be distinctly reflected in the associated descriptive features [27]. Besides, for node-wise events (addition/deletion of nodes, update of node features), they can be perceived as self-interacted events, thus can be defined similarly as described in Definition 1.

2.2 Temporal Graph Neural Networks

Temporal Graph Neural Networks (T-GNNs) have shown their superiority of representation learning on dynamic graphs. Currently, existing state-of-the-art T-GNNs [15, 27, 32] on CTDGs can be generalized into a common architecture, which involves two key operations: node state update and temporal message passing.

Node State Update. Since different nodes may have an interaction history of different lengths, merely using neighborhood sampling may not be adequate to provide information for the dynamic embedding generation. Therefore, most existing T-GNN models [15, 19, 27, 31] adopt the node memory module to summarize the past information of nodes. Specifically, for each node v_i, T-GNN maintains a state vector $s_i(t)$ for it, which encodes the historical information of the interactions containing node v_i. Upon the arrival of an interaction $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$, T-GNN firstly updates the state vectors for the nodes v_i and v_j as follows:

$$s_i(t) = \textbf{UPDATE}(s_i(t^-), s_j(t^-), e_{ij}(t), \phi(t - t^-)), \tag{1}$$

$$s_j(t) = \textbf{UPDATE}(s_j(t^-), s_i(t^-), e_{ij}(t), \phi(t - t^-)), \tag{2}$$

where **UPDATE** is the RNN or GRU memory updater. $s_i(t^-)$ and $s_j(t^-)$ are the latest state vectors for node v_i and v_j before the timestamp t. $\phi(\cdot)$ is a time encoding function, which encodes the time interval $\Delta t = t - t^-$ into a vector.

Temporal Message Passing. When a new interaction $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$ arrives, T-GNN generates the node embedding $h_i^\ell(t)$ for node v_i (and for node v_j as well) in the following steps:

$$N_i^\ell(t) = \textbf{SAMPLE}(\mathcal{G}, v_i, t), \tag{3}$$

$$h_i^\ell(t) = \textbf{AGGREGATE}(\{h_j^{\ell-1}(t^-)||e_{ij}(t^-)||\phi(t-t^-)| \\ (v_j, t^-) \in N_i^\ell(t)\}), \forall \ell = 1, ..., L, \tag{4}$$

$$h_j^0(t^-) = s_j(t^-) + \textbf{MLP}(x_j). \tag{5}$$

Firstly T-GNN conducts temporal neighborhood sampling for v_i (Eq.3). The sampled neighbor set $N_i(t)$ consists of nodes, which interacts with the node v_i before the timestamp t. Note that in temporal sampling, the timestamp needs to be considered. It means that the same nodes may be sampled multiple times but with different timestamps. As for the sampling strategy, top-k recent neighbor sampling is widely adopted in previous literature [19, 27, 31], which samples the top-k neighboring nodes of the target node v_i with the latest timestamps up to t. Then, T-GNN conducts embedding generation at the ℓ-th layer through the neighborhood information aggregation for node v_i at the timestamp t (Eq.4). Generally, **AGGREGATE** is an attention-based aggregator. Moreover, during

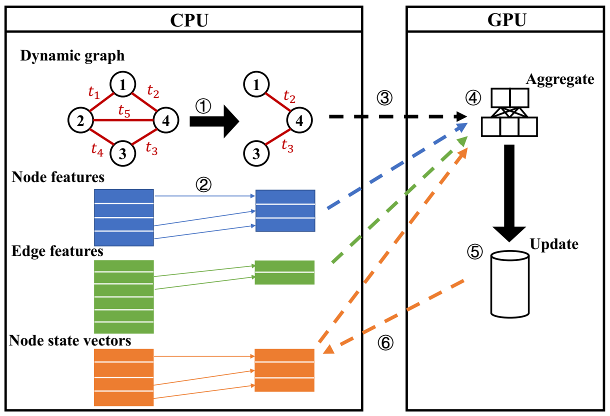


Figure 3: An illustration of hybrid CPU-GPU data layout for T-GNN training over large-scale dynamic graphs. The input dynamic graph, associated node features, edge features and the node state vectors are held in CPU main memory during training. For every training batch, ① neighborhood sampling is performed on CPU. ② Based on the sampling result, the corresponding input data are accessed on CPU. ③ The sampled subgraph together with accessed data is transferred from CPU to GPU for ④ T-GNN model computation. ⑤ The updater on GPU updates the state vectors for the target nodes in the current batch. ⑥ The updated state vectors for target nodes are transferred back from GPU to CPU.

the temporal message passing, the input node features at the first layer are comprised of initial node features and the state vectors of nodes (Eq.5). If the CTDG does not provide the initial node features, the input features would be solely the node state vectors.

2.3 Training of T-GNNs

Generally, T-GNN offline training is conducted in a chronological fashion. We illustrate the data layout, the preprocessing, and main training stages for T-GNN training over large-scale dynamic graphs as follows.

Data Layout. In previous works [15, 19, 27, 31–33], most T-GNNs are trained in an *all-on-GPU* fashion, such that the storage of the necessary input data for training and the training process itself solely rely on GPU. However, when tackling large-scale dynamic graphs, GPU alone is not adequate to take over both the storage and training process. For example, for GDELT [16], a real-world dynamic knowledge graph with near-billion interactions, just the storage of all the input data would already consume over 130GB memory. This vastly exceeds the memory capacity of a single GPU, which is generally equipped with 11GB ~ 40GB memory. To accommodate the memory requirement of large-scale dynamic graphs, TGL [39] points out the necessity of a hybrid CPU-GPU data layout as shown in Figure 3, which leverages large CPU main memory for input data preservation during the training process. Specifically, all the input data are loaded from the secondary storage to the CPU main memory in an one-shot manner right before the starting of the training. Then, the input data including the input dynamic graph, node state vectors, node features, and edge features are maintained in CPU main memory throughout the training process. This CPU main memory storage scheme is also the mainstream choice in systems for static GNN training [20, 34, 37]. On the other hand, GPU is responsible for conducting model computation of the T-GNN.

Preprocessing. Before the training, it is required to preprocess the given dynamic graph and split it into multiple batches with a temporal order constraint as shown in Figure 1. The temporal order constraint preserves the intrinsic temporal characteristic of the dynamic graph. It ensures that the order of the timestamps cannot be violated. For example, in Figure 1, it is not allowed to exchange the interaction $(v_2, v_3, e_{23}(t_2), t_2)$ in Batch 1 with the interaction $(v_3, v_5, e_{35}(t_3), t_3)$ in Batch 2. Since the T-GNN would process the batches in a sequential fashion, no prediction would be made by leveraging the future information. Otherwise, the prediction result is unfair.

