PRISM: A Training System to Unlock the Potential of Temporal Graph Learning Through Staleness Avoidance

Md Ashraful Islam, Hojae Son, Suhaas Kiran, and Marco Serafini
University of Massachusetts Amherst
Amherst, MA, USA
{mdashrafulis,hojaeson,sdg,mserafini}@umass.edu

ABSTRACT

Training memory-augmented Temporal Graph Neural Networks (M-TGNNs) efficiently and accurately remains challenging due to *memory staleness*, which arises when temporally dependent events are processed in the same batch and severely degrades accuracy at large batch sizes. We introduce PRISM, an M-TGNN training system that achieves staleness-freedom without giving up GPU parallelism by using *multi-versioned memory vectors*, so that each event in a batch can consume the memory version that is temporally consistent for it. PRISM formalizes a relaxed notion of staleness-freedom called *lazy freshness*, which allows for more parallelism than existing staleness-free approaches, and implements it through a *multi-versioned memory refinement algorithm* over a lightweight memory computation graph. On five temporal-graph benchmarks and three unmodified M-TGNN models (TGN, TNCN, APAN), PRISM improves accuracy of existing models by up to 28% and surpasses the TGB leaderboard by 9.2%, while keeping training time competitive with parallel stale-memory systems (TGL, ETC) and consistently lower than stricter staleness-free baselines. PRISM thus provides a practical, staleness-free foundation for temporal graph learning.

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

1 INTRODUCTION

Temporal Graph Neural Networks (TGNNs) extend static GNNs to dynamic graphs where nodes, edges, and features evolve over time, typically as streams of events [5, 8, 24]. By modeling temporal dependencies between interactions—such as transactions or communications—TGNNs jointly capture structural and temporal dynamics. This makes them well-suited for tasks including temporal link prediction, dynamic node classification, and event forecasting [14, 15, 18, 20, 30, 33, 39].

Many popular TGNN models are *memory-augmented* (M-TGNN) because they model the propagation of information over time using a module called *memory* [10, 19, 20, 28, 31, 41, 42]. These models associate each node in the graph with a *memory vector*, which is updated every time the node is affected by an *event*, for example when an incident edge is added to the graph. Over time, events propagate memory information across the graph.

Training M-TGNNs is difficult to parallelize because memory updates must follow an order that respects the temporal and structural *dependencies* among events. For example, if two events add edges incident on the same node, their memory updates must be computed and applied sequentially in temporal order. However, achieving sufficient computational parallelism requires processing multiple events concurrently. This tension between considering dependencies, which preserves the semantics of M-TGNN models, and parallelism is a central challenge in the design of M-TGNN training systems [7, 11, 12, 35, 37, 42, 44, 46].

Most of these systems adopt a *stale-memory training* strategy, which achieves parallelism by relaxing model semantics. They process batches of consecutive events and *ignore intra-batch dependencies* to increase parallelism. A simplified overview of this execution flow is shown in Figure 1a. First, the system computes predictions for all events in a batch using the memory vectors carried over from the previous batch (Task 1). Then, it updates the node memories based on the current batch's events and memory values (Task 2). The resulting updated memories are subsequently used to process the next batch. Both tasks rely on memory vectors from the previous batch that may be stale, since they do not incorporate the effects of preceding dependent events within the same batch.

Stale-memory training introduces a tradeoff between parallelism and accuracy. As the batch size increases, more dependencies are ignored, exacerbating memory staleness and ultimately reducing prediction accuracy, as shown in Figure 2. Some systems combine stale-memory training with strategies that reduce the impact of intra-batch dependencies by adaptively defining batch boundaries or using randomization [2, 4, 45]. They can use larger batch sizes to improve training efficiency, but they typically cannot exceed the best accuracy that can be achieved using regular stale-memory training and smaller batches, since they still relax model semantics.

This paper proposes a novel approach to build general M-TGNN training systems that preserve model semantics while achieving high parallelism. It addresses two fundamentals questions: What formal correctness guarantees should a M-TGNN training system achieve to preserve model semantics, respect dependencies, and avoid memory staleness? How to implement them with a parallel algorithm that supports large batch sizes?

One possible approach to preserve model semantics is to serialize the processing of dependent events (batch size 1). This implements what we call the *sequential freshness* property. *Sequential freshness* can also be achieved by forming independent sequential chains of dependent events and executing chains in parallel, as in NeutronStream [1]. However, this design yields limited parallelism: the number of independent chains per time window is small—particularly for large batches—and chain lengths are highly skewed, with some requiring long sequential processing. While tolerable in a CPU-based system, this approach does not scale well to GPU-based training.

In this paper, we formally define a novel correctness property, called *lazy freshness*. Like sequential freshness, lazy freshness preseves model semantics by avoiding the same *staleness anomalies* that

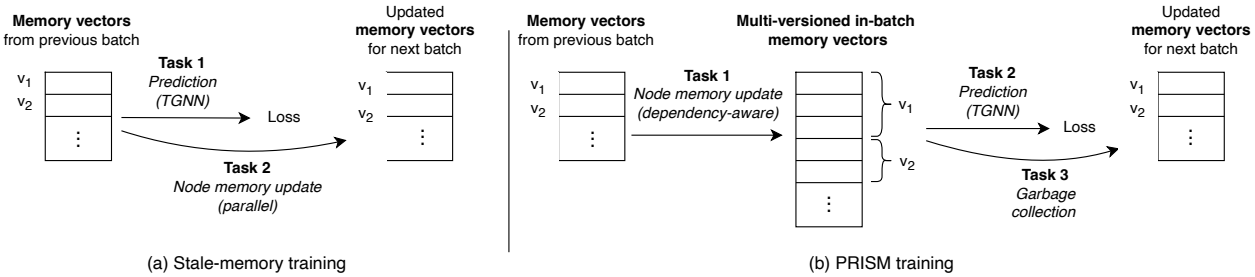


Figure 1: Overview of the execution flow of M-TGNN training for a batch.

are ruled out by sequential freshness. Lazy freshness is, however, weaker than sequential freshness and thus amenable to highly parallel and efficient implementations. Sequential freshness requires that each update to a memory vector consider the latest fresh version of that vector. Lazy freshness, instead, allows updating memory vectors using an older fresh version, and only requires that the update considers all subsequent dependent events that were not observed by that version. This allows updating multiple versions of the memory vector of a node in parallel rather than sequentially.

We leverage lazy freshness in a new staleness-free M-TGNN training system called *PRISM: Parallel Refinement of Intra-batch Stale Memory*, which preserves model semantics and still can utilize GPUs effectively. PRISM is a general training systems, which supports M-TGNN models expressed using the same general abstractions as other existing systems, but its underlying training strategy is different. The high-level idea is shown in Figure 1b. Instead of using memory vectors built in the previous batch, PRISM starts the computation of a batch by calculating *multi-versioned memory vectors* for all the nodes affected by the events in the batch (task 1). For each event affecting a node, PRISM computes a different version of the node's memory, which respects all the intra-batch dependencies in the memory update process. Then, the predictions for each event can use the correct versions of the memory vectors for that event (task 2). At the end of the batch, only the latest memory version for each node is kept (task 3).

The challenge in PRISM is to devise a multi-versioned training algorithm that can be parallelized by event. This ensures that training can scale with the batch size, since each training batch consists of a set of events. Predictions based on multi-versioned memory vectors (task 2) can be parallelized by event like in stale-memory training. The computation of the memory vectors (task 1), however, is challenging to parallelize because it needs to consider dependencies. Our key insight is that scheduling memory updates in an order that respects dependencies, as done by existing work, is inherently hard to parallelize. Instead, we frame the problem as a parallel diffusion algorithm over a data structure called the Memory Computation Graph, which maps dependencies between versions. The algorithm performs multiple passes over the graph, propagating and applying memory updates between memory versions, until it eventually eliminates all stale versions and ensures lazy freshness. Each diffusion pass is parallelized by event, which makes the algorithm scale to large batches.

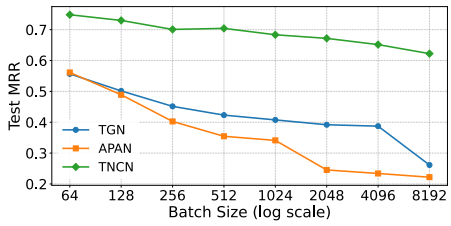


Figure 2: Batch size vs. Mean Reciprocal Rank (MRR) with stale-memory training on the wiki dataset.

A second systems challenge for M-TGNN training is to support efficient predictions over dynamic graphs, where new events can be added incrementally. This requires appropriate dynamic graph data structures that are designed for efficient GPU-based temporal sampling. We propose a new data structure, called TCI, that addresses both these problems and achieves state-of-the-art performance.

By being able to train efficiently with larger batch sizes without loss of accuracy, PRISM boosts the accuracy of unmodified existing M-TGNN models. For example, we achieve up to 28% higher accuracy for TNCN over dependency-agnostic training, exceeding the accuracy of the leader in the Temporal Graph Benchmark leaderboard by up to 9.2% [7]. PRISM is also efficient, generally achieving a lower time to convergence than the baselines. In some cases, the speedup is up to 4.76×, while in some other cases existing systems converge faster, but to a much lower accuracy.

PRISM is also efficient for online inference. On the Wiki dataset, it achieves lower latency than the baselines for both TGN and APAN with the same accuracy target, with speedups of up to 35.39×. Furthermore, PRISM supports incremental sampler-state updates without full data-structure reconstruction.

