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) inference and training on streaming graphs involve systems challenges for optimized latency, memory, and throughput, as well as algorithmic challenges to continuously capture the up-to-date state of the graph and the model. We present D3-GNN as a distributed, hybrid-parallel, streaming GNN system that performs embedding computations, predictions, and training in near real-time for massive event streams. The model updates are captured efficiently with every change in graph topology and features, and account for the cascading updates in the neighborhood aggregation step. D3-GNN involves custom streaming GNN aggregators which are feature-graph synopsis operators capable of incremental updates in a distributed fashion. The active replication policies and asynchronous pipeline ensure that throughput requirements are handled during distributed training, despite challenges of complex inter-dependencies in graph-structured data and evolving topologies of streaming graphs. D3-GNN is built on top of Apache Flink and is designed to benefit from the functionalities of the underlying fault-tolerant streaming middleware. Experiments on large-scale data streams and workloads demonstrate the scalability of D3-GNN with a high degree of parallelism (on 200 CPU cores in our experiments). D3-GNN enjoys high streaming throughput even on a single machine, about 100x that of DGL. A windowed forward pass is also proposed to counteract the exploding computational load of GNNs, which reduces run times by roughly 12x and message volumes by up to 220x at higher parallelism.

PVLDB Reference Format:
Rustam Guliyev, Aparajita Haldar, and Hakan Ferhatosmanoglu. D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

Social networks [46], item-products recommendation systems [49], physical systems [39], and biological networks [34] are among the many real-world examples of large graph data that are inherently dynamic with evolving topology and features over time. There is a wide range of data analytics and machine learning applications over such graph-structured data [7, 37, 50]. Graph Neural Network (GNN) models have been highly successful in generating node and graph representations, classification, and link prediction. However, even with state-of-the-art systems [5, 51], as the sizes of the graphs and workloads grow, it becomes increasingly difficult to meet their latency, memory, and throughput constraints [48]. Additionally, since real-world graphs are rarely static, GNN models need to be continuously re-trained to capture their up-to-date state. Even with pre-trained models, subsequent changes to graph topology cause cascading updates in node representations and model predictions for their neighboring set of influenced nodes. In the inductive setting, node representations obtained via forward propagation must be refreshed to make accurate predictions possible for new/unseen nodes; in the transductive case, all masked nodes performing aggregation on their neighborhoods must also maintain latest representations to keep the entire model updated.

Extensions to popular ML systems (e.g., PyTorch Geometric [9], Tensorflow GNN [20]) were not primarily designed for handling intricate inter-dependencies with streaming updates in distributed graphs. Existing libraries for distributed GNNs (e.g., DGL [52], P3 [11]) are built for static graph inputs. They typically employ static graph partitioning to load the entire graph and rely on mini-batching for transferring ego-graphs (induced subgraph of neighbors) and their raw features to the distributed system (local machines). This violates the data locality principle and forces graph data to be loaded redundantly with every batch iteration, which can take up to 85% of the system's compute time [21]. On the other hand, systems that aim to support dynamic graphs require separate functionality to periodically load the updated graph snapshot on which to produce predictions and perform training. This adds to the complexity, maintainability, and cost of the overall system, as well as causing delayed predictions that are infeasible for latency-critical applications. The problem of poorly utilizing data locality is exacerbated when dealing with heterogeneous or multi-modal graphs, as the more complex raw features introduce significant network traffic.

Keeping node representations abreast of the aforementioned updates requires a forward pass (involving message passing and aggregation steps) to update the embeddings of the entire set of influenced nodes, which can be expensive for massive event streams. The irregular access patterns on graphs also incur additional communication overheads, worsening the problems caused by 'neighborhood explosion' during the aggregation steps [2]. Thus, there is a need for a dynamic, parallel GNN system that can incrementally operate on massive streams in a distributed setting where loading

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the entire graph structure is avoided and fault-tolerance is provided with no loss in accuracy.

To address the above challenges, we present **D3-GNN - a distributed, hybrid-parallel, streaming GNN system** built on top of the Apache Flink dataflow framework. D3-GNN processes streaming graph events as a first-class citizen and supports asynchronous, incremental GNN operations in a **scalable, fault-tolerant, modular, dataflow pipeline**. Designing on top of Apache Flink helps to solve further challenges intrinsic to event streams such as exactly-once processing and handling late events. D3-GNN is hybrid-parallel, i.e., it is both *data-parallel* by using streaming graph partitioning algorithms to assign parts on-the-fly, and *model-parallel* by assigning one operator corresponding to each GNN layer and thus distributing the model across the dataflow pipeline.

D3-GNN continuously maintains generalized node representations under streaming graph updates, with final output layers that can utilize these representations for task-specific predictions. Each embedding layer is designed as a Flink operator that comprises generic Message, Aggregate, and Update functions covering the de facto standard Message-Passing Neural Network paradigm [12]. Intermediate results are stored in each operator's state in a fault-tolerant manner and updates are handled by passing the required GNN messages. Each GNN layer is encapsulated in a separate, independent operator, which, alongside the MPGNN paradigm, enables modular blocks of independent layers that asynchronously communicate to produce node representations. Besides an output prediction layer, D3-GNN provides a distributed lookup table of the up-to-date node representations via Flink Queryable State API. The learned representations can be retrieved from the system (cached aggregator states) without any neighborhood sampling or data movement, for use in models/tasks external to the ecosystem.

The **asynchronous pipeline of operators and incremental state updates** enable lower latency in generating representations under streaming updates on the graph topology and edge/node features. We define common GNN aggregators (*Sum*, *Mean*, *Concatenation*, and *Graph-Attention*) as streaming graph neighborhood synopsis operators that are mergeable, commutative, and invertible. These properties allow incremental updates to be computed in aggregator states consistently without accessing the entire L-hop neighborhood of their corresponding nodes, and without any dependence on the MPGNN function (Message, Update) properties. The synopsis states are actively cached to allow later on-the-fly computations of GNN embeddings and: (i) reduce the forward pass complexity for both training and inference as full neighborhood aggregation is avoided during each update, and (ii) reduce the backward pass communication since the cached aggregator state and embedding are locally accessed to generate gradients, as opposed to repeating the forward pass during training. Furthermore, we develop novel operators in D3-GNN that are essential to support GNN computations, which handle checkpointing and message flushing logic during iterative computations.

Finally, the **replication and storage policies** distribute the streaming dataflow pipeline with minimal communication overheads. Push-based active replication of cut vertices allows the master to synchronize updates with all replicas in other logical partitions, thereby allowing cascading updates to be triggered for all influenced nodes. D3-GNN manages the graph data itself; no external graph database is used. The corresponding part of the graph for each operator is stored locally in RocksDB[1] state backend. This also avoids relying on limited RAM space or external communication.

The experiments demonstrate that D3-GNN achieves its objectives, scaling to a high processor count with linear streaming throughput and keeping communication overheads bounded. Experiments illustrate the impact of various design decisions on throughput, latency, communication costs, and run time, such as windowing during the forward pass, streaming partitioning to distribute the stream, and controlling the neighborhood explosion for parallelizing deeper layers to more threads.