Main Training Stages. Model training on each batch mainly consists of three stages: temporal neighbor sampling, input data access, and model computation. For each target node in a given batch, the temporal neighbor sampling is conducted as shown in Eq. 3. Different from the sampling in static graphs, in which all the neighbors of the target node are sampling candidates, the temporal neighbor sampling only considers the neighbors with past timestamps so that no future neighbors would be included. To enhance the efficiency of such a sampling process, TGL [39] proposes a parallel sampler, such that the temporal neighbor sampling process for different nodes can be conducted simultaneously. Then the corresponding node state vectors, node features, and edge features required for model computation are accessed based on the sampling result. Finally, these accessed input data are fed to the T-GNN model for computation. To accelerate the model computation stage of T-GNN training, Orca [18] utilizes the historical embeddings to reduce the frequency of aggregation, while Zebra [19] changes the aggregation rule of general T-GNN [27], which only aggregates most influential temporal neighbors.

3 BOTTLENECKS OF T-GNN TRAINING ON LARGE-SCALE DYNAMIC GRAPHS

In this section, we provide more detailed analysis for the two pivotal efficiency bottlenecks discussed in Section 1, which prevent scaling existing T-GNNs to large-scale dynamic graphs.

3.1 Ill-suited Batching Scheme in Preprocessing

As illustrated in Section 1, the traditional batching scheme for T-GNN training in previous literature [18, 19, 27, 31, 39] usually adopts a small batch size, which leads to few interactions being processed in parallel by T-GNN models on GPU. As a result, the efficiency of T-GNN computation is dragged down [15]. Naively setting a large batch size under the traditional batching scheme can exaggerate the intra-batch information loss issue. Example 1 provides an explanation for the intra-batch information loss issue. Typically, with the increase in batch size under the traditional batching scheme, the model computation efficiency is boosted while the predictive accuracy can be significantly degraded. Corresponding results can be seen in Section 5.4 later. This calls for an approach to quantify the information loss of batches, and a new design of the batching scheme, which can mitigate the information loss issue,

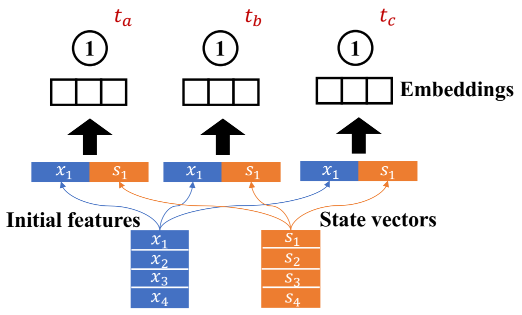


Figure 4: An illustration of data access pattern in T-GNN.

while using possibly large batches for offline T-GNN training to achieve higher model computation efficiency.

Example 1. *consider the two interactions (v_4, v_2, $e_{42}(t_5)$, t_5) and (v_4, v_5, $e_{45}(t_6)$, t_6) of Batch 3 in Figure 1. As they are included in the same batch and processed simultaneously, they can be perceived to be arriving at the same time. Note that v_4 occurs in both of these interactions. Ideally, T-GNN would update the state vector of v_4 twice on t_5 and t_6. However, when performing the state vector update for v_4 in these two interactions within the same batch, the same past state vector of v_4 is utilized. Therefore, when performing the state vector updates for v_4 on t_6, it does not leverage the updated state vector of v_4 by the interaction (v_4, v_2, $e_{42}(t_5)$, t_5), which indicates that such interaction is cast away from the viewpoint of node v_4 on t_6.*

3.2 High Input Data Access

As shown in Figure 1, input data access is the major efficiency bottleneck (around 60% of the overall training time) of T-GNN training on large-scale dynamic graphs. When conducting input data access, the conventional procedure [18, 19, 39] involves selecting and pulling all the required input data directly based on the sampling result. It begins by creating an access list that is identical to the sampled data IDs by the temporal neighbor sampler. Subsequently, the corresponding data are transferred from the CPU main memory to the GPU based on this access list. Despite the straightforwardness of the conventional data access approach, it overlooks the data access pattern in T-GNN training, which can lead to redundant data transfer. A crucial observation of the data access pattern in T-GNN training is that: *within a given batch, the same nodes can occur in various interactions with different timestamps. Even though the T-GNN model would create distinct state vectors and embeddings for these same nodes with different timestamps, they all require accessing the same input node features and past state vectors.* Example 2 provides an illustration for such a data access pattern. In practice, such redundancy can be overwhelming as shown in Figure 5. The same nodes are repeatedly accessed, which can lead to redundant data transfer taking up over 80% of the total data transfer volume.

Example 2. *In Figure 4, node v_1 occurs at timestamp t_a, t_b, and t_c respectively in a given batch. Though they would have different embeddings generated by T-GNN due to different timestamps, the same initial feature x_1 and the same past state vector s_1 are accessed and fed to T-GNN for computation. Under the conventional data access policy, there exists redundant data access such that x_1 and s_1 are selected three times on CPU and transferred altogether to GPU. But actually, x_1 and s_1 can be only transferred once, since the same input data are required to generate different embeddings.*

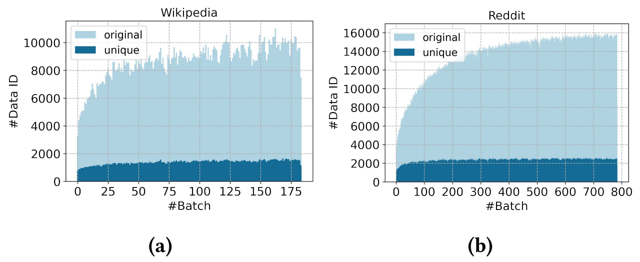


Figure 5: The sampling results (batch_size=600; k=10) on (a) Wikipedia and (b) Reddit datasets [15]. The X-axis denotes the number of batches in an epoch, and the Y-axis denotes the number of data IDs. "original" indicates the total data IDs based on the sampling result, while "unique" represents the unique data IDs in the sampling result.

4 THE ETC FRAMEWORK

In this section, we present ETC, a generic framework tailored for efficient T-GNN training over large-scale dynamic graphs, which resolves the two efficiency bottlenecks discussed in Section 3. The optimizations in ETC reside in the batch split preprocessing stage and the input data access stage, which are orthogonal to existing T-GNN training frameworks [18, 19, 39]. In the preprocessing stage of T-GNN training (Section 4.1), we present an information-loss-bounded batching scheme, which forms sizeable training batches to enhance model computation efficiency while restricting the possible information loss to preserve the effectiveness of the underlying T-GNN models. To address the tremendous input data access costs (Section 4.2), we present a novel three-step data access policy to cut down the redundant data access volume. Also, we incorporate an inter-batch pipeline mechanism in ETC, which parallelizes the input data access and the model computation to further hide the input data access costs.

4.1 Information-loss-bounded Batching Scheme

In contrast to the traditional batching scheme as discussed in Section 3.1, ETC's batching scheme successfully increases the batch size while effectively limiting the impact of information loss. As a result, our approach significantly improves the efficiency of model computation without compromising model performance, surpassing the limitation of the conventional batching scheme.

