DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training

Zhen Song
Northeastern Univ., China
songzhen_neu@163.com

Yu Gu*
Northeastern Univ., China
guyu@mail.neu.edu.cn

Qing Sun
Northeastern Univ., China
sunqing_neu@163.com

Tianyi Li
Aalborg Univ., Denmark
tianyi@cs.aau.dk

Yanfeng Zhang
Northeastern Univ., China
Zhangyf@mail.neu.edu.cn

Yushuai Li
Aalborg Univ., Denmark
yusli@cs.aau.dk

Christian S. Jensen
Aalborg Univ., Denmark
csj@cs.aau.dk

Ge Yu
Northeastern Univ., China
yuge@mail.neu.edu.cn

ABSTRACT

Dynamic Graph Neural Networks (DGNNs) have demonstrated exceptional performance at dynamic-graph analysis tasks. However, the costs exceed those incurred by other learning tasks, to the point where deployment on large-scale dynamic graphs is infeasible. Existing distributed frameworks that facilitate DGNN training are in their early stages and experience challenges such as communication bottlenecks, imbalanced workloads, and GPU memory overflow.

We introduce DynaHB, a distributed framework for DGNN training using so-called Hybrid Batches. DynaHB reduces communication by means of vertex caching, and it ensures even data and workload distribution by means of load-aware vertex partitioning. DyanHB also features a novel hybrid-batch training mode that combines vertex-batch and snapshot-batch techniques, thereby reducing training time and GPU memory usage. Next, to further enhance the hybrid batch based approach, DynaHB integrates a reinforcement learning-based batch adjuster and a pipelined batch generator with a batch reservoir to reduce the cost of generating hybrid batches. Extensive experiments show that DynaHB is capable of up to a 93× and an average of 8.06× speedups over the state-of-the-art training framework.

1 INTRODUCTION

Graph Neural Networks (GNNs) that operate on static graphs have achieved remarkable success, and system aspects of enabling efficient training have been studied comprehensively. However, GNNs

are constrained in their ability to capture the temporal dynamics of evolving graphs [1, 10, 17–19, 22, 55, 56]. Thus, a new category of methods known as Dynamic Graph Neural Networks (DGNNs) has emerged that targets the analysis of dynamic graphs.

Because they manage temporal information, the computational complexity of DGNNs significantly exceeds that of GNNs: (i) DGNNs need to process multiple snapshot graphs, whereas GNNs only process a single snapshot, leading to higher time and space complexity; (ii) DGNNs typically involve communication between different snapshots through Recurrent Neural Networks (RNNs) in addition to the traditional GNN vertex-to-vertex communication along the graph structure, resulting in increased communication; (iii) DGNNs not only require synchronization of vertex embeddings at each layer but also necessitate synchronization between RNNs across snapshots, introducing additional intricacies when enabling parallel processing. Hence, prevailing distributed GNN training frameworks fall short at accommodating large-scale DGNN training.

Currently, several single-machine frameworks exist that are developed specifically for the training of DGNNs, including PyGT [33], CacheG [14], and PiPAD [43]. These frameworks focus primarily on optimizing aspects such as intermediate result caching and reuse, pipeline parallelism, etc. However, as these frameworks are constrained by the limited resources of a single machine, they fall short at supporting large-scale DGNN training. Recently, distributed DGNN training frameworks, including ESDG [3], DGC [5], BLAD [7], and DynaGraph [9], have emerged, but they are still in their early stages of development and face three major challenges.
Challenge I: High Communication Overhead. In addition to requiring vertex communication for graph convolution, DGNNs also require vertex communication between different snapshots for RNNs, leading to significant communication overhead. In a dynamic graph with T snapshots and N vertices, each DGNN training iteration incurs T times the vertex communication of traditional GNN training for graph convolution. Furthermore, an additional $O(NT)$ overhead is incurred for RNN communication. Existing techniques [3, 5, 7, 9] aim to reduce communication through partitioning strategies, but the impact is limited by snapshot constraints, resulting in communication bottlenecks.
Challenge II: Inefficiency at Large-Scale DGNN Training. Many existing frameworks [3, 5, 14, 35] train DGNNs in full batch mode, reducing efficiency and increasing GPU costs. Snapshot-batch (i.e., sliding-window) training [9, 33], which uses a set of consecutive snapshots, faces reduced computational efficiency and increased GPU memory usage as the number of vertices increases. On the

other hand, vertex-batch (i.e., mini-batch) training [57, 60] where a subset of vertices across all snapshots is processed, faces challenges with large snapshot volumes and high batch generation costs due to edge re-encoding.

Challenge III: Skewed Workload and Data Distribution. Real-world graphs often exhibit skewed degree distributions. For example, power-law graphs include small numbers of vertices with high degrees. As a result, techniques that partition vertices uniformly may cause uneven computational loads to be assigned to different machines. Unbalanced workload causes increased synchronization overhead, as well as uneven distribution of training data, especially for distributed DGNN training. The prevailing DGNN frameworks (e.g., [3, 7, 9]) predominantly employ snapshot partitioning, a coarse-grained partitioning technique that results in limited load balancing control.

To address the above challenges, we propose an asynchronous distributed training framework for **Dyna**mic Graph Neural Network using **H**ybrid **B**atches, called DynaHB. To address ***Challenge I***, we implement a vertex partitioning strategy that minimizes communication through caching. We employ asynchronous local model updates alongside periodic synchronization of the global model to reduce the otherwise high synchronization costs. To address ***Challenge II***, we introduce a novel GPU-friendly hybrid-batch structure, which strikes a purposeful balance between efficiency and convergence. To dynamically determine a carefully selected hybrid batch size during training, we introduce a reinforcement learning-based batch adjuster. Further, to decrease the cost of generating hybrid batches, we introduce a Pipeline-based Hybrid Batch Generator that utilizes the batch-reservoir technique. To address ***Challenge III***, we enable balanced workloads and data distributions by providing means of load-aware vertex partitioning coupled with a load-balancing training strategy. Finally, we provide a convergence proof for DynaHB training.

Our contributions are summarized as follows.

- We propose an asynchronous framework for distributed DGNN training that alleviates communication and synchronization overheads. We tailor a load-aware vertex partition strategy coupled with a load-balanced training strategy.
- We propose a flexible, GPU-friendly Hybrid-Batch training mode. Moreover, we design a reinforcement learning based batch adjuster, aiming to minimize the overall convergence time by selecting carefully a batch size during DGNN training.
- We propose a Pipeline-based Hybrid Batch Generator aimed at overlapping training time with batch generation time. The generator incorporates a batch-reservoir technique to preserve generated hybrid batches, thereby enabling their reuse and effectively reducing batch generation costs.
- Extensive experiments conducted with three common DGNN models on eight public datasets offer insight into the performance of DynaHB. DynaHB achieves up to a 93× and an average of 8.06× speedups over snapshot batch techniques, the state-of-the-art DGNN training mode.

The paper is structured as follows. Section 2 reviews related work of DGNN models, optimizations, and systems. Section 3 provides preliminaries on dynamic graphs and DGNN training. Section 4 elaborates an overview of DynaHB and load-aware partitioning and

training, while Section 5 details the reinforcement learning-based hybrid batch adjustment and optimizations. Experimental results are presented in Section 6, and the paper concludes with a summary in Section 7.

2 RELATED WORK

2.1 DGNN Models

Existing DGNN models are based on either snapshot graph processing or event stream processing.

Snapshot graph processing. These methods focus on analyzing discrete graph snapshots over time. DCRNN [22] captures spatial information through bidirectional random walks and uses an encoder-decoder framework for temporal learning. To fully capture the changes in dynamic graphs, SEIGN [30] proposes a dual evolution technique for graph model parameters and the representation of each snapshot graph. ROLAND [52] treats node embeddings at different GNN layers as hierarchical node states and proposes a scalable dynamic graph learning framework, which can help researchers apply static GNNs to dynamic graphs. DGNNs [1, 6, 16, 22, 26, 30, 34, 37, 52, 56] typically exhibit variations in their temporal and spatial encoding methodologies, while the foundational communication patterns, encompassing GCN and RNN communication topologies, remain invariant.

Event stream processing. These methods aim to handle continuous flows of graph events. JODIE [13] uses recursive neural networks to understand user-item interactions, predicting future activities. Zebra [21] simplifies recursive temporal message passing by connecting it with time-random-walk processes, selecting key temporal neighbors to compute node representations. However, these models [13, 46, 47, 50] cannot handle scenarios where data arrives in batches or data is collected at regular intervals. Thus, they fall outside the scope of this paper.

2.2 Performance Optimizations for DGNNs

Efforts to speed up DGNN training involve optimizations for single servers and distributed servers.

Single-server optimizations. PyGT [33] introduces a programming library for DGNN training. CacheG [14] reduces redundant computations by caching and reusing hidden layer embeddings. Pi-PAD [43] uses a pipelined parallel DGNN training architecture with overlap-aware data organization. TGOpt [48] adopts a redundancy-aware training strategy for temporal graph attention networks. These systems face challenges in supporting large-scale DGNN training due to single machine resource constraints.

Distributed-servers optimizations. ESDG [3] avoids GNN communication with snapshot-based partitioning, but RNN computations still require communication. DynaGraph [9] employs sliding-window (SW) based snapshot partitioning, eliminating communication during training but requiring communication for data exchange. DGC [5] combines snapshot and vertex partitioning to reduce communication costs. ADGNN-T extends a GNN framework [35] with vertex-partition-based distributed training. These architectures still face communication bottlenecks.

Next, we compare DynaHB with the state-of-the-art proposals in terms of partitioning framework and batch modes.
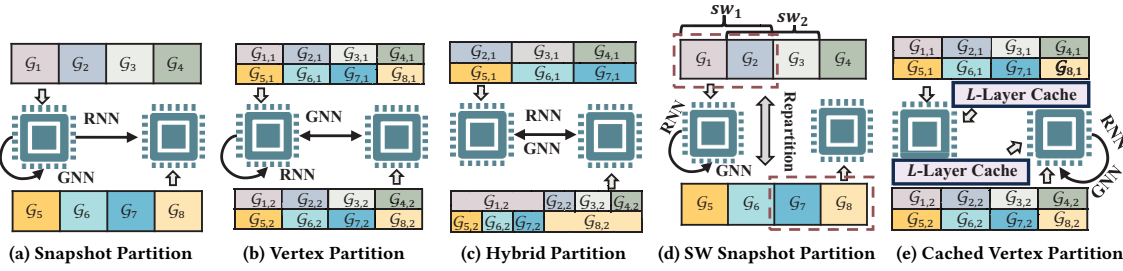
Figure 1: Partition frameworks. Here, $\mathcal{G}_i$ represents the ith snapshot, with each snapshot depicted in a different color; $\mathcal{G}_{i,j}$ represents the part of the ith snapshot allocated to the jth worker; sw_i represents the ith sliding window.