2 RELATED WORK

2.1 Distributed graph analytics systems

A number of frameworks have been developed to process iterative graph algorithms in a distributed environment. Google's Pregel [26] debuted the dataflow notion of *Bulk Synchronous Parallel (BSP)* message passing abstraction. This vertex-centric process is based on synchronously exchanging node messages until the algorithm converges. PowerGraph [13] introduced the *GAS model (Gather, Apply, Scatter)* for graph algorithms, extending Pregel with edge-granular computations to overcome the node degree imbalance problem for skewed power-law graphs. PowerGraph also supports asynchronous edge-wise computations, to mitigate the problem of 'straggler' tasks in the distributed pipeline; this also motivates the asynchronous behavior in D3-GNN. GPS [38] improves Pregel by supporting both vertex-centric and global computations, checkpointing for fault-tolerance, and dynamic re-partitioning strategies. X-Stream [36] argues for the use of edge-centric computations and data partitioning of sequentially ingested edge lists to exploit sequential (instead of random) memory access bandwidth; D3-GNN similarly partitions incoming edge streams to improve data locality. GraphX [14] aims to unify optimizations from specialized graph processing environments (e.g., Pregel) with general-purpose processing environments (e.g., MapReduce, Apache Spark), by adding graph processing support via basic dataflow operators to express *GAS model* computations within the Spark distributed system. By contrast, D3-GNN is built atop the Apache Flink framework to avail of its stream processing support. All the above are systems for distributed graph-based computations primarily tackle analytics workloads (e.g., PageRank) rather than GNN-based learning tasks, and do not deal with streaming graphs explicitly.

2.2 Distributed GNN systems

Several distributed processing systems have been developed for data/model-parallel ML [6, 29], and also for graph learning tasks. GraphLab [22] uses asynchronous distributed shared-memory abstraction for iterative ML algorithms. The Tux graph engine [47] uses *MEGA (Mini-batch, Exchange, GlobalSync, Apply)*, an extension of the *GAS model* for graph-based ML, and a vertex-cut approach to designate master/replica nodes. NeuGraph [23] proposed *SAGA abstraction (Scatter, ApplyEdge, Gather, ApplyVertex)* for better expressing GNN pipelines by recasting graph-specific optimizations

[1] http://rocksdb.org

into TensorFlow dataflow design. It uses backward gather functions to distribute accumulated gradients on backward pass, thereby supporting distributed parallel GNN training; D3-GNN applies a similar logic to streaming dataflow middleware for distributed GNNs.

To overcome the communication overhead of loading and processing an entire graph for training, sampling methods such as neighborhood sampling (e.g., GraphSAGE [15]) have been proposed. Large-scale training is supported by systems like AliGraph [53] and DistDGL [52] with the help of mini-batch sampling operators. These employ distributed graph storage with parallel mini-batch training on CPU/GPU clusters. DistGNN [27], another extension using the DGL [45] interface (like DistDGL), optimizes for full-batch training using a shared memory, which reduces communication overheads with a vertex-cut graph partitioning strategy and delayed aggregate functions. ROC [17] instead utilizes dynamic programming for parallel GNN training. Recently, Sancus [32] was devised which adaptively skips broadcasts to avoid communication in data-parallel GNNs. All of these systems achieve reduction of network communication via graph partitioning and optimizing data locality while maintaining computational balance. However, these systems are all designed for static graphs. Moreover, the above models reduce communication by mini-batching and only computing GNN backpropagations periodically, however, such batching causes spiky loads and delays the computation of the latest inference results by the length of the batch window (in the best case). In other words, none of these solutions are designed to handle streaming graphs as input, nor do they support latency-critical applications, whereas D3-GNN supports evolving graph topology and features and boasts scalable performance in terms of throughput, latency, and communication oveheads.

2.3 Stream processing systems

Stream processing systems (e.g., Apache Storm, Apache Spark Streaming, Apache Flink) are designed for processing independent data streams rather than graph learning. There is no streaming dataflow platform for dynamic graph learning. A few methods have been devised for iterative graph algorithms on dynamic graphs. Naiad [28] executes cyclic dataflow tasks for iterative, incremental, low-latency algorithms. GraphTau [16] is a time-evolving graph processing dataflow framework built on top of Spark that uses graph snapshots and incremental computations, alongside checkpoints for fault tolerance and speculative re-execution of straggler tasks. Tornado [40], built atop Storm, is designed for real-time iterative analysis with bounded asynchronous behavior using data dependencies to ensure correctness over evolving graph streams. Non-recoverable edge deletions invalidate the intermediate incremental results, thus KickStarter [43] defines node trimming approximations to tackle them; D3-GNN supports incremental addition, deletion, and update events without needing any such additional considerations. D3-GNN provides streaming primitives to support GNNs with incremental computations while supporting fault-tolerance and latency-critical applications. Studies on enhancing GNN models (e.g., dynamic updates, edge event streams, continual learning, memory modules [24, 30, 35, 44]) are orthogonal to our work, as most can be implemented within the modular D3-GNN ecosystem.

2.4 Streaming graph partitioners

For distributed processing, partitioning algorithms are widely employed to enable data-parallel computations with load balance and low communication. Low-latency streaming partitioners aim to perform partitioning at the same time that the graph is being loaded into the cluster [10, 41, 42]. 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 shows that data-model-specific techniques (e.g., FENNEL [42] for vertex stream, HDRF [33] for edge stream) offer better communication performance while data-model-agnostic methods (e.g., hash) trade off data locality for better balanced workloads [1]. Despite these studies on various graph analytics workloads (e.g., shortest paths, connected components, PageRank), there is no work that tests the utility of streaming partitioners on graph-based learning models. In particular, we examine the effect of streaming partitioning on GNN training and inference performance, with a focus on producing high quality node representations from streaming graph data.

3 BACKGROUND

This section briefly presents the background needed to cover how D3-GNN achieves distributed GNN inference/training on streaming graphs via a fault-tolerant dataflow pipeline.

3.1 Graph Streams

Real-world graphs are often dynamic in nature, displaying both a topology that evolves with time as well as node/edge features that update 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 $x_e \ \forall e \in E$. To represent topological or feature updates on the graph, we assume that the data is ingested in the form of an event stream (streaming vertices along with their local neighborhood, or streaming edge lists). Each such timestamped event may be either an add, delete, or update operation on a graph element (vertex/edge/feature).

3.2 Streaming Dataflow Systems

Traditional batch-processing systems suffer from poor latency when faced with streaming input, thus streaming systems have been developed to handle such data. Apache Flink is a state-of-the-art streaming dataflow system that is stateful, scalable, fault-tolerant, and provide exactly-once semantics with event-times [3].

A streaming dataflow is a pipeline of data transformation tasks, also referred to as 'operators', that consume input streams and emit output streams. The pipeline starts at one or more 'data-source' operators and ends at one or more 'data-sink' operators. While these pipelines are typically Directed Acyclic Graphs (DAGs), we also introduce ways to add cycles to support iterative computations in a fault-tolerant manner for some complex tasks as part of our solution. Operators are parallelizable across threads and machines, with each parallel 'sub-operator' instance performing local computations.

Apache Flink guarantees fault-tolerance by having 'replayable' source operators and taking intermittent 'checkpoints' to a reliable storage (e.g., HDFS[2], S3[3]). States can be recovered from the storage and assigned to re-scaled partitions as necessary. Flink uses a variation of the Chandy-Lamport algorithm [4] for distributed checkpointing, where the source sub-operators broadcast 'checkpoint barriers' that never overtake any 'in-flight' messages in the pipeline. Only upon receiving barriers successfully from all its immediately preceding sub-operators does a given sub-operator begin its own broadcast of barriers.