Information Loss Qunatification. To quantify the information loss of each batch, we define a novel information loss score $\beta(B_i)$ for a given batch B_i as follows:

$$\beta(B_i) = \sum_{v \in N_i} \beta(v, B_i), \tag{6}$$

$$\beta(v, B_i) = \left|\{\alpha(t)|\alpha(t) \in B_i, v \in \alpha(t)\}\right| - \mathbf{1}\left(\left|\{\alpha(t)|\alpha(t) \in B_i, v \in \alpha(t)\}\right| > 0\right). \tag{7}$$

N_i is the node set derived from B_i. Eq.7 quantifies the gap between the actual update times and the ideal update times (single interaction per batch) for a given node v. Then Eq.6 shows the information loss score of a given batch, which sums up the information loss score of each node appearing in B_i. Based on Eq.6 and Eq.7, we can also derive the following equivalent formulation for the information loss score of a target batch B_i:

$$\beta(B_i) = 2\left|B_i\right| - \left|N_i\right|. \tag{8}$$

Problem Formulation. Generally, given an input dynamic graph, we would like to split it into multiple batches for T-GNN training. The optimization goal is about minimizing the total number of batches required for the training process. Since the total number of interactions in an input dynamic graph is fixed, minimizing the total number of batches is equivalent to enlarging the sizes of batches. Also, an information loss constraint is required for each batch, such that no batch would incur excessive information loss. We formally define the batch split problem as follows.

Definition 2 (Batch Split Problem for T-GNN Training). *Given all the interactions $\mathcal{G} = \{\alpha(t_1), \alpha(t_2), ..., \alpha(t_e)\}$, split them into K batches $B_1, B_2, ..., B_K$. Each batch contains a set of continuous interactions such that $B_i = \cup_{l=p}^{q-1}\{\alpha(t_l)\}, p < q, \forall i$. Each batch is associated with an information loss score $\beta(\cdot)$. We aim to find a batch split $f(\mathcal{G}) = \{B_1, B_2, ..., B_K\}$, such that:*

$$\min_f \quad K$$
$$s.t. \quad \beta(B_i) \le \varepsilon, \ \ \forall i, \tag{9}$$
$$B_i \cap B_j = \emptyset, \ \ i \ne j, \ \ \forall (i,j), \tag{10}$$
$$\cup_{i=1}^{K} B_i = \mathcal{G}, \tag{11}$$
$$t_m < t_n, \ \ \forall \alpha(t_m) \in B_i, \ \ \forall \alpha(t_n) \in B_j, \ \ i < j \tag{12}$$

Eq. 9 indicates that the information loss score of each batch cannot be larger than a threshold ε. Eq. 10 is a disjoint constraint, which requires that there is no overlap among all the batches. Eq. 11 is a union constraint, which means that the union of all the batches is the given input dynamic graph $\mathcal{G}$. As for Eq. 12, it is a temporal order constraint, which ensures that the batch split generated would not break the intrinsic order of interactions in the given dynamic graph.

Single-pass Batching Algorithm. The batch split problem is a variant of the known bin packing problem [4] with an additional temporal order constraint. To solve the batch split problem, we develop Algorithm 1, which is an efficient single-pass batch split algorithm. We further theoretically prove that the proposed algorithm is an optimal solution to the batch split problem.

Algorithm 1 begins by initializing an empty first batch, an empty node set, and a counter for update times (Lines 1-4). It then proceeds with a sequential scan of all the interactions (Line 5). During the scan, each target interaction is added to the current batch, followed by the calculation of the information loss score for that batch (Lines 6-10). If adding the interaction to the current batch does not violate the information loss constraint, it is retained in the current batch (Lines 11-12). However, if adding the interaction would violate the information loss constraint, a new batch is created that includes the interaction (Lines 13-17). Since Algorithm 1 performs a single scan of all the interactions and decides whether each scanned interaction should be placed in the current batch or a new batch, its time complexity is $O(|E|)$, where $|E|$ represents the total number of interactions in the given dynamic graph.

Algorithm 1 Single-pass Batch Split Algorithm.

```
Input: Dynamic graph $\mathcal{G}$; information loss threshold $\varepsilon$.
Output: Batches $B_1, B_2, ..., B_K$.
1: Initialize an update counter: $C_u \leftarrow 0$;
2: Initialize batch ID: $i \leftarrow 0$;
3: Initialize an empty node set: $N \leftarrow \{\ \}$;
4: Initialize an empty batch: $B_i \leftarrow \{\ \}$;
5: for $\alpha(t) \in \mathcal{G}$ do ▷ Iterate over the interactions sequentially.
6:     $C_u \leftarrow C_u + 2$
7:     for $v \in \alpha(t)$ do
8:         $N \leftarrow N \cup \{v\}$
9:     end for
10:    $\beta(B_i) \leftarrow C_u - |N|$
11:    if $\beta(B_i) \le \varepsilon$ then          ▷ Stick to the current batch.
12:        $B_i \leftarrow B_i \cup \{\alpha(t)\}$
13:    else                                       ▷ State a new batch.
14:        $i \leftarrow i + 1$
15:        $C_u \leftarrow 0$
16:        $B_i \leftarrow \{\alpha(t)\}$
17:        $N \leftarrow \{v \in \alpha(t)\}$
18:    end if
19: end for
```

Theoretical Analysis. We present a theoretical analysis of the proposed single-pass batch split algorithm. Despite the simplicity of Algorithm 1, we can prove that its output is an optimal solution to the batch split problem. To establish this claim, we begin by proving the following two useful lemmas.

Lemma 1. *Given a batch $B(p,q)$, which starts at the p-th interaction and ends right before the q-th interaction, we have:*

$$\beta(B(p,q)) \le \beta(B(p,q+c)), \tag{13}$$
$$\beta(B(p-c,q)) \le \beta(B(p,q)), \tag{14}$$

where c is a constant integer.

Proof. For the batches $B(p,q)$ and $B(p,q+c)$, denote the node sets in them as $N(p,q)$ and $N(p,q+c)$ respectively. Based on the definition of the information loss score as described in Eq.8, we know that:

$$\beta(B(p,q)) = 2(q-p) - |N(p,q)|,$$
$$\beta(B(p,q+c)) = 2(q+c-p) - |N(p,q+c)|.$$

Then we can derive:

$$\beta(B(p,q+c)) - \beta(B(p,q)) = 2c - (|N(p,q+c)| - |N(p,q)|).$$

We know that $N(p,q) \subseteq N(p,q+c)$. Also, $B(p,q+c)$ can be seen as $B(p,q)$ adding c more interactions right after the last interaction it contains. In this way, c more interactions bring at most $2c$ more new nodes to $N(p,q)$. Therefore, we know that:

$$|N(p,q+c)| - |N(p,q)| \le 2c,$$

which ends the proof for Eq.13. The proof for Eq.14 can be done in a similar fashion. □

Lemma 1 shows the non-decreasing characteristic of the information loss function. For a given batch B, if adding new interactions into it, its information loss score $\beta(B)$ would only either increase or remain the same.

LEMMA 2. *An optimal solution (OPT) to the batch split problem generates K^* batches, and the output of Algorithm 1 (ALG1) generates K batches. Denote the index of the last interaction in a batch as $e(\cdot)$. Then for all $0 < i \leq K^*$, we have $e(i, OPT) \leq e(i, ALG1)$.*

PROOF. We can prove the lemma above by induction. The base case is when $i = 1$, and the first batch produced by both OPT and $ALG1$ starts at the first interaction of the input dynamic graph. We know that $ALG1$ would not stop adding the subsequent interactions to the first batch until the information loss constraint is violated. If the first batch produced by OPT has more interactions than those in the first batch produced by $ALG1$, the information loss constraint would be violated. Therefore, we have $e(1, OPT) \leq e(1, ALG1)$.