(i) **Partition frameworks.** The snapshot partitioning in Figure 1a stores four snapshots per worker. Thus, each worker can independently perform GNN convolutions, while vertex exchange is needed for RNN computation. Vertex partitioning in Figure 1b stores half of the vertices per worker. Convolution of each snapshot needs remote embeddings, while RNN can execute locally. The hybrid partitioning in Figure 1c requires communication to compute RNN and GNN, while minimizing communication. The SW-based snapshot partitioning in Figure 1d does not require communication during training; however, communication is needed when exchanging snapshots to achieve random data distribution.

In contrast to the above, DynaHB employs cache-based vertex partitioning for five key reasons: (a) it resolves the communication bottleneck; (b) DGNNs are relatively shallow, distributed CPU memory is often sufficient to accommodate the caching of vertices; (c) synchronization overhead is nearly eliminated, achieving close to linear scalability; and (d) vertex partitioning enables distributed processing of a single snapshot graph, making it scale better to large data sizes. Figure 1 shows the above five partitioning strategies for distributed DGNN training. Overall, cache-based vertex partitioning is the most suitable for distributed DGNN training.

(ii) **Batch modes.** ESDG [3], DGC [5], and ADGNN-T [35] primarily use full-batch training, which can be challenging due to GPU memory limitations. Two common mini-batch modes are snapshot-batch and vertex-batch. DynaGraph [9] supports a snapshot batch mode (sliding windows), grouping consecutive snapshots to reduce computational complexity. However, fitting large snapshot graphs into GPU memory remains difficult. Vertex-batch mode [57, 60] is used in GNN training to reduce computational load. However, small snapshot loads may not fully utilize GPU performance, and sampling vertices introduces additional encoding costs.

In contrast, DynaHB employs a hybrid batch training mode, combining vertex batches and snapshot batches. First, the mode enhances GPU training by improving memory efficiency and adaptability. Second, the hybrid batch training mode allows flexible control over data size, balancing efficient training with high accuracy.

EXAMPLE 1. *Figure 2 illustrates two consecutive rounds of DGNN training, showcasing three different training modes. In each training round, the vertex batch includes 2 vertices from all 4 snapshots, resulting in a computational and storage cost of 16 target vertices. Similarly, the snapshot batch includes 3 snapshots for all 4 vertices, totaling 24 target vertices. In contrast, the hybrid batch requires computation and storage for only 7 target vertices.*
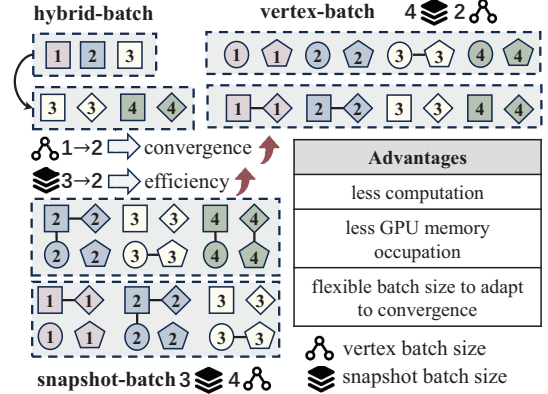

Figure 2: Batch type comparison. Colors and numbers indicate snapshots, while shapes represent vertices.

2.3 Techniques for Other Graph Learning Tasks

Training optimizations for static GNNs. Various optimizations have been proposed for static GNN training, including techniques such as caching [23, 60], message compression [36], disk optimization [28], pipeline parallelism [38, 41, 42], training frameworks [29, 44, 45], and sampling [35, 51, 54]. While effective in static environments, these optimizations encounter challenges in distributed DGNN training.

Training optimizations for event stream based DGNNs. Event-based DGNNs [13, 31, 39, 46, 47, 50] handle event streams by updating node representations when relevant events occur. Orca [20] improves training efficiency by caching and reusing intermediate embeddings. EARLY [15] addresses quality-deficit and neighbor-redundancy issues by identifying influential nodes and proposing a diversity-aware sampling technique. These systems [4, 15, 20, 49, 53, 58, 59] target event-based DGNN models in event stream processing. The difference in design goals distinguishes DynaHB's system training architecture from the requirements of event-stream based DGNN training, which is beyond the scope of our study.

Partitioning for static GNN training. BNS-GCN [40] ensures balance by evenly distributing vertices but lacks load consideration. P^3 [8] splits features, but struggles with low dimensions and communication caching. Graph Ladling [11] divides vertices into partitions but ignores inter-cluster edges. CoFree-GNN [2] cuts vertices for load balance but unsuitable for DGNN training as it cannot control the placement of vertices. Moreover, these techniques are

not tailored to the computational load of DGNNs, while DynaHB simulates the cost of DGNN training and balances training loads.

3 PRELIMINARIES

We follow existing studies [5, 43] to define dynamic graphs.

DEFINITION 1. **Dynamic Graph.** *A dynamic graph is an ordered set of snapshot graphs* $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_T\}$. $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ $(1 \leq t \leq T)$ *is the snapshot at timestamp* t *with vertex set* $\mathcal{V}_t$ *and edge set* $\mathcal{E}_t$. *Each vertex* $v_{i,t} \in \mathcal{V}_t$ *is associated with a feature vector* $\mathbf{x}_{i,t} \in \mathcal{X}_t$ *and a label* $y_{i,t} \in \mathcal{Y}_t$, *where* $\mathcal{X}_t$ *and* $\mathcal{Y}_t$ *are the sets of feature vectors and labels for all vertices at timestamp* t. *Each edge* $e_{(i,j),t}$, *connecting* $v_{i,t}$ *and* $v_{j,t}$, *is associated with an edge weight* $w_{e_{(i,j),t}} \in \mathcal{W}_{\mathcal{E}_t}$, *where* $\mathcal{W}_{\mathcal{E}_t}$ *is the set of weights for all edges at timestamp* t.

Each DGNN layer includes structural and temporal encoding modules. Structural encoding uses GNNs to aggregate neighboring information. We simplify notation by representing vertex embeddings of snapshot t at layer ℓ as $\boldsymbol{h}_{v,t}^{\ell}$.

$$\text{GNN}(\boldsymbol{h}_{v,t}^{\ell}) = \text{UPDGNN}(\text{AGG}(\boldsymbol{h}_{u,t}^{\ell-1}|u \in \mathcal{N}(v_t)), \boldsymbol{h}_{v,t}^{\ell-1}), \quad (1)$$

where $\mathcal{N}(v_t)$ denotes the neighbors of vertex v_t and $\boldsymbol{h}_{u,t}^{\ell-1}$ denotes the embedding representation of neighbor u of v at layer $\ell - 1$ at timestamp t. For the snapshot $\mathcal{G}_t$, AGG$(\cdot)$ aggregates neighboring vertex features on vertex v at layer ℓ. UPDGNN$(\cdot)$ combines neighbor embeddings and self-embedding, followed by updating the self-embedding using the parameters of the GNN.

After structural encoding, each target vertex obtains its embedding representation. Next, the embeddings of identical vertices across different snapshots are amalgamated using Recurrent Neural Networks (RNNs) for temporal encoding.

$$\text{RNN}(\boldsymbol{h}_{v,t}^{\ell}) = \text{UPDRNN}(\boldsymbol{h}_{v,t}^{\ell}, hidden_{v,t-1}^{\ell}), \quad (2)$$

where $hidden_{v,t-1}^{\ell}$ denotes the hidden state of vertex v from $\mathcal{G}_{t-1}$, and UPDRNN$(\cdot)$ gets a new embedding $\boldsymbol{h}_{v,t}^{\ell}$ for v_t and passes the hidden state $hidden_{v,t}^{\ell}$ to snapshot $\mathcal{G}_{t+1}$. Finally, each vertex attains an embedding that encapsulates both spatial and temporal information. The stacking of DGNN layers facilitates the realization of the multiple layers of a DGNN.

4 SYSTEM OVERVIEW AND LOAD-AWARE OPTIMIZATION TECHNIQUES

4.1 System Overview

We present an overview of DynaHB in Figure 3. DynaHB functions through four modules. *Dynamic Graph Partitioner* obtains a logical vertex partition for a given dynamic graph. *Hybrid-batch Generator* generates hybrid batches and feeds them into DGNN training. *DGNN Trainer* trains the provided hybrid batch, executing both forward and backward propagations to obtain the gradients and updating local models. *Asynchronous Model Controller* coordinates model averaging across all workers through a global timer.

We elucidate the specific functions of each module according to the execution order in Figure 3. Given a dynamic graph $\mathcal{G}$, the *Load-computation Unit* calculates the workload of each vertex during DGNN training (Step ①). Second, the load information is conveyed to the *Load-aware Partitioner* (Step ②). This procedure gets a balanced logical partition $P = \{P_1, P_2, \ldots, P_M\}$, where M denotes the

number of workers. Next, the *L-layer Cache Constructor* retrieves iteratively remote vertices from the L-hop neighbors of local vertices for the ith $(1 \leq i \leq M)$ worker, i.e., $C_i = \{u|u \in \bigcup_{l=1}^{L} \mathcal{N}^l(v) \wedge u \notin P_i\}$, where $\mathcal{N}^l(v)$ represents the lth-hop neighbors of v (Step ③). The training of DGNNs, similar to GNN training, features L-hop neighbor isolation. This means that the information within L hops of the target vertex contains all the necessary data for its DGNN training. Consequently, we achieve communication avoidance by caching neighbors within this L-hop range. Finally, the *Data Assignment Module* allocates data to the designated workers according to the partition results and L-hop remote vertices, e.g., worker 1 receives P_1 and C_1 (Step ④).

During training, DynaHB generates hybrid batches. The RL-based Hybrid Batch Size Adjuster determines the optimal size, combining snapshot and vertex counts (Step ⑤). Next, the Vertex Sampler selects target vertices (Step ⑥), which, along with the size parameters, are input into the Hybrid Batch Generator to form a new hybrid batch (Step ⑦). This batch is then sent for GPU training (Step ⑨). Hybrid batches are saved in the Hybrid Batch Reservoir, allowing us to choose between reusing an existing batch or generating a new one for subsequent training (Step ⑧).

DynaHB effectively resolves communication and GPU memory bottlenecks. In scenarios with ample network bandwidth and GPU memory, DynaHB also offers benefits: a) the asynchronous mode avoids synchronous waiting costs, even in resource-rich environments; and b) its hybrid batch training allows for flexible adjustment of vertex and snapshot batch sizes, reducing computational costs on each worker.

