D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks

Rustam Guliyev
University of Warwick
Coventry, UK
rustam.guliyev@warwick.ac.uk

Aparajita Haldar*
University of Warwick
Coventry, UK
aparajita.haldar@warwick.ac.uk

Hakan Ferhatosmanoglu†
University of Warwick
Coventry, UK
hakan.f@warwick.ac.uk

ABSTRACT

Graph Neural Network (GNN) models on streaming graphs entail algorithmic challenges to continuously capture its dynamic state, as well as systems challenges to optimize latency, memory, and throughput during both inference and training. We present D3-GNN, the first distributed, hybrid-parallel, streaming GNN system designed to handle real-time graph updates under online query setting. Our system addresses data management, algorithmic, and systems challenges, enabling continuous capturing of the dynamic state of the graph and updating node representations with fault-tolerance and optimal latency, load-balance, and throughput. D3-GNN utilizes streaming GNN aggregators and an unrolled, distributed computation graph architecture to handle cascading graph updates. To counteract data skew and neighborhood explosion issues, we introduce inter-layer and intra-layer windowed forward pass solutions. Experiments on large-scale graph streams demonstrate that D3-GNN achieves high efficiency and scalability. Compared to DGL, D3-GNN achieves a significant throughput improvement of about 76x for streaming workloads. The windowed enhancement further reduces running times by around 10x and message volumes by up to 15x at higher parallelism.

PVLDB Reference Format:
Rustam Guliyev, Aparajita Haldar, and Hakan Ferhatosmanoglu. D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks. PVLDB, 17(11): 2764 - 2777, 2024.
doi:10.14778/3681954.3681961

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Rustam-Warwick/d3-gnn.

1 INTRODUCTION

The ubiquity of large-scale, semi-structured data, such as knowledge graphs, social networks, financial transactions, and e-commerce networks, has fostered graph learning [41]. To this end, Graph Neural Networks (GNNs) have proven to achieve greater performance on tasks like node classification [17], link prediction [57], and graph classification [9], compared to traditional approaches [53]. GNNs combine neural networks (NN) with graph topology, allowing them to generate semantically richer node embeddings used for downstream prediction tasks. Applications of these models and the various tasks have since been extensively studied in recommendation systems [13, 55], computer vision [2, 18], social networks [52], fraud detection [26], and more.

Several frameworks have been developed to facilitate distributed GNN model development and deployment in static settings [46]. However, most of the graphs observed in the real world are dynamic or streaming [5]. For example, in social networks, new users join and existing users may update their profiles or interests, leading to node updates and additions. The rate of ingestion in a streaming graph can be high, such as 30K edges/sec in Alibaba graph [38]. In addition to streaming updates, it is also important to consider low-latency query settings in the design of GNN systems. The state-of-the-art methods typically follow the **ad-hoc** querying setting, where an external actor is expected to initiate the execution of a GNN query.

However, a rather unexplored setting, which is of high importance for latency-critical applications, is the **online** setting where queries are implicit to graph topology. Consider a dynamic social network where the system needs to autonomously identify and monitor potential spam accounts or harmful content creators without external prompt. This requires the GNN system to continuously analyze the graph for such behaviors, a task that cannot be efficiently handled by ad-hoc systems due to their reliance on periodic entire-graph inferences, thus making them ill-suited for near-real-time requirements. We bridge these gaps by introducing a distributed, end-to-end solution that is highly flexible and is designed to handle streaming graphs in online query settings.

Under streaming graph updates, previously generated representations become stale and require cascading GNN inference operations over a set of influenced nodes to maintain an up-to-date state. During inference (forward pass), the cascading nature of operations causes *neighborhood explosion* [5], which makes it challenging to maintain consistent throughput while keeping representations abreast of the updates. Identifying the set of influenced nodes (i.e., nodes affected by graph updates) is also resource-intensive, requiring a full L-hop traversal of the out-neighborhood. The irregular access patterns and data-skew on graphs also incur additional overheads, due to central (hub) nodes being involved in the majority of computations. This worsens the problems caused by 'neighborhood explosion' and introduces significant load-imbalance. Also, updates

*Currently with Fujitsu Research of Europe. This publication describes work performed at the University of Warwick and is not associated with Fujitsu.
†Also with Amazon Web Services. This publication describes work performed at the University of Warwick and is not associated with Amazon.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681961

can incur *concept drifts* over time, which deems periodic model re-training (backward pass) necessary to sustain model accuracy. All these challenges have to be tackled while keeping in mind the scalability and fault-tolerance of the system, which becomes more challenging in high-volume, streaming workloads. Hence, as the system requirements get closer to near-real-time, it becomes increasingly difficult to manage its latency, memory, and throughput constraints using traditional approaches.

Nowadays, we have libraries for GNN processing like DGL [51], Pytorch Geometric [10] and Pytorch Geometric Temporal [1]. These provide APIs for storing, manipulating and performing spatial operations on graphs. However, under the streaming workloads of large graphs, these fall short due to their inability to utilize a distributed environment. By contrast, the few libraries for distributed GNNs (e.g., DistDGL [58], AliGraph [61], DynaGraph [16]) are not built for streaming graphs or online queries. Some recent work considers temporal GNN models [20, 60], however, they all treat the graph as a static one with temporal edges and are mostly designed only for training workloads. These employ static data structures and static graph partitioning algorithms as a pre-processing step to distribute the graphs. For processing, they follow a synchronous, mini-batch execution model which requires transferring ego-graphs and their raw features to distributed machines to then perform local GNN computations. This violates the data locality principle and forces graph data to be migrated redundantly with every batch iteration, which can take up to 85% of the system's compute time [25]. Furthermore, no prior system considers the impact of external data-skews on GNN workloads. Retrofitting these features into such systems introduces further maintainability problems, and delayed predictions, rendering them unsuitable for latency-critical applications. Hence, there is a need for a holistic GNN system that incrementally operates on large graph updates while also efficiently tackling the **online** query settings.

To address the above challenges, we present D3-GNN - the first distributed and hybrid-parallel dataflow-based system for streaming GNNs. D3-GNN enables asynchronous, incremental GNN inference within a scalable dataflow pipeline. As node representations are fundamentally the building block for virtually all graph-learning tasks, the D3-GNN system is designed to maintain up-to-date node representations under streaming graph updates in an online fashion. This core capability allows D3-GNN to be versatile across different models and applications, in other words fundamentally task-agnostic. To build D3-GNN, we leverage Apache Flink[2], which helps us to develop an extendable and fault-tolerant system and tackle intrinsic challenges in event streams, such as exactly-once processing and handling late events.

At its heart, D3-GNN employs a novel design of unrolled computation graph, with each dataflow operator representing a GNN layer, facilitating both data and model parallelism via streaming graph partitioning and the separation of GNN layers across operators, respectively. To efficiently handle streaming graph updates, we pivot on node memory and streaming aggregators, which perform consistent, incremental updates in node states. Furthermore, we propose and analyze several windowing algorithms for tackling the neighborhood explosion and data-skew issues persistent in GNNs with only a minor latency trade-off. Lastly, while continual learning algorithms for GNNs is an orthogonal research problem, providing an effective system support for it in an inference-first pipeline solves commonly occurring problems of model staleness, data migration, and the need for periodic resource provisioning. Hence, although D3-GNN is primarily designed as an inference-first system, it also supports efficient distributed training.

We further enhance our system by solutions for efficient graph & model storage, stale-free training, feature replication, tensor management, nested iterations, and termination detection. The decoupled nature of D3-GNN allows independent parallelization of GNN layers and flexibility in modifying the dataflow egress depending on the specific use-case. For example, streaming node representations can be used to generate real-time predictions or act as a materialized embedding table that can be further queried. D3-GNN's model-agnostic approach, based on MPGNN, enables support for a wide range of common GNN models.

2 RELATED WORK

To the best of our knowledge, there is no other streaming dataflow solution for GNN computations tackling the online query execution model, either as a research or as an industrial offering. Existing systems are not well-suited for low-latency streaming graph learning and inference operations.

2.1 Streaming Graph Processing Systems

Over the past decade, a variety of graph management systems and algorithms have been designed for streaming graph analytics [6, 30, 34, 44]. These systems are optimized for running analytics algorithms, such as subgraph matching and subgraph counting, on such dynamic data [29]. However, none of these systems are designed to perform on graph learning tasks such as GNNs [8].

Recently, with the advent of streaming systems, dataflow-based graph systems have been developed. The Gelly-streaming[3] library, for instance, built on top of Apache Flink, tackles streaming graph algorithms such as finding connected components and bipartite matching. GraphX [15] aims to unify optimizations from specialized graph processing environments (e.g., Pregel) with general-purpose processing environments (e.g., MapReduce, Apache Spark). Other dataflow-based methods have been devised for incremental graph algorithms, such as Naiad [31], GraphTau [21], Tornado [42], and KickStarter [48]. None of these tackle graph learning and GNN computations in the streaming setting.