Assume for the $i - th$ batch, $e(i, OPT) \leq e(i, ALG1)$. Then we need to prove that for the $i + 1 - th$ batch, $e(i + 1, OPT) \leq e(i + 1, ALG1)$ still holds.

Case1: $e(i+1, OPT) \leq e(i, ALG1)$**.** Based on $ALG1$, we can know that $e(i, ALG1) \leq e(i + 1, ALG1)$. Then based on the assumption, we get $e(i + 1, OPT) \leq e(i + 1, ALG1)$.

Case2: $e(i + 1, OPT) > e(i, ALG1)$**.** By the information loss constraint, we know that the $i + 1 - th$ batch produced by OPT would not violate the information loss constraint, such that:

$$\beta(e(i, OPT) + 1, e(i + 1, OPT)) \leq \varepsilon.$$

Then based on Lemma 1, we can obtain that:

$$\beta(e(i, ALG1) + 1, e(i + 1, OPT)) \leq \beta(e(i, OPT) + 1, e(i + 1, OPT)).$$

In this way, we can know that if the $i + 1 - th$ batch produced by $ALG1$ ends at $e(i + 1, OPT)$, the information loss constraint would not be violated. In other words, the $i + 1 - th$ batch produced by $ALG1$ at least ends at $e(i+1, OPT)$. Therefore, we get $e(i+1, OPT) \leq e(i + 1, ALG1)$. □

Based on Lemma 1 and Lemma 2, we provide the following Theorem 1.

THEOREM 1. *The output of Algorithm 1 is an optimal solution to the batch split problem that minimizes the number of batches under the information loss score constraint as described in Definition 2.*

PROOF. Denote the number of interactions in the input dynamic graph as $|E|$. For the batch split problem, assume the optimal solution (OPT) produces K^* batches, while Algorithm 1 ($ALG1$) produces K batches, such that $K^* \leq K$. Since OPT splits all the interactions of the input dynamic graph into K^* batches, we have $e(K^*, OPT) = |E|$. Then based on Lemma 2, we know $e(K^*, OPT) \leq e(K^*, ALG1)$. Since Algorithm 1 performs a single scan over all the interactions of the input dynamic graph, we know that the algorithm terminates at the last interaction, which means $e(K^*, ALG1) = |E|$. In this way, $K = K^*$. □

4.2 Efficient Input Data Access

ETC incorporates two key optimizations to efficiently conduct input data access. Firstly, the three-step data access policy largely shrinks the data access volume, which significantly cuts down the cost of input data access. Besides, to reduce the additional overhead incurred by the three-step data access policy, ETC adopts an inter-batch pipeline mechanism, which decouples the input data access from the model computation. It assists in further reducing the cost of input data access and improving the overall training efficiency.

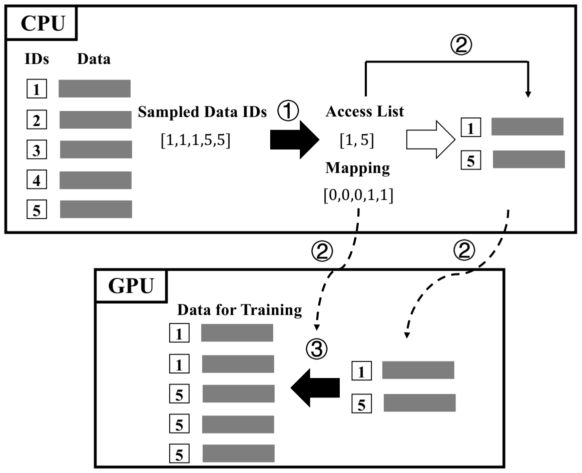


Figure 6: An illustration of the three-step data access policy.

4.2.1 Three-step Data Access Policy. Motivated by the observation as illustrated in Section 3.2, ETC leverages a novel three-step data access policy named Supra to efficiently conduct the input data access. Figure 6 provides an illustrative example. In the first step, Supra performs data ID transformation. Starting with the original data IDs obtained from the sampling result, an access list is created, containing only unique data IDs. Besides, Supra maintains a mapping between the original data IDs and the unique data IDs. In the second step, utilizing the access list generated in the first step, the corresponding data (node state vectors, node features, and edge features) are selected on CPU and then transferred from CPU to GPU. Finally, in the third step, once the unique data has been transferred to GPU, the originally required input data can be reconstructed using the mapping generated in the first step.

Data Transfer Volume Analysis. With the conventional data access policy, the total resultant data transfer volume for a batch is $O(Q \cdot d)$, where Q represents the number of data IDs in the batch, and d is the dimension of a single data point (e.g., dimension of a node feature). By contrast, with the utilization of Supra, the data transfer volume is reduced to $O(R \cdot d + Q)$, where R is the number of unique data IDs in the batch. Additionally, an extra term of Q accounts for transferring the ID mapping. Considering the data access pattern in T-GNN training, it is observed that Q can be multiple times greater than R. Consequently, the implemented data access policy Supra can significantly reduce the total data transfer volume, thereby mitigating the heavy data transfer costs.

4.2.2 Inter-batch Pipeline. ETC is also equipped with a lightweight inter-batch pipeline mechanism. While the data access policy Supra in ETC boosts the input data access efficiency, the design introduces some additional overhead. Specifically, the first step of Supra

Table 1: Summary of statistics of the dynamic graphs. $|V|$ and $|E|$ represent the number of nodes and edges. d_v and d_e denote the dimension of node features and edge features. α and β respectively represents the average degree and diameter. Θ denotes the average update distance of all nodes. Specifically, for a particular node v, its average update distance θ_v is defined as the average number of interactions between two consecutive updates among all updates of node v. Then the average update distance for all nodes is $\Theta = \sum_{v \in V} \theta_v / |V|$.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	d_v	d_e	α	β	Θ
LastFM	2K	1.3M	128	128	1306	1	2873
Wiki-Talk	1.1M	7.8M	172	172	14	11	458149
Stack-Overflow	2.6M	63.4M	172	172	49	13	1354646
GDELT	17K	191.3M	413	186	22934	7	4876113

involves searching for unique data IDs and generating mappings between the unique data IDs and the original data IDs in every training batch. This operation has a time complexity of $O(Q)$, where Q represents the number of data IDs in a batch. However, when training on a large-scale dynamic graph, a substantial number of batches are required. The accumulated costs of this operation can become massive in practice, limiting the potential efficiency gains offered by Supra. Moreover, in cases where the cost of ID transformation outweighs the reduction in data transfer costs, the effect of Supra may be neutralized or even counterproductive.