4.2 Load-aware Optimization Strategies

Each target vertex has its own workload during DGNN training. During DGNN training, the workload of a vertex mainly includes operations related to GNNs and RNNs. The computation of GNNs is related to the number of L layers of neighbors for a vertex. Below, we will provide the relevant definitions for workload. We disregard the RNN computation cost when calculating the workload since each vertex has the same computation cost.

DEFINITION 2. *The* **vertex workload** $\mathcal{L}_i^t$ *of a vertex* $v_{i,t} \in \mathcal{V}_t$ *is the recursive sum of degrees from layer* 1 *to layer* L, *i.e.,* $\mathcal{L}_i^t = \sum_{l=1}^{L} (\sum_{l'=1}^{l} |\mathcal{N}^{l'}(v_{i,t})|)$, *where* $\mathcal{N}^{l'}(v_{i,t})$ *denotes the set of* l'-*hop neighbors of* $v_{i,t}$ *in* $\mathcal{G}_t$.

We provide an example of calculating the workload for vertex v_2 in a 2-layer DGNN model in Figure 4.

EXAMPLE 2. *We obtain the neighbors within 2-hops for vertex* v_2 *in snapshots* $\mathcal{G}_1$ *and* $\mathcal{G}_2$, *as shown in Figure 4. In* $\mathcal{G}_1$, $\mathcal{N}^1(v_{2,1}) = \{v_1, v_3\}$, *and* $\mathcal{N}^2(v_{2,1}) = \{v_3\}$, *where we omit the snapshot index for clarity. Thus* $\mathcal{L}_2^1$ *is calculated as* $|\mathcal{N}^1(v_{2,1})| + (|\mathcal{N}^1(v_{2,1})| + |\mathcal{N}^2(v_{2,1})|) = |\{v_1, v_3\}| + (|\{v_1, v_3\}| + |\{v_3\}|) = 5$, *where* $|\cdot|$ *denotes the size of the set. Similarly,* $\mathcal{L}_2^2 = 5$. *Consequently, the total workload for* v_2 *in DGNN training is 10. The table in the top right corner of Figure 4 shows the workload for each vertex.*

We calculate vertex workloads using label aggregation. Initially, each vertex is assigned a label of 1 to indicate a workload contribution of 1 unit as a neighboring vertex. We use label propagation along edges to update vertex labels and perform aggregation at
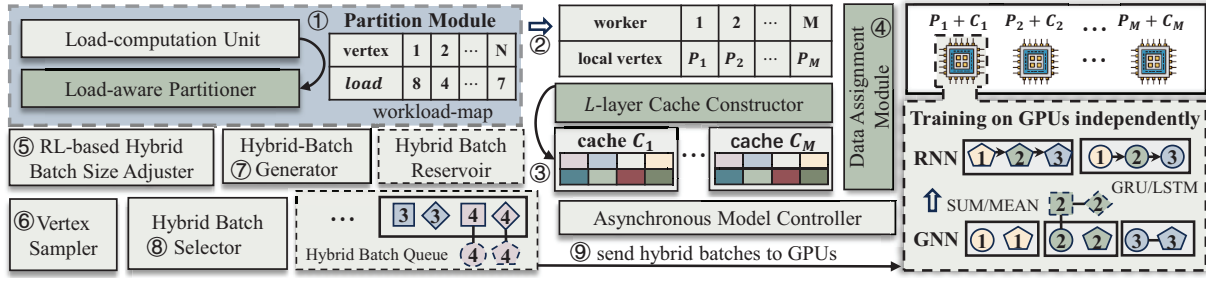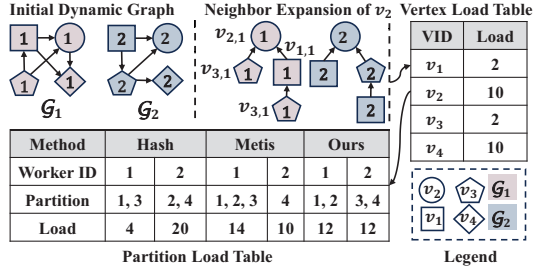
Figure 3: System overview

Figure 4: An example of workload calculation

target vertices using addition. For an L-layer DGNN, we repeat this process L times to determine the final workload for each vertex. The resulting vertex workloads are sorted in descending order and used to allocate vertices, ensuring balanced local workloads.

Algorithm 1 summarizes the proposed Load-aware Partition strategy. First, we initialize *vertex_load* to store the final vertex load (lines 1-2), which is the sum of the load of each vertex across all layers in all snapshots. Then, we start computing the load for each layer of vertices (lines 3-7). We initialize the load of each edge to 1, representing the initial aggregation load of an edge in the initial layer, and store it in *edge_loads*[0]. We obtain the edge load *edge_load* and the destination vertex *dst* for each edge. Based on this information, we aggregate the edge values according to the destination vertex (line 6). We accumulate the vertex load across all layers in all snapshots into *vertex_load_raw* (line 7). Next, we compute the sum of the load of each vertex across all layers in all snapshots (line 10). We sort the loads and partition them using the round-robin method (line 11), simultaneously dividing the vertices into *group_size* load-balanced groups. Example 3 demonstrates the efficiency of our Load-aware Partition.

EXAMPLE 3. *Continuing Example 2, we provide the workload for each worker of the Hash, Metis [12], and our Load-aware Partition strategy. Hash uses a round-robin partitioning. As shown in the partition load table (the bottom of Figure 4), v_1 and v_3 are assigned to worker 1, and v_2 and v_4 are assigned to worker 2. Thus, in this scheme, the workload for worker 1 is 4, and the workload for worker 2 is 20, which is highly unbalanced. Metis [12] is a partition approach aimed at minimizing the edge cut, thus v_1, v_2, and v_3 are assigned to worker 1, and vertex 4 is assigned to worker 2. The workloads for each worker are 14 and 10, respectively, also unbalanced. In contrast, our technique results in a workload of 12 for both workers, which is balanced.*

After getting the optimal hybrid batch size through reinforcement learning, which will be detailed in Section 5, it is necessary to

(i) select snapshots based on the snapshot batch size and (ii) select target vertices based on the vertex batch size. For (i), an initial snapshot ID is selected from the range $[0, T - N_S - 1]$, where N_S denotes the snapshot batch size. Following this, the next N_S snapshots are sampled sequentially. However, for (ii), random sampling can lead to imbalanced workloads due to differences in computational loads for each vertex. This can result in an overall decrease in training efficiency and convergence issues caused by imbalanced distributions. During training, coordinating the selection of a balanced hybrid batch for each worker is an important challenge.

Algorithm 1: Load-aware partition

Input: Vertex Number *node_num*, Layer Number *layer_num*, Dynamic Graph *graph*
Output: Partition results *partition*

1 *vertex_load*=zeros(*node_num*)
2 *vertex_load_raw*=zeros((*layer_num*,*node_num*))
   // calculate vertex loads for each layer
3 **for** l **from** 1 **to** *layer_num*+1 **do**
4   **for** t **from** 0 **to** *graph.snap_num* **do**
5   *edge_load, dst*=*edge_loads*[l-1][t], *graph.edges*[t][1]
6   *edge_loads*[l][t] = scatter(*edge_load, dst*, dim=0, dim_size=*node_num*, reduce="sum")
7   *vertex_load_raw*[l] += *edge_loads*[l][t]

   // calculate total vertex loads
8 **for** i **from** 0 **to** *layer_num* **do**
9   **for** j **from** 0 **to** i **do**
10   *vertex_load* += *vertex_load_raw*[j]

   // get partition and balanced groups
11 *partition*=div_array_by_round_robin(*vertex_load*)
12 *balanced_groups*=split_into_n_parts(*partition*[*worker_id*],*group_size*)
13 **return** *partition*

We design a target vertex sampler based on balanced groups, which selects the same number of target vertices within each group of vertices with different computational loads to achieve balanced training. First, during dynamic graph partitioning, we calculate the workload for each vertex and allocate target vertices to each worker. Within each worker, we can obtain the local vertex load through the vertex workload table and arrange them in descending order. Next, we divide the vertices, sorted by load, into N_{bg} groups on average, ensuring balance in the workload for the majority of vertices within each group.

During the training process, we divide the vertex batch size $N_\mathcal{V}$ into N_{bg} parts. Hence, we sample $N_\mathcal{V}/N_{bg}$ target vertices for each balanced group and concatenate them to form a complete set of sampled target vertices for training. This not only ensures balanced workloads and shortens training time but also allows for uniform sampling from vertices with different loads in each training iteration, simulating the overall distribution effectively.

5 JOINT OPTIMIZATIONS FOR HYBRID BATCH GENERATION

5.1 Reinformance Learning Based Strategy

We provide the formal definition of a hybrid batch below.

DEFINITION 3. *In a dynamic graph comprising T snapshots, the size of a snapshot batch, denoted as N_S, and the size of a vertex batch, denoted as $N_\mathcal{V}$, are defined. A **target vertex set** $\mathcal{V}_\mathcal{H}$ of a hybrid batch Hb is given by $\{v_{i,t}|i \in B_\mathcal{V}, t \in B_S\}$. Here, $B_S = \{\mathcal{I}_S^t, \mathcal{I}_S^{t+1}, \dots, \mathcal{I}_S^{t+N_S}\}$ and $B_\mathcal{V} = \{\mathcal{I}_\mathcal{V}^{i_1}, \mathcal{I}_\mathcal{V}^{i_2}, \dots, \mathcal{I}_\mathcal{V}^{i_{N_\mathcal{V}}}\}$, where $\mathcal{I}_S = \{s|1 \le s \le T\}$ and $\mathcal{I}_\mathcal{V} = \{v|1 \le v \le N\}$ are index sets of snapshots and vertices, respectively. A **hybrid batch** Hb represents a trainable subgraph constructed based on $\mathcal{V}_\mathcal{H}$.*

During training, the size of a hybrid batch $N_\mathcal{H}$ affects the variance of stochastic gradients and training efficiency. Our goal is to reduce the variance and increase training efficiency. However, different dynamic graphs exhibit varying sensitivities to the snapshot batch size N_S and vertex batch size $N_\mathcal{V}$. Simultaneously, various factors such as different iteration stages, DGNN models, layer numbers, dynamic graph distributions, etc., may all impact the convergence. This motivates us to develop a reinforcement learning based strategy to adaptively adjust the size.

Reinforcement Learning (RL) is an online learning method designed to efficiently determine the optimal execution strategy for policy-based tasks. We propose a reinforcement learning model tailored to dynamically adjust the hybrid batch size. DQN [25] is effective for discrete action spaces and continuous state spaces, aligning with batch sizes and convergence states. The quick convergence of experience replay is vital as we need rapid policy selection. To achieve optimal performance, it is essential to define the appropriate state space, action space, and reward metric. We propose to simplify the state and action spaces to make DQN converge in the early iterations by the following design principles.