3.3 Graph Partitioning

Graph partitioning is used to enable data-parallel processing by splitting a graph into smaller sub-graphs. The approaches usually involve identifying sets of 'cut edges' or 'cut vertices' that can be removed to create disconnected components (i.e., partitions or parts). Two main constraints are desirable for the GNN setting: (i) **Balanced partitions**: Graph data, and the computational tasks/workload associated with every node, should be split evenly across processors (physical parts), e.g., during message aggregations cascading through the graph. D3-GNN achieves data-parallelism by distributing the incoming graph with data locality preserved through streaming partitioning. (ii) **Fewer cut edges/vertices**: Communication/replication overheads should be minimized, e.g., inter-partition message exchanges (in edge-cut partitioning) or synchronization of node representations across master/replicas (in the vertex-cut case). The sub-operators in D3-GNN communicate between master and replicas for feature and model synchronization.

A k-way partition $P_k = V_1, V_2, \ldots, V_k$ of the original graph G is well-balanced if $|V_i| \leq (1+\epsilon)|V|/k$ for all parts $V_i \in P_k$ where $|V_i|$ is the size of each part and ϵ is the error threshold.

P_k reduces communication overheads if it minimizes the global cost associated with cut edges/vertices. In the edge-cut case, this is generally expressed as the total weight of all cut edges. For vertex-cut, 'replication factor' $(\sum_{i=1}^{k} |V_i|/|V|)$ is often used.

3.4 Graph Neural Network

GNN models typically first learn graph representations, i.e., node or edge embeddings, that are then used for a downstream task, e.g., node classification, link prediction. As per the Message Passing GNN (MPGNN) paradigm [12], the computation task for a GNN layer at each of its nodes can be viewed as message generation along each incoming edge along with an aggregation operation at the receiving node, after which the node updates its representation for use by the next 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:

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

[2]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[3]https://aws.amazon.com/s3/

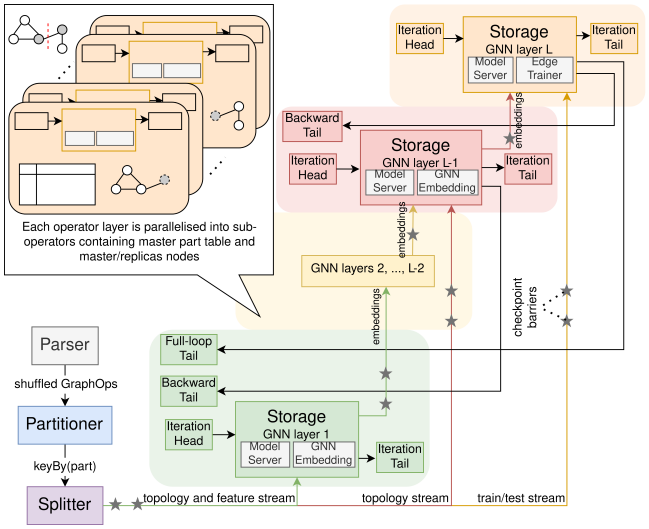


Figure 1: D3-GNN dataflow pipeline

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 $l+1$.

In D3-GNN, a concrete implementation of any GNN embedding layer can therefore be obtained by defining these MESSAGE (ϕ), AGGREGATE (ρ), and UPDATE (ψ) functions. Typically, ϕ and ψ are neural networks in themselves, whereas ρ is one of *Concat*, *Sum*, *Mean*, *Min*, *Max*, *LSTM*, or *Attention* functions. A multi-layer GNN is constructed by stacking these computations. In D3-GNN, this is achieved by chaining GNN layer sub-operators following the message-passing path of destination nodes until L-hop is reached. Note that D3-GNN avoids bottlenecks from potential neighborhood explosion by using asynchronous computations.

4 D3-GNN DESIGN

In this section we present the D3-GNN system components and describe the asynchronous forward and backward pass operations.

4.1 Overview of Operators

The overall process comprises a distributed dataflow pipeline of operators that transform and store data and compute up-to-date node representations incrementally under streaming graph updates and perform on-the-fly inference/training asynchronously. 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, thus can only be used by `KeyedStreams`, and each arriving element can only access data that is assigned to its particular key. Below are the descriptions of operators present in the D3-GNN pipeline (illustrated in Figure 1).

4.1.1 ***Parser (GraphOp).*** A parser operator is needed to translate different data source formats (e.g., ProtoBuf[4], XML[5], JSON[6]) into a unified message type for ingestion. We define this type in a `GraphOp` class that encapsulates the operations to be performed on the graph and is the primary unit of communication for the subsequent D3-GNN operators. Each `GraphOp` is composed of a `GraphElement`, `part`, and `communication type`:

- **`GraphElement`** represents the element – such as vertex, edge, feature, or Plugin – to be stored, passed as messages, or processed by the subsequent operators. These are described in Section 4.2.
- **`Part`** is an integer number representing the logical partition that a specific `GraphOp` is directed to. The actual physical part (sub-operator) is determined after hashing (see Section 4.3).
- **`Communication type`** is either `P2P` (point-to-point between specific `parts`) or `Broadcast` (sent to all of the logical `parts`). While most messages are `P2P`, `Broadcast` type is useful during the backward pass (i.e., exchanging gradients, synchronizing the model).

4.1.2 ***Partitioner.*** This operator manages the streaming graph partitioning logic. It uses a local dynamic summary of the graph seen so far and assigns `parts` to the incoming `GraphOps`. D3-GNN treats the first assigned `part` as the **`master part`** and subsequent ones as **`replica parts`**. In this way, communication is possible between `Storage` operators that maintain a partitioned graph by replicating and synchronizing vertices and their desired features. Apart from partitioning the graph, the `Partitioner` operator also routes the workload to the relevant logical `parts`. On receipt of an external node feature update, it is the job of the `Partitioner` to identify the `master part` of its corresponding node and assign it to this `GraphOp`.

4.1.3 ***Splitter.*** Different subsets of the external data are required at different points along the GNN pipeline. The `Splitter` operator acts as a filter to eliminate memory inefficiencies by ensuring that a given `Storage` operator only receives necessary `GraphOps`. We identify three types of data splits: The first (input) GNN layer receives the graph topology (which includes data about the `master parts` of vertices) and external features that are necessary to generate messages for aggregation. For intermediate (hidden) layers, `Splitter` delivers only the graph topology without attached features, to enable forward propagation through neighborhoods. The final (output) layer receives the training/testing data streams. `GraphOps` delivering such data (e.g., true labels, existing links, negative labels, etc) are exclusively sent to this last GNN layer since it requires the actual outputs for learning.

4.1.4 ***Storage and Plugins.*** The `Storage` operator interacts with Flink State Backend and updates its local graph snapshot with the incoming `GraphOps`. It can store vertices, edges, and heterogeneous features. Additionally, each `Storage` operator maintains a list of attached `GraphElements` called **`Plugins`**, that it interacts with through callbacks. `Plugins` are immediately notified of changes to `Storage` via these callbacks and can respond with their own implemented logic (see Section 4.2). In our implementation, each `Plugin` processes a layer of the GNN model. We create multi-level GNN models by chaining multiple `Storage` operators containing respective GNN embedding and model serving `Plugins`.

4.1.5 ***Iteration Head and Iteration Tail.*** Since Flink currently does not support stream iterations with checkpointing that accounts for in-flight messages, we design new operators to ensure consistency and fault tolerance guarantees. These new operators facilitate stream iterations by co-locating, i.e., placing in the same JVM (Java Virtual Machine), the head and tail operators of two sub-operators that require iterative communication between them. The `GraphOps` are then transferred through a shared, in-memory, MPSC (multi-producer single-consumer) queue. `GraphOps` that are sent to the `Iteration Tail` get published to the queue which are then asynchronously consumed by its paired `Iteration Head`. Such design eliminates the costly overheads of serialization between the iteration operators. We introduce 3 main types of tail operators: `Iteration Tail` sends messages within the same operator, for replication or aggregation. `Backward Tail` sends gradients backwards in the pipeline, during the backpropagation phase of training. `Full-loop Tail` sends messages back to the first layer operator (e.g., positive edges after training, or features in case of auto-regressive GNN models).