Overall, our work shows that using the right training system that carefully avoids approximations is essential to unlock the accuracy potential of M-TGNN models. PRISM tackles foundational staleness challenges that arise in single-GPU settings. This paper makes the following contributions:

- We formalize *sequential freshness*, introduce a parallel implementation (BNS), and characterize its limitations.
- We formalize *lazy freshness*, a relaxed correctness condition that avoids staleness anomalies while enabling more parallelism.
- We propose an implementation of lazy freshness in *PRISM*, an M-TGNN training system. PRISM materializes multiple in-batch memory versions and computes them using a parallel diffusion-style refinement algorithm over a Memory Computation Graph.

- We design *GRN-Stream*, a GPU-accelerated temporal neighbor sampler based on a novel *Temporal Chunk Index (TCI)*, enabling efficient incremental updates and high-throughput sampling for large dynamic graphs.

2 BACKGROUND

We introduce key concepts and terminology used throughout this paper to formalize the problem setting of M-TGNNs. To improve readability, we summarize the core notation used in the paper in Table 1. Superscripts in parentheses denote time (e.g., $s_u^{(t)}$); without parentheses, they denote memory versions within a batch (e.g., s_u^k).

Temporal Graph Neural Networks. A temporal graph consists of a node set $\mathcal{V}$ and a set $\mathcal{E}$ of timestamped *events* $e = (u, v, x, t)$, where $u, v \in \mathcal{V}$ interact at time $t \in \mathbb{R}^+$ with associated features x. Both $\mathcal{V}$ and $\mathcal{E}$ may expand over time as new nodes or events appear. *TGNNs* extend conventional *GNNs* to dynamic graphs and are widely used to learn node representations for various prediction tasks [21, 28, 31, 36, 43]. They process evolving connections by sampling and aggregating information from each node's *temporal neighborhood*, restricted to interactions occurring before a given timestamp.

Memory-Augmented TGNNs. A key challenge in temporal graph learning is preserving long-term history. *M-TGNNs* tackle this by maintaining a per-node memory vector $s_u^{(t)} \in \mathbb{R}^d$ that encodes a node's past interactions and is combined with node features during prediction. The *node memory update* task (see Figure 1) updates memory vectors as new events occur. Each event produces messages that are appended to the mailboxes of the recipient nodes, and the aggregated messages are then used to update their memory vectors.

Messages for memory updates. Let a (mini)batch be an ordered list of observed events, which correspond to edges in the graph:

$$\mathcal{B} = \{e_k = (u_k, v_k, x_k, t_k)\}_{k=1}^{K}, \qquad t_1 \le t_2 \le \cdots \le t_K,$$

Each event $e_i = (u, v, x, t) \in \mathcal{B}$ generates two messages via a message function $\mathrm{msg}(\cdot)$ to update the memories of u and v:

$$m_{i,u} = \mathrm{msg}(s_u^{(t^-)}, s_v^{(t^-)}, x, t), \quad m_{i,v} = \mathrm{msg}(s_v^{(t^-)}, s_u^{(t^-)}, x, t) \quad (1)$$

where $s_u^{(t^-)}$ and $s_v^{(t^-)}$ denote the memory states of u and v immediately before time t. The term *message* in this paper always refers to these messages, which are solely used for updating memory vectors and are distinct from the messages passed in the TGNN's embedding module during the *prediction* task (Figure 1).

Affected nodes and write set. The generated messages are delivered to the nodes *affected* by the e, collectively referred to as the *write set* $WS(e) = WS(e, u) \cup WS(e, v)$. The specific definition of $WS(e, u)$ depends on the model's delivery mechanism:

- *Self-delivery:* Messages are delivered only to the interacting nodes themselves (as in TGN [20], TNCN [41], and JODIE [10]). Formally, $WS(e, u) = \{u\}$.
- *Neighbor-delivery:* Messages are additionally propagated to the neighbors of the interacting nodes at time t (as in APAN [31] and DyRep [28]). Formally, $WS(e, u) = \{u\} \cup \mathcal{N}(u, t)$.

Mailbox. After messages are generated, they are delivered by appending them to a data structure associated to the recipient of the message called the *mailbox*. Formally, the mailbox of node u at time t, denoted by $\mathcal{M}_u^{(t)}$, contains all messages generated by events e' in the interval $(t^-, t]$ such that $u \in WS(e')$, where t^- is the timestamp of a prior memory update to u.

Table 1: Core notation used in PRISM.

Symbol	Description
$\mathcal{B}_i$	i-th mini-batch of events.
$s_u^{(t)}$	State (memory) vector of node u at time t.
s_u^k	Version k of node u's state within a batch (refinement index).
$\mathrm{msg}(\cdot)$	Message generation function.
$m_{e,u}$	Message generated by event e for node u.
$\mathcal{M}_u^{(t)}$	Mailbox of node u at time t (set of received messages).
$WS(e, u)$	Write set of event e at node u.
$WS(e)$	Write set of event e (union over interacting nodes).
$\mathrm{MFG}(e)$	Union of nodes in the message flow graph for event e.
$\mathcal{N}(u, t)$	Temporal neighbor set of node u at time t.
$\mathrm{AGG}(\cdot)$	Message aggregation function.
$\mathrm{UPD}(\cdot)$	State update function.

Updating a memory vector. To update a memory vector, the messages in $\mathcal{M}_u^{(t)}$ are first aggregated into a single vector using an *aggregation function* $\mathrm{AGG}(\cdot)$. The *memory update function* $\mathrm{UPD}(\cdot)$ combines this aggregated message with the previous memory state $s_u^{(t^-)}$ to produce the updated memory:

$$s_u^{(t)} = \mathrm{UPD}\big(s_u^{(t^-)}, \mathrm{AGG}(\mathcal{M}_u^{(t)})\big). \quad (2)$$

These two functions are model-specific. Examples of aggregation functions used by existing models are mean or attention-based aggregators. For the memory update function, possible choices are sequence models such as an `RNNCell`, `GRUCell`, or `LSTMCell`.

3 THE CHALLENGE OF MEMORY STALENESS

This section examines anomalies caused by stale memory during training. While existing staleness-free systems eliminate these anomalies by enforcing sequential equivalence, this limits GPU parallelism. We present a more GPU-friendly parallel design that remains sequentially equivalent, but show that this constraint still restricts scalability. We therefore conclude that a weaker notion of freshness is needed to enable higher parallelism without reintroducing stale-memory anomalies.

3.1 Characterizing Memory Staleness

In stale-memory training, both prediction and memory update steps (Figure 1a) rely on memory from the previous batch, ignoring intra-batch events. This leads to *memory staleness*, causing two anomalies: *temporal discontinuity* and *approximation error*. While temporal discontinuity has been noted in prior work [26], we are the first to describe approximation error. Both effects intensify with larger batch sizes due to increased intra-batch dependencies.

Figure 3 illustrates these anomalies for events involving node u across two batches. In the example, let t_0 be the timestamp of the the last event in batch $\mathcal{B}_{i-1}$, and assume that the memory vectors of the nodes are updated at time t_0.

Approximation error. For batch $\mathcal{B}_i$, Figure 3 shows how stale-memory training updates the memory vector of u. Three messages

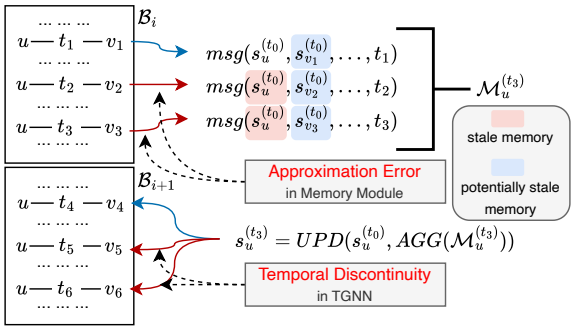


Figure 3: Anomalies in M-TGNN Stale-memory Training

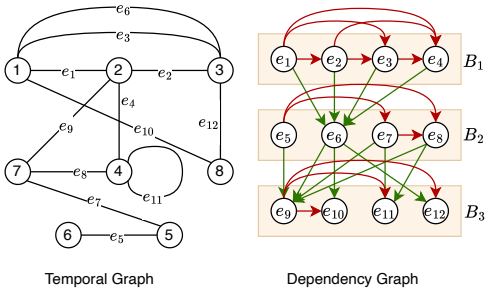


Figure 4: Temporal Graph Dataset and Dependency Graph. Events are indexed in chronological order of their timestamps. Green and red arrows represent inter-batch and intra-batch dependencies, respectively.

for u are generated (see Eqn. 1) for the three events affecting u, which occur at time $\{t_1, t_2, t_3\}$. These messages are appended to the mailbox of u. Then, the memory vector of u is updated (see Eqn. 2) using the value of the memory vector at the beginning of the batch, which is $s_u^{(t_0)}$, and the messages in the mailbox $\mathcal{M}_u^{(t_3)}$.

This approach to update memory vectors introduces staleness. After the event at time t_1, s_u should be updated. Since memory vectors are updated only once per batch, in $\mathcal{M}_u^{(t_3)}$, the messages corresponding to events at times t_2 and t_3 use the stale memory vector $s_u^{(t_0)}$. The memory vectors of the other nodes $\{v_1, v_2, v_3\}$ may also be stale if those nodes have other events in the batch. Consequently, the resulting $s_u^{(t_3)}$ is no longer fresh at the next event after t_3. We call this staleness anomaly an *approximation error*.