State space. The state space should capture the core factors influencing action selection. A factor is considered unreasonable if two distinct states result in the same action. This is because the introduction of redundant state factors increases the complexity of DQN, leading to slower convergence. We formulate the designed redundancy-free state space in Definition 4.

DEFINITION 4. *A **redundancy-free state space** S is designed as loss reduction, i.e., $S = \Delta loss$, which is a single-factor state space.*

Action space. The design of the action space should adhere to two principles: a small number of actions and a large distinction between actions. Two strategies are typically employed to define numerical action space: (i) action combination (ii) change trend. In our scenario, the former leads to enormous potential combinations, whereas the latter triggers more frequent changes to the hybrid

batch, thereby impeding convergence. As a result, we propose a streamlined and clear-cut action space design in Definition 5.

DEFINITION 5. *An **action space** $\mathcal{A}$ is formed by pairing down the number of snapshots and vertices to a set of common empirical sizes, i.e., $\mathcal{A} = SS \times VS$, where SS and VS represent the sets of the sizes of snapshots and vertices.*

By condensing the snapshot and vertex batch sizes to widely recognized empirical values and using these as our action space, we achieve a more manageable set of actions. On the one hand, a smaller action space is easier to converge; on the other hand, it addresses the challenges of making precise adjustments to the hybrid batch size and alleviates the problem of slow responsiveness. **Reward.** Reward setting is a critical step in RL. Our goal in adjusting the batch is to minimize the overall convergence time. However, without prior knowledge of whether an action leads to the shortest convergence time among all possible choices, creating an effective reward system is complex. To address the problem, we define the reward as follows.

DEFINITION 6. *The **reward** is defined as the sum of the time reward $\mathcal{R}_{time}$ and the loss reward $\mathcal{R}_{loss}$.*

$$\mathcal{R} = \mathcal{R}_{time} + \mathcal{R}_{loss} = \frac{t_l - t_c}{t_l} + \frac{\Delta loss_c - \Delta loss_l}{\Delta loss_c}, \quad (3)$$

where t_l and t_c represent the training time of the last and current iterations, respectively, while $\Delta loss_c$ and $\Delta loss_l$ represent the loss reduction of the current and last iterations, respectively. Clearly, $\mathcal{R}_{time}$ is the percentage improvement in speed compared to the previous solution, while $\mathcal{R}_{loss}$ is the percentage decrease in $\Delta loss$.

Reinforcement learning model parameters. We use a consistent set across all datasets: $\gamma = 0.99$, $\epsilon = 1$, $\epsilon_{decay} = 0.95$, $\epsilon_{min} = 0.1$, $lr = 0.005$. Here's a brief explanation of each parameter: γ balances immediate and future rewards; ϵ, ϵ_{decay}, ϵ_{min} control the probability of random selection, its decay rate, and minimum value. These parameters ensure that strategy selection avoids local optima by promoting early exploration and gradually reducing it as training stabilizes. A learning rate of 0.005 is chosen for faster convergence.

5.2 Pipeline Based Hybrid Batch Generator

In distributed DGNN training, the process of generating a hybrid batch involves multiple operations such as iteratively generating target vertices for each layer, vertex encoding, and mapping edges to new encodings. This process, conducted on CPUs, is time-consuming. To alleviate this, we employ a reservoir.
Hybrid Batch Reservoir is employed to store the generated hybrid batches. Since we adopt an adaptive schema for dynamically adjusting the hybrid batch size, the reservoir needs to establish storage areas for different sizes of hybrid batches. When a new hybrid batch is generated, it is added to the designated storage area.
Hybrid Batch Selector determines whether to create a new hybrid batch or retrieve one from the reservoir during training, guided by a probability function.

$$p = \frac{N_{rsv}}{N \cdot T/(N_S \cdot N_\mathcal{V})/N_\mathcal{A}}, \quad (4)$$

where $N_\mathcal{A}$ is the number of batch size combinations, N_{rsv} is the size of hybrid batches in the reservoir, N_S is the snapshot batch
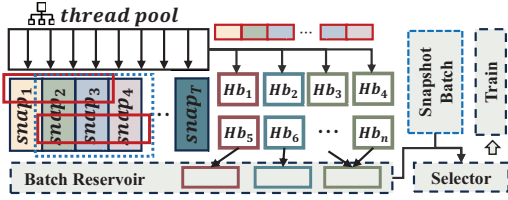
Figure 5: Pipelined Hybrid Batch Generator

size, N_V is the vertex batch size, and N and T represent the total number of vertices and snapshots, respectively. The intuition behind Equation 4 is straightforward: (i) the probability of using the reservoir decreases with smaller batch sizes but increases as the reservoir's batch size grows; and (ii) the more hybrid batches of a given size N_{rsv} there are in the reservoir, the higher the chance of selecting a batch from the reservoir.

The reservoir reduces the time needed to generate hybrid batches, but initially contains a limited number of batches. Consequently, the selector tends to generate more new batches, leading to a bottleneck as it waits for their creation, significantly impeding efficiency. To mitigate the blocking issue, we suggest decoupling the training process and the hybrid batch generator. Both processes run concurrently, with separate threads for each size of the generator. This allows them to independently generate batches. Since generating batches primarily involves reading from the original graph, parallel execution is feasible without conflicts. Locking is only required during the writing process to the reservoir, which involves a simple pointer assignment, resulting in minimal lock overhead.

Algorithm 2: Training process of DynaHB

Input: Dynamic Graph $\mathcal{G}$, Model Update Time T_{mu}
Output: Trained Model *model*

1 **for** i **from** 0 **to** *action_size* **in parallel do**
2   *thread*.start(generate_batch, args=[*hb_size*[i][0], *hb_size*[i][1]])
3 *adap_rl* = DQN()
4 *start_time*=*time*.time()
5 **for** e **from** 0 **to** *epoch* **do**
6   *hybrid_batch* = get_batch(*graph*, *hybrid_batch_size*)
7   **for** *time*, *snapshot* **in enumerate**(*hybrid_batch*) **do**
8   *y_hat*, *hidden_state* = model(*snapshot*, *hidden_state*)
9   *loss* += mean((*y_hat* - *snapshot*.y)**2)
10   *loss* = *loss* / (*time* + 1)
11   *excuting backward, parameter update, clear gradients*
12   *hybrid_batch_size* = *adap_rl*.train(*e*, *loss*, *train_time*)
13   *adap_rl*.distributed_strategy_update(*hybrid_batch_size*)
14   **if** *time*.time() - *start_time* >= T_{mu} **then**
15   *model*.average()

We refine the selector to shorten the impact of hybrid batch generation on training time. When a batch is selected from the reservoir during training, it indicates that enough randomly generated batches are available, ensuring an efficient choice with minimal impact on convergence. However, if a reservoir batch isn't selected, we adopt a different approach. Generating new hybrid batches can be time-consuming and may cause trainer wait time, potentially

---

Algorithm 3: Batch reservoir approach *get_batch*()

Input: Vertex Batch Size N_V, Snapshot Batch Size N_S, Action Space Size $N_{\mathcal{A}}$, Vertex Set Size N, Snapshot Set Size T
Output: Hybrid Batch Hb

1 *action_id*= get_action_id (N_V,N_S)
2 *max_batch_num* =int($N \cdot T$ / ($N_S \cdot N_V$) / $N_{\mathcal{A}}$)
3 N_{rsv} = len(*batch_reservoir*[*action_id*])
4 p = N_{rsv} / max_batch_num
5 **if** *random.rand*() < p **then**
6   *index*=random.randint(0, N_{rsv})
7   **return** *batch_reservoir*[*action_id*][*index*]
8 **else**
9   *snap_id*=random.randint(0, T - N_S + 1)
10   **return** *graph*[*snap_id* : *snap_id* + *snap_size*]

---

exceeding the duration of full-batch training and leading to inefficiency. Therefore, depending on GPU memory availability, if the GPU can accommodate a snapshot batch of the corresponding window size, we generate a new snapshot batch to reduce vertex batch reconstruction costs. Otherwise, we generate a new hybrid batch. Example 4 demonstrates the pipeline-based hybrid batch generator.

EXAMPLE 4. *As shown in Figure 5, threads use a hybrid batch generator that takes the hybrid batch size for the subset it's responsible for, to generate trainable hybrid batches on the dynamic graph. For example, Thread 1 (red) manages to generate hybrid batches of $N_S = 3$ and $N_V = N/3$, and Thread 2 (blue) is responsible for generating hybrid batches of $N_S = 3$ and $N_V = N$. As a result, Thread 1 generates Hb_1 and Hb_5, Thread 2 generates Hb_2 and Hb_6, and Thread 3 generates Hb_3, Hb_4, Hb_n, each storing them in their respective batch reservoirs. During training, the trainer calls the selector to choose from three options: extract a hybrid batch from the batch reservoir, generate a snapshot batch, or generate a new hybrid batch.*

Algorithm 2 describes the training process of DynaHB. Firstly, the batch generator generates each type of hybrid batch (lines 1-2). Before training, hybrid batches are obtained (line 6). Then, processing begins for each snapshot within the hybrid batch, sequentially passing the hidden states to the next snapshot and accumulating the loss function (lines 6-10). After the forward propagation computation is completed, backward propagation, parameter updates, and gradient clearing are executed (line 11). We utilize the current execution time of the hybrid batch, along with the loss reduction, to feed into the reinforcement learning model for training. DQN reward calculation module, which calculates a reward based on the loss and updates the DQN model. After updating its model, the DQN action selection module chooses a new batch size (line 12). Then, all workers update the current strategy followed by obtaining a strategy for the new hybrid batch size (line 13). The Asynchronous Model Controller utilizes a global timer and a model update interval T_{mu}. After each iteration, workers check whether the update time has been reached. (line 14). Once all workers have reached the update interval, their models are averaged and updated (line 15).

Algorithm 3 outlines the specific process of obtaining batches using the batch reservoir. Firstly, based on the size of the hybrid batch, we retrieve the corresponding action ID (line 1). Then, we define a probability threshold p, where the batch selector chooses

to use the hybrid batch with this probability (lines 5-7); otherwise, it selects the snapshot batch training mode (lines 8-10). This probability is defined as the number of hybrid batches of that size already generated in the batch reservoir divided by the total number of hybrid batches in the training dataset (line 4).