By contrast, D3-GNN provides a dataflow-based streaming graph system targeting GNN computations and applications. It enables incremental GNN computations while preserving primary aspects of streaming systems [3]. Note that studies on enhancing dynamic GNN training (e.g., edge events, continual learning, memory modules [28, 33, 40, 49]) are orthogonal to our work, and most can be implemented within our modular D3-GNN ecosystem.

2.2 Distributed GNN Systems

Extending from graph analytics systems, the recent advent of GNNs has brought attention to push-based distributed systems. These

[1] https://pytorch-geometric-temporal.readthedocs.io/
[2] https://flink.apache.org
[3] https://github.com/vasia/gelly-streaming

aim to provide data/model-parallel ML capabilities to message-passing models [7, 32]. NeuGraph [27] proposed an abstraction for expressing GNN pipelines with graph-specific optimizations. Here, backward gather functions are used to distribute the accumulated gradients on the backward pass, thereby supporting distributed parallel GNN training. D3-GNN proposes a streaming aggregator perspective with the help of a dynamic dataflow middleware.

To overcome communication overheads in large-scale training, sampling methods such as neighborhood sampling (e.g., GraphSAGE [17]) have been proposed. Motivated by this, recent developments of GNN systems like AliGraph [61], DistDGL [58], DistDGLv2 [59], AGL [56], and FlexGraph [50] employ distributed, mini-batch sampling operators to perform pull-based GNN computations. These collect (pull) sampled subgraphs and run data-parallel GNN on top of them. Similarly, P3 [12] introduced a dimension-based partitioning with layer-wise model distribution. Sancus [36] adaptively skips broadcasting to avoid communication in data-parallel GNNs. Recently, a few systems have appeared to tackle temporal GNN models by providing more efficient temporal sampling algorithms for such scenarios (TGL [60], T-GCN [20]). DynaGraph [16] proposed caching mechanisms for efficient dynamic GNN execution. However, it still tackles only ad-hoc queries and is primarily useful for training workloads.

The GraphTides [8] framework was introduced to assess graph-based streaming platforms. Within this framework, streaming graph processing encompasses the capability to manage dynamic graphs (both topology and feature updates) and execute real-time algorithms directly on the graph without external queries. This is achieved while meeting the demands of the streaming domain, such as fault-tolerance, high availability, and scalability. To the best of our understanding, D3-GNN stands out as the premier distributed GNN system adept at handling such workloads. While it might be feasible to partially mimic this use-case on the mentioned systems by periodically inferring the entire graph, the mini-batching in these models leads to uneven loads and postpones the latest inference results by the duration of the batch window at best. Simply put, these systems neither cater to streaming graphs as input nor are they tailored for latency-sensitive applications.

2.3 Streaming Graph Partitioners

Partitioning algorithms are widely used in distributed graph processing to enable data-parallelism for load balance and low communication. Low-latency partitioners are applied at the same time that the graph is being loaded into the cluster [11, 43, 45]. This online approach is well-suited to dynamic graphs where offline repartitioning of the entire new graph snapshot is inefficient. An experimental comparison across different applications with Apache Flink [1] shows that data-model-specific techniques (e.g., FENNEL [45] for vertex stream, HDRF [37] for edge stream) offer better communication performance while data-model-agnostic methods (e.g., hash) trade off data locality for better balanced workloads. Despite these studies on various graph algorithms (e.g., shortest paths, PageRank), there is no work that adapts the utility of streaming partitioners for distributed graph learning. We investigate the effect of streaming partitioning while producing node representations using GNNs on streaming graph data.

3 BACKGROUND

This section presents the background needed to cover how D3-GNN provides a fault-tolerant dataflow pipeline, and how it achieves distributed GNN inference and training on streaming graphs.

3.1 Graph Streams

Real-world graphs are often dynamic in nature, exhibiting changes in their topology as well as node and edge features over time. We denote a multi-modal graph as $G = (V, E, X_V, X_E)$ comprising nodes $v \in V$, edges $e_{u,v} \in E \subseteq V \times V$. Additionally, nodes and edges may contain some features. To simplify the presentation, we consider a single feature associated with each node, denoted by $x_v \ \forall v \in V$, and similarly consider edge features $x_e \ \forall e \in E$. We use edge streams to ingest topological data, while node features as feature stream. Nonetheless, the granularity of updates is not limited to this particular paradigm. For example, one can employ vertex streams (streaming nodes along with their local neighborhoods) jointly with a matching partitioner like Fennel [45]. More generally, each streaming event is timestamped and may be a create, delete, or update operation on a graph element (vertex/edge/feature/sub-graph).

3.2 Distributed Streaming Dataflow

To build D3-GNN, we leverage Apache Flink[4], a stateful stream processing system that started as a research project and is now widely used in the industry for processing data streams. Flink uses a pipeline of data transformation tasks, called operators, to consume input streams and emit output streams. Operators can be parallelized across threads and machines, with each parallel sub-operator instance performing local computations. Fault-tolerance is guaranteed through replayable source operators and intermittent checkpoints to reliable storage. States can be recovered from this storage and used for the re-scaling of partitions. Flink employs a variation of the Chandy-Lamport algorithm [4] for distributed checkpointing, ensuring that correctness is maintained.

Although Flink pipelines are typically represented as Directed Acyclic Graphs (DAGs), we extend the framework by introducing a novel way to add nested cycles to enable sending back the gradients and performing state replication. This supports fault-tolerant, iterative computations for more complex tasks as part of our solution. We enhance the expressiveness and flexibility of Flink and develop a set of novel modules to better handle a wider range of streaming use cases. Our end-to-end solution enables efficient and scalable processing of iterative streams, while maintaining fault-tolerance by including in-flight iterative events within checkpoints.

3.3 Graph Neural Network

In Graph Neural Networks (GNNs), the learning process often involves the generation of node (or edge) embeddings, which are used to perform downstream tasks such as node classification, link prediction, and similarity queries. The Message Passing GNN (MPGNN [14]) paradigm views the computation task for each node in a GNN layer as a message generation process along incoming edges, followed by an aggregation operation at the receiving node. The node then updates its representation, which is used by the next

[4] https://flink.apache.org

GNN layer. Below is a common MPGNN formulation where the GNN layer l generates embeddings x_v for layer l+1 at every node v:

$$m_e^{(l+1)} = \phi(x_u^{(l)}, x_v^{(l)}, x_e^{(l)}) \qquad \forall\ (u,v,e) \in N_{in}(v)$$
$$a_v^{(l+1)} = \rho(m_e^{(l+1)} : (u,v,e) \in N_{in}(v))$$
$$x_v^{(l+1)} = \psi(x_v^{(l)}, a_v^{(l+1)})$$

The messages m_e are generated along each incoming edge $e = e_{u,v}$ in the node's in-neighborhood $N_{in}(v)$, as some function of the features (x_u, x_v, x_e). All $|N_{in}(v)|$ messages at v are then aggregated into a_v, which is used in combination with its old feature $x_v^{(l)}$ to generate the new representation $x_v^{(l+1)}$ for layer l+1.

Adhering to the common message passing GNN (MPGNN) formulation, our approach supports a diverse range of concrete GNN models, such as Graph Convolutional Networks [23], GraphSAGE [17], Graph Attention Networks [47], and Jumping Knowledge Networks [54], through the variation of the MESSAGE (ϕ), AGGREGATOR (ρ), and UPDATE (ψ) components. Typically, ϕ and ψ are neural networks in themselves, whereas ρ is one of *Concat*, *Sum*, *Mean*, *Min*, *Max* functions. A multi-layer GNN is constructed by stacking these computations, and an output layer (implemented as a neural network) is used to generate predictions based on the task.

3.4 Graph Partitioning

Graph partitioning divides a graph's nodes or edges into disjoint subsets, or partitions, aiming to minimize 'cut-edges' or 'replicated-nodes'. This NP-Hard problem seeks to balance computing loads and reduce network communication. However, most existing algorithms focus on static graphs, making them ill-suited for dynamic, real-time inputs due to lengthy computation times.

In response, D3-GNN involves streaming graph partitioning that compute partitions in real-time as each graph event arrives. We separate logical from physical partitioning for enhanced system scalability. Utilizing HDRF, CLDA [39], METIS and Random streaming vertex-cut partitioners, we achieve data-parallelism by distributing incoming edge streams. Additionally, D3-GNN presents a feature-granular master-replica synchronization method to manage vertex replication among sub-operators.