4.2 Graph Elements

`GraphElements` are objects that are involved in the graph processing pipeline. Similar to ORMs (object-relational mappings), `GraphElements` are used to interact with the `Storage` operators, and are the base class from which we can derive more specific functionality, specifically the `Edge` and `ReplicableElement` classes. `Vertex`, `Feature`, and `Plugin` classes are further derived from `ReplicableElement`. Except `Plugins`, these are all stored in `Keyed State`. All of these are detailed below:

4.2.1 ***GraphElements.*** Each `GraphElement` instance has a `type`, `id`, and a list of attached `Features`. Together, `type` and `id` are assumed to be compositely unique identifiers for a `GraphElement`. Along with data, `GraphElements` also contain CUD (Create, Update, Delete) logic and trigger corresponding `Plugin` callbacks upon successfully writing changes to the `Storage`.

4.2.2 ***Edges.*** Edges are not replicated by the partitioner and hence are regular `GraphElements`.

4.2.3 ***ReplicableElements.*** These are extensions of `GraphElements` that additionally contain synchronization and replication logic. In order to communicate with their masters, `ReplicableElements` store their `master part`. For some use-cases, the replicas do not need access the `Features` stored in their masters, but instead the replicas and master merely need to know the existence of each other for synchronization. For instance, as we will later describe, our GNN aggregators are stored in master vertices, and need to only receive messages from replicas. In order to support such logic, `ReplicableElements` include an additional Boolean **`halo`** flag. Halo elements (`halo=1`) only exist in their `master parts` and are not actively replicated. Instead, upon creation, an empty stub is sent to replicas through which they communicate with their master. Masters do need to know parts they are replicated in, hence the list of replicated parts is also a halo `Feature` that is used by the master

[4]https://developers.google.com/protocol-buffers
[5]https://www.w3.org/standards/xml
[6]https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

for synchronization. In this manner, the replicas can alter the array (e.g., to delete themselves) while avoiding storing data redundantly.

4.2.4 ***Vertex.*** Vertices are replicated by the vertex-cut partitioner.

4.2.5 ***Feature.*** These can store an arbitrary value. We define `Features` as either *standalone* or *attached* to some other `GraphElement`. Standalone `Features` are used to store global features about the graph or sub-graph, and act as regular `ReplicableElements`. Attached `Features`, instead, are bound to the life-cycle of their parent element. That is, they share the same replication pattern as their parent elements and get deleted upon their deletion. `Features` may even contain nested sub-features. Such semantics permit multi-modal, heterogeneous graphs to be represented with finer granularity, allowing individual feature updates to generate new messages and refresh node representations. Additionally, this allows for more specific master-replica synchronizations in multi-modal graphs, i.e., there is no need to sync an entire `GraphElement` if only a subset of its `Features` are updated.

4.2.6 ***Plugin.*** Various GNN tasks involve different computations at each layer, which are performed by `Plugins` in D3-GNN: `ModelServer`, `StreamingGNNEmbedding`, `WindowedGNNEmbedding`, `GNNEmbeddingTrainer`, `EdgeTrainer`, and `VertexTrainer`. `Plugins` are added to the `Storage` operator through dependency injection during the job start-up time, to ensure that computation is always local to data. `Plugins` are stored in the `Operator State` with one instance for each sub-operator. Ultimately, the specific combination of `Plugins` chosen on job start-up define the responsibilities and processing logic of that `Storage` operator. Each `Storage` operator is standalone and can have different `Plugins`, e.g., the first few layers can produce streaming updates while the last one performs windowing on those updates. Such decoupling and modular code avoids redundancy in implementation and provides the ability to replace parts of the `Storage` operator logic as desired. `Plugins` react to callbacks to perform some computation. Algorithms 3 and 4 demonstrate the functionality of some out of the following callbacks available in D3-GNN: *addElementCallback*(), *updateElementCallback*(), *deleteElementCallback*(), *onTimer*(), *onOperatorEvent*().

4.3 Graph Partitioning

When distributing the workload, data is transferred with the help of the `Partitioner` that identifies the correct destination sub-operators. That is, `Partitioner` assigns integer `part` numbers to `GraphOps`. We replace Flink's default round-robin partitioning by specifying which field of the data to use as a key to use for partitioning. Here we present the distributed partitioner logic and how the logical parts are re-scaled.

4.3.1 Distributed partitioner logic. D3-GNN is built to support any partitioner. In our implementation, we utilize HDRF [33] with streaming vertex-cut graph partitioning that replicates high-degree vertices for better load balance [13]. Distributing the HDRF partitioner requires a shared-memory model for storing partial degree and partition tables, something currently unavailable in Flink. Without it, a single thread needs to be allocated to the HDRF partitioner, which causes a 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 `GraphOps`. Furthermore, we develop vertex-locking mechanisms for correctness. That is, edges with common vertices are assigned to their logical parts one at a time. Note that for shuffled graph streams (streams not generated via BFS or DFS traversals), we find that non-locking partitioning does not significantly impede the partitioning quality (load balance and replication factor).

Our `Partitioner` maintains a **`master part table`** where the `ReplicableElements` can store the first `part` that the element is assigned to. This helps replicas to sync and communicate with their masters upon arriving at `Storage` operators. Thus the replicated vertices used in vertex-cut partitioning to minimize communication overheads are managed using `ReplicableElements`. Algorithm 1 describes the pseudo-code for the streaming partitioner logic.

Algorithm 1 Streaming Partitioner

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

4.3.2 Re-scaling logical parts. Assigning physical partitions alone does not allow flexible re-scaling of `Storage` operators (e.g., if the number of physical partitions changes due to failure), nor does it support different parallelisms across the chained `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 `GraphOp` 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 its keys (i.e., our logical parts) in complete isolation. That is, each part maintains its own context (state tables, timers, etc). 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. Thus, using a `KeyedStream` ensures that intermediate results are stored without inconsistencies and offers a wider range of Flink storage and operator primitives, such as: *Keyed State, Keyed Windows, Joining, etc.* Having a 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,

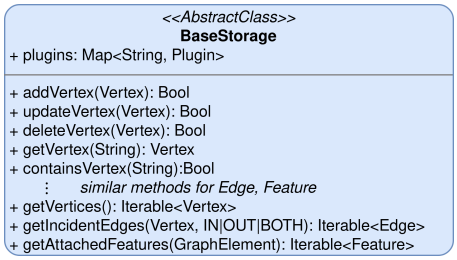


Figure 2: BaseStorage UML Diagram

thereby ensuring 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*[7] algorithm on top of the key's *hashCode*[8] to compute physical parts, we use Algorithm 2. Each operator gets assigned at least one key, and overall, logical parts are evenly distributed to operators depending on the current *parallelism*.

Algorithm 2 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*

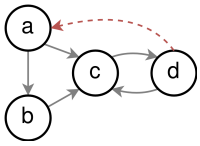
4.4 Storage and Serialization

The graph `Storage` class within D3-GNN is necessary to incrementally construct/destruct a graph from streaming `GraphOps` and to interact with Flink State Backend to enable fault tolerance. Having efficient data serialization is crucial as it decreases both memory consumption and network traffic between various operators.