Temporal discontinuity. Stale-memory training can also introduce anomalies during prediction within a batched execution setting. As illustrated in Figure 3, even if $s_u^{(t_3)}$ is fresh at the end of batch $\mathcal{B}i$, it can become stale during the prediction phase of the subsequent batch $\mathcal{B}i + 1$.

In the prediction task, node memories are used to augment the input features and used to compute node representations using a TGNN. The same memory vector $s_u^{(t_3)}$ is used to perform predictions for all three events involving node u in batch $\mathcal{B}_{i+1}$, since memory vectors are updated only at the end of the batch. After the event at time t_4, however, the memory vector of u should be updated and used for predicting the events at time t_5 and t_6. We refer to this reuse of stale memory for node embedding computation as *temporal discontinuity*.

3.2 Sequential Freshness

Earlier M-TGNN works define sequential training, i.e., using a batch size of 1, as the ideal staleness-free setup [9, 16]. Each event is processed sequentially: a prediction is made, and the memories of all affected nodes are immediately updated. This guarantees that all memory vectors remain fresh, avoiding both *approximation error* and *temporal discontinuity*, a property we term *sequential freshness*.

Sequential freshness. We say that sequential training ensures *sequential freshness*, which we formally define as follows:

Definition 1 (Sequential freshness). *Let t be a timestamp and s_u the memory vector of node u. Let $s_u^{(t_s)}$ be the current value of s_u at time t, which was computed (Eqn. 2) at t_s using a previous version $s_u^{(t_s^-)}$ and a mailbox $\mathcal{M}_u^{(t_s)}$.*

We say that s_u is fresh at t iff all the following conditions are met:

(i) t_s (resp. t_s^-) is the timestamp of the latest event e_s prior to t (resp. t_s) such that $u \in WS(e_s)$;

(ii) $\mathcal{M}_u^{(t_s)}$ contains a message generated (Eqn. 1) by e_s using memory vectors that are fresh at t_s;

(iii) $s_u^{(t_s^-)}$ is fresh at t_s.

Sequential training enforces freshness but is impractical due to limited parallelism. Prior work improves parallelism along two directions: (i) stale-memory training (Figure 3), which mitigates anomalies without guaranteeing freshness [2, 4, 45]; and (ii) parallelized sequential freshness, as proposed by NeutronStream [1].

Chain-wise sequential training. NeutronStream proposes a parallel training algorithm that we term *chain-sequential*, based on a dependency graph over events. An event e_j is dependent on a preceding event e_i if e_i updates memory vectors used to predict e_j. Formally, let $MFG(e_j)$ denote the union of nodes in the message flow graph used for predicting e_j (Section 2); we define $e_i \rightarrow e_j$ iff $WS(e_i) \cap MFG(e_j) \neq \emptyset$, inducing a dependency graph (Figure 4).

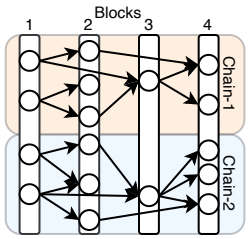
NeutronStream parallelizes training by identifying *chains* of independent events, corresponding to weakly connected components of the dependency graph. Events within each chain are totally ordered and processed sequentially, with predictions followed by immediate memory updates, thereby preserving the semantics of sequential training and ensuring sequential freshness.

We observe that this approach exposes limited parallelism (see Figure 5a). As the batch size increases, overlapping dependencies cause chains to merge, yielding longer and more skewed chains while the number of independent chains grows only marginally. At batch size 8192, chain lengths exceed 6500 events and 79% of events are serialized within a single sequential chain. NeutronStream targets CPU-based training and assigns each chain to a thread, but GPU-based training requires far more parallelism than this method can expose. Furthermore, NeutronStream performs dependency analysis offline with $O(K^2)$ worst-case complexity (for window size K), which accounts for up to 5% of the runtime overhead.

Block-wise node sequential (BNS) training. We introduce *block-wise node sequential training* (BNS), a parallel design that preserves sequential freshness while increasing parallelism. BNS

Batch	Chain-Seq		BNS
	Count	Max len	Block len
64	23	14	8
128	31	34	14
256	39	101	24
512	49	263	42
1024	61	631	72
2048	75	1434	121
4096	86	3125	220
8192	89	6508	406

(a) Parallelism metrics



(b) BNS vs NeutronStream

Figure 5: Parallelism with sequential freshness (Wiki).

serves as a baseline to demonstrate the limitations of sequential freshness even with a more parallel implementation.

Prior to training, BNS performs a preprocessing step that partitions events into *blocks* using the dependency graph (see Figure 5b). The first block contains events with no intra-batch dependencies; each subsequent block contains events that depend only on events in earlier blocks. During training, blocks are processed sequentially, while events in the same block are processed in parallel.

BNS exposes more parallelism than NeutronStream's chain-wise execution. NeutronStream processes weakly connected components sequentially, whereas BNS groups events into blocks, enabling a more parallel execution. Figure 5a shows that BNS block chains are much shorter than NeutronStream chains, especially for larger batches. However, parallelism remains constrained: although early blocks are large, later blocks are often sparse and must execute sequentially, creating a long tail that limits performance. Furthermore, dependency graph analysis still incurs preprocessing overhead, like in NeutronStream. We experimentally evaluate the limitations of BNS in more detail in Section 6.

Conclusion. Based on these results, we conclude that the sequential freshness requirement is too strict to achieve sufficient parallelism. We propose a new freshness semantic that can allow for more parallelism while avoiding stale-memory anomalies.

4 PRISM TRAINING

4.1 Lazy Freshness

In the previous section, we discussed the problem of stale memory in M-TGNN training and showed the challenges of implementing sequential freshness. We now argue that a *strictly weaker* notion of freshness, called *lazy freshness*, is sufficient to avoid the stale-memory anomalies that arise in M-TGNN training: approximation error and temporal discontinuity (see Section 3.1). We then present PRISM, our efficient training system that preserves lazy freshness.

Decoupling subsequent memory updates. Sequential freshness requires a *coupling* of memory updates: each event produces messages and these are used to update the *current* version of the memory vectors of the affected nodes. Figure 6a shows how to update the memory of node u at timestamps t_4 and t_5 in the example of Figure 3. Sequential freshness couples the updates of $s_u^{(t_4)}$ and $s_u^{(t_5)}$, which must occur in sequential order.

Our insight is that the semantics of M-TGNN models, provide enough flexibility to *decouple* updates so that they can use *previous versions* of the memory vector of a node without introducing stale-memory anomalies. This decoupling is essential to unlock additional parallelism, since multiple versions can be computed in parallel.

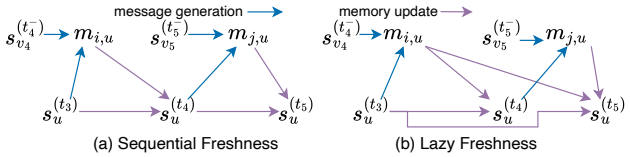


Figure 6: Decoupling memory updates with lazy freshness.

Figure 6b shows how decoupled memory updates operate in the previous example. The memory versions $s_u^{(t_4)}$ and $s_u^{(t_5)}$ can be computed independently, rather than sequentially, by using the memory vector computed at the end of the previous batch, $s_u^{(t_3)}$. To avoid stale-memory anomalies, it is necessary that all messages generated by relevant in-batch events are considered by the update. M-TGNN models can do that by including these messages in a mailbox $\mathcal{M}$ (see in Eqn. 2). This approach decouples the updates to $s_u^{(t_4)}$ and $s_u^{(t_5)}$, which can now be computed in parallel from $s_u^{(t_3)}$ as two separate versions of the memory vector s_u.

Decoupling memory updates does not completely eliminate dependencies between memory versions. In Figure 6b, $s_u^{(t_4)}$ is used to generate a message that is necessary to update $s_u^{(t_5)}$. In the next section, we will show that messages can be generated efficiently despite these dependencies using a parallel refinement algorithm.

Lazy freshness. Based on these observations, we formalize a new lazy freshness correctness property to enable decoupling and multi-versioning while avoiding stale-memory anomalies.

Definition 2 (Lazy freshness). *Let t be a timestamp and s_u the memory vector of node u. Let $s_u^{(t_s)}$ be the current value of s_u at time t, which was computed (Eqn. 2) at t_s using a previous version $s_u^{(t_s^-)}$ and a mailbox $\mathcal{M}_u^{(t_s)}$.*

We say that s_u is fresh at t *iff all the following conditions are met:*

(i) t_s is the timestamp of the latest event e_s prior to t such that $u \in WS(e_s)$;

(ii) $\mathcal{M}_u^{(t_s)}$ contains the messages generated by all the events e' with timestamp in $(t_s^-, t_s]$ such that $u \in WS(e')$. Each $m \in \mathcal{M}_u^{(t_s)}$ is generated (Eqn. 1) using memory vectors that are fresh at t_m, where t_m is the generation time of m;

(iii) $s_u^{(t_s^-)}$ is fresh at the timestamp of the first event after t_s^-.

Lazy freshness has some key differences compared to sequential freshness (Definition 1). Condition (i) does not require t_s^- to be the timestamp of the latest event affecting u prior to t_s, as done by sequential freshness. This enables parallel processing of multi-versioned memory. Nonetheless, lazy freshness still avoids temporal discontinuity, since condition (ii) ensures that all relevant messages generated after t_s^- are included in the mailbox $\mathcal{M}_u^{(t_s)}$, as required by the model semantics (see Section 2). The condition also avoids approximation error, since only fresh memory vectors must be used to generate messages. Finally, condition (iii) requires that $s_u^{(t_s^-)}$ is itself fresh.