4 METHODOLOGY

D3-GNN performs asynchronous and incremental inference, enabling the production of up-to-date node representations under streaming graph updates. In cases where re-training is necessary, it performs synchronous backpropagation in a distributed fashion, while avoiding stale states and eliminating bursty resource provisioning. Stale states are a result of outdated information during training, causing asymmetric forward and backward functions. This can occur due to asynchronous updates in graph topology and node embeddings, while backpropagation is taking place, and can negatively impact model accuracy. To be able to tackle this, current systems would maintain a training environment that pulls a static graph snapshot to perform backpropagation, which results in bursty resource provisioning and extensive data migration. Our approach eliminates the need for a separate training environment, reducing the computational burden and improving overall efficiency while providing full-graph, static training guarantees.

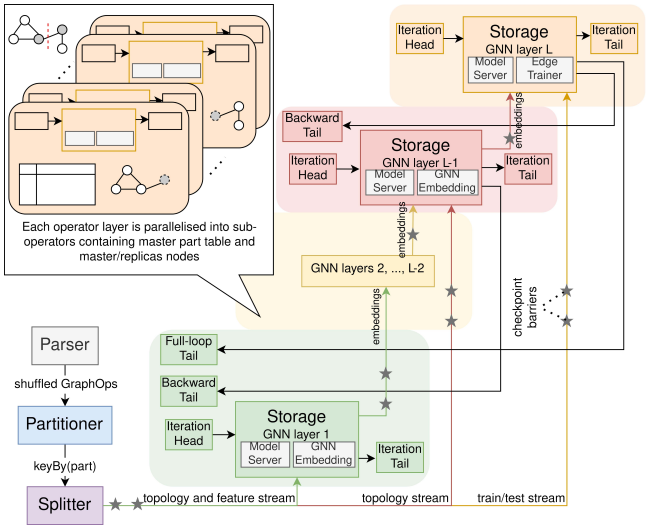


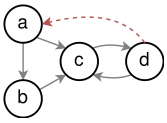
Figure 1: D3-GNN dataflow pipeline

4.1 System Overview

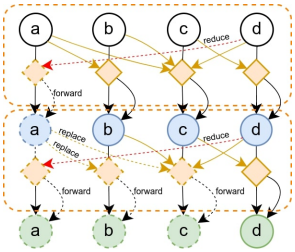
D3-GNN involves a dynamic dataflow pipeline, as depicted in Figure 1. This pipeline leverages a unified message format that supports create, update, and delete events for edges, vertices, and features. The process begins with a source **`Dataset`** parser. While we utilize temporal edge-list files to stream per-edge addition events and per-node features, in practice, any stream source, such as database logs, edge devices, or message queues, can be used. Coupling our system with Flink allows us to seamlessly integrate a broad range of such external data sources. After the events are generated, the streaming **`Partitioner`** consumes them, assigning and routing the events to their designated parts. This facilitates data-parallelism for subsequent operators. Before the GNN operators receive them, incoming events pass through a **`Splitter`**. This filter improves memory efficiency by ensuring that GNN layers do not process unnecessary events. We categorize events into three classes to cover the majority of inductive use-cases in GNNs:

- Topology data: Received by all GNN layers, except for the output layer when the task is node-level.
- Feature data: Only the first GNN layer receives this, as subsequent layers inherit their features from the preceding one.
- Train & test data: Solely received by the final output layer for training and reporting real-time metrics.

Subsequent GNN layers are distributed into their associated **`Graph Storage`** operators, facilitating model-parallelism. Furthermore, this design allows us to control the parallelism of GNN layers independently, thereby better accommodating explosive workloads. These operators incrementally store their graph partition as events arrive. Each sub-operator handles a subset of graph parts and houses the actual GNN layer. In addition, it includes a set of **`Plugins`** that monitor local graph updates and execute computations at a granular level, like individual feature updates. This design allows us to devise incremental GNN computations, ensuring that the graph partition and GNN code remain co-located, unlike in sampling-based systems. Such a configuration grants quick access to essential data



(a) Example graph with new edge d to a



(b) Unrolled computation graph with updates (dotted elements)

Figure 2: GNN Computation Graph

and provides flexibility for other applications. Moreover, `Graph Storage` features a tensor manager for rapid native memory management and supports iterations for master-replica synchronization and gradient feedback. We delve deeper into these optimizations in Section 5.

In this context, "feature" pertains to graph elements linked to their parent element (e.g., vertices or edges) and holds a value of an arbitrary data type. Embeddings, aggregators, and class labels all qualify as features. Some features might be designated as **halo**, signifying that they will not be duplicated even if their parent element is replicated.

4.2 GNN Inference

In order to maintain up-to-date node representations in an online fashion, the system must continuously track all influenced nodes I whose representations have become outdated due to new topological or feature update to the graph. This is necessary due to cascading effect of updates during model inference. For a GNN with L layers and an input graph having average in-degree δ_{in} and average out-degree δ_{out}, an edge addition is expected to influence $|I| = \sum_{l=0}^{L-1} \delta_{out}^{l}$ nodes. Computing the set of influenced nodes is $O(\delta_{out}^{L-1})$ as it requires fetching L-1 out-neighborhood. Updating node representations, in turn, requires retrieving their L-hop in-neighborhood (roots of computation graph) and performing forward pass on them. This in turn means constructing $|I|$ computation graphs each with δ_{in}^{L} source vertices (i.e., $O(\delta_{in}^{L})$ each). As such, the total cost of supporting a single edge update becomes $O(\delta_{out}^{L-1} + [\sum_{l=0}^{L-1} \delta_{out}^{l}] * \delta_{in}^{L})$. Consequently, when dealing with large graphs, performing low-latency inference can quickly become an impractical and difficult task.

In our approach, we avoid explicitly tracking influenced nodes or repeatedly pulling neighborhoods for constructing local computation graphs. Instead, as depicted in Figure 1, we treat the chained `GraphStorage` dataflow as an implicit computation graph, naturally partitioned by breadth (data-parallelism) and depth (model-parallelism), and enable incremental cascades based on changes on graph topology and features. In doing so, D3-GNN achieves a much lower cost $O(\delta_{out}^{L-1})$ of updating influenced node representations with a single edge addition.

4.2.1 ***Incremental aggregation.*** In MPGNN (Section 3.3), aggregators summarize the messages that arrive to the node from all of its in-neighbors. These are typically permutation-invariant functions such as sum, mean, and concatenation. We note that, such functions can be incrementally updated by maintaining a relatively small state using exact or probabilistic data structures. To formalize this approach, in D3-GNN, we develop **Aggregators** as instances of synopsis operations that are cached at each master node to maintain incremental computations. The states of `Aggregators` are updated by remotely invoking one of the following method interfaces at the master node:

$reduce(msg, count = 1)$ to add a new message
$replace(msg_{new}, msg_{old})$ to update a message
$remove(msg, count = 1)$ to delete a message

Our incremental formulation for the Aggregator is customizable, being contingent only on the restrictions of synopsis operators, as they have to be *mergeable, commutative, and invertible*. It can thus support any Update and Message neural network definitions, with their properties having no impact on the incremental functionality. Moreover, under massive graph updates, we do not require any graph locking mechanisms and allow many cascades to be simultaneously updating the system. Since the aggregators are permutation-invariant and feature updates follow causally consistent dataflow, the incremental model is **eventually consistent**.

Therefore, the tracking of influenced nodes and the retrieval of node neighborhoods occur automatically, by following the dataflow between operators. These are triggered by external updates at each layer of the GNN.

Most temporal GNN architectures make use of memory modules which are new embeddings that do not conflict with the above essential properties of the aggregators. This allows our model to be adaptable to temporal GNNs without compromising the integrity of its core functions. Integrating nonlinear recurrent units like LSTMs and GRUs, which diverge from the traditional aggregator paradigm, presents a more complex challenge. These units, by design, necessitate the retention of an extensive message history for each node. A permutation-invariant version of this approach would enable us to streamline the model by retaining only the most recent hidden state for each aggregator, aligning seamlessly with the operational protocols of our described D3-GNN interfaces. For instance, the process for replacing a message could be elegantly bifurcated into two distinct phases: initially, the model would apply the sign inverse of the old message to effectively nullify its impact, followed by the computation of the new message.

4.2.2 ***Streaming forward pass.*** This method describes pure streaming approach for inference, where the next layer representations are immediately updated by cascading through the computation graph as described earlier. The arriving edges cause Messages to be sent to destination aggregators. As the `Aggregators` receive updates, the algorithm generates up-to-date representations through `Update` function and forwards it to the next `GraphStorage` along the chain. Note that, we employ vertex-cut partitioning in our system, therefore some vertices are replicated and the corresponding

AGGREGATOR resides only with the master vertex (and not with replicas). This distributes edge-based computations across machines, achieving greater load balance.