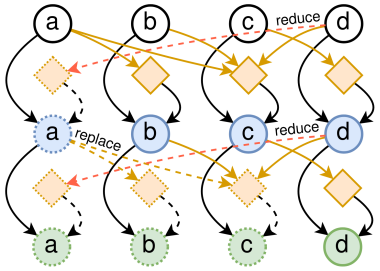
4.4.1 Flink State Backend interface. We define a common **`BaseStorage`** interface (Figure 2) to easily extend and add custom `Storage` logic in D3-GNN. It manages the serialization/deserialization of graph elements present in the partitioned subgraph and interfaces with Flink State Backend to maintain fault tolerance. The `BaseStorage` abstraction is introduced to have support for various compressed graph storage representations (i.e., CSR, STINGER [8], LLAMA [25]). However, Flink State Backend is currently limited to *HashMapStateBackend* or *EmbeddedRocksDBStateBackend*. The former stores smaller graphs efficiently in JVM Heap memory while the latter handles larger graphs using slower local disk storage.

4.4.2 Data serialization. By default, Flink uses its custom type serializers when dealing with primary data types or POJOs (Plain Old Java Objects), while depending on the Kryo[9] library otherwise. Although Kryo fallback is generalizable and easily applied, it has proven to be significantly slower than the built-in Flink serializer [18]. To improve performance and memory footprint, we design

[7]https://sites.google.com/site/murmurhash/
[8]Java Object method which generates an integer hash depending on implementation
[9]https://github.com/EsotericSoftware/kryo



(a) Example graph with new edge d to a



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

Figure 3: GNN Computation Graph

custom serializers for `GraphOps` and `GraphElements`, where we use bitmasks and lower-range data types where necessary as well as implement recursive serialization for nested `Features`. Moreover, as machine learning libraries store tensors outside of JVM heap (i.e., in machines' native memory), serialization in this case needs careful attention. Interaction with native memory is provided by Java ByteBuffers class, while we implement 2 variants for the tensor serializer: raw and compressed. In the first variant, the raw tensor byte array is sent along with its shape array and data type. The second variant additionally applies LZ4 compression algorithm[10] to the tensor data array.

4.5 GNN Inference

4.5.1 Aggregation in computation graph. Figure 3 illustrates how the computation graph is generated based on MPGNN formulation (Section 3.4). Given the example graph in Figure 3a having new edge connection arriving as shown (red dotted directed edge), the breakdown of the computation graph is as shown in Figure 3b, unrolled for 2-hop GNN operations. 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.

To support dynamic graph updates, the model must keep track of all influenced nodes (I) whose representations get outdated because of new topological updates arriving to the graph. For example, the new edge from node d to a (Figure 3a) affects the representations for vertices a, b, and c. This can be traced by looking at affected leaf nodes (dashed borders) in the unrolled computation graph illustrated in Figure 3b.

In traditional non-streaming systems, for a GNN with L layers and an input graph having average in-degree δ_{in} and average out-degree δ_{out}, the cost to update node representations would quickly become intractable. Computing the set of influenced nodes is $O(\delta_{out}^{L-1})$ and constructing $|I|$ computation graphs each with δ_{in}^{L}

[10]https://github.com/lz4/lz4

number of source vertices and cost of $O(\delta_{in}^{L})$ leads to an overall cost of $O(\delta_{out}^{L-1}+[\sum_{l=0}^{L-1}\delta_{out}^{l}]*\delta_{in}^{L})$. Within D3-GNN `Plugins`, we instead use caching to perform incremental computations by defining Aggregators as instances of synopsis operators. Each node's Aggregator receives one of the following Messages from each incoming edge: $reduce(msg)$ to add a new message, $replace(msg_{new}, msg_{old})$ to update a message, or $remove(msg)$ to delete a message. Consider, for example, the edge update (dotted red edge) in Figure 3a that adds a completely unseen Message to the computation graph. The Aggregator for node a has to be reduced with this new message from d, as per the two new Messages in Figure 3b (dotted red arrows to Aggregators). An Aggregator maintains an internal summary of the Messages seen so far, and stores the processed value in its `Storage` operator as a `halo Feature` attached to the corresponding `Vertex`. Any change in Aggregator state (dotted diamond) is carried out using the CUD logic of the `GraphElement`, and triggers a cascading Update to the next layer node embedding (dotted black arrow to node), which is changed using its CUD logic. This node subsequently sends Messages to its neighboring aggregator `Features` in the computation graph. In the example, the modified Aggregator for a updates its (dotted circle) embedding, which updates Aggregators of b and c following its out-edges. However, since we have already reduced Messages for those Aggregators, they are now replaced. Note that topology updates due to edge/node addition/deletion are streamed by the `Splitter` to reflect in relevant `Storage` operators of the GNN model. So the second GNN layer will also reflect a reduced Message from d to the aggregator of a. Hence, in D3-GNN, the cost of updating influenced node representations with a single edge addition is $O(\delta_{out}^{L-1})$.

Our incremental formulation for the Aggregator is only contingent on the restrictions to the synopsis operators (mergeable, commutative, and invertible), and can support any Update and Message neural networks function definitions. That is, the properties of those functions have 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 Aggregator function is permutation invariant and `Vertex Feature` updates are causally consistent, the incremental model is eventually consistent.

4.5.2 Streaming/windowed forward pass. In D3-GNN, each GNN layer is a separate `Storage` operator responsible for storing incoming features, doing aggregation, and outputting next layer embeddings. This makes for *model-parallelism* across the `Storage` operators and *data-parallelism* across their sub-operators. We implement the following `Plugins` for streaming GNN inference:

- **StreamingGNNEmbedding** incrementally updates the next layer embeddings by following the cascading computation graph.
- **WindowedGNNEmbedding** additionally employs timers to output next layer embedding only if node session has terminated, i.e., if a particular node has not received new updates for a specified duration.

`GNNEmbedding Plugins` are used within the `Storage` to encapsulate each embedding layer, in either streaming or windowed manner, and are decoupled for ease of adding deeper GNN layers. This `Plugin` runs Message and Update based on changes happening in `Storage`, and induces `GraphElement` CUD logic to update the next layer embedding `Vertex Feature`. `Vertex Features` are defined as non-halo, hence are actively replicated among replica nodes. `Edges` thereby use locally available `Features` to generate their Messages. Aggregators only update the `Vertex master parts` and use the results in Update function, thus are not replicated.

Algorithm 3 contains pseudo-code for **`StreamingGNNEmbedding Plugin`**. Note that this illustrative example is for a simplified MPGNN-style model (GraphSAGE) which is uni-modal and does not contain edge features. $createAggregator()$ attaches `Features` to master nodes (for trainable embeddings in the first layer, and Aggregators in all GNN layers). $msgReady()$ and $updReady()$ check if all the data dependencies are in place for Message and Update functions respectively. $reduce()$ and $replace()$ functions send messages to the corresponding Aggregators. $forward()$ function computes the next layer embedding x_u^{l+1} and sends it to the subsequent operator.