Comparison to potential adaptive strategies. (i) *The cost-based adaptive strategy* offers a solution to the trade-off problem posed by batch size selection. However, it cannot capture the benefits in different states. (ii) *The other RL-based adaptive strategy* involves setting the action space as increase, decrease, or remain unchanged. However, It can only continuously increase or decrease the hybrid batch size at predefined intervals.

Generalizability of DynaHB. It efficiently manages snapshots of varying sizes and various tasks. Its communication-free design resolves synchronization bottlenecks. The load-aware partitioning balances training loads, and the hybrid batch adjustment suits different graph scales and training convergence rates.

5.3 Theoretical Analysis

Complexity analysis. We present the time complexity of the training modes, focusing on two key aspects: batch generation cost and computation cost. Sampling target vertices involves re-encoding the edge list, which is time-intensive. BoV (Batch of Vertex) and HB (Hybrid Batch) modes require vertex encoding, with complexities of $O(N_{\mathcal{V}} T \bar{d}^L)$ and $O(N_{\mathcal{V}} N_{\mathcal{S}} \bar{d}^L)$, respectively. Here, $N_{\mathcal{V}}$ and $N_{\mathcal{S}}$ are vertex batch and snapshot batch sizes, T is the number of snapshots, and $\bar{d}$ is the average vertex degree. Note that batch generation complexity exponentially increases with the number of layers. Conversely, BoS batch generation only extracts $N_{\mathcal{S}}$ snapshots. PipeHB, our optimized technique, has a constant cost if drawing from the batch reservoir by a probability p; otherwise, it incurs $N_{\mathcal{S}}$ cost with a probability 1-p, resulting in $O((1-p)N_{\mathcal{S}})$ complexity. Also, we present the theoretical complexity of preparing the hybrid batch reservoir in Equation 5, where $N_{\mathcal{V}}^i$ and $N_{\mathcal{S}}^i$ respectively represent the size of the vertex and snapshot batch for the ith action.

$$O_{rsv} = O\left(\sum_{i \in \mathcal{A}} \frac{(T \cdot N)}{(N_{\mathcal{V}}^i \cdot N_{\mathcal{S}}^i) \cdot |\mathcal{A}|} N_{\mathcal{V}}^i \cdot N_{\mathcal{S}}^i \cdot \bar{d}^L\right) = O(T \cdot N \cdot \bar{d}^L) \quad (5)$$

The computational framework of DGNNs involves GNNs and RNNs. The GNN complexity is attributed to convolution computation and neural network transformations. Each DGNN iteration requires k_g GNN computations, involving L-layer calculations for T snapshots. For each snapshot, GNN complexity includes convolution computation $N \bar{d} \bar{f}$ and neural network transformation $N \bar{f}^2$. Thus, the GNN computational complexity in Full Batch mode is $O(k_g TL(N \bar{d} \bar{f} + N \bar{f}^2))$. The main RNN cost lies in neural network transformations, accounting for $k_r TL(N \bar{f}^2)$. Given k_g and k_r are usually comparable, we unify them as k. Hence, the simplified DGNN computational complexity in Full Batch mode is $O(kTLN(\bar{d}\bar{f} + \bar{f}^2))$.

BoV utilizes vertex batches, reducing N to $N_{\mathcal{V}}$; BoS uses snapshot batches, reducing T to $N_{\mathcal{S}}$; HB employs a hybrid batch, reducing both vertex and snapshot sizes to $N_{\mathcal{V}}$ and $N_{\mathcal{S}}$ respectively. PipeHB, with a probability 1-p of choosing BoS and p of choosing HB, incurs a cost of $O((1-p)C_{BoS} + pC_{HB})$, where C_{BoS} is the GNN cost for BoS, and C_{HB} is the cost for the hybrid batch.

The main cost components are the storage of features and edge lists. The host and GPU memory costs of communication-based frameworks are $O(TN\bar{f} + T\bar{e})$, where $\bar{e}$ is the average number of edges per snapshot. Frameworks caching L-hop neighbors cost $O(TN\bar{d}^L \bar{f})$ on CPU and $O(T_{\mathcal{B}} N_{\mathcal{B}} \bar{d}^L \bar{f})$ on GPU. BoS: $T_{\mathcal{B}} = N_{\mathcal{S}}$, $N_{\mathcal{B}} = N$; BoV: $T_{\mathcal{B}} = T$, $N_{\mathcal{B}} = N_{\mathcal{V}}$; DynaHB: $T_{\mathcal{B}} = N_{\mathcal{S}}$, $N_{\mathcal{B}} = N_{\mathcal{V}}$.

Additionally, we encapsulate the system optimizations and default parameters, eliminating the need for additional costs associated with implementation or parameter tuning.

Convergence analysis. We provide some definitions and symbols to aid in the analysis of convergence.

DEFINITION 7. *The gradients of hybrid batch training are denote as $\nabla g(x) = \nabla f(x) - \xi$, where $\nabla f(x)$ is the gradient under snapshot batch mode, and ξ is the error. The expected value $\mathbb{E}[\xi] = 0$ and variance $\mathbb{E}||\nabla f(x) - \nabla g(x)||^2 = \mathbb{E}||\xi||^2$*

In order to demonstrate the convergence of our update scheme, we make the following assumptions:

- Lipschitzian gradient: $f(x)$ is L-Lipschitz smooth, i.e., $||\nabla f(x) - \nabla f(y)|| \leq L||x - y||, \forall x, y \in \mathbb{R}^d$.
- Bounded variance: the variance of the stochastic gradient, denoted by ξ, is bounded such that $\mathbb{E}[||\xi||^2] \leq \sigma^2$.

DEFINITION 8. *The gradients of adaptive hybrid batch updating scheme are given by: $\nabla g(x) = p \cdot \nabla g_{hb}(x) + (1 - p)\nabla f(x)$, i.e., $\nabla g(x) = p \cdot (\nabla f(x) - \xi) + (1 - p)\nabla f(x) = \nabla f(x) - p \cdot \xi$. The parameter update equation at the tth iteration can be represented as $x_{t+1} = x_t - \gamma \nabla g(x_t)$.*

LEMMA 1. *Under the aforementioned assumptions, we can deduce the convergence rate $\frac{1}{T} \sum_{t=0}^{T-1} \mathbb{E}||\nabla f(x_t)||^2 \leq 2(\mathbb{E}f(x_0) - \mathbb{E}f(x_t))/(T(2\gamma - L\gamma^2)) + L\gamma^2 p^2 \sigma^2/(2\gamma - L\gamma^2)$.*

LEMMA 2. *With the asynchronous model controller, when we set $\gamma = \sqrt{(f(\tilde{x}_1) - f^*)M/(K^2 L(\theta + \check{p}^2 \sigma^2))}/\sqrt{T}$, the convergence formula of our update scheme is as follows, and it can achieve the convergence of $O(1/\sqrt{T})$, where $\check{p}$ represents the maximum probability.*

$$\frac{1}{T}\mathbb{E}\sum_{j=1}^{T} ||\nabla f(\tilde{x}_j)||_2^2 \leq \sqrt{\frac{(f(\tilde{x}_1) - f^*)L(\theta + \check{p}^2 \sigma^2)}{M}} * \frac{4K}{(K - 1 + \delta)\sqrt{T}} \quad (6)$$

In fact, a higher convergence rate does not guarantee the minimum overall convergence time. In practice, to balance the execution time of individual batches and the convergence rate, we adopt the probability formula given in Equation 4. The detailed proof of Lemmas 1 and 2 is available online[1].

6 EXPERIMENTS

6.1 Experimental Setup

DGNN models. We use three popular models: Temporal Graph Convolutional Network (T-GCN [55]), Diffusion Convolutional Recurrent Neural Network (DCRNN [22]), and Evolve Graph Convolutional Network (EvolveGCN [27]). T-GCN is designed for traffic prediction, combining GCN for spatial information convolution and GRU for temporal information learning. DCRNN utilizes bidirectional random walks on the graph to capture spatial dependencies
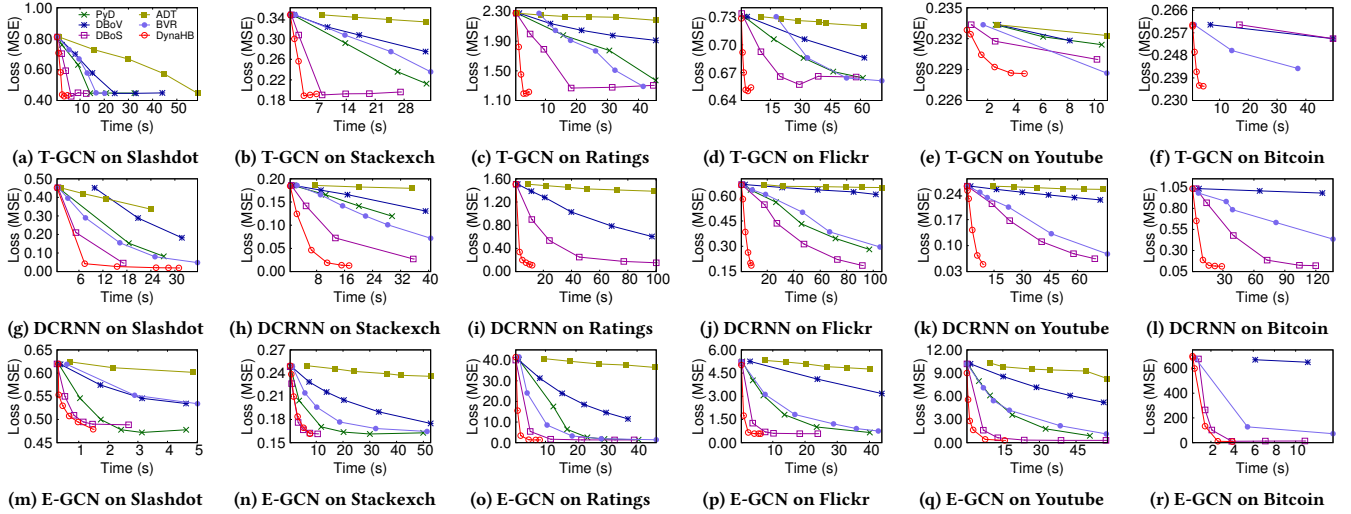
---
[1]https://github.com/songzhen-neu/DynaHB/Proof.pdf

(a) T-GCN on Slashdot **(b) T-GCN on Stackexch** **(c) T-GCN on Ratings** **(d) T-GCN on Flickr** **(e) T-GCN on Youtube** **(f) T-GCN on Bitcoin**

(g) DCRNN on Slashdot **(h) DCRNN on Stackexch** **(i) DCRNN on Ratings** **(j) DCRNN on Flickr** **(k) DCRNN on Youtube** **(l) DCRNN on Bitcoin**

(m) E-GCN on Slashdot **(n) E-GCN on Stackexch** **(o) E-GCN on Ratings** **(p) E-GCN on Flickr** **(q) E-GCN on Youtube** **(r) E-GCN on Bitcoin**

Figure 6: Convergence comparison

Table 1: Datasets (M=million and B=billion)

Attributes	$|S|$	$|V|$	$|E|$	$|TV|$	$|EL|$	$|F|$	Task
Cora-T	50	0.05M	0.27M	2.71M	1.30M	1433	CL
Pubmed-T	50	0.39M	1.77M	19.72M	8.86M	500	CL
Slashdot	80	0.05M	0.14M	4.088M	3.69M	2	RG
Stackexch	170	0.55M	1.30M	92.68M	33.6M	2	RG
Ratings	70	2.15M	5.84M	150.2M	125M	2	RG
Flickr	70	2.30M	33.1M	161.2M	263M	2	RG
Youtube	80	3.22M	12.2M	257.9M	332M	2	RG
Bitcoin	90	24.6M	122M	2.211B	1.05B	2	RG

and an encoder-decoder architecture to capture temporal dependencies. EvolveGCN (E-GCN) uses GCN for spatial information convolution and an RNN to evolve GCN parameters.