Under the standard execution order in T-GNN training, CPU and GPU are utilized sequentially. Specifically, CPU is responsible for conducting temporal neighbor sampling and processing input data access. When the model computation of a training batch is performed by T-GNN on GPU, CPU would remain relatively idle and underutilized. Recognizing this, we have made a slight adjustment to the T-GNN training by workload decoupling. While the model training is carried out on GPU for a given batch, the temporal neighbor sampling and input data access for the subsequent batch are concurrently performed on CPU. This modification well reduce the additional costs associated with Supra. Although this pipeline mechanism is lightweight, it effectively facilitates a further reduction on the data access overhead.

5 EXPERIMENTS

5.1 Experiment Setups

Implementations. We implement ETC based on TGL [39] and DGL [30], using Pytorch [24] as the backend deep learning framework for the T-GNN model training phase. We adopt the efficient parallel temporal sampler in TGL with a CSR-based data structure for rapid access to temporal neighbors. We utilize DGL to generate the Message Flow Graph (MFG) for each batch, which contains the sampled dynamic subgraph as well as the associated input data. For the implementation of the three-step data access policy, we utilize NumPy [12] for a fast search of unique data IDs and the generation of the ID mapping. For the inter-batch pipeline, we implement it using Python threading.

Datasets. As summarized in Table 1, we use four real-world large-scale dynamic graphs with millions of interactions that exhibit distinct graph characteristics. LastFM [15] consists of one-month listener-music interactions. Wiki-Talk [3] and Stack-Overflow [1] record user-user interactions from the corresponding websites. GDELT [39] is a near billion-scale dynamic temporal knowledge graph, which originates from the Event Database in GDELT 2.0 [16]. As for the data splits, we adopt the same data split on LastFM, Wiki-Talk, and Stack-Overflow as used in [19, 27, 33], which chronologically splits the input graph into the training set (70%), the validation set (15%), and the test set (15%). On GDELT, we also use the same data split in [39], which uses the interactions before 2019, in 2019, and in 2020 as training set, validation set, and test set, respectively.

Backbone Models. To verify the effectiveness of ETC framework, we use three representative T-GNNs as the backbone models.

- TGAT [33] uses random Fourier features to encode the time information and adopts the attention mechanism, which imitates the message passing scheme in static GNNs.
- TGN [27] is a general T-GNN framework, which includes several existing T-GNN models [15, 28, 33] as its special cases. It dynamically maintains state vectors for nodes in the input dynamic graphs to capture the temporal information.
- APAN [31] is an asynchronous and attention-based T-GNN model, which decouples model computation and message propagation.

As for the other T-GNN models [13, 15, 32], they can be thought of as extensions on top of the above representative T-GNN models with more complex model structure designs [6]. Since the training procedures for these T-GNN models are similar to the ones for the three selected representative T-GNNs, we omit the results for these T-GNN models in the main experiments.

Baseline Frameworks. To verify the efficiency of ETC, we utilize three state-of-the-art T-GNN training frameworks as baselines.

- TGL [39] is a generic training framework that is applicable to a wide range of T-GNN models. It is equipped with an efficient parallel sampler that resolves the high complexity issue in temporal neighbor sampling.
- Orca [18] focuses on resolving the model computation bottleneck in T-GNN training. It leverages a dynamic caching mechanism and utilizes historical representations to avoid substantial computation workload. Also, it incorporates a gradient blocking strategy to improve the generalization ability of T-GNN models.
- Zebra [19] also aims at reducing the model computation costs in T-GNN training. It fundamentally changes the aggregation rule of underlying T-GNN by only aggregating the most influential temporal neighbors.

ETC and TGL are generic frameworks, which do not change the functionality of backbone T-GNNs. Their optimizations reside in managing the data during T-GNN training such as faster sampling and data access. While Orca and Zebra resort to modifying the model structure design of T-GNN for better model computation efficiency and model effectiveness. Note that Orca and Zebra can only be generalized to synchronous T-GNNs [15, 27, 33], and they only implement the classic TGN [27] model. Therefore, we present their results on their currently supported TGN model in the main experiments.

Table 2: **Comparison results of T-GNN training frameworks. Time refers to per-epoch execution time (s). The best average precision (%) and the fastest execution time are marked in bold. "TLE" indicates the time limit exceed such that the training of one epoch cannot finish within 12 hours.**

Dataset	Model	Framework	AP(%)	Time(s)	Dataset	Model	Framework	AP(%)	Time(s)
LastFM	TGN	Orca	80.11	22.0 (1.6×)	Stack-Overflow	TGN	Orca	**97.64**	12428.7 (25.7×)
		Zebra	**82.44**	76.0 (5.4×)			Zebra	97.57	30167.4 (62.4×)
		TGL	80.92	29.7 (2.1×)			TGL	83.32	1504.4 (3.1×)
		ETC	80.79	**14.1**			ETC	86.10	**483.5**
	TGAT	TGL	67.11	8.5 (2.4×)		TGAT	TGL	87.47	339.4 (1.6×)
		ETC	**67.58**	**3.6**			ETC	**87.55**	**215.9**
	APAN	TGL	**70.05**	19.1 (1.8×)		APAN	TGL	63.10	1696.3 (1.8×)
		ETC	69.79	**10.8**			ETC	**66.42**	**937.4**
Wiki-Talk	TGN	Orca	96.05	343.3 (5.8×)	GDELT	TGN	Orca	TLE	TLE
		Zebra	**96.08**	1029.5 (17.5×)			Zebra	TLE	TLE
		TGL	92.94	135.5 (2.3×)			TGL	98.08	4001.5 (3.3×)
		ETC	93.41	**58.9**			ETC	**98.46**	**1222.4**
	TGAT	TGL	**86.10**	40.7 (2.0×)		TGAT	TGL	**98.08**	1338.2 (2.2×)
		ETC	86.08	**20.1**			ETC	98.07	**619.9**
	APAN	TGL	86.73	160.4 (2.0×)		APAN	TGL	96.62	3302.9 (2.5×)
		ETC	**88.11**	**81.8**			ETC	**96.93**	**1304.5**

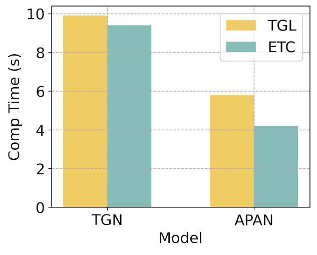
Training Configurations. We focus on the link prediction task, which is widely adopted in previous works [19, 27, 31–33, 39]. Such a task aims at predicting whether there exist interactions between given pairs of nodes in the future timestamps. Since the original datasets only contain positive interactions between nodes, an equal number of false links are sampled. As for the evaluation metric, we utilize average precision (AP) of models on the test set as in previous works [19, 27, 31, 39]. For the sampling strategy, we utilize top-k recent neighbor sampling [27] for all the underlying T-GNN models with k set as 10, since it is shown in previous works [19, 27] that the top-k recent neighbor sampling can provide better predictive performance compared to the other sampling strategies. For TGN and APAN, we set the base batch size on LastFM, Wiki-Talk, Stack-Overflow, and GDELT as 1000, 1500, 2000, and 2500 respectively. For a fair comparison, we first calculate the upper bound of information loss score for all batches generated by the conventional batch split, and then use the derived upper bound as the threshold for our proposed batch split algorithm. In this way, the two approaches are bounded by the same extent of information loss. The resultant average batch sizes by ETC's batch split algorithm on four datasets are 1180, 2180, 2938, and 2635 respectively. For TGAT model without the node memory module, the large batch size does not cause the information loss issue. Therefore, the batch split algorithm in ETC is not adopted for TGAT and we fix the batch size to be 5000 for TGAT on all the datasets. We train all the models on all the datasets for five epochs. For the other training related hyper-parameters, we do not tune and fix them the same as in TGL [39] to verify the robustness of our framework. All the experiments are done on a server with 96 CPU cores and 256 GB main memory. The model training is done on a single RTX 3090 (24GB). All the experiments are repeated three times and the mean results are reported.