Algorithm 3 Streaming Inference

Require: node u with feature x_u and aggregator agg, out-neighborhood $(u, v, e) \in N_{out}(u)$, functions Message (ϕ), Update (ψ)

function $addElementCallback(u)$
  if $u.state() == MASTER$ **then** $createAggregator(u)$

function $addElementCallback(e)$
  if $msgReady(e)$ **then** $v.agg.reduce(\phi(e))$

function $addElementCallback(x_u)$
  if $updReady(u)$ **then** $forward(\psi(x_u, u.agg))$
  for all $N_{out}(u)$ **do** $v.agg.reduce(\phi(e))$

function $updateElementCallback(x_u^{new}, x_u^{old})$
  if $updReady(u)$ **then** $forward(\psi(x_u^{new}, u.agg))$
  for all $N_{out}(u)$ **do** $v.agg.replace(\phi(e^{new}), \phi(e^{old}))$

function $updateElementCallback(u.agg^{new}, u.agg^{old})$
  if $updReady(u)$ **then** $forward(\psi(x_u, u.agg))$

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

We introduce a new system hyper-parameter called 'explosion factor' (λ) to vary the parallelism p_i of each individual `Storage` operator. Namely, given an initial parallelism p and L GNN layers, we assign the actual parallelism for each `Storage` operator (layer) as $p_i = p * \lambda^{i-1}$ for $i \in [1 \dots L]$. To tackle neighborhood explosion (which has reverse effect on layer-wise workload during backward pass), this parameter must be selected based on the frequency of training, even though high λ benefits the forward pass.

In order to address the other bottlenecks, we develop a **`WindowedGNNEmbedding Plugin`** which does not immediately forward the vertex updates to the next operator. Instead, it uses a timer to

forward updates only if the node has not received new updates for W milliseconds. Algorithm 4 provides pseudo-code for the forward function and $onTimer()$ callback that are unique to this `Plugin`. This windowing delays cascades from highly active vertices until such time that they have a lower workload. Note that we use timer coalescing of 200ms to not overwhelm timer threads.

Algorithm 4 Windowed Inference

function $forward(vertex)$
  $updateTime \leftarrow currentTimeMs + W$
  $timerTime \leftarrow ceil(updateTime/200) * 200$
  $pendingList.update([vertex.id, timerTime])$
  $startTimer(timerTime)$
function $onTimer(timerTime)$
  for $pendingVertex \in pendingList$ **do**
    if $pendingVertex[1] \leq timeTime$ **then**
      $output(pendingVertex)$

4.6 GNN Training

For training, the last GNN layer needs labeled data delivered by the `Splitter` (in case of supervised learning) and a loss function ($\mathcal{L}$) when there is a specific task. To this end, we define the following `Trainer Plugins` to initiate and carry out the distributed backpropagation and synchronization of the model:

- **GNNEmbeddingTrainer** manages the distributed training of a single embedding layer. It receives $\partial\mathcal{L}/\partial x_v^{(l+1)}$ per participating node, does backward pass and sends back $\{\partial\mathcal{L}/\partial x_u^{(l)} \mid u \in N_{in}(v) \cup v\}$ for the previous `GNNEmbeddingTrainer` to consume.
- **EdgeTrainer** stores the loss function and begins distributed training for an edge classification/regression task. It incrementally matches edge labels to node representations and attempts to start training once the pre-defined batch size gets filled.
- **VertexTrainer** similarly starts distributed training for a vertex classification/regression task.
- **ModelServer** manages the model storage (weights and biases for one embedding layer, parameters for Message and Update functions) and has primitives for gradient synchronization.

The distributed training needs to be coordinated to avoid any inconsistencies due to performing backward pass while having streaming graph updates. The gradients sent back through the computation graph will be invalid if the topology changes in the meantime. Moreover, once the training is finished, intermediate node embeddings and aggregators will need to be rebuilt to reflect the updated model. The high-level process for distributed training is shown in Algorithm 5. The `Splitter` halts the incoming stream while waiting for all messages to flow through the computation graph. Distributed backpropagation takes place, followed by synchronization of model parameters. Finally, a full-batch forward pass on the graph is used to recompute intermediate values, before resuming the stream. These steps are detailed below.

4.6.1 Coordinating trainers. Each distributed `EdgeTrainer` or `VertexTrainer` plugin must coordinate when to begin and end the backward pass, for which we extend Flink's **`OperatorCoordinator`** interface. These special objects live outside of the main Flink dataflow (in the JobManager) and can directly talk with each sub-operator via `OperatorEvent` messages. We create the **`TrainingCoordinator`** class by adding support for communication with all operators across the GNN chain starting with `Splitter` and ending with `Output Storage` operator.

Algorithm 5 Distributed Training

function startTraining
  $stopExternalUpdates()$
  while $hasInFlightMessages()$ **do** $wait()$
  $model.backward(); model.update(); model.forward()$
  $resumeExternalUpdates()$

`OperatorEvents` are handled by the $onOperatorEvent()$ callbacks when received by a `Trainer Plugins`. `TrainingCoordinator` waits to receive special `StartTraining` message from all individual sub-operators of the final `Trainer` plugin before sending the `StartTraining` to the `Splitter`. The decision on starting training could be made periodically (e.g., in D3-GNN, it is triggered when a pre-defined batch size is attained in the final `Storage` operator) or adaptively (e.g., based on test performance).

Upon entering training mode, the `Splitter` starts buffering incoming `GraphOps` into a file and broadcasts the received `StartTraining` to subsequent operators in the pipeline. `Trainer Plugins` similarly continue broadcasting this upon receiving them from all previous operators. This enables all in-transit messages to be flushed from the stream, since Flink messages are not allowed to overtake each other. However, since our graph is partitioned and replicated, there might still be iterative messages remaining in the `Iteration Heads and Tails`. To flush these, in `GNNEmbeddingTrainer`, `StartTraining` messages are broadcast three times within its own sub-operators before emitting to the next operator. The length of three represents the longest chain of iterative messages possible in the GNN embedding layers ($replica \xrightarrow[\text{Sync request}]{} master \xrightarrow[\text{Sync}]{} replica \xrightarrow[\text{reduce}]{} master$). Note that for `WindowedGNNEmbedding Plugin`, we also fire the timers preemptively to flush all the up-to-date embedding updates. When the `StartTraining` message is received at the last `Output Trainer Plugin`, the backpropagation begins.

4.6.2 Distributed backpropagation. Once the final `Trainer Plugin` is sure that the computation graph is frozen, it begins the backpropagation in each logical partition:

(1) Fetch the prediction layer inputs per each train label from the current batch (node embeddings for `Vertex Trainer`, or source and destination node embeddings for `Edge Trainer`)
(2) Perform predictions and evaluate the loss function $\mathcal{L}$
(3) Run 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

Backpropagation for each embedding layer is handled by the `GNNEmbeddingTrainer Plugin`. It maintains a **GradientCollector** per logical part, which is a standalone, non-replicated `Feature` that accumulates (per-vertex) the $\partial\mathcal{L}/\partial x_v^{(l+1)}$ that are sent back to it. Similar to 'checkpoint barriers', `Trainer Plugins` broadcast back 'train barriers' to trigger backprop in the previous operator only

after all logical parts of the current sub-operator complete their job, and backprop continues until the first layer. Once the 'train barrier' is received by a layer's `GNNEmbeddingTrainer Plugin` from all sub-operators, no more embedding gradients are expected. The embedding layer performs its own backpropagation in two synchronous phases (within steps 1–4 computations can proceed asynchronously, and also within steps 5–8):

(1) Compute $x_v^{(l+1)}$ for each accumulated `Vertex` in the local `GradientCollector`
(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 `replica parts` of the given `Vertex`, along with $a_v^{(l+1)}$ for gradient computation (since these halo stubs do not have them locally)
(5) Once all the Aggregators are received, compute the Messages $m_e^{(l+1)}$ from the locally available in-edges
(6) 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
(7) 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
(8) Send gradients back to corresponding master `Vertex`

As can be seen, we make use of the 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.