Datasets. We use eight public datasets. The first two are for node classification (CL) tasks, extended from static graphs using techniques [9]; the remaining six [32] are for regression (RL) tasks, representing dynamic graphs and available on the website[2]. The statistics of the datasets are summarized in Table 1, where $|S|$, $|V|$, $|E|$, $|F|$, and $|TV|$ represent the number of snapshots, vertices, edges, feature dimension and total vertices, respectively. We employ a graph smoothing technique named edge-life [3, 27, 43], which can densify the dynamic graph and ensure snapshot continuity. Following the common practice [3, 5, 24], we treat in-degrees and out-degrees as features of each vertex for the latter six datasets.

Baselines. We compare DynaHB with five state-of-the-art distributed DGNN systems with different training modes.

- **PyGT-Dist (PyD)** [33] is a distributed implementation of PyGT based on the cache-based vertex partitioning architecture, utilizing full-batch mode for training.
- **Dist-Batch-of-Vertex (DBoV)** [57] is a distributed architecture that trains using complete snapshots and mini-batches of vertices, which is extended from GNN training.

- **Dist-Batch-of-Snapshot (DBoS)** [3] is a distributed architecture that trains using full vertices and a sliding window, which is the state-of-the-art DGNN training mode.
- **DBoV-Reservoir (BVR)** employs reservoir technology with a pipeline based batch generator, based on the vertex batch mode.
- **ADGNN-Temporal (ADT)** [35] is a distributed architecture based on vertex partitioning, involving communication during training but avoiding redundant computations.

Hybrid partition (DGC [5]), snapshot-partition (ESDG [3]), and sliding-window-based snapshot-partition (DynaGraph [9]) are not open-source systems with architectures distinct from ours. Although designed to reduce communication, they still rely on synchronous operations, which can be less efficient than DynaHB. In particular, the size of the sliding window–bound by the local snapshot count–is restricted, and as the number of workers increases, the restrictions become tighter, resulting in poor scalability.

Environments and parameter settings. We conduct experiments on two environments: **E1:** a cluster consisting of five machines employing 10Gbps Ethernet connections, each equipped with 250 GB DRAM, an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 32 cores, and an NVIDIA RTX 2080Ti with 11GB memory; **E2:** equipped with 2 Intel(R) Xeon(R) Gold 6326 CPUs @ 2.90GHz with 755GB DRAM, and configured with 8 NVIDIA RTX A6000 GPUs, each with 48GB of GPU memory. For all tests, excluding ablation study, we process the data in batches—5% of vertices and 20% of snapshots per batch. We vary hybrid batch sizes between 1%, 5%, 10%, and 20% to evaluate performance in ablation study. The model is considered to have converged if its loss does not decrease for ten consecutive epochs. The train/test set ratio is set to 0.4/0.6.

6.2 Convergence and Speedup Comparison

We perform the convergence comparison results as shown in Figure 6 with the Mean Squared Error (MSE) loss over time.

Clearly, DynaHB achieves the fastest convergence and the lowest final loss values. For example in Figure 6c, DynaHB converges to a

Table 2: Test loss and convergence time (sec) comparison

System	Model	Slashdot				Stackexch				Ratings				Flickr				Youtube				Bitcoin			
		ST	#CN	TT	TL	ST	#CN	TT	TL	ST	#CN	TT	TL	ST	#CN	TT	TL	ST	#CN	TT	TL	ST	#CN	TT	TL
PyD	T-GCN	0.174	90	15.7	0.436	0.426	80	48.1	0.191	0.699	120	59.9	1.278	1.487	170	252	0.655	2.212	200	442	0.228			OOM*	
ADT		0.725	90	72.5	0.436	7.304	80	584	0.191	8.037	120	964	1.278	9.926	170	1687	0.655	11.67	200	2335	0.228			OOM*	
DBoV		0.244	90	21.9	0.436	0.804	80	64.3	0.192	0.986	110	108	1.288	2.784	170	473	0.655	1.450	190	275	0.228	6.028	100	602	0.235
DBoS		0.062	110	_6.87_	0.427	0.101	80	_8.04_	0.191	0.150	150	_22.5_	1.254	0.320	300	_96.1_	0.657	0.416	150	_62.4_	0.228	3.381	100	_338_	0.235
BVR		0.207	100	20.6	0.446	0.546	80	43.6	0.191	0.344	120	41.3	1.289	0.743	160	118	0.655	0.538	20	10.7	0.228	1.049	100	104	0.235
DynaHB		0.019	97	**1.84**	0.427	0.036	90	**3.21**	0.188	0.016	201	**3.22**	1.190	0.016	252	**4.06**	0.650	0.034	140	**4.76**	0.229	0.027	133	**3.61**	0.235
PyD	DCRNN	0.123	1800	220	0.019	0.491	500	246	0.013			OOM*		0.321	500	161	0.187			OOM*				OOM*	
ADT		0.603	1800	1084	0.019	6.97	500	3483	0.013	9.04	990	8947	0.100	8.933	500	4466	0.187	10.83	460	4981	0.049			OOM*	
DBoV		0.187	1530	286	0.021	0.720	520	374	0.013	0.954	990	945	0.100	2.723	510	1388	0.187	1.398	460	643	0.049	5.775	701	4210	0.118
DBoS		0.042	1580	_65.8_	0.019	0.164	500	_81.9_	0.013	0.178	1070	_190_	0.100	0.158	580	_91.6_	0.186	0.192	490	_93.8_	0.050	0.142	921	_135_	0.118
BVR		0.108	2140	232	0.019	0.353	520	183	0.013	0.219	1000	218	0.100	0.343	520	178	0.186	0.286	480	137	0.049	0.591	700	413	0.118
DynaHB		0.011	2993	**32.7**	0.019	0.026	675	**17.3**	0.013	0.010	1429	**13.8**	0.100	0.010	823	**7.97**	0.186	0.014	659	**9.08**	0.049	0.015	1348	**20.0**	0.118
PyD	E-GCN	0.077	40	3.06	0.472	0.166	200	33.2	0.161	0.121	980	118	1.319	0.330	240	79.2	0.578	0.454	1080	491	0.294			OOM*	
ADT		0.575	40	22.2	0.472	5.154	200	1028	0.161	7.125	980	6983	1.319	7.254	240	1741	0.578	8.051	1080	8694	0.294			OOM*	
DBoV		0.145	110	16.0	0.496	0.664	200	133	0.161	0.723	640	462	1.318	2.029	300	608	0.579	1.380	970	1339	0.270	3.452	300	1036	8.462
DBoS		0.034	60	_2.05_	0.480	0.050	210	_10.5_	0.161	0.040	890	_35.2_	1.319	0.043	460	_19.9_	0.580	0.084	580	_48.8_	0.294	0.076	60	_4.54_	9.492
BVR		0.109	160	17.46	0.499	0.232	220	51.2	0.164	0.161	441	72.7	1.319	0.342	220	75.2	0.580	0.268	620	166	0.458	0.501	320	160	24.71
DynaHB		0.012	124	**1.48**	0.479	0.022	351	**7.60**	0.162	0.008	943	**7.10**	1.319	0.008	817	**6.22**	0.579	0.010	1516	**15.8**	0.293	0.015	208	**3.04**	7.559

loss of 1.190 at 3.22s with T-GCN on Ratings, demonstrating the fastest convergence, while DBoS, the state-of-the-art training mode, converges to a loss of 1.254 at 22.5s. The superiority of DynaHB stems from the following reasons. First, DynaHB leverages hybrid batches, which overcomes the limitation of a single snapshot size, resulting in smaller loads. Second, the asynchronous mode, along with other optimizations, further enhances efficiency.
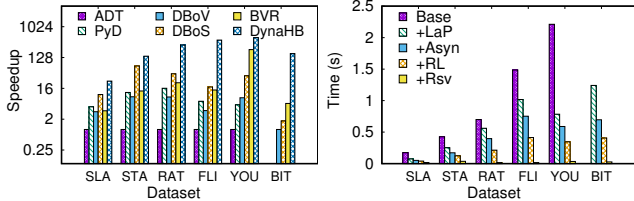

Figure 7: Speedup comparison **Figure 8: Ablation study**

Next, in higher-load DGNN model training, such as T-GCN and DCRNN, DynaHB exhibits greater performance improvement compared to DBoS in terms of acceleration. This is primarily due to the more complex computation processes of DGNNs, leading to a significantly increased load on individual vertices. In this case, the use of vertex batches effectively reduces the overall training time.

Table 2 shows iteration time (ST), convergence rounds (#CN), total convergence time (TT), and test loss (TL) for all systems using three models on six datasets. "OOM*" indicates memory overflows, while bold and underlined values indicate the best and second-best convergence efficiency, respectively. DynaHB stands out for its rapid convergence on all datasets and achieves comparable loss to full-batch methods. For example, with T-GCN on Flickr, DynaHB's loss of 0.650 is the lowest, outperforming all baselines and even full-batch methods like PyD and ADT. DynaHB introduces variability in the number of snapshots, which can help the model escape local optima, in turn mitigating overfitting.

In most cases, DynaHB requires more convergence rounds compared to baselines. For instance, when running T-GCN on Flickr, DynaHB converges in 252 rounds, while PyD and ADT only require 170 rounds. This is due to the increased randomness in gradients introduced by the hybrid batch mode. Similarly, the vertex batch system DBoV, BVR, and the snapshot batch system DBoS generally have more convergence rounds compared to the full-batch PyD and ADT. However, DynaHB's quick iterations (small ST) ensure it still outperforms baselines in total convergence time.