5.2 Main Results

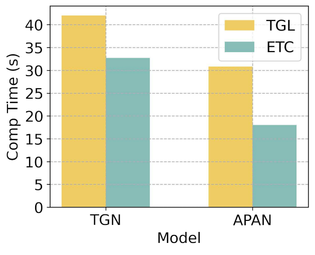
We first provide an overall comparison between ETC and baseline frameworks. We apply ETC and the baseline frameworks to train the same T-GNNs on different datasets. The results are summarized in Table 2.

Efficiency of ETC. We notice that compared to the baseline T-GNN training frameworks, ETC is able to further provide 1.6× ∼ 62.4× training speedup for different T-GNN models, and the achieved speedup is over 2× in most cases. Compared with Zebra, ETC exhibits 5.4× ∼ 62.4× training speedup. Zebra utilizes the conventional data access policy and suffers from prohibitive data access costs. In contrast, ETC is able to remove a huge amount of redundant data access thanks to its data access policy Supra. Compared with Orca, ETC shows 1.6× ∼ 25.7× efficiency gain. Orca leverages a dynamic caching mechanism for the intermediate embeddings, which not only avoids certain computation workload, but also reduces the data access volume in an indirect way. Nonetheless, it falls short of reducing a large amount of redundant data access. Compared with the next best T-GNN training framework TGL in terms of efficiency, ETC is able to provide 1.6× ∼ 3.3× faster training execution. Though the parallel sampler incorporated in TGL is able to resolve the high time complexity of sampling, it still overlooks the overwhelming data access costs.

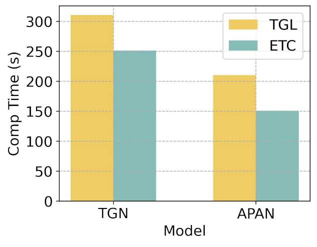
Effectiveness of ETC. By comparing the predictive results by ETC and the other generic T-GNN training framework TGL, we can see that ETC is able to well-maintain the effectiveness of different T-GNN models, as there is no prominent accuracy degradation caused by ETC framework. We also notice that the other two frameworks Orca and Zebra can present a large accuracy improvement for TGN. This is because these two frameworks modify the model structure in TGN, which can also improve the effectiveness and generalization ability of TGN. By contrast, ETC and TGL do not contain the optimizations in TGN structure design and utilize the



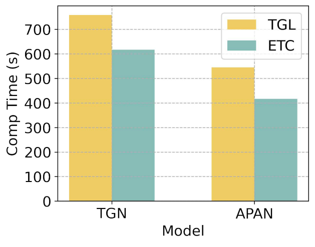
(a) LastFM



(b) Wiki-Talk



(c) Stack-Overflow



(d) GDELT

Figure 7: Per-epoch model computation time of TGN and APAN models by ETC and TGL.

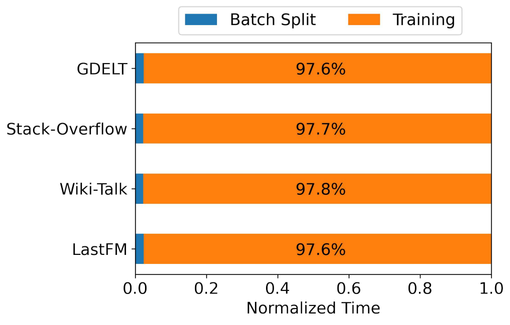


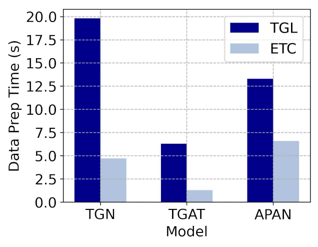
Figure 8: Comparison of normalized time between the batch split preprocessing and the training.

vanilla implementation of TGN. In fact, the techniques incorporated in ETC are orthogonal to those in Orca and Zebra. ETC as a versatile T-GNN training framework, allows the incorporation of the model-side optimizations to improve the effectiveness of the underlying T-GNN. In practice, as we train the Orca's version of the TGN model using ETC, we can achieve prominent enhancement on the predictive performance of TGN. Due to the page limit, we put the corresponding results in our technical report [2].

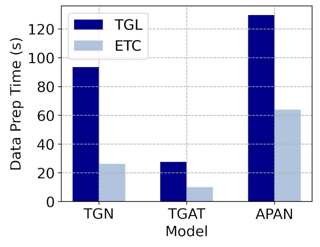
5.3 Analysis of ETC Framework

In this section, we conduct one to one comparison between the proposed ETC and the baseline TGL. Since they are both generic frameworks that support a wide range of T-GNN variants, such comparison helps to investigate the impact of different optimizations incorporated in ETC.

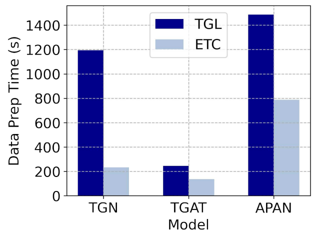
Improved Model Computation Efficiency. The batch split algorithm in ETC enables training existing T-GNNs with larger



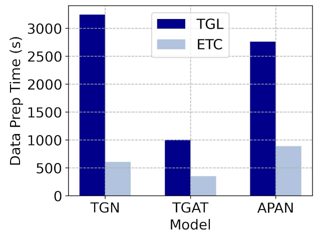
(a) LastFM



(b) Wiki-Talk



(c) Stack-Overflow



(d) GDELT

Figure 9: Per-epoch data preparation time comparison between ETC and TGL.

Table 3: Per-epoch data access time (s) comparison between TGL and ETC.

	LastFM	Wiki-Talk	Stack-Overflow	GDELT
TGL	14.2 (4.9×)	66.3 (4.2×)	906.1 (3.9×)	2599.6 (6.1×)
ETC	2.9	15.9	233.3	427.5

batch sizes, thus increasing GPU utilization. In this way, the model computation efficiency can be enhanced. We compare the model computation time of TGN and APAN under ETC and TGL implementations in Figure 7. ETC is able to provide 1.3× faster model computation on average than TGL for these models, which verifies the effectiveness of our batch split algorithm. We also evaluate the cost of the batch split algorithm on different datasets. As shown in Figure 8, the batch split algorithm does not incur a heavy preprocessing cost and it merely accounts for around 2.5% of the total end-to-end training time.