4.6.3 Model synchronization. In the first layer, instead of sending back gradients, `Trainer Plugin` starts the model update and forward pass cycle to recompute up-to-date embedding `Feature` and Aggregator states. This procedure is similar to streaming forward pass, with synchrony maintained by thrice broadcasting a special message (as in Section 4.6.1). However, since our graph is now static (external updates are halted in `Splitter`), we introduce several optimizations with layer-by-layer `GNNEmbedding Plugin` 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., x_v^0 are not received as input), which triggers replica synchronization. Once completed, each `ModelServer Plugin` broadcasts its local parameters to other sub-operators, which compute the mean of the received values. Algorithm 6 gives a pseudo-code for this process.

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 streaming 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.

Algorithm 6 Model update

function updateModel
  $W_i^+ = optimizer(W_i, \Delta W_i)$ ▷ In each logical part $i = 1, \ldots, P$
  $broadcast(W_i^+)$; $W^+ \leftarrow collect()$
  $W_i = \frac{1}{P}\sum_{j=1}^{P} W_j^+$ ▷ In each logical part $i = 1, \ldots, P$

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. The *batchReduce*() result is not stored in any internal state, hence it can be used from `halo` replica stubs as well. Note that 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. Update is invoked for the vertices and sent to the next operator, from where a new synchronous cycle begins. This continues until the last `Output` layer, where end of training (`StopTraining` message) is triggered to restart the stream. When stream is restarted `Splitter` operators first emit their local buffers and then continue their normal execution.

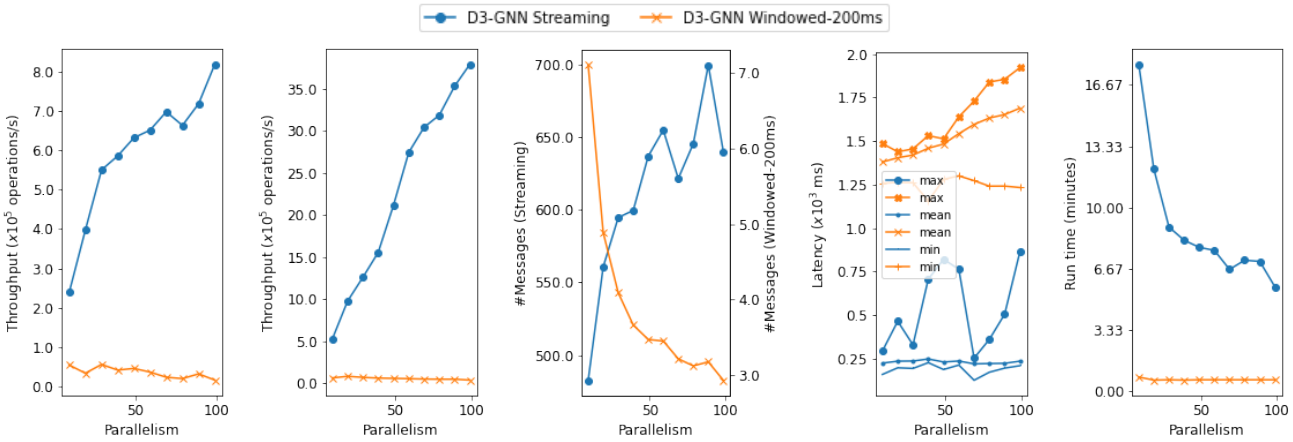
5 PERFORMANCE EVALUATION

To evaluate the performance of D3-GNN and provide insights on distributed streaming GNN inference and training, we evaluate the following: (i) Variation in system performance with different `Plugins` when increasing the number of compute nodes (i.e., strong scaling); (ii) Impact of different partitioners and their settings; (iii) Impact of *explosion factor* (a coefficient to reflect neighborhood explosion). We 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.

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[11] and Deep Java Library[12] with PyTorch[13] as our primary ML framework.

For the experiments, we build a distributed 2-layer GraphSAGE model using D3-GNN, with the layers containing 64 and 32 vertex features respectively. For the vertex features, we use a Mean aggregator. As one of the baselines, we employed DGL[45] and implemented the algorithm described in Section 4.5.1 which computes the updated representations of the influenced nodes for each new edge by sampling the 2-hop neighborhood. We note that DistDGL[52]) does not provide API support for dynamic graph edge additions.

[11] https://flink.apache.org
[12] https://djl.ai
[13] https://pytorch.org



(a) Mean throughput **(b) Max throughput** **(c) Communication volume** **(d) Latency** **(e) Runtime**

Figure 4: Performance comparison of D3-GNN Streaming and D3-GNN Windowed on Reddit ($\lambda = 3$)

For distributing D3-GNN tasks, we use HDRF vertex-cut partitioning algorithm with balance coefficient $\theta = 8$ unless otherwise specified, and also include results for Random partitioning. We also use an explosion factor (λ) in our distributed task allocations to tackle variable workloads at each layer due to neighborhood explosion. That is, when we say *parallelism* $= x$ of the system, the second layer's parallelism is λx, the third layer's is $\lambda^2 x$, and so on. We use the LZ4 compression algorithm for tensor serialization and *HashMapStateBackend* for all the following experiments.

Datasets. To test the performance of our system on streaming graphs, we use two temporal network datasets: Reddit [19] and StackOverflow [31]. Reddit dataset contains an edge-list of directed sub-reddit mentions derived from the Reddit Social Network, with 286K temporal edges and 36K nodes. StackOverflow data has question-answers and comments from the StackOverflow social network, with 63.5M temporal edges and 2.6M nodes. The data is processed in sequence and treated as an incoming stream of edge addition and feature update events to the graph. Edge deletion events are also supported in D3-GNN but are not present in the datasets evaluated.

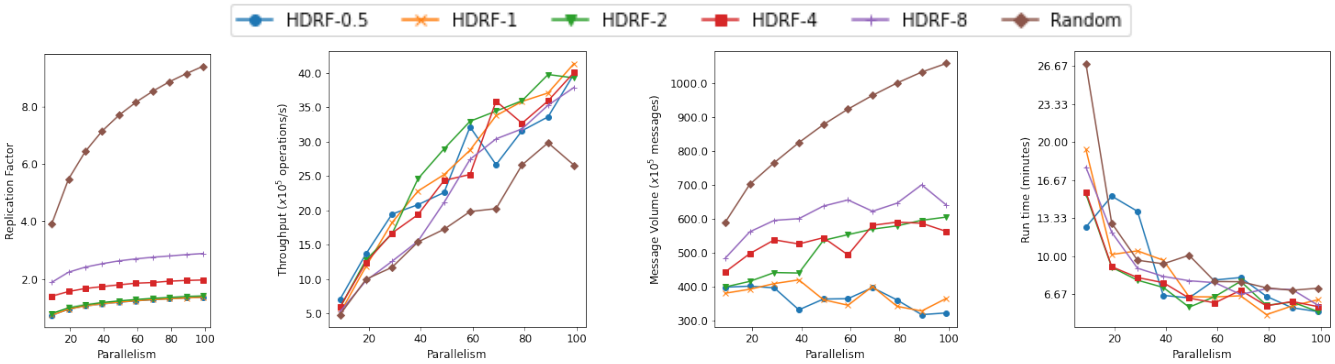
Strong scaling. We increase the threads available for allocation to D3-GNN operators, and examine the impact on throughput, total communication volume, latency, and run time (Figure 4). We compare the effects for two different `Plugins`, i.e., the D3-GNN Streaming (D3-S) model uses `StreamingGNNEmbedding` while the D3-GNN Windowed (D3-W) model uses `WindowedGNNEmbedding`. Results are reported for an average over 3 runs, with the D3-GNN system being scaled up from 1 machine (20 cores) to 10 machines (200 cores). For these experiments we set the number of logical parts to be the parallelism of final layer (λx). Hence, as we scale up, increase in logical parts is expected to increase the replication factor of the graph resulting in higher communication overheads. To accurately reflect the system throughput, we ensure that the entire pipeline is completely busy. We congest the system operators and network buffers by first partitioning all the incoming edges and then sending them through the GNN pipeline all at once. Additionally, we compare D3-S and D3-W with a single node, in-memory DGL implementation (Table 1). Since we run experiments in Slurm, DGL also uses 20 CPU cores.