Finally, under our settings (with snapshot batches set at 20% and vertex batches at 5%), the batch distribution remains relatively stable, which minimizes the effects of random fluctuations in gradients. Consequently, almost all baselines ultimately converge to values close to or identical with each other. As shown in Table 2, across all models and datasets, the MSE loss for all batch techniques relative to the full-batch mode does not exceed 5.085% (e.g., on Slashdot, DBoV has a loss of 0.496, while the full-batch loss is 0.472 for E-GCN). In most cases, DynaHB is superior or equal to the full-batch mode in terms of loss convergence.

Due to space limitation, we use T-GCN to illustrate the speedups of all baselines in Figure 7 against ADT, with similar trends observed in other models: DCRNN and E-GCN. As observed, PyD achieves a speedup of 4.62-16.09×, DBoV achieves 1.78-8.93×, BVR achieves 3.52-218.2×, DBoS achieves 9.09-42.84×, and DynaHB significantly leads with 26.08-490.55×. Moreover, for Bitcoin, where ADT and PyD face GPU memory issues, we base speedup calculations on DBoV's performance. DBoS and BVR achieves 1.78× and 5.78× speedups on the Bitcoin dataset compared to DBoV, while DynaHB achieves a 166.76× speedup. In summary, DynaHB outperforms the state-of-the-art system, DBoS, by 2.47-93.63× in speedup.

6.3 Breakdown and Memory Use

We provide a breakdown of convergence time for T-GCN on Flickr in Table 3. This breakdown includes constructing reservoir time

Table 3: Breakdowns of total convergence time (sec)

System	rsv	gen_bat	to_gpu	comp	comm	upda	rl
PyD	N/A	N/A	0.115	243.3	N/A	0.542	N/A
ADT	N/A	N/A	0.303	53.33	1517	0.522	N/A
DBoV	N/A	443.1	3.445	25.29	N/A	0.489	N/A
DBoS	N/A	0.003	0.060	14.63	N/A	0.803	N/A
BVR	5.271	0.000	1.233	40.27	N/A	0.321	N/A
DynaHB	8.158	0.000	0.115	3.352	N/A	0.554	0.522

(a) Partition time   (b) Scalability

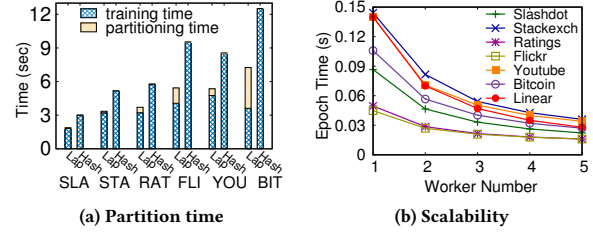**Figure 9: Partition time and scalability**

(*rsv*), batch generation time (*gen_bat*), data transfer to GPU time (*to_gpu*), computation time (*comp*), communication time (*comm*), parameter aggregation and update time (*upda*), and reinforcement learning time (*rl*). N/A indicates that the comparison system does not require that step. Meanwhile, 0.000 denotes that the time is less than one-thousandth of a second and is thus negligible.

As shown in Table 3, the primary time consumption of PyD lies in computation since, under its architecture, data does not require communication. Conversely, the ADT system faces a significant bottleneck in communication, reaching up to 30 times the computation time. This is primarily due to (i) the significantly higher GPU computational performance compared to network transfer performance, (ii) the existence of multiple snapshots in DGNNs, where each snapshot involves communication and synchronization, incurring high costs, and (iii) in addition to the actual communication data, some communication preparation processes such as data serialization, deserialization, and message construction are also costly. ADT is more suitable for scenarios with a small number of snapshots and a high load per individual snapshot graph. Additionally, we observe that the computation time for ADT is lower than PyD, primarily due to PyD's adoption of a caching mechanism involving redundant calculations.

The primary bottleneck for the DBoV system lies in the generation of vertex batches. Each vertex batch necessitates resampling of target vertices, which entails re-encoding and updating vertex mappings in the edge list. This process is time-intensive and can surpass computation time. BVR addresses the bottleneck by incorporating our pipeline reservoir technique. Conversely, DBoS operates at the snapshot graph level, where each snapshot graph retains its initial mapping, eliminating the need for vertex re-encoding and mapping updates. During batch generation, DBoS merely requires specifying the initial snapshot ID and retrieving the necessary number of snapshots for the snapshot batch, incurring low time cost.

DynaHB mitigates bottlenecks by eliminating the need for communication, and minimizes computation time through its hybrid batch techniques. The reservoir stops when training ends, thus completely covered by online time. Also, it can overlap with testing intervals during training. With no communication overhead, minimal computational load, and negligible batch generation time, DynaHB achieves optimal performance.

Table 7 reports the average host memory and GPU memory use of DynaHB and baselines. In terms of host memory, ADT requires the least memory since it does not involve caching. The host memory use of other frameworks is similar. However, in our cluster, each machine has 250GB of host memory, which is far above the upper limit of required host memory. In terms of GPU memory use, DynaHB has the smallest memory footprint among all systems.

6.4 Evaluation of Scalability

We evaluate the scalability, considering layer-wise, machine-wise, graph-type-wise, and device-wise expansion. Initially, we validate the sustained optimal performance of our system, ensuring it maintains speedups similar to those observed in single-layer approximations, even with multi-layer DGNNs. In Table 4, DynaHB remains the top-performing system across 2-layer and 3-layer DGNN configurations, showing negligible declines in speedups with increasing layer count. For example, on Bitcoin, DynaHB achieves speedups over DBoS of 2.77×, 3.71×, and 3.38× for the first layer, two layers, and three layers of the T-GCN model, respectively.

In addition, we assess the scalability of our system concerning the number of workers. We present the experimental results of DynaHB utilizing one to five machines across six datasets in Figure 9b. Furthermore, we include a reference line for linear speedup. Notably, DynaHB achieves nearly linear speedups across all datasets. This is primarily attributed to the independence of hybrid batches within the asynchronous architecture, effectively eliminating communication overhead.

Scalability of other graphs and devices. As shown in Table 5, DynaHB achieves similar accuracies to the PyD and ADT under the full-batch mode, with losses of 0.11% and 0.05%, respectively, but achieves acceleration ratios of 2.56× and 3.26×. Compared to the fastest DBoS in the baselines, we achieve acceleration ratios of 1.56× and 2.24×. Therefore, DynaHB remains effective at node classification tasks on high-dimensional attribute datasets. Additionally, we conduct further scalability experiments on a new device in Table 6. In the environment with 8 A6000 GPUs, DynaHB still achieves the best training efficiency, with acceleration ratios of 1.74-3.81× compared to the DBoS mode.

We conduct new experiments to test the speedup of DynaHB compared to other systems on different graph sizes. Using the Stackexch as an example, we set the training dataset to 20%, 40%, 60%, 80%, and 100% of the full dataset size. As shown in Table 9, DynaHB achieves the fastest convergence time across all dataset proportions. Compared to DBoS, DynaHB achieves speedup ratios of 2.22×, 2.24×, 3.25×, 3.23×, and 2.50×, respectively, for the different dataset sizes. Furthermore, although smaller datasets can reduce the convergence time per iteration, due to changes in convergence characteristics, smaller datasets do not necessarily lead to faster overall convergence.

6.5 Ablation Study

In Figure 8, we present the findings from ablation experiments conducted using T-GCN. The term "Base" denotes the baseline performance obtained without any optimizations, employing a

Table 4: Layer scalability test (sec)

System	Slashdot			Stackexch			Ratings			Flickr			Youtube			Bitcoin		
	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
PyD	0.11	0.19	0.28	0.24	0.45	0.71	0.18	0.50	0.82	0.13	0.46	0.85	0.19	0.40	0.81	0.45	0.93	OOM*
ADT	0.27	3.32	6.28	3.24	16.4	29.2	2.83	12.0	20.7	2.59	10.2	17.6	3.82	17.8	31.4	3.04	70.8	139
DBoV	0.16	0.30	0.51	0.47	1.05	2.69	0.54	2.15	3.50	1.71	8.89	23.42	0.80	3.77	8.67	2.06	3.65	5.45
DBoS	0.04	0.05	0.07	0.06	0.10	0.15	0.04	0.10	0.17	0.03	0.12	0.20	0.03	0.09	0.15	0.10	0.37	0.55
BVR	0.14	0.21	0.31	0.34	0.55	0.88	0.14	0.42	0.64	0.41	1.32	2.64	0.37	2.18	5.40	0.36	0.48	0.67
Ours	0.02	0.03	0.04	0.03	0.04	0.05	0.01	0.02	0.03	0.02	0.06	0.13	0.02	0.04	0.09	0.04	0.10	0.16

Table 5: Test on classification tasks

System	Cora-T		Pubmed-T	
	Acc	Time	Acc	Time
PyD	74.14%	46.6	70.81%	140
ADT	74.14%	914	70.81%	1548
DBoV	74.14%	169	70.82%	768
DBoS	73.55%	28.3	69.23%	96.0
BVR	72.19%	33.7	70.62%	264
Ours	74.03%	18.2	70.76%	42.8

Table 6: Convergence time (sec) on 8 A6000 GPUs

Dataset	PyD	ADT	DBoV	DBoS	BVR	Ours
Slashdot	11.11	24.40	14.04	3.84	12.85	2.16
Stackexch	18.84	262.22	37.71	4.89	26.85	2.82
Ratings	21.78	342.86	65.32	4.28	16.63	1.62
Flickr	13.08	442.93	171.13	6.72	41.34	1.76
Youtube	11.15	769.38	47.99	5.46	22.21	1.67
Bitcoin	35.80	246.27	165.10	8.32	29.17	3.34

Table 10: Hyperparameter test for DQN models

Strategy	#1	#2	#3	#4	#5	#6	Ours	Random
ϵ_{decay}	0.999	0.75	0.95	0.95	0.95	0.95	0.95	N/A
ϵ_{min}	0.1	0.1	0	0.5	0.1	0.1	0.1	N/A
lr	0.005	0.005	0.005	0.005	0.1	0.001	0.005	N/A
Time (s)	4.57	4.52	4.99	4.46	4.49	4.18	3.21	5.82

Table 7: Average memory use (GB)

System	Host	GPU
PyD	40.88	9.95
ADT	18.58	5.45
DBoV	40.63	0.97
DBoS	40.04	3.05
BVR	40.34	1.71
DynaHB	40.25	0.47

Table 8: Convergence time (sec) with different fanouts

Dataset	1%	5%	10%	20%	Ours
Slashdot	2.47	2.49	2.51	2.72	**1.84**
Stackexch	3.62	5.13	10.1	25.1	**3.21**
Ratings	13.7	22.0	40.4	82.5	**3.22**
Flickr	43.5	94.6	197	424	**4.06**
Youtube	13.7	28.9	50.9	135	**4.76**
Bitcoin	13.5	33.6	88.9	309	**3.61**

Table 9: Scalability w.r.t different graph sizes (sec)

Size	PyD	ADT	DBoV	DBoS	BVR	Ours
20%	10.45	145.95	23.83	5.72	17.04	**2.58**
40%	42.10	361.80	51.90	6.19	54.83	**2.76**
60%	63.35	544.91	131.17	11.69	77.17	**3.60**
80%	102.42	574.27	143.61	12.58	109.61	**3.90**
100%	48.10	584.00	64.30	8.04	43.60	**3.21**

We provide experimental results for the overall convergence time with different sizes of hybrid batches, i.e., 1%, 5%, 10%, and 20%. In this configuration, hybrid batches ultimately converge to MSE loss close to those of the full batch. As seen in Table 8, bold values represent the best performance. DynaHB achieves better efficiency across all sizes of hybrid batches. For instance, on Ratings, DynaHB achieves speedups of 4.25×, 6.83×, 12.55×, and 25.62× for hybrid batch sizes of 1%, 5%, 10%, and 20%, respectively. The average speedup ratios for hybrid batch sizes of 1%, 5%, 10%, and 20% are 4.01, 8.08, 16.82, and 42.22, respectively.