To illustrate, consider Figure 2b, which unfolds the computation graph generated, based on MPGNN formulation, given the example graph in Figure 2a. The dotted lines represent edge addition event and the cascading computations taking place in the computation graph respectively. The illustrated computation graph represents layers sequentially from top to bottom, i.e., node representations are seen for input and two GNN layers. AGGREGATORS are diamonds. Orange arrows directed towards AGGREGATORS depict MESSAGES, and black arrows towards nodes are UPDATES. For example, the new edge from node d to a (Figure 2a) affects the representations for vertices a, b, and c. This can be traced by looking at affected leaf nodes (dashed borders) in computation graph.

Algorithm 1 illustrates the pseudo-code for streaming inference for a simplified MPGNN-style model (GraphSAGE) which is unimodal and does not contain edge features. $createAggregator()$ creates and attaches `Aggregator` to its vertex in all gnn layers. $msgReady()$ and $updReady()$ check if all the data dependencies are in place for MESSAGE and UPDATE functions to be computed. $reduce()$ and $replace()$ functions send `Remote Method Invocation` messages to the corresponding AGGREGATORS at the master. $forward()$ function computes the next layer representation x_u^{l+1} and sends it to the subsequent operator as vertex feature update.

Algorithm 1 Streaming Inference

Require: node u with feature $u.f$ and AGGREGATOR $u.agg$, out-neighborhood $(u, v, e) \in N_{out}(u)$, functions MESSAGE (ϕ), UPDATE (ψ)

function $addElement(u)$
  if $u.state() == MASTER$ **then** $createAggregator(u)$
function $addElement(e)$
  if $msgReady(e)$ **then** $v.agg.reduce(\phi(e))$
function $addElement(u.f)$
  if $updReady(u)$ **then** $forward(\psi(u.f, u.agg))$
  for all $N_{out}(u)$ **do** $v.agg.reduce(\phi(e))$
function $updateElement(u.f^{new}, u.f^{old}))$
  if $updReady(u)$ **then** $forward(\psi(u.f^{new}, u.agg))$
  for all $N_{out}(u)$ **do** $v.agg.replace(\phi(e^{new}), \phi(e^{old}))$
function $updateElement(u.agg^{new}, u.agg^{old})$
  if $updReady(u)$ **then** $forward(\psi(u.x, u.agg))$

The streaming setting can cause three possible bottlenecks: (i) Neighborhood explosion in GNNs can cause `Graph Storage` operators at deeper layers to receive higher workload, increased by a factor of δ_{out} with every additional layer. (ii) Vertices with high centrality scores (hub vertices, especially in power-law graphs) emit new features more frequently, hence can overwhelm the subsequent sub-operators. (iii) Changing external workload patterns (e.g., because of seasonality, real-world events) can concentrate graph updates in a narrow region of its topology.

4.2.3 ***Explosion Factor.*** Since our GNN layers are fully decoupled and our graph is only logically partitioned, we can vary the *parallelisms* of `Graph Storage` operators independently. Hence, to tackle GNNs' neighborhood explosion challenge, we introduce a new system hyper-parameter called '***explosion factor***' (λ). This enables us to vary the parallelism p_i of each individual `Graph Storage` operator, i.e., the number of sub-operators that perform the same task in a data-parallel manner. Namely, given an initial parallelism p and L layers of the GNN, we assign the actual parallelism for each `Storage` operator (layer) as $p_i = p * \lambda^{i-1}$ for $i \in [1, \ldots, L]$. This parameter must be selected considering the frequency of training, as even though the forward pass is always benefited by higher λ, because neighborhood explosion has reverse effect on layer-wise workload during backward pass for training.

4.2.4 ***Windowed forward pass.*** We propose **intra-layer** and **inter-layer** windowing to mitigate neighborhood explosion, data skews, and master node imbalances. Unlike the streaming algorithm (Algorithm 1), which executes the $forward$ and $reduce$ functions immediately, the windowing approaches introduce a time-based window that delays their execution.

Intra-layer windowing (which delays the $forward$ functions for each vertex) is especially beneficial for hub-vertices. It allows us to send a single, most up-to-date update for a batch of $forward$ requests. By doing this, we can optimize network usage and reduce cascades in the subsequent layer, thereby minimizing effects of neighborhood explosion.

Although our system utilizes vertex-cut partitioners to balance per-edge computations, the presence of nodes that receive updates only at master nodes can introduce additional skews. Thus, inter-layer windowing delays the emission of reduce messages for each destination node. It batches the corresponding edges, calculates a local, partial aggregation for each destination node, and emits a single reduce message summarizing the batched edges.

Algorithm 2 provides the pseudo-code for the abstract windowed forward pass. Depending on the $intraLayerWindow$ and $interLayerWindow$ functions, we propose three windowing algorithms. We employ timers to manage windowing with a 10ms coalescing interval, ensuring that timer threads are not overwhelmed.

In the **Tumbling Windowed** approach, each $forward$ node and $reduce$ destination are allocated a window of a specific duration. This naturally enables the batching of intra and inter-layer computations based on the frequency of the corresponding cascades during its interval. By adjusting the window interval, we can effectively control the latency overheads of this method.

Shifts in dynamic patterns can cause some edges to become highly active for a specific duration. For instance, a concert might lead to a sudden surge in merchandise sales. Such phenomena can introduce workload skews in sub-operators, even when using tumbling windowing. To address this, we propose **Session Windowing**. In this approach, the aforementioned functions are evicted after a certain, fixed period of inactivity. This algorithm is similar to the tumbling one, with the primary distinction being that adding a vertex already in a window further postpones its eviction time.

Nodes in the real world can exhibit varied and evolving frequencies. Therefore, the inactivity duration considered a "session" for a specific node is dynamic. To account for this, we introduce **Adaptive Session Windowing**, where session intervals are determined based on the windowed exponential mean of past frequencies. To enable low-storage computations of the windowed exponential

mean, we designed a thread-safe CountMinSketch that is periodically averaged.

Algorithm 2 Windowed Inference

Require: node u with feature x_u and AGGREGATOR agg, out-neighborhood $(u, v, e) \in N_{out}(u)$, $forwardBatch$ holding delayed forward vertices, $reduceBatch$ holding delayed reduce edges per dest vertex, functions MESSAGE (ϕ), UPDATE (ψ)

▷ Other functions same as streaming algorithm

function $addElement(e)$
  if $msgReady(e)$ **then**
    $e.delete()$
    $intraLayerWindow(e)$

function $forward(vertex)$
  $interLayerWindow(vertex)$

function $evictForward(timestamp)$
  $vertices \leftarrow forwardBatch.lessThan(timestamp)$
  $updates \leftarrow \psi(vertices.f, vertices.agg)$
  for $i \in [0, vertices.len)$ **do**
    $send(updates[i], vertices[i].master)$

function $evictReduce(timestamp)$
  $edges \leftarrow reduceBatch.lessThan(timestamp)$
  $edges.create()$
  $srcMessages \leftarrow \phi(edges.src.unique.f)$
  $reduceMsgs \leftarrow scatterAggregate(srcMessages, edges)$
  for $(dest, count, index) \in edges.groupby(\text{"}dest\text{"})$ **do**
    $dest.agg.reduce(reduceMsgs[index], count)$

function $onTimer(timestamp)$
  $evictReduce()$
  $evictForward()$

4.3 GNN Training

Aside from causing load imbalance, external workloads occasionally change the distribution of "true" representations. This phenomenon is more commonly referred to as "concept drift." To stay abreast of these changes, model re-training becomes necessary. In this section, we describe our approach in D3-GNN that allows for this re-training on the same cluster while preventing staleness.

The distributed training must be coordinated to ensure there are no inconsistencies when performing the backward pass during streaming graph updates. Gradients sent back through the computation graph will become invalid if the topology changes during this time. Furthermore, after the training is concluded, intermediate node embeddings and aggregators need to be recalculated to mirror the updated model.

To address this, we introduce a specialized, fault-tolerant **`Training Coordinator`** process within the job manager. This process oversees the entire GNN training life-cycle, which includes initiating and terminating the distributed training loop, computing epochs and batch sizes, and averting staleness.

In GNN training, we append an output `GraphStorage` operator following the final embedding layer. This operator captures the final node representation, true labels, and a loss function ($\mathcal{L}$). During job definition, the loss function is seamlessly integrated into the designated trainer `Plugin`. Concurrently, the true labels and node representations are introduced as streams, in line with the approach detailed in Section 4.1.

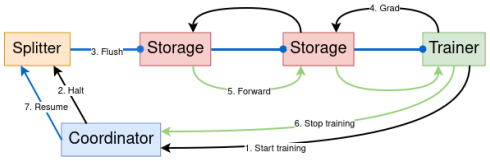


Figure 3: Distributed Training Overview

An overview of the sequence of events in distributed training is presented in Figure 3. The incoming stream is paused at the `Splitter`, allowing all messages to be processed through the pipeline. Once paused, the requisite epoch and batch counts are determined based on the volume of training data available. This is followed by distributed backpropagation, synchronization of model parameters, and a full-batch forward pass. Upon completion of the training, D3-GNN reactivates the `Splitter` and resumes its standard operation using the refreshed model. Subsequent portions of this section delve deeper into each stage.