4.2 Overview of PRISM

The PRISM system is designed to implement lazy freshness efficiently. It associates each node in a batch with multiple versions of its memory vector and updates them in parallel using an iterative refinement algorithm.

As discussed previously, lazy freshness decouples state updates but it still needs to respect dependencies for message generation. Parallelizing the memory updates while respecting these dependencies remains a challenging problem. The key idea of PRISM is that instead of tracking and scheduling memory updates in an order that respects dependencies, which is hard to parallelize, we can encode direct dependencies between versions and messages as a graph and then formulate the memory update problem as a parallel diffusion algorithm over that graph.

PRISM introduces the concept of *Memory Computation Graph (MCG)* a bipartite graph that connects memory versions to messages according to the definition of lazy freshness. In Figure 6b, the edges of the MCG are shown as message generation and memory update edges. At each batch, PRISM first builds the MCG and then runs a diffusion algorithm over it. Each pass of the algorith updates each version using messages for all relevant events at each pass. These messages, however, may still be constructed using stale memory versions, which results in approximation errors that violate lazy freshness. Iterative refinement, however, makes more and more versions fresh at each pass, eliminating approximation.

Initially, the versions of all nodes are set to be equal to the last version obtained in the previous batch. In our example, version $s_u^{(t_4)}$ is stale initially because it is set to the initial version $s_u^{(t_3)}$. At each pass, the diffusion algorithm propagates versions to messages over the message generation edges of Figure 6b, generates new messages, sends them back to the versions over the message update edges, and updates the versions. The first pass generates message $m_{i,u}$ using fresh memory versions and then uses it to update $s_u^{(t_4)}$, which becomes fresh at the end of the pass. Message $m_{j,u}$, however, is still generated using an outdated version of $s_u^{(t_4)}$ in the first pass, so $s_u^{(t_5)}$ is still stale at the end of the pass.

4.3 Parallel Iterative Refinement in PRISM

PRISM implements parallel iterative refinement as a diffusion algorithm over a *Memory Computation Graph (MCG)*. The algorithm is parallelized by event, so it can be executed efficiently on a GPU and scale with the batch size. In this section we describe this graph and the diffusion algorithm, showing that it implements lazy freshness. Then, we discuss how to build the MCG.

Memory Computation Graph. The MCG connects each memory vector version to the messages that must be included in the mailbox to update it, and each message to the memory versions that are necessary to generate it. Unlike the dependency graph used by previous work (see Section 3.2), which is more complex and is typically built offline, the MCG is a simple bipartite graph that can be efficiently built online. The MCG for a batch $\mathcal{B}$ is defined as a bipartite graph (S, M, E), where S is the memory version set for $\mathcal{B}$ and M is the set of messages generated by all the events in $\mathcal{B}$. Edges in the MCG describe the information flow used to update the memory versions in a way that respects lazy freshness.

Algorithm 1 PRISM algorithm for batch $\mathcal{B}$

Require: mini-batch of events $\mathcal{B} = \{(u, v, t, x)\}$; initial states $\{s_u^0\}$ obtained from previous batch, for each $u \in WS(e) : e \in \mathcal{B}$; number of passes K.
Ensure: fresh memory versions $\{s_u^i\}$.

```
Build MCG = (S, M, E) for B
s_u^i ← s_u^0 for each s_u^i ∈ S
// Iterate over passes
for pass ∈ [1, K] do
  // BSP superstep 1: generating messages
  for s_u^i ∈ MCG.S do
    send s_u^i to each m ∈ MCG.M : (s_u^i, m) ∈ MCG.E
  for m ∈ MCG.M do
    receive s_u^i, s_v^j
    m.msg ← MSG(s_u^i, s_v^j, m.e(x), m.e(t))
  // BSP superstep 2: updating memory versions
  for m ∈ MCG.M do
    send m.msg to each s_u^i ∈ MCG.S : (m, s_u^i) ∈ MCG.E
  for s_v^i ∈ MCG.S do
    empty mailbox M
    append all received messages to M in timestamp order
    s_v^i ← UPD(s_v^0, AGG(M))
```

Edges from memory versions in S to messages in M represent message generation. Each message is connected to the memory version that should be used to respect lazy freshness. Edges from messages in M to memory versions in S indicate which messages are used to update a memory version. A new memory version of a node u must be generated for each in-batch event e affecting u. That memory version must use messages for all in-batch events that affect u and precede e to avoid temporal discontinuity.

Iterative memory refinement. The PRISM algorithm (Algorithm 1) implements iterative refinement as a parallel diffusion algorithm over the message computation graph $MCG(S, M, E)$. Each node $s \in S$ holds a memory vector. The initial memory version of a node u in a batch is denoted as s_u^0 and subsequent versions are denoted as $s_u^1 \dots s_u^n$. Each node $m \in M$ holds a message vector $m.msg$ and the generating event $m.e$.

At the beginning of a batch, the algorithm builds the MCG and initializes all memory vector versions of node u to s_u^0. The algorithm then proceeds in multiple passes of information diffusion, each updating the value of all memory vector versions. Each pass consists of two Bulk-Synchronous Parallel super-steps. In the first super-step, nodes in S propagate memory versions to nodes in M, which generate messages. In the second super-step, nodes in M propagate messages to nodes in S, which update their memory versions. The algorithm runs for a fixed number of passes.

Correctness. We now argue that the PRISM algorithm is correct.

Theorem 4.1. *The PRISM algorithm (Algorithm 1) satisfies lazy freshness (Definition 2).*

Proof. We need to prove that when the algorithm terminates for a batch $\mathcal{B}$, each version $s_v^j \in S$ is fresh at the time of the first event after the latest event that updates that version, which we call $e_{\phi(v,j)}$. Suppose that we run the algorithm for a number of passes K equal to half the diameter of the MCG, which is a DAG.

The initial versions $s_u^0 \in S$ computed at the end of the batch before $\mathcal{B}$ are fresh by induction on the sequence of batches. We now show that all the other versions in S eventually become fresh.

We start by introducing some notation. Given a version $s \in S$, consider the set of maximal paths of length $\leq 2k$ terminating in s, and let $P(s,k)$ be the set of source nodes of those paths. Note that each node in $P(s,k)$ is in S, since MCG is a bipartite graph and each message in M has incoming edges from versions in S. The algorithm runs as many passes as half the diameter of the MCG.

We now show by induction that *after k passes, every version $s_u^i \in S$ such that all versions $s_v^j \in P(s,k)$ satisfy $j = 0$ is fresh* at the time of the first event after $e_{\phi(v,j)}$. All versions satisfy this condition after executing all the passes of the algorithm. This is because any reverse path in the MCG that begins at a version in S and repeatedly follows predecessor edges will eventually terminate at an initial version. This follows by construction: the MCG is a DAG, the initial versions $s_u^0 \in S$ have no incoming edges, all other versions $s_u^i \in S$ have incoming edges from one or more messages, and each message $m \in M$ has incoming edges from two versions. The statement trivially holds for $k = 0$, since it must hold that $i = 0$ and we have already established that s_u^0 is fresh. We now consider a pass $k > 0$ and a version s_u^i such that all $s_v^j \in P(s_u^i, k)$ satisfy $j = 0$. We show that s_u^i is fresh at the time of the first event after $e_{\phi(v,j)}$.

Condition (i) of Definition 2 holds because at every pass the algorithm updates the memories with messages generated by all previous events in the batch, according to the MCG. Condition (iii) holds because the algorithm updates all memory versions using $s_u^{(t_s^-)} = s_u^0$, which we have previously established to be fresh. For condition (ii), we need to show that all messages in the mailbox $\mathcal{M}_u^{(t_s)}$ used to update s_u^i are generated using fresh versions. These versions are all at distance 2 from s_u^i in the MCG, by construction. Suppose by contradiction that one message in the mailbox is generated using a stale version s. By induction, it must hold that some $s_v^j \in P(s, k-1)$ has $j \neq 0$. Since s is at distance 2 from s_u^i, this implies that $s_v^j \in P(s_u^i, k)$ and $j \neq 0$, a contradiction. □

Building the MCG efficiently. The MCG is constructed by parallelizing across the events in a batch. First, we build tensors for the multi-versioned memory vectors and the messages that enable constant-time retrieval of the latest memory versions needed to compute messages for later events. Each event e produces $|WS(e)|$ new memory versions. All versions are globally ordered such that for two events e_i and e_j with $j > i$, the versions generated by $WS(e_i)$ precede those from $WS(e_j)$. Edges from memory versions in S to messages in M are established through a single parallel reverse sweep over the versions, sorted by the event that updated them. Edges from M to S are constructed with a parallel forward sweep that links each message to the subsequent versions they update. Both sweeps incur constant per-thread work and a linear overall pass in the number of memory versions, allowing them to execute concurrently on the GPU.

5 EFFICIENT TEMPORAL SAMPLING

Temporal neighbor sampling is central to TGNNs, yet systems such as GNNFlow [44] rely on data structures that limit scalability. GNNFlow stores temporal events in block-based linked lists in host memory, where each block contains ordered tuples (neighbor ID, edge ID, timestamp). To locate the recent neighbors of a node with n temporal events and block size c, sampling requires traversing $\lceil n/c \rceil$ blocks to locate edges preceding a query time, followed by a binary search within a block, yielding complexity $O(\lceil n/c \rceil + \log c)$. This pointer chasing induces *irregular memory access*, preventing coalesced GPU reads and introducing traversal latency.