Improved Data Preparation Efficiency. The three-step data access policy Supra, as well as the inter-batch pipeline mechanism, helps reduce the data preparation costs in training existing T-GNNs over large-scale dynamic graphs. Figure 9 summarizes the data preparation costs in training existing T-GNN models with TGL and ETC frameworks. ETC can provide up to 5.5× faster data preparation (3.3× on average) compared with using TGL.

The Effect of Three-step Data Access Policy Supra. We conduct experiments on all the datasets using TGN [27] as an example to verify the effectiveness of the proposed three-step data access policy Supra. The results with the other T-GNN models are similar. Table 3 shows that Supra largely reduces the data access costs in data preparation. ETC is able to achieve 4.9×, 4.2×, 3.9×, and 6.1× faster data access compared with TGL. We notice that for denser dynamic graphs, the higher data access efficiency can be

Table 4: Per-epoch time (s) comparison for three steps of the data access policy Supra.

	LastFM	Wiki-Talk	Stack-Overflow	GDELT
Step I	2.1	10.3	137.3	276.2
Step II	2.9	15.9	233.3	427.5
Step III	0.7	2.5	22.2	60.5

Table 5: Per-epoch time (s) comparison of ETC with and without the inter-batch pipeline.

	LastFM	Wiki-Talk	Stack-Overflow	GDELT
w/o pipeline	16.9 (1.2×)	75.7 (1.3×)	783.6 (1.6×)	1850.0 (1.5×)
w pipeline	14.1	58.9	483.5	1222.4

Table 6: Comparison of batch size by the traditional and ETC's batching varying information loss score upper bound.

Method	Batch Size								
Traditional	1000	1500	2000	2500	3000	3500	4000	4500	5000
ETC's	1442	2118	2771	3399	4033	4631	5243	5860	6480

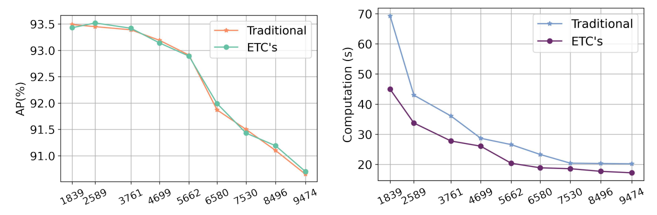
enjoyed. Specifically, on much denser datasets LastFM and GDELT, the extent of data access time reduction is more evident than that on Wiki-Talk and Stack-Overflow. This is reasonable since dense dynamic graphs are likely to incur more repeated data IDs in the neighbor sampling results. Table 4 evaluates the cost of each individual step of Supra. It is observed that the second step, which transfers the unique input data from CPU to GPU, is the most time-consuming part. Besides the second step, the first step also causes a substantial overhead, which is for the searching of unique data IDs and generating reverse mapping in each batch. While for the last step, such cost of reconstructing input data on GPU does not incur much overhead.

The Effect of the Inter-batch Pipeline Mechanism. We also evaluate the effect of the lightweight inter-batch pipeline mechanism incorporated in ETC. We conduct experiments with TGN as the backbone model on all the datasets, and compare the training efficiency of ETC with or without the pipeline mechanism. The comparison result is shown in Table 5. We see that the pipeline mechanism helps ETC further achieve 1.2× ~ 1.6× faster training speedup compared with ETC without such a pipeline mechanism.

5.4 Analysis on the Batching Scheme

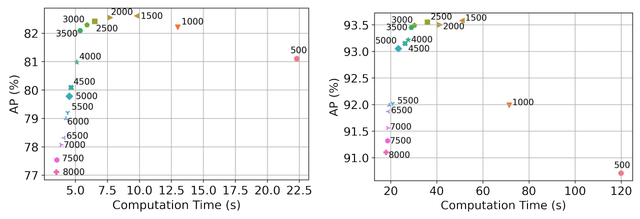
In this section, we conduct case studies with TGN [27] as the backbone model, which (1) compares ETC's batching scheme with the traditional one varying the information loss score upper bound, and (2) analyzes the impact of different threshold values by ETC. Due to the page limit, we report a few representative results that are useful for discussion. Please see our technical report [2] for comprehensive comparison results.

Varying Information Loss Score Upper Bound. As the input for the traditional batching scheme is the pre-determined batch size, we set a batch size range from 1000 to 5000 with the step as 500 for the traditional batching scheme, and calculate the upper bound



(a) Average Precision (b) Per-epoch Computation Time

Figure 10: Comparison of (a) average precision and (b) per-epoch computation time between the traditional and ETC's batching varying the information loss score upper bound on Wiki-Talk dataset.



(a) LastFM (b) Wiki-Talk

Figure 11: The effect of different values of threshold ε. X-axis denotes the per-epoch model computation time (s) and Y-axis denotes the average precision (%). Specific threshold values are represented by dots with numbers next to them.

of information loss score by the traditional batching scheme, and make them the input for ETC's batching scheme to ensure the same constraint of information loss. Figure 10 shows the comparison results varying the information loss score upper bound on Wiki-Talk. We can see that both approaches exhibit similar predictive performance when subjected to the same extent of information loss in their respective batches. Nonetheless, ETC's batching scheme showcased higher model computation efficiency under the same condition. This improvement is resulted from larger (average) batch size achieved by ETC's batching scheme as shown in Table 6.

Impact of Threshold Value & Guidance for Practitioners. We further perform a case study to investigate the impact of threshold ε and provide corresponding guidance for practitioners. Specifically, we conduct experiments varying the ε value from 500 to 8000 with a step size of 500 on LastFM and Wiki-Talk datasets, which exhibit quite distinct graph characteristics. We observe the following findings: **(1)** Increasing the threshold value improves the model computational efficiency of T-GNNs. However, we notice an effect of diminishing marginal utility when the threshold becomes sufficiently large. Further increasing the threshold value beyond a certain point only results in minimal improvements in model computational efficiency. **(2)** Interestingly, lower threshold values do not necessarily guarantee better predictive performance. It is possible that the original dynamic graphs may contain some noisy interactions, which may result in an overfitted model [26]. Therefore, we recommend a reasonably large lower limit for the threshold value (e.g., over 1000) to mitigate the overfitting issue thus improving the generation ability of the underlying T-GNN models on the test set. On the other hand, as the threshold value increases to a certain point, the model performance can degrade drastically due to too much information loss. **(3)** Besides, we can see that there exists a broad range of threshold values that can present satisfactory runtime-accuracy tradeoffs which allow practitioners to fine-tune the specific value based on their individual preference. For practitioners who prioritize higher model computation efficiency, choosing a relatively larger value within the range is advisable. Conversely, those aiming for better model performance should consider a relatively smaller threshold value within the range. **(4)** Moreover, we also notice the upper limit of the suitable threshold range can significantly vary on two datasets. In order to determine such an upper limit, it is important to consider the average update distance of the input dynamic graph (as illustrated in Table 1). The average update distance can reflect the update frequency of the node state vectors for a given dynamic graph. For the input dynamic graph with a small average update distance such as LastFM, a small threshold value upper limit should be considered (e.g., around 3000). Otherwise, a large threshold value upper limit is recommended (e.g., around 5000).