4.3.1 ***Coordination of distributed training.*** The distributed sub-operators of the final layer need to coordinate to determine the start and end times for their training. To initiate this process, we employ a majority-voting mechanism, wherein the coordinator begins the training loop once more than half of the available output sub-operators signal the `StartTraining` command.

The decision to start training, made by the output sub-operators, could be either periodic (e.g., in D3-GNN, it is triggered when a pre-defined batch size is reached in the final `Storage` operator) or adaptive (e.g., based on test performance).

Upon entering the training mode, the coordinator halts the `Splitter` from consuming external updates. Thanks to its streaming design, such stoppage gradually cascades back to the stream source (`Dataset`), preventing memory bloating issues for prior operators. It is important to note that iterative messages still flow freely during the training phase.

The asynchronous pipeline implies that there might still be in-flight (unprocessed) events flowing through the pipeline. To flush out these remaining messages, the coordinator employs a termination detection algorithm as described in Section 5. Correctly flushing the pipeline, combined with the halt of external updates, ensures that staleness issues do not arise during backpropagation.

Lastly, before initiating the training loops, the output sub-operators share their training data sizes, aggregate them, and then determine a batch size. We choose epochs to be static during the definition of the pipeline.

4.3.2 ***Distributed backpropagation.*** Once the computation graph is frozen, backpropagation begins in each logical partition of the final (output) layer:

(1) Fetch the prediction layer inputs for each train label from the current batch (node embeddings for node-based tasks or source and destination node embeddings for edge-based tasks).
(2) Perform predictions and evaluate the loss function $\mathcal{L}$.

(3) Run the local backpropagation algorithm to generate model gradients as well as embedding gradients $\partial\mathcal{L}/\partial x_v^{(L)}$.
(4) Send $\partial\mathcal{L}/\partial x_v^{(L)}$ back to each corresponding master vertex in the previous layer.

Backprop is triggered (via broadcast instructions) in the previous layer operator only after all logical parts of the current sub-operator complete their job. Each logical part accumulates (per-vertex) the $\partial\mathcal{L}/\partial x_v^{(l+1)}$ for the layer that gets sent back to it. Once a broadcast instruction is received by a layer from all sub-operators, no more embedding gradients are expected. That layer is then free to perform its backpropagation in two phases. The first phase, within which computations can proceed asynchronously, is as follows:

(1) Compute $x_v^{(l+1)}$ for each accumulated vertex in the logical part.
(2) Perform local backprop (i.e., Jacobian-vector product) using stored Aggregator state $a_v^{(l+1)}$, vertex feature $x_v^{(l)}$, and accumulated gradients to get $\partial\mathcal{L}/\partial a_v^{(l+1)}$ and $\partial\mathcal{L}/\partial x_v^{(l)}$.
(3) Send $\partial\mathcal{L}/\partial x_v^{(l)}$ back to each corresponding master vertex.
(4) Send $\partial\mathcal{L}/\partial a_v^{(l+1)}$ to all replicas of the given vertex, along with $a_v^{(l+1)}$ for gradient computation (since replicas do not have them locally).

The second phase then proceeds (also asynchronously) as follows:

(1) Once all the Aggregators are received, compute the Messages $m_e^{(l+1)}$ from the locally available in-edges.
(2) Compute $\partial\mathcal{L}/\partial m_e^{(l+1)} : (u,v,e) \in N_{in}(v)$ from $\partial\mathcal{L}/\partial a_v^{(l+1)}$ and local Messages at the Aggregator.
(3) Continue the backpropagation by calculating the $\partial\mathcal{L}/\partial x_u^{(l)}$ and $\partial\mathcal{L}/\partial x_v^{(l)}$ using the above message gradients collected.
(4) Send gradients back to the corresponding master vertex.

Once the vertex gradients are received the process repeats until the first GNN layers is reached. We make use of the cached Aggregator and vertex feature states that had been calculated and stored during the last forward pass (before training was triggered). Such caching minimizes redundant communication and computations during training. Moreover, our incremental Aggregators calculate gradients using only locally available data and their stored state. Synchronous training phases also allow the use of *vectorization* to perform matrix operations efficiently in bulk.

4.3.3 ***Model synchronization and forward pass.*** In the first layer, instead of sending back gradients, the system starts the model update and forward pass cycle to recompute up-to-date embedding representations and Aggregator states. This procedure is similar to the streaming forward pass. However, since our graph is now static (external updates are halted due to buffering the incoming graph stream), we introduce several optimizations with layer-by-layer computations in three synchronous phases:

Phase 1 (Model Update). Since our model is distributed across sub-operators, the gradients and model parameters must be synchronized after training. Each distributed model runs its local optimizer (e.g., SGD, Adam, Adamax) to update its model parameters. Vertex embeddings are also updated if trainable (i.e., if x_v^0 are not received as input), which triggers their replica synchronization. Once completed, each sub-operator broadcasts its local parameters to other sub-operators, which compute the mean of the received values. Algorithm 3 provides pseudo-code for this process.

Algorithm 3 Model update

```
function updateModel
    W_i^+ = optimizer(W_i, ΔW_i)   ▷ In each logical part i = 1,...,P
    broadcast(W_i^+)
    W^+ ← collect()
    W_i = (1/P) Σ_{j=1}^{P} W_j^+   ▷ In each logical part i = 1,...,P
```

Phase 2 (Aggregate). Once the model is updated, we can safely continue re-building our computation graph. This involves computing Messages and performing *reduce*() at each Aggregator, as done in the inference case. However, since the graph is now static, separate synchronous phases for aggregation and update can avoid redundant Update messages. In this phase, we only update the Aggregators, without producing next layer embeddings.

The aggregate phase starts with each master vertex resetting its Aggregator state (usually to a zeros tensor). We perform a *batchReduce*() on all locally available in-edges, and send only the resulting *reduce*() message to the master Aggregator. As such, the number of *reduce*() messages sent per vertex is only proportional to its replica counts, not in-degree.

Phase 3 (Update). Once the model is guaranteed to be updated and all Messages reduced, all the local master vertex embeddings must be updated. The Update is invoked for the vertices and sent to the next operator, from where a new synchronous cycle begins. This continues up to the final layer. At this point, the `StopTraining` instruction is activated, prompting the `Splitter` to resume and transitioning the system back to inference mode.

4.4 Streaming Graph Partitioning

The above inference and training methodologies allow for incremental updates and caching within logical parts alongside master/replica synchronization steps. We now present our partitioning scheme to support the distributed execution of our hybrid-parallel pipeline. In particular, we discuss how we optimize the latency of a streaming partitioner operator for our purposes, and how this operator is used to assign as well as re-scale parts.

When distributing the workload, the `Partitioner` operator identifies the correct destination sub-operators by assigning `part` numbers to the incoming stream data.

4.4.1 ***Distributed partitioner logic.*** We build D3-GNN to utilize any streaming partitioning algorithm within its `Partitioner` operator. In our implementation, we utilize HDRF, CLDA and Random vertex-cut streaming partitioners. Distributing those requires a shared-memory model for storing partial degree and partition tables, which is not supported by Flink. Without it, a single thread needs to be allocated to the partitioner, which causes a significant bottleneck when scaling up the system. Hence, we develop a novel `Partitioner` operator to support correct, concurrent thread distribution for streaming partitioners. It distributes the main partitioning logic among arbitrary number of threads while having

synchronized access to the output channel. The latter is necessary to avoid corrupt data during network transfer, as output channels consume data in smaller units than the graph data being streamed. A vertex-locking mechanism is also developed for correctness, where edges with common vertices are assigned to their logical parts one at a time.

Our `Partitioner` maintains a master part table where we can store the first `part` that an element is assigned to. Because vertex-cut partitioning is used, it results in replicated vertices that are assigned to parts different from the first master vertex. The master part table enables replicas to sync and communicate with their masters from their respective `Graph Storage` operators. Algorithm 4 describes the pseudo-code for our streaming partitioner.

Algorithm 4 Streaming Partitioner

Require: *state*, *master_table*, *num_partitions*, *operator*
  part ← *assignPart(state, master_table, num_partitions, operator)*
  if *master_table*(*operator.element*) = ∅ **then**
    master_table(*operator.element*).*insert*(*part*)
  assignMaster(*operator.element*, *master_table*)
  operator.part ← *part*
  return *operator*

4.4.2 ***Re-scaling logical parts***. Assigning physical partitions alone does not allow flexible re-scaling of `Graph Storage` operators (e.g., if the number of physical partitions changes due to failure), nor does it support different parallelisms across the chained `Graph Storage` operators (e.g., to better cope with the exponential load induced by neighborhood explosion). To tackle this issue, we define the total number of available parts (*num_partitions*) to be the same as the maximum possible parallelism of the system (*max_parallelism*), while actually partitioning the graph events using *keyBy*(*operator.part*). In other words, the streaming `Partitioner` assigns only *logical parts* while the *physical part* is computed using a hash of the assigned logical part. As a consequence, multiple logical `parts` may map to the same sub-operator. Flink treats the logical parts (keys) in complete isolation; each part maintains its own context (state tables, timers, etc).