In contrast, *PRISM* introduces a GPU-accelerated Recent Neighbor Sampler (*GRN-Stream*) built on a custom *Temporal Chunk Index (TCI)*. By eliminating pointer chasing and moving temporal search entirely to the GPU, GRN-Stream reduces CPU–GPU transfer overhead and enables high-throughput, logarithmic-time neighbor retrieval for large dynamic graphs.

Temporal Chunk Index (TCI)— is a GPU-aware data structure that replaces GNNFlow's linked-list traversal with a *contiguous, chunk-based layout*. Each node's temporal events are partitioned into fixed-size, contiguous *chunks* of size c (similar to block size), each storing sorted (timestamp, edge ID) pairs. Chunks are stored in *CPU pinned memory* and transferred asynchronously to the GPU only when required. A compact *chunk map*, permanently resident in GPU global memory, stores terminal timestamps and memory pointers for all chunks, enabling fast GPU-side indexing without host involvement.

GRN-Stream sampler executes a *two-stage GPU-based binary search* for each query $(node, ts)$: (1) a GPU search over the chunk map locates the earliest chunk whose terminal timestamp exceeds ts; (2) a second GPU search within that chunk finds the most recent interactions preceding ts. This design reduces search complexity from $O(\lceil n/c \rceil + \log c)$ in GNNFlow to $O(\log w + \log c)$ in GRN-Stream, where w denotes the number of active chunks per node. If all events were retained, w would equal $\lceil n/c \rceil$. However, chronological training enables a further optimization: we maintain a *sliding active window* of w consecutive chunks per node, where $w \ll n/c$. As training progresses, the window advances so that all events required for the upcoming batch remain within the active set. This confines the search space to the active window, further lowering complexity while preserving full GPU locality.

GRN-Stream targets recent-neighbor sampling, where only the k most recent neighbors are required. By selecting a chunk size $c > k$, all k neighbors of any query node are guaranteed to lie within the identified chunk or its immediate predecessor. As a result, *at most two chunks per node* are transferred to the GPU for any sampling operation. In practice, k is small (typically 10–20) and c is large (256 or higher), so one or two chunks are sufficient for all node appearances *in the batch*. This significantly reduces data movement between CPU and GPU. Moreover, since TGNN training progresses *chronologically*, the same chunks are frequently reused across consecutive batches, enabling efficient GPU caching and further minimizing transfer overhead.

By co-locating the lightweight chunk index on the GPU and storing bulk edge data in pinned CPU memory, GRN-Stream ensures contiguous, coalesced GPU access while minimizing data transfers by loading at most two reusable chunks per node. This design enables scalable asynchronous prefetching and supports incremental updates as new temporal data arrives. Overall, GRN-Stream provides a GPU-accelerated, logarithmic-time temporal

neighbor sampler that removes structural bottlenecks and enables high-throughput, large-batch TGNN training at scale.

6 EVALUATION

We conduct extensive experiments to evaluate our proposed system against state-of-the-art baselines. All experiments were conducted using CUDA v.11.8, PyTorch v.2.1, PyTorch Geometric v.2.3, RTX 2080 Ti 11GB GPUs (TGN, APAN), and A100 80GB GPUs (TNCN).

6.1 Datasets and Evaluation Protocol

We evaluate on three commonly used temporal benchmarks—Wiki (9,227 nodes, 157,474 edges), Reddit (10,984 nodes, 672,447 edges), and Lastfm (1,980 nodes, 1,293,103 edges) [10]—and two large-scale TGB datasets [7]: *tgbl-coin* (638,486 nodes, 22.8M edges) and *tgbl-comments* (994,790 nodes, 44.3M edges) [17, 22]. These datasets differ in temporal dependency range. We evaluate continuous-time dynamic link prediction using a 70/15/15 train/validation/test split. The task is to predict the next interaction partner of a source node at a given timestamp based on its historical interaction sequence, capturing evolving graph structure over continuous time. For each event $e_k = (u_k, v_k, x_k, t_k)$, the model predicts the destination node v_k using only events with timestamps $< t_k$, ensuring strict temporal causality.

At evaluation time, the true node is ranked against a fixed candidate set consisting of the true destination and sampled negatives (≈1000 for Wiki/Reddit; 5 for Lastfm). Ranking is performed independently per event using scores computed at time t_k.

Let r_k denote the raw rank of the true node within its candidate set. We report Mean Reciprocal Rank (MRR), $\text{MRR} = \frac{1}{|\mathcal{T}|} \sum_{k \in \mathcal{T}} \frac{1}{r_k}$, where $\mathcal{T}$ denotes the validation or test event set.

6.2 Benchmarks

We evaluate three M-TGNN backbones. **TGN** [20] maintains node memories updated by events and generates temporal embeddings via neighborhood sampling and aggregation. **TNCN** [41] extends Neural Common Neighbor [32] to temporal graphs with a memory-based backbone and efficient temporal common-neighbor extraction. **APAN** [31] adopts a neighbor-delivery mechanism that propagates messages to both root nodes and their neighbors; we pair it with a GAT-based GNN (as in TGN) for stronger node differentiation. Together, these models cover both self-delivery (TGN, TNCN) and neighbor-delivery (APAN) paradigms.

We compare PRISM against five representative training systems. **TGB** [7], **TGL** [45], and **ETC** [4] rely on stale-memory training with different batching strategies. **PRES** [26] approximates memory updates via a GMM-based helper module to mitigate staleness. **BNS** is our GPU-accelerated implementation of sequential freshness, executing dependency-free blocks concurrently.

6.3 Comparative Analysis

In this section, we evaluate PRISM against baseline systems in terms of accuracy, convergence time, batch-size robustness, sampling cost, memory usage, and online inference throughput, jointly analyzing the interplay among freshness, parallelism, and efficiency.

Accuracy and Convergence Time By enabling large batch sizes without staleness, *PRISM* substantially improves model accuracy while maintaining competitive or faster convergence. Table 2 summarizes accuracy and the convergence times for the best test MRR across datasets, models, and systems. For each combination, we consider the batch size that achieves the highest accuracy. PRISM tends to reach optimal performance with larger batch sizes than other systems, thanks to its staleness-free training approach.

Across all configurations, *TNCN* consistently achieves the highest test MRR, confirming its strong temporal modeling capacity and effective message-passing design under our training framework. Among systems, *PRISM* attains the highest accuracy and competetive convergence speed across every dataset, particularly for *TNCN*. For example, on Wiki, *PRISM* reaches a test MRR of 0.92—the highest across all systems—while converging in only 225 seconds, nearly an order of magnitude faster than stalenes-free baselines such as *BNS* and substantially higher in accuracy than *ETC* or *TGL*. Similar trends are observed on Reddit and Lastfm, where *PRISM* sustains high accuracy across large batch sizes, demonstrating robustness to both short- and long-term temporal dependencies. In some cases, *ETC* or *TGL* converge slightly faster, but they settle at suboptimal minima, while *PRISM* converges to a more accurate and stable solution, yielding higher test MRR.

In terms of convergence time, *PRISM* is generally faster to converge than other systems, up to 4.76× for the Wiki dataset and *APAN*. In some cases, existing systems achieve faster convergence than *PRISM*, but they converge to a lower accuracy.

We did not conduct *APAN* experiments with *BNS* because *APAN*'s neighbor-delivery mechanism creates a large number of memory versions per node, which in turn increases the number of dependency blocks within each batch and severely limits GPU parallelism. As a result, the per-epoch computation time becomes prohibitively high, making *BNS* training impractical for *APAN*.

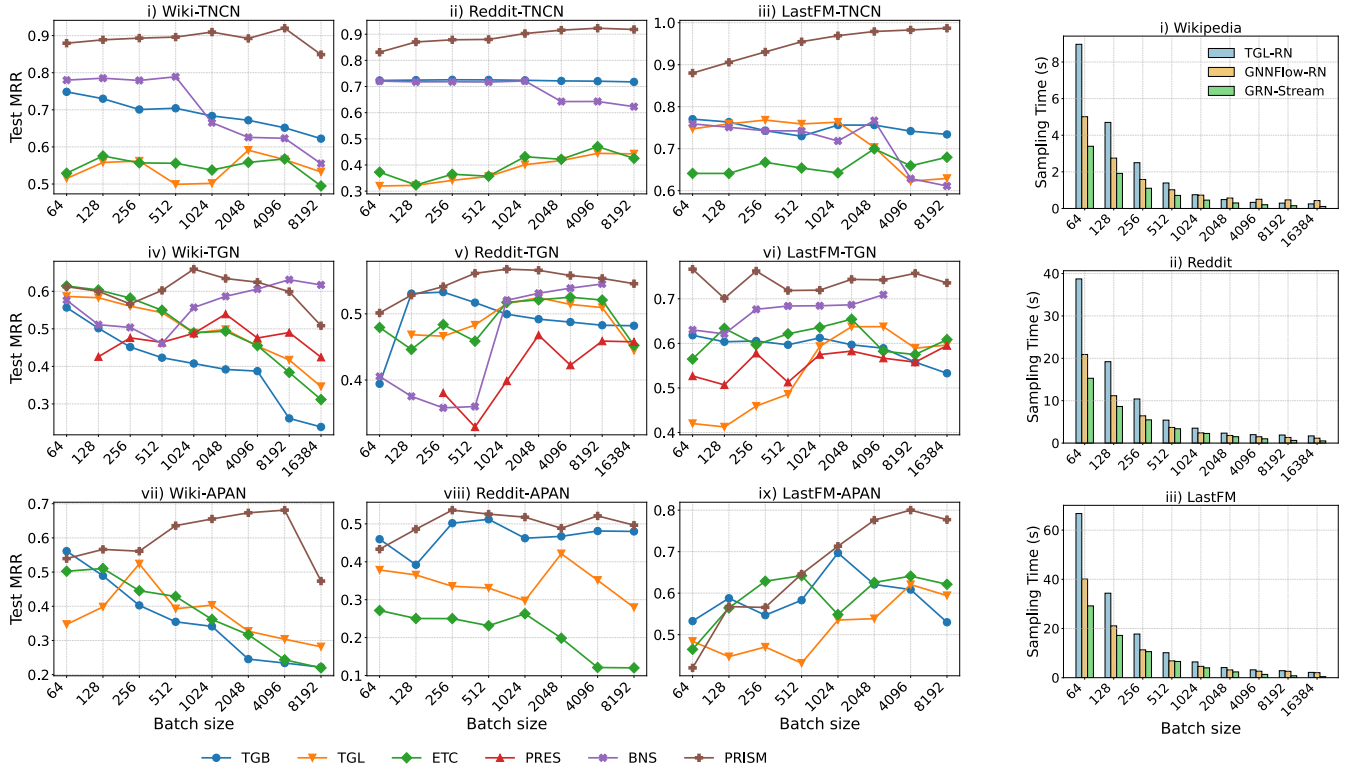
We also perform a similar evaluation on three datasets provided by the *TGB benchmark*, which include a leaderboard [27]. Table 3 reports the test MRR on *TNCN* model running on top of the stale-memory TGB system, the same model running on top of PRISM *PRISM-TNCN*, and the current benchmark leaderboard on those datasets *TPNet* [14]. *PRISM-TNCN* consistently achieves higher accuracy, surpassing the best *TGB*-reported results.