	Run time (minutes)			Mean throughput ($\times 10^3$ operations/s)		
#Machines	D3-S	D3-W	DGL	D3-S	D3-W	DGL
1	17.7	0.78	96.8	239.3	55.4	2.2
5	7.83	0.63	-	631.2	46.8	-
10	5.68	0.63	-	817.6	17.1	-

Table 1: Performance comparison with DGL on Reddit ($\lambda = 3$)

Through partitioning, tasks are allotted to sub-operators across logical parts that process data simultaneously, therefore significantly reducing the run time of D3-GNN and improving its throughput. We measure throughput as the mean/max value across operator's run time of producing final layer embeddings (Figures 4a– 4b). We find that the streaming throughput scales roughly linearly with the amount of parallelism. The windowed approach throughput, however, slightly decreases. This is because the faster the system is able to consume data, the more postponements (to embedding update computations) can be expected to happen. Throughput measurements are significantly lower with D3-W (with a 200 milliseconds window interval), which is due to actively postponing node updates in graph.

Communication operations are incurred due to replication when distributing tasks across a large number of processors. Therefore, the overall communication volume of the system must be taken into account. The network traffic (total number of iterative messages) while generating updated embeddings are shown for the second GNN layer, in Figure 4c. The first layer message volumes are identical in streaming and windowed cases, since we are only receiving graph updates and communicating with the aggregators, irrespective of the `Plugin` being used. However, we see that having a 200ms window for updates from layer-1 significantly reduces the communication overheads in the subsequent GNN layer, as feature



(a) Replication factor **(b) Max throughput** **(c) Communication volume** **(d) Run time**

Figure 5: Performance comparison of D3-GNN Streaming with different partitioners on Reddit ($\lambda = 3$)

synchronization and *replace*() messages explode less frequently. Moreover, for D3-W, as we increase parallelism the communication volume decreases. This happens because as the system is able to consume more messages the frequency of postponements also increases, resulting in further delays to node feature updates.

Overall, the windowed setting enjoys a dramatic decrease in communication volume as well as the reduced run time compared to the streaming setting, trading off latency and throughput. Note that distributed termination detection is needed to measure the system run time. We consider the sub-operator ready for termination if it was idle for past 30 seconds. In other words, run times for D3-S and D3-W are reported along with this additional 30 second wait time before termination. Furthermore, termination detection in Flink relies on broadcasting messages between sub-operators, which itself incurs some extra cost at higher parallelism. These requirements make throughput and latency more reliable indicators of system performance, particularly since we focus on a continuous data stream workload.

Results for StackOverflow also demonstrate the weak scaling to millions of edges, where the run time is found to decrease following similar trends to that of Reddit (58 mins on 5 machines to 20 mins on 10 machines). These updates are windowed over a longer interval of 40 seconds. The figures are omitted due to lack of space.

Effect of partitioner. Since identifying the most suitable partitioning scheme is orthogonal to our goals, we present results for the **HDRF** vertex-cut partitioner and **Random** partitioner. However, we also measure the effect of changing the balance coefficient θ, which at higher values assigns more importance to the load balancing constraint. The performance comparison between various partitioner settings in D3-GNN Streaming is in Figure 5.

During the partitioning stage, θ helps to tackle the bandwidth limitations of the distributed infrastructure that D3-GNN is deployed on. Lower θ improves network communication overheads at the cost of load balance, mainly due to a lower replication factor. Using Random partitioning results in the highest communication volume needed to perform streaming GNN, thereby also lowering its throughput. While there are visible trends, the power-law nature of the input graph does induce some fluctuations in the observed metrics depending on the assignments of high-degree nodes.

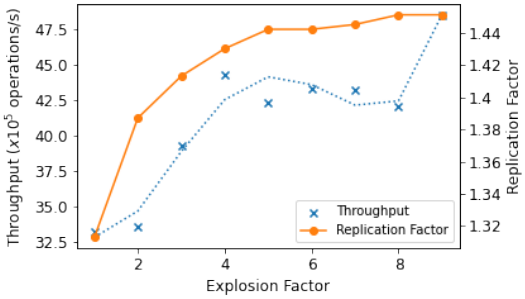


Figure 6: Performance comparison of D3-GNN Streaming with varying explosion factor on Reddit

Explosion factor. The effect of the explosion factor is given in Figure 6. D3-GNN system utilizes λ to optimally allocate tasks to all available processors when at its peak congestion level (when the neighborhood explosion is highest). A default explosion factor $\lambda = 3$ is used for all our previous experiments.

6 CONCLUSION

We introduced D3-GNN, a distributed, hybrid-parallel system for efficient GNN training and inference under streaming graph updates. D3-GNN continuously computes the embeddings with low latency and manages the graph data and model with fault-tolerance. We demonstrated the strong scalability of the system at higher parallelism, and with high throughputs of millions of streaming edges (about 100x that of DGL) even using only a single machine. D3-GNN also achieves improvements in communication volume, as well as with a 10–20x faster windowed approach that reduces network congestion by over 200x.

ACKNOWLEDGMENTS

Rustam is supported by an EPSRC Doctoral Training Partnership award, and Aparajita is supported by a Feuer International Scholarship in Artificial Intelligence at the 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] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. 2021. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[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] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[6] 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.

[7] Chi Thang Duong, Trung Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. 2021. Efficient streaming subgraph isomorphism with graph neural networks. *Proceedings of the VLDB Endowment* 14, 5 (2021), 730–742.

[8] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[9] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[10] 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.

[11] 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)*. 551–568.

[12] 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.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[14] 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.

[15] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[16] 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.

[17] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[18] Nico Kruber. 2020. Flink Serialization Tuning Vol. 1: Choosing your Serializer — if you can. https://flink.apache.org/news/2020/04/15/flink-serialization-tuning-vol-1.html. Accessed: 2022-07-12.

[19] 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.

[20] Sibon Li, Jan Pfeifer, Bryan Perozzi, and Douglas Yarrington. 2021. Introducing TensorFlow Graph Neural Networks. https://blog.tensorflow.org/2021/11/introducing-tensorflow-gnn.html

[21] 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.

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012).

[23] 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.

[24] 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.

[25] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[26] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[27] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. *arXiv preprint arXiv:2104.06700* (2021).

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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

[33] 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.

[34] Sungmin Rhee, Seokjun Seo, and Sun Kim. 2018. Hybrid approach of relation network and localized graph convolutional filtering for breast cancer subtype classification. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 3527–3534.

[35] 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).

[36] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[37] 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.

[38] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th international conference on scientific and statistical database management*. 1–12.

[39] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. 2018. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*. PMLR, 4470–4479.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *IJCAI*.

[47] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. TUX2: distributed graph computation for machine learning. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. 669–682.

[48] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. 2012. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 517–528.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[50] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3125–3137.

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

[52] 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. *arXiv preprint arXiv:2010.05337* (2020).

[53] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.