Operators store data (state) in two ways: **`Operator State`** stores data for a given sub-operator, which can be accessed by all elements arriving at sub-operator, while **`Keyed State`** stores data at the granularity of a unique key, and each arriving element can only access data that is assigned to its particular key. Upon re-scaling D3-GNN, `Operator State` is either randomly redistributed to new sub-operators, or broadcast (in entirety) to all remaining sub-operators to then perform recovery logic. `Keyed State`, however, is distributed to the new sub-operator containing that key. Our flexible mapping of keys to sub-operators allows for re-scaling of the physical partitions based on availability, and a fixed hash function (for logical to physical parts) guarantees fault-tolerant recovery. Hence, we are able to delegate the fault tolerance logic to Flink and ensure state redistribution and correct operation even under variable parallelisms.

When operator's *parallelism* gets closer to *max_parallelism*, some sub-operators may remain constantly idle due to never being assigned with any logical parts. To tackle this, instead of using Flink's default *Murmurhash*[5] algorithm on top of the key's *hashCode*[6] to compute physical parts, we develop Algorithm 5. Each operator is assigned at least one key, and logical parts are evenly distributed to operators depending on the current *parallelism*.

Algorithm 5 Compute physical part from logical

Require: *logical_part*, *parallelism*, *max_parallelism*
  key_group ← *logical_part* % *max_parallelism*
  physical_part ← *key_group* * *parallelism* / *max_parallelism*
  return *physical_part*

5 SYSTEM COMPONENTS AND OPTIMIZATIONS

This section describes the system-level functionalities and optimizations we introduce within D3-GNN to further facilitate streaming GNN pipeline.

5.1 Communication

To enhance communication efficiency within D3-GNN, we introduce custom serializers for frequently used data types, including vertex, edge, and remote method invocation. Additionally, for tensor serialization, we employ compression techniques. We also introduce a *selectiveBroadcast* primitive, which enables broadcasting an event to specific portions of the graph. For instance, replicating a vertex from a master to its replicas. This method circumvents the repeated serialization typically found in P2P communication, thus conserving computational resources.

To ensure fault-tolerant, iterative communication in D3-GNN, we use in-memory, SPSC, array-based queues. We apply a unified **`IterationHead`** logic to wrap operators with terminal feedback edges, which allows for thread-safe event consumption without racing with external ones. Events directed to specific head-operators are collected in separate **`IterationTail`** operators, which are co-located with their respective heads to maintain their queues. Our iteration model can handle nested, multi-layered iterations. Fault-tolerance is achieved by stopping queue consumption from heads and resuming it from tails after checkpointing in-queue messages.

5.2 Storage

D3-GNN involves a custom in-memory storage backend which takes advantage of unboxed data structures. The backend uses two adjacency lists (one for in-edges and one for out-edges) to store edges. A **task-manager-local storage** option is provided to avoid duplicating data in the cluster. This serves for storing vertex master tables and any other global data structures like CountMinSketch for Adaptive Windowing plugin.

Dealing with tensor garbage collection at large scales could result in excessive memory consumption and potential heap overflows, primarily because their memory is allocated outside the JVM. To counteract this challenge, we've established a per-thread **tensor**

[5]https://sites.google.com/site/murmurhash/
[6]Java Object method which generates an integer hash depending on implementation

cleaning module grounded on counter-based caching. As each sub-operator processes a unit-event, newly spawned tensors are temporarily stored in local cleaners. Each cleaner updates its counter based on the volume of new tensors and deallocates those surpassing a specific age threshold. This mechanism ensures a restricted count of active tensors, leading to optimized memory utilization.

5.3 Termination Detection

Introducing nested iterations in D3-GNN requires a valid distributed termination detection algorithm. For that, similar to `TrainingCoordinator` (Section 4.3), we develop a **`TerminationCoordinator`**. It periodically collects termination states from all iteration "heads" and proceeds with regular dataflow termination once all are ready to be terminated. The "heads" are ready to be terminated if they have not received events since the last collection and if they have not got scheduled timers. Last condition is necessary to avoid staleness in window-based inference methods (Section 4.2.4). This algorithm is also used to flush the pipeline in the case of GNN training (Section 4.3).

6 PERFORMANCE EVALUATION

Datasets. To test the performance of D3-GNN on streaming graphs, we use five datasets: `sx-superuser` [35], `reddit-hyperlink` [24], `stackoverflow` [35], `ogb-products` [5] and `wikikg90Mv2` [19]. The data is treated as an incoming stream of edge addition and feature update events to the graph, ordered by the edge timestamps. `sx-superuser` is temporal network of user interactions on a stack exchange website. It contains 1.4M edges and 200k nodes. `reddit-hyperlink` dataset contains an edge-list of directed subreddit mentions derived from the Reddit Social Network, with 286K edges and 36K nodes. `stackoverflow` dataset has question-answers and comments from the StackOverflow social network, with 63.5M edges and 2.6M nodes. `ogb-products` dataset is a co-purchasing network from Amazon with 62M edges and 2.6M nodes. `wikikg90Mv2` is a knowledge-graph dataset spanning 601M edges and 91M nodes. Edge deletion events are also supported in D3-GNN but are not present in the datasets evaluated.

Experimental setup and baselines. Experiments are executed on a Slurm cluster with 10 machines, where each machine contains Xeon E5-2660 v3 @ 2.6 GHz (20 cores/40 threads) and 64GB RAM. We use Apache Flink[7] and Deep Java Library[8] with PyTorch[9] as our primary ML framework.

The generated graph representations from the temporal network datasets in the experiments can be further used in a wide range of applications, including fraud detection, social modelling and recommendations by adding an output layer to the final representations. Hence, we focus on generating streaming node representations and build a distributed 2-layer, GraphSAGE model with 64 output dimensions using D3-GNN. As a baseline, we employ the distributed version of DGL [58] with our enhancements to emulate our incremental algorithm described in Section 4.2. Since DGL does not support dynamic edge additions, we label each graph edge with a timestamp. For each edge, we simulate topology updates in DGL via a sampling process. That is, our DGL baseline updates the representations for influenced nodes by only sampling from edges prior to its current timestamp.

We also provide partitioning-based performance evaluation with respect to HDRF, CLDA, METIS and Random vertex-cut partitioners. In HDRF and CLDA we use a balance coefficient of $\theta = 2$ and $\epsilon = 1$. Furthermore, we have empirically determined the value of $\lambda = 3$ as our explosion factor, based on runtime performance.

To evaluate the performance of D3-GNN, we evaluate the following: i) Scalability of different inference approaches and partitioners in terms of throughput, running times, network volume and load imbalance; ii) Comparison of inference and training with respect to DGL; iii) Impact of explosion factor on the runtime performance; iv) Analysing the latency overheads of the windowed inference. Note that D3-GNN and its streaming incremental aggregators produce the same embeddings as those from a static model executed on the equivalent final graph snapshot, therefore accuracy remains unaffected.

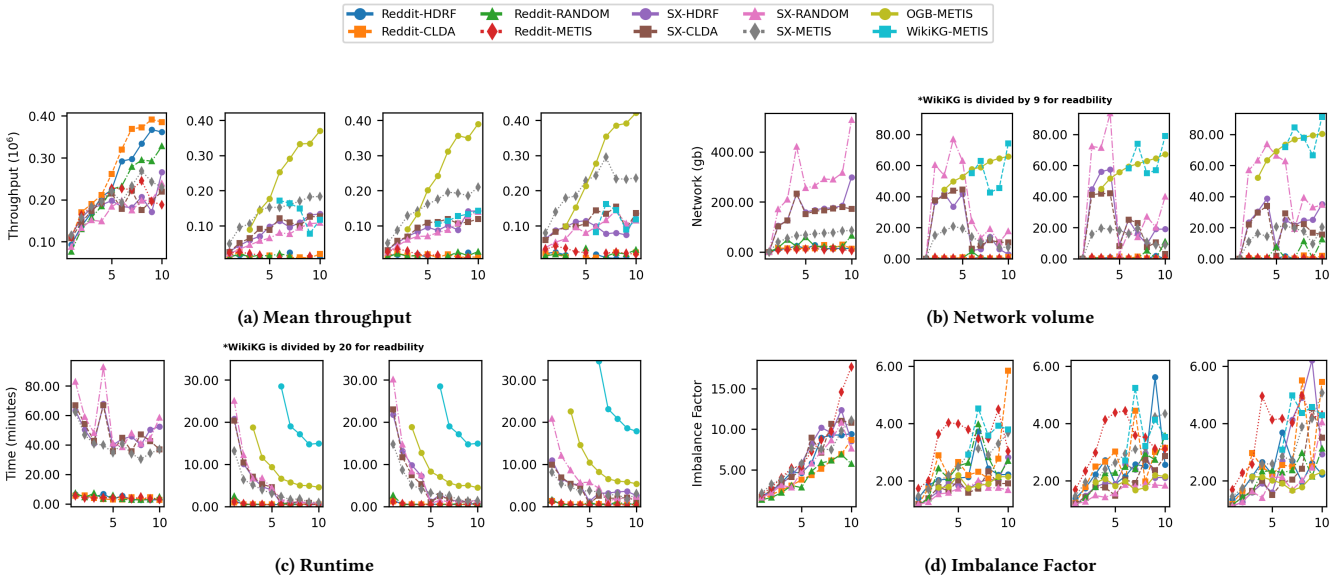
We examine the impact on throughput, total communication volume, latency, load imbalance, and run time when increasing the number of task managers available for allocation. Figure 4 provides introspective analysis of D3-GNN. Whereas, in Figure 5 we compare D3-GNN against DGL using `reddit-hyperlink`. Figure 6 details the impact of explosion factor on the runtime performance of the system. Lastly, in Figure 7, we evaluate the latency overheads of windowing on `sx-superuser`.