Predictive Stability with Varying Batch Sizes Figure 7a shows the effect of batch size on test accuracy (MRR). *PRISM* achieves the most stable accuracy across all datasets as the batch size varies. It maintains near-constant accuracy on Wiki and Reddit and preserves strong long-range consistency on Lastfm, confirming its adaptability across varying temporal dependency scales. In general, the staleness-free training systems, *BNS* and *PRISM*, consistently outperform all stale-memory approaches by explicitly enforcing event dependencies during memory updates. *BNS* achieves sequential freshness, which limits parallelism as batch size increases. Consequently, the model experiences longer effective update horizons—information flows correctly but must traverse many intermediate events before reaching distant nodes. This deep, multi-step propagation can dilute temporal signal strength and accumulate numerical or gradient noise.

The *TGB* baseline performs well at small batch sizes but degrades sharply as batch size grows. Because it updates node memories only once per batch, *TGB* fails to capture within-batch temporal order,

Table 2: Comparison of best test MRR and convergence time with batch size across systems, models, and datasets.

Dataset	Model	TGB			TGL			ETC			BNS			PRISM (Ours)		
		MRR	Time (s)	Batch	MRR	Time (s)	Batch	MRR	Time (s)	Batch	MRR	Time (s)	Batch	MRR	Time (s)	Batch
WIKI	TNCN	0.7484	838	64	0.5912	379	2048	0.5676	304	4096	0.789	2682	512	**0.9199**	**225**	4096
	TGN	0.5567	1590	64	0.5863	3007	64	0.6141	2024	64	0.6309	3345	8192	**0.6589**	**531**	1024
	APAN	0.5613	2229	64	0.5241	3335	256	0.5106	1267	128	-	-	-	**0.6813**	**266**	4096
REDDIT	TNCN	0.7254	1564	512	0.4445	1156	4096	0.4703	1442	4096	0.7213	2958	1024	**0.923**	**1025**	4096
	TGN	0.5328	4028	256	0.5244	1072	2048	0.5211	**415**	2048	0.5451	7026	8192	**0.5534**	1286	8192
	APAN	0.5121	5499	512	0.4213	**1772**	2048	0.2629	3420	1024	-	-	-	**0.521**	1818	4096
LASTFM	TNCN	0.7562	1214	2048	0.7632	12419	1024	0.6997	4332	2048	0.767	**601**	2048	**0.9869**	1326	8192
	TGN	0.6036	1730	128	0.6373	1602	2048	0.6542	**358**	2048	0.7091	51625	4096	**0.7571**	2266	8192
	APAN	0.6968	6572	1024	0.6206	4833	4096	0.6411	**2763**	4096	-	-	-	**0.8001**	2956	4096



(a) batch size vs test MRR **(b) batch size vs sampling time**

Figure 7: Comparative analysis of MRR and sampling time for different batch sizes across systems

leading to strong staleness effects—especially in Wiki and Reddit, where short-interval dependencies are common. Lastfm's longer interaction gaps make it less sensitive to staleness, producing a slower accuracy decline. *TGL* and *ETC* alleviate accuracy degradation with larger batch sizes. *TGL*'s random chunking preserves limited temporal locality but disrupts true dependency chains, causing oscillatory MRR trends. *ETC*'s threshold-based dependency-aware batch boundaries yield smoother trends and moderate gains by grouping temporally close events. In both cases, accuracy plateaus with larger batch sizes.

Sampling Time with Varying Batch Sizes Figure 7b compares sampling time across batch sizes for the OpenMP-based *TGL-RN*, the GPU-based *GNNFlow-RN*, and our *GRN-Stream* sampler. At smaller batches (64–256), all samplers incur higher overheads, but GRN-Stream is already faster than TGL-RN—about 2× on Wiki and

Table 3: TNCN vs PRISM-TNCN vs TGB Best Model

Dataset	TNCN MRR	PRISM-TNCN MRR	TGB Best Method	TGB MRR
tgbl-wiki	0.7190	**0.9199**	TPNet	0.8280
tgbl-coin	0.7660	**0.9130**	TPNet	0.8330
tgbl-comment	0.7030	**0.8577**	TPNet	0.8310

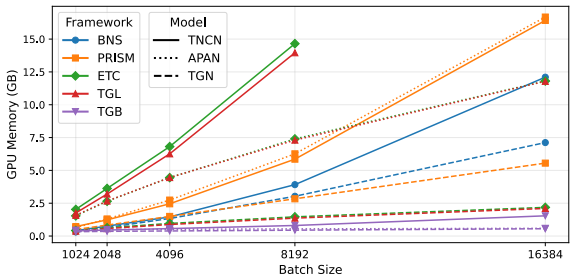


Figure 8: GPU memory usage on the *Wiki* dataset.

up to 2.5× on Reddit and Lastfm. As batch size increases, sampling time decreases for all methods, yet the reduction is sharper for *GRN-Stream*. *TGL-RN* plateaus due to thread scheduling and synchronization overhead, while *GRN-Stream* continues to scale smoothly, sustaining 2–2.5× speedup at batch sizes beyond 8192. *GNNFlow-RN* shows similar scaling behavior but remains consistently slower than *GRN-Stream*.

Memory Requirement PRISM increases memory consumption by using multiple versions of each memory vector. We analyze GPU memory consumption across frameworks and varying batch sizes to understand their scalability limits (Figure 8). *TGB*, *TGL*, and *ETC* each maintain a single memory version per node, which explains the similar memory footprints of the *TGN* model across these frameworks. However, the temporal sampler used by *TGL* and *ETC* materializes separate entries for every interaction with a neighbor, even when multiple interactions involve the same node inflating intermediate memory usage, particularly for *APAN*, which must send messages to all temporal neighbors. For *TNCN*, the effect is amplified since it computes representations not only for root nodes but also for one-hop neighbors to form common-neighbor features. Without temporal deduplication, redundant neighbor entries propagate through the memory flow graph, expanding the effective frontier. At large batch sizes (e.g., 16384), this can lead to oversized tensors in the NCN decoder, and the subsequent dense-to-sparse conversion may exceed PyTorch's `INT_MAX` element limit.

PRISM and *BNS* require more GPU memory due to multi-version node-memory management. *PRISM*'s multi-pass mechanism further increases its footprint by holding additional intermediate gradients during training. As shown in Figure 8, PRISM exhibits a visible regime transition as batch size increases. However, beyond this transition, its memory growth does not further steepen and remains predictable across all three models. Overall, PRISM exhibits monotonic and controlled memory scaling behavior consistent with its explicit multi-version design.

Table 4: Online inference throughput (events/sec) across systems and speedup vs. PRISM (Wiki dataset).

Model	System (batch size)	Throughput (events/s)	Speedup (×)
TGN	PRISM (8192)	679.4	1.00×
	TGB (128)	85.6	7.94×
	TGL (128)	19.2	35.39×
	ETC (128)	29.7	22.86×
APAN	PRISM (4096)	224.9	1.00×
	TGB (64)	62.5	3.60×
	TGL (256)	39.0	5.77×
	ETC (128)	29.5	7.62×

Inference Performance We now evaluate inference performance on a dynamic stream of events. Inference does not need a backward pass over the model, which reduces computational costs compared to training and helps *PRISM* achieve similar performance benefits as with training. It is now necessary, however, add a stream of events to the graph, which can become a major bottleneck.

Table 4 reports incremental inference throughput on the Wiki test split for *TGN* and *APAN* across different systems. For each model, batch sizes are selected such that all systems achieve similar MRR, enabling a fair comparison that isolates inference-time effects rather than accuracy differences. We exclude *TNCN* because *PRISM* consistently attains substantially higher MRR across batch sizes.