6 RELATED WORK

Representation Learning on Dynamic Graphs. To support the dynamic graph related downstream tasks, current approaches can be generally divided into two main categories, namely snapshot-based GNNs and T-GNNs. Snapshot-based GNNs focused on discrete-time dynamic graphs (DTDGs), which are represented by graph snapshots captured at different time intervals. While the most recent efforts are devoted to continuous-time dynamic graphs (CTDGs). In real-world applications, where interactions occur at varying time granularities, T-GNNs are more suitable approaches and exhibit better performance than snapshot-based GNNs.

JODIE [15] is the pioneering work of T-GNNs, which updates the node representation involved in the edges sequentially using RNN. Dyrep [28] further takes the 2-hop neighbor information into consideration to perform node representation update. TGAT [33] follows the message passing scheme in static GNNs but incorporates random Fourier features to encode the timestamp in the continuous interactions. TGN [27] is a comprehensive T-GNN framework that encompasses previous works [15, 27, 28] as specific instances. To enhance the inductive performance on unseen nodes, CAW [32] anonymizes node identities with the frequency of node occurrences based on a set of sampled walks. NeurTWs [13] utilizes temporal walks and motif structure to further improve the inductive ability. GraphMixer [6] adopts a simplified MLP-based architecture and bypasses the temporal neighborhood aggregation.

Training Frameworks for Efficient Representation Learning on Graphs. Besides developing more advanced representation learning models on graphs, researchers in the database and system community focus more on the efficient execution of such models. For representation learning on dynamic graphs, Li *et al.* [17] propose a cache-based framework to accelerate the training of static GNNs on DTDGs. DynaGraph [9] leverages cached message passing and timestamp fusion mechanisms to efficiently train existing snapshot based GNNs on DTDGs. Chakaravarthy *et al.* [5] parallelize the process of different snapshots of DTDGs to speed up the existing snapshot-based GNNs. TGL [39] is a generic framework for scaling existing T-GNN models on large-scale CTDGs, which is equipped with a parallel temporal sampler to resolve the high sampling overhead in previous implementations. Zebra [19] proposes temporal personalized PageRank and performs temporal aggregation only for top-k influential neighbors to improve model computation efficiency. Orca [18] incorporates a dynamic caching mechanism and utilizes historical embedding to reduce certain model computation and input data access workload.

For representation learning on static graphs, some works resort to mini-batch training to scale GNNs to large-scale static graphs on single machine, which aim at faster execution of sampling [7, 11, 35, 36] and CPU-GPU data transfer [20, 34]. By contrast, some others utilize distributed training approach to scale existing graph representation learning models to large-scale static graphs, and propose different optimizations to reduce heavy inter-device communication [21, 22, 25, 38, 40].

7 CONCLUSION

In this paper, we present ETC, a generic framework for efficient T-GNN training over large-scale dynamic graphs. It incorporates a novel batching scheme to improve model computation efficiency, which allows T-GNN training with a large batch size while controlling the information loss issue. Moreover, it removes the redundant input data access for temporal neighbors with a novel three-step data access policy. It further decouples the input data access and the model computation by T-GNNs using a simple yet effective pipeline mechanism. Extensive experimental results show that ETC can achieve $1.6\times \sim 62.4\times$ faster training compared to state-of-the-art training frameworks for various underlying T-GNNs on multiple large-scale dynamic graphs. A promising future direction is to design a caching mechanism tailored for T-GNN training to further reduce data access associated costs, which is capable of capturing the data access dynamics in T-GNN training.

ACKNOWLEDGMENTS

Yanyan Shen's work is supported by the National Key Research and Development Program of China (2022YFE0200500), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102), the Tencent Wechat Rhino-Bird Focused Research Program, and SJTU Global Strategic Partnership Fund (2021 SJTU-HKUST). Yingxia Shao's work is supported by the National Natural Science Foundation of China (Nos. 62272054, 62192784), Beijing Nova Program (No. 20230484319), and Xiaomi Young Talents Program. Lei Chen's work is partially supported by National Science Foundation of China (NSFC) under Grant No. U22B2060, the Hong Kong RGC GRF Project 16209519, CRF Project C6030-18G, C2004-21GF, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab grant and HKUST Global Strategic Partnership Fund (2021 SJTU-HKUST).

REFERENCES

[1] [2023]. Stack-Overflow. https://snap.stanford.edu/data/sx-stackoverflow.html

[2] [2023]. The technical report. https://github.com/eddiegaoo/ETC/blob/main/Technical-report.pdf

[3] [2023]. Wiki-Talk. http://snap.stanford.edu/data/wiki-talk-temporal.html

[4] Korte Bernhard and Jens Vygen. 2008. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.* (2008).

[5] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[6] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. 2023. Do We Really Need Complicated Model Architectures For Temporal Networks?. In *International Conference on Learning Representations*.

[7] Jialin Dong, Da Zheng, Lin F Yang, and George Karypis. 2021. Global neighbor sampling for mixed CPU-GPU training on giant graphs. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 289–299.

[8] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. 2020. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems* 187 (2020), 104816.

[9] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

[10] Ehsan Hajiramezanali, Arman Hasanzadeh, Krishna Narayanan, Nick Duffield, Mingyuan Zhou, and Xiaoning Qian. 2019. Variational graph recurrent neural networks. *Advances in neural information processing systems* 32 (2019).

[11] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[12] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[13] Ming Jin, Yuan-Fang Li, and Shirui Pan. 2022. Neural Temporal Walks: Motif-Aware Representation Learning on Continuous-Time Dynamic Graphs. In *Advances in Neural Information Processing Systems*.

[14] Thomas N Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[15] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 1269–1278.

[16] Kalev Leetaru and Philip A Schrodt. 2013. Gdelt: Global data on events, location, and tone, 1979–2012. In *ISA annual convention*, Vol. 2. Citeseer, 1–49.

[17] Haoyang Li and Lei Chen. 2021. Cache-based gnn system for dynamic graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 937–946.

[18] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Orca: Scalable Temporal Graph Neural Network Training with Theoretical Guarantees. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[19] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When Temporal Graph Neural Networks Meet Temporal Personalized PageRank. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1332–1345.

[20] 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.

[21] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: a graph-based system approach to scaling large embedding model training. In *Proceedings of the 2022 International Conference on Management of Data*. 470–480.

[22] 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.

[23] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 5363–5370.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[25] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: Staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1937–1950.

[26] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *International Conference on Learning Representations*.

[27] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *ICML 2020 Workshop on Graph Representation Learning*.

[28] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*.

[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[30] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[31] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*. 2628–2638.

[32] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *International Conference on Learning Representations*.

[33] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations*.

[34] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNlab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 417–434.

[35] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[36] Xin Zhang, Yanyan Shen, and Lei Chen. 2022. Feature-Oriented Sampling for Fast and Scalable GNN Training. In *2022 IEEE International Conference on Data Mining (ICDM)*. IEEE, 723–732.

[37] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

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

[39] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: a general framework for temporal GNN training on billion-scale graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1572–1580.

[40] 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.