We compare 5 types of algorithms - Streaming and Windowed. In **D3-GNN Streaming** and **DGL Streaming**, influenced nodes are updated with each incoming edge. For D3-GNN, this approach corresponds to Algorithm 1 which generates new node embeddings by cascading computation graph updates. In the Windowed case, edges are processed after some delay (20ms except for `wikikg90Mv2` where delay is set as 10 seconds), that batches these updates. We have labeled the windowing algorithms in accordance with the ones in Section 4.2.4. Additionally, to compare against DGL with equal amount of batching, we include **D3-GNN WCount-2000** and **DGL WCount-2000**. These, instead process a specific number of edges in a fixed batch size, rather than enforcing a timer-based delay. In our evaluations, we process 2000 edges in a single batch. To ensure consistent throughput as the system scales, we set the window size of each distributed process at $2000/parallelism$.

The reported results are an average of three runs, and the systems were gradually scaled up from 1 to 10 task managers (x-axes were omitted to reduce space). To accurately reflect the performance of the system, we made sure that the entire pipeline was fully busy by congesting it first.

Scalability of streaming inference. To measure inference throughput, we calculated the average and maximum rates, across an operator's run time, of producing final layer representations. We observed that D3-GNN scales almost linearly with the number of machines for all algorithms when it comes to throughput performance (Figures 4a, 5a, 5b). However, we also observed that in the case of highly imbalanced graphs, such as `sx-superuser`, streaming algorithm can suffer at higher levels of parallelism. This is explained by high workload imbalance, which is also reflected in Figure 4d. When faced with such circumstances, adopting one of the windowing algorithms has been demonstrated to reduce workload imbalance by

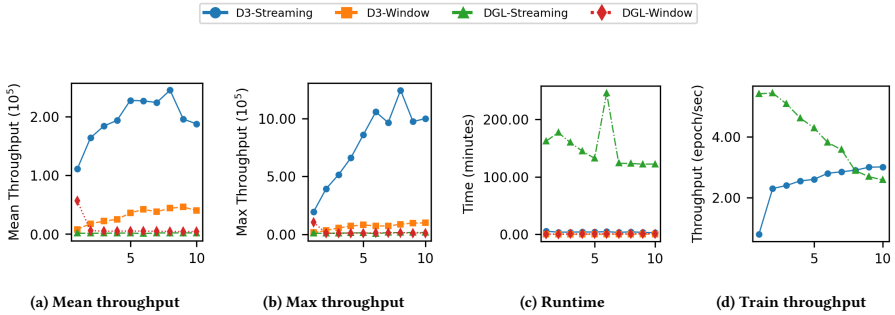
[7] https://flink.apache.org
[8] https://djl.ai
[9] https://pytorch.org



(a) Mean throughput (b) Network volume

(c) Runtime (d) Imbalance Factor

Figure 4: Scalability of inference algorithms: Streaming, Session, Sliding, and Adaptive.



(a) Mean throughput (b) Max throughput (c) Runtime (d) Train throughput

Figure 5: Comparing inference and training against DGL on `reddit-hyperlink`

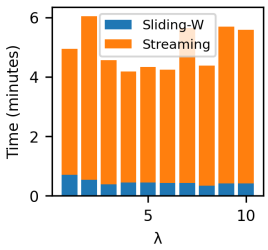


Figure 6: Explosion factor

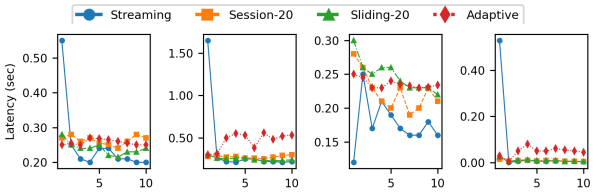


Figure 7: Scalability of mean, max, min, and standard deviation of latencies for ingesting 10k edges/sec from `sx-superuser`

nearly $x4$, resulting in enhanced scalability. We calculate imbalance factor by averaging the busy time for each sub-operator and then dividing the maximum by the average. Upon comparing our three windowing algorithms, we find that the adaptive approach yields higher throughput, while the Sliding and Session-based windowing methods are comparable.

Compared to DGL, D3-GNN demonstrates superior inference task performance, surpassing DGL by a factor of $x76$ on streaming, and by $x15$ on WCount-2000 tasks. Furthermore, we observe that DGL WCount-2000 encounters a significant performance drop when entering distributed mode with two or more machines. These findings are also reflected in the runtime metrics.

Running time efficiency. When dealing with bounded computations, we use runtime rather than throughput to accurately gauge scalability. To calculate runtime, we utilize the termination detection algorithm explained in Section 5.3.

Our study revealed sub-linear scalability in terms of runtime for D3-GNN's streaming algorithm. Notably, the runtime scalability of windowed approaches are **super-linear**. Additionally, for `sx-superuser` dataset, the streaming algorithm exhibited poor scalability on higher levels of parallelism. This was consistent with the inference throughput, underscoring the advantages of windowing algorithms. Figure 4c highlights that these algorithms can improve system runtime by a factor of almost 10, while resulting in steeper runtime curves.

Although, DGL WCount-2000 algorithm shows competitive running times in lower parallelisms, for increased parallelisms its runtime consistently worsens. On the other hand, DGL Streaming, which has sub-linear scalability, takes $25x$ of D3-GNN runtime. It is important to highlight the sudden runtime jump of DGL Streaming being run on 6 machines. This was persistent behavior concluded to be due to the partitioning strategy (METIS [22]) for that setting, and was resolved by replacing the partitioner. This further suggests that our asynchronous vertex-cut GNN pipeline provides greater resilience against incorrectly partitioned hub-vertices.

Results for `stackoverflow` also demonstrate the weak scaling to millions of edges, where the running time is found to decrease following similar trends to that of `reddit-hyperlink` (58 mins on 5 machines to 20 mins on 10 machines). The figures are omitted due to lack of space.

Communication volume. Communication operations are incurred due to replication when distributing tasks across a large number of processors. The experiments (Figure 4b) suggest significant reduction in communication volume (around $15x$) by employing millisecond-scale, windowing algorithms. These measure the volume of iterative communication in the second GNN layer in GB. Furthermore, while the streaming communication volume increases sub-linearly with system scaling, the windowing algorithm's communication volume is **consistent**. This improvement occurs because as the windowing algorithm is able to consume events more rapidly the delay between node updates shortens, hence decreasing the neighborhood explosion. This further supports the significance of our windowing methods for scalable operation. In practice, depending on the network infrastructure, communication volume can be tweaked by using larger windowing intervals.

Scalability of training. We conducted a performance comparison of D3-GNN and DGL when training a 2-layer GraphSAGE model for vertex classification (Figure 5d). We did not consider any batching or neighborhood sampling for either system. Despite D3-GNN being designed as a streaming, inference-first system, our results indicate that it performs competitively with DGL. In particular, D3-GNN is more effective at higher scalability, whereas DGL struggles to scale for high-communication workloads.

Inference latency. Our latency measurements involve determining the time interval between the ingestion of a graph event and the production of the corresponding node representation in the final layer. We included a throttling mechanism to the `Partitioner` to cap the ingestion rate at $10k$ $edges/sec$. This is necessary to avoid ending in a back-pressured (congested) state. According to our experiments (Figure 7), when the streaming algorithm can keep pace with the ingestion rate, which it failed to do only for parallelism 1, its latency is the lowest one amongst all algorithms and it shows minimum variance.