PRISM achieves much lower inference latency than the baseline systems. This advantage stems from two factors. First, its robustness to large batch sizes enables higher event ingestion rates at a given accuracy, whereas stale-memory baselines are constrained to small batches. Second, its GRN-Stream sampler operates on an incremental data structure, TCI. While *TGB* also supports incremental updates, it suffers from stale-memory limitations; in contrast, *TGL* and *ETC* rebuild temporal CSR structures from scratch, increasing update latency in streaming settings.

Memory–Parallelism–Accuracy Tradeoffs Training batch size is tightly coupled with GPU memory capacity and has a direct impact on both convergence speed and final accuracy. For PRISM, memory usage is further influenced by the need to maintain multiple memory versions. In Figure 9 we evaluate different batch sizes for different systems. For each batch size/system combination, we report the resulting GPU memory requirements, training time, and accuracy. The best batch size/system combinations lie at the top-left corner of the figure (high accuracy, low memory consumption) and have cooler colors (shorter training time).

PRISM operates effectively with the larger batches by maintaining multi-version memory and applying refinement passes to mitigate staleness. This shifts the accuracy–memory frontier upward: PRISM achieves higher MRR at comparable or lower training time under moderate memory budgets (under 4 GB). Baselines remain competitive only with small memory budgets (under 1 GB), which constrain training to small batches where staleness is negligible. These results define a clear operating envelope: PRISM is most beneficial with moderage memory budgets.

6.4 Ablation and Sensitivity Analysis

In this section, we analyze the sensitivity of *PRISM*'s two key parameters, the number of refinement passes and TCI chunk size. We

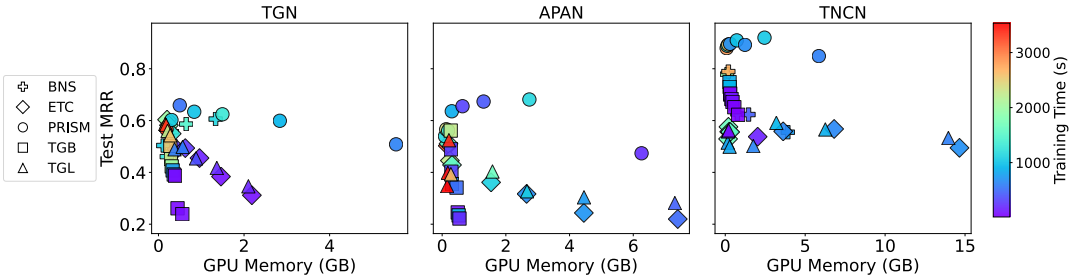


Figure 9: Accuracy (MRR) and training time for different memory budgets (Wiki dataset).

evaluate how these parameters affect convergence behavior, and computational cost across datasets and models.

MCG Construction Cost. We profile memory computation graph (MCG) construction overhead across models on Wiki dataset. For *TGN* and *TNCN*, *MCG* construction accounts for less than 2% of total training time across batch sizes 102 and 8192. In contrast, *APAN*, which propagates updates to neighbors and induces denser intra-batch dependencies, exhibits moderately higher construction cost: 4.4% at batch size 1024 and 8.3% at batch size 8192. These results indicate that *MCG* construction remains a small fraction of overall training time across architectures, while naturally scaling with the density of intra-batch dependency patterns.

Multi-Pass Refinement Analysis. PRISM uses an iterative refinement algorithm that runs for a fixed number of passes. Our theoretical analysis shows that to achieve lazy freshness, it is sufficient that a batch uses a number of passes equal to half the diameter of the Message Computation Graph. Empirically, however, we observe that a small number of passes are sufficient to make most memory vector versions fresh and achieve high accuracy while increasing training efficiency.

Figure 10 illustrates how the number of passes affects the convergence and accuracy of *TGN* across the three datasets. Using zero passes is equivalent to stale-memory training, where memory is updated only once per batch. Increasing the number of passes introduces iterative refinement. With one pass, a single version of each node's memory is updated considering all the events that affect that node in the batch. This setting updates node memories in-batch using the same logic adopted by stale-memory training systems across batches, which already improves convergence substantially.

Using $k > 1$ passes ensures that each version update receives information from all k-hop chains of dependent events affecting it. Figure 10 shows that using more than one pass is beneficial. However, beyond a certain number of passes, there are diminishing returns and eventually convergence starts to degrade. This is likely due to oversquashing: propagation over a large number of hops can lose discriminative information. Furthermore, a larger number of passes also results in longer epochs, increasing convergence time.

The number of refinement passes is a dataset-dependent hyperparameter that can be tuned to further optimize accuracy if needed. For example, Figure 10 shows that LastFM behaves better than the other datasets with a larger number of passes. We use three passes in our evaluation for all datasets and models because it achieves a good tradeoff between convergence time and accuracy.

Overall, the ability of leveraging a bounded number of passes is a major advantage of PRISM over existing systems, since it allows integrating k-hop dependency information at a moderate performance cost. By contrast, sequential training approaches like NeutronStream or BNS can only offer full sequential freshness.

Impact of Multiple Passes on GPU Memory usage Figure 11 presents GPU memory requirements across different numbers of refinement passes for the three models (*TGN*, *TNCN*, and *APAN*). Each sub-block corresponds to a fixed model, where columns denote batch sizes and rows represent the number of passes. Memory usage increases smoothly with both batch size and the number of passes, demonstrating near-linear scaling behavior. For smaller batch sizes (up to 4096), *TNCN* shows the highest memory footprint due to its two-hop aggregation and dense intermediate tensors. At larger batch sizes, *APAN* overtakes *TNCN* because its message-passing mechanism sends updates to all temporal neighbors, requiring substantial mailbox and buffer storage. *TGN* remains the most memory-efficient across all configurations. Overall, PRISM exhibits bounded and predictable growth, confirming that multi-pass refinement introduces only moderate overhead on GPU usage.

Impact of TCI on training performance. *TCI* reduces sampling overhead and improves end-to-end training efficiency, particularly at larger batch sizes. While absolute sampling time is similar across models, its relative share varies due to model-specific compute costs: *APAN*'s memory module and *TNCN*'s decoder are more computation-intensive, reducing the sampler's percentage contribution. On Wiki, sampling accounts for 13.2% (*TGN*), 8.5% (*APAN*), and 8.5% (*TNCN*) at bs=512, dropping to roughly 3–4% at bs=8192. By reducing neighbor sampling time and ensuring it scales smoothly with batch size, *TCI* enables larger batches to achieve throughput gains without hitting sampling bottlenecks.

TCI Chunk Size Sensitivity. To study the effect of chunk size in GRN-Stream, we fix the batch size at 8,192 and vary the chunk size (Figure 12). Each sampling query performs two binary searches—first over per-node chunk indices on GPU and then within the selected chunk on CPU—while recently accessed chunks are cached on the GPU. Larger chunks reduce GPU overhead by shrinking the node–chunk mapping table but increase search latency and host memory usage. Empirically, sampling time grows with chunk size (about 3× on WIKI, 5× on REDDIT, and 3× on LASTFM), whereas GPU overhead drops sharply (up to 50× on REDDIT), illustrating a clear trade-off between latency and memory efficiency.

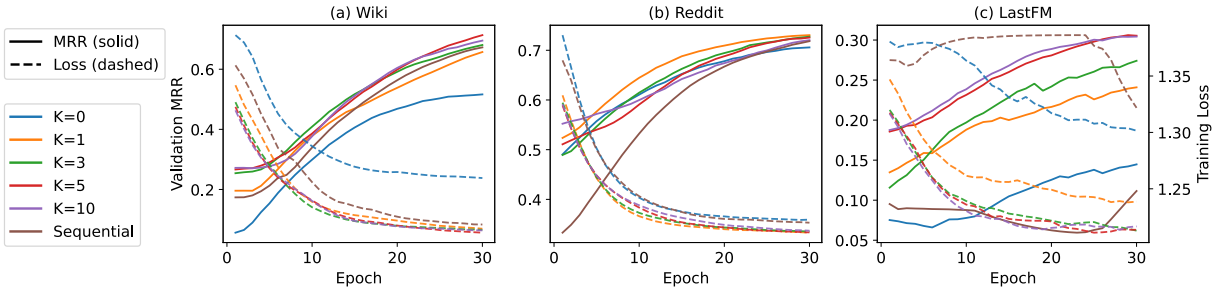


Figure 10: Loss vs epoch for different pass counts on Wiki, Reddit, and LastFM for Self-Delivery (PRISM-TGN).

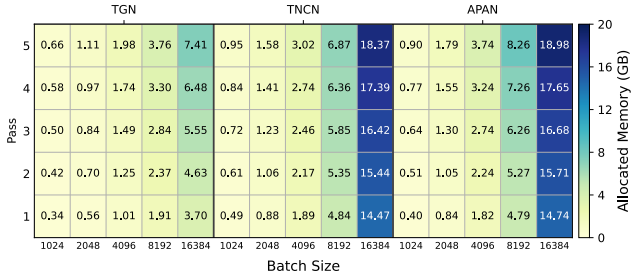


Figure 11: Memory requirement vs refinement passes.

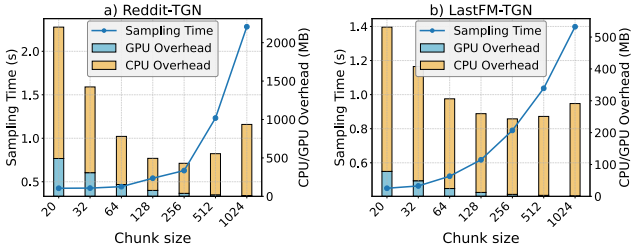


Figure 12: Sampling time and CPU-GPU memory overhead for different chunk size for GRN-Stream Sampler.