Hyperparameter sensitivity test for DQN. We evaluate the hyperparameter sensitivity of the DQN model, as shown in Table 10, using six different settings and a random action selection strategy for comparison. The DQN model accelerates convergence compared to random selection. However, setting parameters excessively high or low results in inferior performance compared to our default values. With our default parameters, we achieve significant acceleration effects, eliminating the need for manual tuning.

7 CONCLUSION

We present a cache-based architecture tailored for distributed DGNN training, aiming to fully exploit CPU memory while eliminating communication overhead. To further enhance the efficiency of the cache-based architecture and mitigate GPU memory constraints, we introduce a novel hybrid batch training mode. We propose a suite of optimizations for this hybrid batch mode, encompassing RL-based batch size adjustment, a load-aware balanced hybrid-batch partition strategy and training modes, and a reservoir technique founded on pipeline-generated hybrid batches. Our extensive experiments demonstrate that DynaHB achieves a remarkable speedup of up to 93× and an average of 8.06× speedups compared to the state-of-the-art training mode.

full-batch execution mode. "+Lap" signifies the incorporation of a partitioning methodology based on hybrid-batch load balancing atop the base configuration. Analogously, "+Asyn" denotes the adoption of an asynchronous strategy, while "+RL" involves the utilization of reinforcement learning to dynamically adjust the size of the hybrid batch. Lastly, "+Rsv" integrates a reservoir technique centered on pipeline batch generation. In Figure 8, each column signifies the inclusion of the specified technique along with all the techniques mentioned in the preceding columns.

Using YouTube as a case study, we observe that the Lap, Asyn, and RL methods yield individual speedups of 2.82×, 1.33×, and 1.70×, respectively, illustrating the efficacy of each technique. Furthermore, the combination of these three techniques results in a speedup of 6.39×. However, with the enhancement of the system's training performance, the overhead associated with hybrid-batch generation emerges as a bottleneck. Consequently, with the introduction of the reservoir technique grounded in pipeline batch generation, DynaHB achieves a further speedup of 10.18×.

REFERENCES

[1] Jiandong Bai, Jiawei Zhu, Yujiao Song, Ling Zhao, Zhixiang Hou, Ronghua Du, and Haifeng Li. 2021. A3T-GCN: Attention Temporal Graph Convolutional Network for Traffic Forecasting. *ISPRS Int. J. Geo Inf.* 10, 7, 485.

[2] Kaidi Cao, Rui Deng, Shirley Wu, Edward W Huang, Karthik Subbian, and Jure Leskovec. 2023. Communication-Free Distributed GNN Training with Vertex Cut. *CoRR* abs/2308.03209 (2023).

[3] 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.

[4] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. 2023. NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams. *Proc. VLDB Endow.* 17, 3, 455–468.

[5] Fahao Chen, Peng Li, and Celimuge Wu. 2023. DGC: Training Dynamic Graphs with Spatio-Temporal Non-Uniformity using Graph Partitioning by Chunks. *Proc. ACM Manag. Data* 1, 4 (2023), 237:1–237:25.

[6] Jinyin Chen, Xueke Wang, and Xuanheng Xu. 2022. GC-LSTM: graph convolution embedded LSTM for dynamic network link prediction. *Appl. Intell.* 52, 7 (2022), 7513–7528.

[7] Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. 2023. BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 37:1–37:13.

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

[9] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *GRADES-NDA*. 6:1–6:10.

[10] Danlei Hu, Lu Chen, Hanxi Fang, Ziquan Fang, Tianyi Li, and Yunjun Gao. 2024. Spatio-temporal trajectory similarity measures: A comprehensive survey and quantitative study. *IEEE Trans. Knowl. Data Eng.* 36, 5 (2024), 2191–2212.

[11] Ajay Kumar Jaiswal, Shiwei Liu, Tianlong Chen, Ying Ding, and Zhangyang Wang. 2023. Graph ladling: Shockingly simple parallel gnn training without intermediate communication. In *International Conference on Machine Learning*. 14679–14690.

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

[13] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *SIGKDD*. 1269–1278.

[14] Haoyang Li and Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs. In *CIKM*. 937–946.

[15] Haoyang Li and Lei Chen. 2023. EARLY: Efficient and Reliable Graph Neural Network for Dynamic Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 163:1–163:28.

[16] Jia Li, Zhichao Han, Hong Cheng, Jiao Su, Pengyun Wang, Jianfeng Zhang, and Lujia Pan. 2019. Predicting Path Failure In Time-Evolving Graphs. In *SIGKDD*. 1279–1289.

[17] Tianyi Li, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2021. TRACE: Real-time compression of streaming trajectories in road networks. *Proc. VLDB Endow.* 14, 7 (2021), 1175–1187.

[18] Tianyi Li, Lu Chen, Christian S Jensen, Torben Bach Pedersen, Yunjun Gao, and Jilin Hu. 2022. Evolutionary clustering of moving objects. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2399–2411.

[19] Tianyi Li, Ruikai Huang, Lu Chen, Christian S Jensen, and Torben Bach Pedersen. 2020. Compression of uncertain trajectories in road networks. *Proc. VLDB Endow.* 13, 7 (2020), 1050–1063.

[20] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Orca: Scalable Temporal Graph Neural Network Training with Theoretical Guarantees. *Proc. ACM Manag. Data* 1, 1 (2023), 52:1–52:27.

[21] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When temporal graph neural networks meet temporal personalized pagerank. *Proc. VLDB Endow.* 16, 6 (2023), 1332–1345.

[22] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *ICLR*.

[23] 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.

[24] Osman Asif Malik, Shashanka Ubaru, Lior Horesh, Misha E Kilmer, and Haim Avron. 2021. Dynamic graph convolutional networks using the tensor M-product. In *Proceedings of the 2021 SIAM international conference on data mining (SDM)*. 729–737.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[26] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2021. Transfer Graph Neural Networks for Pandemic Forecasting. In *AAAI*. 4838–4845.

[27] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *AAAI*. 5363–5370.

[28] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching. *Proc. VLDB Endow.* 15, 11 (2022), 2626–2639.

[29] 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. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.

[30] Xiao Qin, Nasrullah Sheikh, Chuan Lei, Berthold Reinwald, and Giacomo Domeniconi. 2023. SEIGN: A Simple and Efficient Graph Neural Network for Large Dynamic Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2850–2863.

[31] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637* (2020).

[32] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. 4292–4293.

[33] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Sinziana Astefanoaei, Oliver Kiss, Ferenc Béres, Guzmán López, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *CIKM*. 4564–4573.

[34] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. 2018. Structured Sequence Modeling with Graph Convolutional Recurrent Networks. In *ICONIP*, Vol. 11301. 362–373.

[35] Zhen Song, Yu Gu, Tianyi Li, Qing Sun, Yanfeng Zhang, Christian S. Jensen, and Ge Yu. 2023. ADGNN: Towards Scalable GNN Training with Aggregation-Difference Aware Sampling. *Proc. ACM Manag. Data* 1, 4 (2023), 229:1–229:26.

[36] Zhen Song, Yu Gu, Jianzhong Qi, Zhigang Wang, and Ge Yu. 2022. EC-Graph: A Distributed Graph Neural Network System with Error-Compensated Compression. In *ICDE 2022*. 648–660.

[37] Aynaz Taheri and Tanya Y. Berger-Wolf. 2019. Predictive temporal embedding of dynamic graphs. In *ASONAM*. 57–64.

[38] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 495–514.

[39] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.

[40] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. 2022. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems* 4 (2022), 673–693.

[41] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428* (2022).

[42] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23.

[43] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. In *PPoPP*. 405–418.

[44] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: a flexible and efficient distributed framework for GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 67–82.

[45] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. Neutronstar: distributed GNN training with hybrid dependency management. In *Proceedings of the 2022 International Conference on Management of Data*. 1301–1315.

[46] 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.

[47] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via causal anonymous walks. *arXiv preprint arXiv:2101.05974* (2021).

[48] Yufeng Wang and Charith Mendis. 2023. TGOpt: Redundancy-Aware Optimizations for Temporal Graph Attention Networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 354–368.

[49] Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2023. Redundancy-free high-performance dynamic GNN training with hierarchical pipeline parallelism. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 17–30.

[50] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *ICLR*.

[51] Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. 2021. Performance-adaptive sampling strategy towards fast and accurate graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2046–2056.

[52] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2358–2366.

[53] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. 2023. Towards Better Dynamic Graph Learning: New Architecture and Unified Library. *Advances in Neural Information Processing Systems* (2023).

[54] 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)*. 723–732.

[55] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2020. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Trans. Intell. Transp. Syst.* 21, 9 (2020), 3848–3858.

[56] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. GMAN: A Graph Multi-Attention Network for Traffic Prediction. In *AAAI*. 1234–1241.

[57] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 36–44.

[58] Yuchen Zhong, Guangming Sheng, Tianzuo Qin, Minjie Wang, Quan Gan, and Chuan Wu. 2023. GNNFlow: A Distributed Framework for Continuous Temporal GNN Learning on Dynamic Graphs. *arXiv preprint arXiv:2311.17410*.

[59] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor K. Prasanna. 2023. DistTGL: Distributed Memory-Based Temporal Graph Neural Network Training. In *SC*. 39:1–39:12.

[60] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12 (2019), 2094–2105.