We also observed that adaptive windowing produces a comparable latency to the session one despite having an edge on throughput performance. Overall, D3-GNN achieves sub-second inference latency for at a rate $10k$ $edges/sec$.

Effect of partitioner. Figure 4 shows HDRF and CLDA surpass Random partitioning across all metrics. Consequently, Random partitioning underperforms, adversely affecting scalability metrics like network volume and runtime. Comparing streaming to METIS static partitioning, static partitioning shows slight superiority. Specifically, METIS significantly cuts network volume in streaming setups. Yet, adopting a windowed approach narrows this performance gap, particularly at higher parallelism levels where network volumes stabilize. This finding suggests that combining HDRF or CLDA with a windowing algorithm can match METIS's performance, a feat not achievable with Random partitioning.

Effect of explosion factor. Figure 6 demonstrates the beneficial impact of the explosion factor on the `reddit-hyperlink` dataset. While the theory suggests that the optimal explosion factor should align with the graph's average out-degree, practical outcomes reveal that partitioning schemes, load imbalances, and ingestion order significantly influence performance. For instance, the performance peaks at $\lambda = 2$ or $\lambda = 7$ can be attributed to load imbalances, as the mapping from logical to physical partitions was not contiguous. Nonetheless, incorporating a windowing approach significantly mitigates these issues, reducing the influence of such factors on overall performance.

7 CONCLUSION

In this work, we introduced D3-GNN, the first distributed, hybrid-parallel system optimized for GNN inference and training in the face of streaming graph updates. Diving into the relatively untouched domain of online query settings for GNNs, D3-GNN excels by incrementally maintaining node embeddings with minimal latency and ensuring fault-tolerant graph data management. We demonstrated the strong scalability of the system at higher parallelism, with high throughput and low runtime. Furthermore, D3-GNN introduced several algorithmic and systems optimizations, such as improving load balance by intra-layer and inter-layer windowing that reduced runtime by $x10$ and communication by $x7$ in our experiments. Significant improvements over potential alternative designs, including DGL, also highlight the contributions of D3-GNN in handling streaming GNN workloads that require near real-time processing. Lastly, the introduction of a stale-free, synchronous training algorithm by D3-GNN underscores its potential in the machine learning landscape, eliminating the need for separate training environments and addressing bursty resource provisioning challenge.

ACKNOWLEDGMENTS

This research is supported in part by EPSRC Doctoral Training Partnership award (Grant EP/T51794X/1) and Feuer International Scholarship in Artificial Intelligence at University of Warwick.

REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603.

[2] Asish Bera, Zachary Wharton, Yonghuai Liu, Nik Bessis, and Ardhendu Behera. 2022. SR-GNN: Spatial Relation-Aware Graph Neural Network for Fine-Grained Image Categorization. *IEEE Transactions on Image Processing* 31 (2022), 6017–6031. https://doi.org/10.1109/TIP.2022.3205215

[3] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 2651–2658.

[4] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[5] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/3292500.3330925

[6] Sutanay Choudhury, Lawrence Holder, George Chin, Abhik Ray, Sherman Beus, and John Feo. 2013. Streamworks: a system for dynamic graph search. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1101–1104.

[7] Gunduz Vehbi Demirci and Hakan Ferhatosmanoglu. 2021. Partitioning sparse deep neural networks for scalable training and inference. In *Proceedings of the ACM International Conference on Supercomputing*. 254–265.

[8] Benjamin Erb, Dominik Meißner, Frank Kargl, Benjamin A Steer, Felix Cuadrado, Domagoj Margan, and Peter Pietzuch. 2018. Graphtides: a framework for evaluating stream-based graph processing platforms. In *Proceedings of the 1st ACM SIGMOD joint international workshop on graph data management experiences & systems (GRADES) and network data analytics (NDA)*. 1–10.

[9] Qijing Feng, Hongmei Zhang, and Haoran Li. 2021. Hierarchical graph classification method based on graph pool topology learning. In *2021 IEEE 21st International Conference on Communication Technology (ICCT)*. IEEE, 1183–1187.

[10] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds* (New Orleans, USA). https://arxiv.org/abs/1903.02428

[11] Ioanna Filippidou and Yannis Kotidis. 2015. Online and on-demand partitioning of streaming graphs. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 4–13.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[13] Chen Gao, Yu Zheng, Nian Li, Yinfeng Li, Yingrong Qin, Jinghua Piao, Yuhan Quan, Jianxin Chang, Depeng Jin, Xiangnan He, and Yong Li. 2023. A Survey of Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions. *ACM Trans. Recomm. Syst.* 1, 1, Article 3 (mar 2023), 51 pages. https://doi.org/10.1145/3568022

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.

[16] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3534540.3534691

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

[18] Kai Han, Yunhe Wang, Jianyuan Guo, Yehui Tang, and Enhua Wu. 2022. Vision gnn: An image is worth graph of nodes. *Advances in Neural Information Processing Systems* 35 (2022), 8291–8303.

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

[20] Chengying Huan, Shuaiwen Song, Yongchao Liu, Heng Zhang, Hang Liu, Charles He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2023. T-GCN: A Sampling Based Streaming Graph Neural Network System with Hybrid Architecture. 69–82. https://doi.org/10.1145/3559009.3569648

[21] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the fourth international workshop on graph data management experiences and systems*. 1–6.

[22] George Karypis and Vipin Kumar. 1999. Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359-392. *Siam Journal on Scientific Computing* 20 (01 1999). https://doi.org/10.1137/S1064827595287997

[23] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[24] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 933–943.

[25] Haiyang Lin, Mingyu Yan, Xiaocheng Yang, Mo Zou, Wenming Li, Xiaochun Ye, and Dongrui Fan. 2022. Characterizing and Understanding Distributed GNN Training on GPUs. *IEEE Computer Architecture Letters* 21, 1 (2022), 21–24.

[26] Mingxuan Lu, Zhichao Han, Susie Xi Rao, Zitao Zhang, Yang Zhao, Yinan Shan, Ramesh Raghunathan, Ce Zhang, and Jiawei Jiang. 2022. BRIGHT - Graph Neural Networks in Real-Time Fraud Detection. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 3342–3351. https://doi.org/10.1145/3511808.3557136

[27] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 443–458.

[28] 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.

[29] Andrew McGregor. 2014. Graph stream algorithms: a survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[30] Jayanta Mondal and Amol Deshpande. 2012. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 145–156.

[31] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[32] Kabir Nagrecha. 2021. Model-Parallel Model Selection for Deep Learning Systems. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2929–2931.

[33] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-time dynamic network embeddings. In *Companion proceedings of the The Web Conference 2018*. 969–976.

[34] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.

[35] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.

[36] 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 (may 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614

[37] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM international on conference on information and knowledge management*. 243–252.

[38] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1876–1888. https://doi.org/10.14778/3229863.3229874

[39] Hadis Cheraghzade Rad and Reza Azmi. 2017. CLDA: vertex-cut partitioning for streaming power-law graphs. In *2017 9th International Conference on Information and Knowledge Technology (IKT)*. IEEE, 104–110.

[40] 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).

[41] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[42] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*. 417–430.

[43] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1222–1230.

[44] Yuanyuan Tian. 2023. The World of Graph Databases from An Industry Perspective. *ACM SIGMOD Record* 51, 4 (2023), 60–67.

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

[46] Jana Vatter, Ruben Mayer, and Hans-Arno Jacobsen. 2023. The Evolution of Distributed Systems for Graph Neural Networks and Their Origin in Graph Processing and Deep Learning: A Survey. *ACM Comput. Surv.* 56, 1, Article 6 (aug 2023), 37 pages. https://doi.org/10.1145/3597428

[47] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[48] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[49] Junshan Wang, Guojie Song, Yi Wu, and Liang Wang. 2020. Streaming graph neural networks via continual learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 1515–1524.

[50] 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* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3447786.3456229

[51] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[52] Zhouxia Wang, Tianshui Chen, Jimmy Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 1021–1028. https://doi.org/10.24963/ijcai.2018/142

[53] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[54] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*. PMLR, 5453–5462.

[55] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S Yu. 2021. Consisrec: Enhancing gnn for social recommendation via consistent neighbor aggregation. In *Proceedings of the 44th international ACM SIGIR conference on Research and development in information retrieval*. 2141–2145.

[56] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3125–3137. https://doi.org/10.14778/3415478.3415539

[57] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/53f0d7c537d99b3824f0f99d62ea2428-Paper.pdf

[58] 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)*. IEEE, 36–44.

[59] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU Training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 4582–4591. https://doi.org/10.1145/3534678.3539177

[60] 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 (apr 2022), 1572–1580. https://doi.org/10.14778/3529337.3529342

[61] 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, 12 (aug 2019), 2094–2105. https://doi.org/10.14778/3352063.3352127