Summary. PRISM shows low sensitivity to batch size, allowing it to be scaled with available GPU memory to improve throughput. Increasing refinement passes improves convergence and final accuracy but raises memory and latency; beyond a dataset-dependent point, gains diminish and may reverse. Empirically, three passes offer a strong overall tradeoff. Chunk size does not affect accuracy but impacts memory usage and sampling speed; values between 128–512 provide a good balance.

7 RELATED WORK

Throughout the paper, we thoroughly discussed and evaluated the prior work that, like PRISM, addresses the problem of memory staleness in M-TGNN training: TGL [45], ETC [4], PRES [26], and NeutronStream [1]. We now review additional related work.

Numerous systems accelerate training on evolving graphs. Early discrete-time dynamic graph (DTDG) frameworks, for example ROLAND [37], BLAD [3], and PiPAD [29] optimize snapshot-based computation through pipelined and load-balanced execution. DynaHB [25] extends this paradigm to distributed DTDG training via hybrid batches. While effective for large static snapshots, these designs do not explicitly address fine-grained temporal dependencies central to memory-augmented TGNNs.

Several works target orthogonal system bottlenecks than the ones we discuss in this paper. TGOpt [34] compiles redundancy-aware temporal attention operators; TIGER [42], ORCA [11], and ZEBRA [12] integrate temporal propagation or personalized diffusion into model architectures; RTGA [38] provides a redundancy-free inference accelerator; and Retrofit [6] augments TGNNs with transformer-based temporal reasoning. These improve efficiency or expressiveness but do not resolve event-dependency ordering during training. TGLite [35] offers a lightweight API for continuous-time TGNNs while retaining single-version memory semantics.

Scaling to multiple GPUs introduces additional staleness challenges, including cross-partition synchronization and replicated state consistency. GNNFlow [44] pioneers a distributed CPU-GPU pipeline for streaming graphs; DistTGL [46] extends memory-based TGNNs across GPUs; PipeTGL [13] minimizes idle bubbles through pipeline optimization; and Hotspot [40] shares active-node memory among workers to cut redundancy. MSPipe [23] incorporates staleness-aware pipelining to balance throughput and temporal consistency. These distributed frameworks tackle system-level scalability, whereas PRISM complements them by focusing on avoiding staleness in single-GPU training. In our future work, we plan to expand PRISM's iterative refinement mechanism to multi-GPU systems.

8 CONCLUSION

We introduced PRISM, a dependency-aware training system that enables accurate and scalable M-TGNN learning by addressing memory staleness without compromising GPU parallelism. PRISM introduces multi-versioned memory refinement, which models intra-batch dependencies through iterative updates on a Memory Computation Graph. Together with the GRN-Stream sampler and its GPU-aware Temporal Chunk Index, PRISM delivers efficient large-batch training and high-throughput temporal sampling. Extensive experiments demonstrate that PRISM achieves up to 28% absolute accuracy improvement over existing systems while retaining competitive convergence speed and memory efficiency.

REFERENCES

[1] 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 (2023), 455–468.

[2] Yue Dai, Xulong Tang, and Youtao Zhang. 2025. Cascade: A Dependency-aware Efficient Training Framework for Temporal Graph Neural Network. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 95–110.

[3] 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*. 1–13.

[4] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. ETC: Efficient Training of Temporal Graph Neural Networks over Large-Scale Dynamic Graphs. *Proc. VLDB Endow.* 17, 5 (2024), 1060–1072.

[5] Alessio Gravina, Giulio Lovisotto, Claudio Gallicchio, Davide Bacciu, and Claas Grohnfeldt. 2024. Long range propagation on continuous-time dynamic graphs. In *Proceedings of the 41st International Conference on Machine Learning*. 16206–16225.

[6] Qiang Huang, Xiao Yan, Xin Wang, Susie Xi Rao, Zhichao Han, Fangcheng Fu, Wentao Zhang, and Jiawei Jiang. 2024. Retrofitting temporal graph neural networks with transformer. *arXiv preprint arXiv:2409.05477* (2024).

[7] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2023. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems* 36 (2023), 2056–2073.

[8] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research* 21, 70 (2020), 1–73.

[9] Boris Knyazev, Carolyn Augusta, and Graham W Taylor. 2021. Learning temporal attention in dynamic graphs with bilinear interactions. *Plos one* 16, 3 (2021), e0247936.

[10] 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* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1269–1278. doi:10.1145/3292500.3330895

[11] 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.

[12] 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.

[13] Jun Liu, Bingqian Du, Ziyue Luo, Sitian Lu, Qiankun Zhang, and Hai Jin. 2025. PipeTGL:(Near) Zero Bubble Memory-Based Temporal Graph Neural Network Training via Pipeline Optimization. *Proceedings of the VLDB Endowment* 18, 8 (2025), 2722–2734.

[14] Xiaodong Lu, Leilei Sun, Tongyu Zhu, and Weifeng Lv. 2024. Improving temporal link prediction via temporal walk matrix projection. *Advances in Neural Information Processing Systems* 37 (2024), 141153–141182.

[15] Yuhong Luo and Pan Li. 2022. Neighborhood-aware scalable temporal network representation learning. In *Learning on Graphs Conference*. PMLR, 1–1.

[16] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming graph neural networks. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 719–728.

[17] Amirhossein Nadiri and Frank W Takes. 2022. A large-scale temporal analysis of user lifespan durability on the Reddit social media platform. In *Companion Proceedings of the Web Conference 2022*. 677–685.

[18] 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.

[19] Katarina Petrović, Shenyang Huang, Farimah Poursafaei, and Petar Veličković. 2024. Temporal graph rewiring with expander graphs. *arXiv preprint arXiv:2406.02362* (2024).

[20] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs.

[21] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*. 519–527.

[22] Kiarash Shamsi, Friedhelm Victor, Murat Kantarcioglu, Yulia Gel, and Cuneyt G Akcora. 2022. Chartalist: Labeled graph datasets for utxo and account-based blockchains. *Advances in Neural Information Processing Systems* 35 (2022), 34926–34939.

[23] Guangming Sheng, Junwei Su, Chao Huang, and Chuan Wu. 2024. Mspipe: Efficient temporal gnn training via staleness-aware pipeline. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2651–2662.

[24] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *iEEE Access* 9 (2021), 79143–79168.

[25] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3388–3401.

[26] Junwei Su, Difan Zou, and Chuan Wu. 2024. PRES: Toward Scalable Memory-Based Dynamic Graph Neural Networks. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=gjXor87Xfy

[27] TGB Team. 2025. Temporal Graph Benchmark (TGB) — Leaderboard for Dynamic Link Property Prediction. https://tgb.complexdatalab.com/docs/leader_linkprop/. Accessed: 2025-11-01; continuously updated.

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

[29] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: pipelined and parallel dynamic GNN training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 405–418.

[30] Lu Wang, Xiaofu Chang, Shuang Li, Yunfei Chu, Hui Li, Wei Zhang, Xiaofeng He, Le Song, Jingren Zhou, and Hongxia Yang. 2021. Tcl: Transformer-based dynamic graph modelling via contrastive learning. *arXiv preprint arXiv:2105.07944* (2021).

[31] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2628–2638. doi:10.1145/3448016.3457564

[32] Xiyuan Wang, Haotong Yang, and Muhan Zhang. 2024. Neural Common Neighbor with Completion for Link Prediction. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=sNFLN3itAd

[33] 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).

[34] 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.

[35] Yufeng Wang and Charith Mendis. 2024. Tglite: A lightweight programming framework for continuous-time temporal graph neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1183–1199.

[36] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *International conference on learning representations* (2020).

[37] 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.

[38] Hui Yu, Yu Zhang, Andong Tan, Chenze Lu, Jin Zhao, Xiaofei Liao, Hai Jin, and Haikun Liu. 2024. RTGA: A Redundancy-free Accelerator for High-Performance Temporal Graph Neural Network Inference. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[39] 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* 36 (2023), 67686–67700.

[40] Longjiao Zhang, Rui Wang, Tongya Zheng, Ziqi Huang, Wenjie Huang, Xinyu Wang, Can Wang, Mingli Song, Sai Wu, and Shuibing He. 2025. Effective and Efficient Distributed Temporal Graph Learning through Hotspot Memory Sharing. *Proceedings of the VLDB Endowment* 18, 9 (2025), 3093–3105.

[41] Xiaohui Zhang, Yanbo Wang, Xiyuan Wang, and Muhan Zhang. 2024. Efficient Neural Common Neighbor for Temporal Graph Link Prediction.

[42] Yao Zhang, Yun Xiong, Yongxiang Liao, Yiheng Sun, Yucheng Jin, Xuehao Zheng, and Yangyong Zhu. 2023. Tiger: Temporal interaction graph embedding with restarts. In *Proceedings of the ACM Web Conference 2023*. 478–488.

[43] Ying Zhong and Chenze Huang. 2023. A dynamic graph representation learning based on temporal graph transformer. *Alexandria Engineering Journal* 63 (2023), 359–369.

[44] 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* (2023).

[45] 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. *Proc. VLDB Endow.* 15, 8 (2022), 1572–1580.

[46] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. 2023. Disttgl: Distributed memory-based temporal graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.