InkStream: Instantaneous GNN Inference on Dynamic Graphs via Incremental Update

Dan Wu
School of Computing
National University of Singapore
Singapore
danwu20@comp.nus.edu.sg

Zhaoying Li ✉
School of Computing
National University of Singapore
Singapore
zhaoying@comp.nus.edu.sg

Tulika Mitra
School of Computing
National University of Singapore
Singapore
tulika@comp.nus.edu.sg

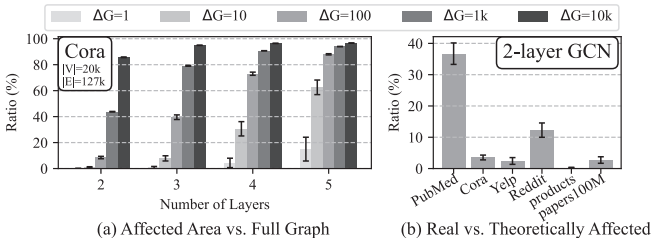
***Abstract*—Graph Neural Network (GNN) on dynamic graphs that evolve with time necessitates constant updates. Current approaches aim to mitigate computational costs by limiting updates to the affected areas, essentially the k-hop neighborhood surrounding modified edges/vertices in k-layer GNNs. However, we identified that these strategies often involve unnecessary computation: (1) Within the k-hop neighborhood, a substantial number of nodes remain unaffected by changes in edges/vertices when GNN employs `max` or `min` as its aggregation function; (2) For certain model architectures, the node embeddings can be incrementally updated with minimal memory access and computation. In response to these observations, we developed InkStream, an innovative and general method for real-time GNN inference by avoiding unnecessary updates, significantly reducing inference time and energy cost. InkStream supports all common GNN aggregation functions while imposing minimal constraints on model architecture. It is grounded in the principle of minimalistic propagation and data retrieval, employing an event-based system to manage both the inter-layer propagation of effects and the intra-layer incremental updates of node embeddings. Additionally, InkStream offers remarkable extensibility and ease of configuration, making it adaptable to evolving GNN model structures. Our evaluation across three GNN models on six graph datasets reveals that InkStream significantly accelerates inference time from hours to mere milliseconds. The code is available at https://github.com/WuDan0399/InkStream.**

***Index Terms*—Graph Neural Networks, Dynamic Graphs**

I. Introduction

Graph Neural Networks (GNNs) have become essential in high-performance computing (HPC) due to their significant computational and memory requirements when processing large-scale graph-structured data. Most existing GNN architectures are designed for static graphs; however, many real-world applications, such as social networks, financial systems, and communication networks, involve dynamic graphs where nodes and edges evolve continuously over time [1]–[3]. This dynamic nature necessitates frequent updates to nodes' latent embeddings [4]–[6], leading to a substantial increase in computational demands and highlighting the challenges GNNs pose as an HPC workload.

Dynamic graphs exhibit unique characteristics in the GNN inference task: most of the nodes have their latent embeddings unchanged in two consecutive timestamps. The reason is two-fold: (1) There is only a small amount of edges changed in the large-scale graphs [7]–[9]; (2) GNNs are usually shallow (k layers, $k \leq 5$) to prevent over-smoothing problem [10], and one changed node can only affect its k-hop neighborhood, which is limited compared to the full graph. This k-hop neighborhood is named as the *affected area*. Therefore, it is redundant to recompute for all nodes in the graph, as practiced in efficient GNN inference works [11]–[13]. Suppose there are ΔG requests for edge changing, either insertion or removal, between two timestamps. Figure 1a shows the size of the affected area compared with the whole graph for the Cora dataset in terms of the number of nodes. It can be seen that the affected area can be tiny (less than 1%), compared with the whole graph, when k is small. A straightforward way to take advantage of this insight is to only compute for the nodes in the k-hop neighborhood [1], [14]. Unfortunately, it still takes seconds to insert 100 edges into the Cora dataset. We propose InkStream, a general method for GNN inference on dynamic graphs, which further reduces the inference time from seconds to milliseconds based on two novel findings.

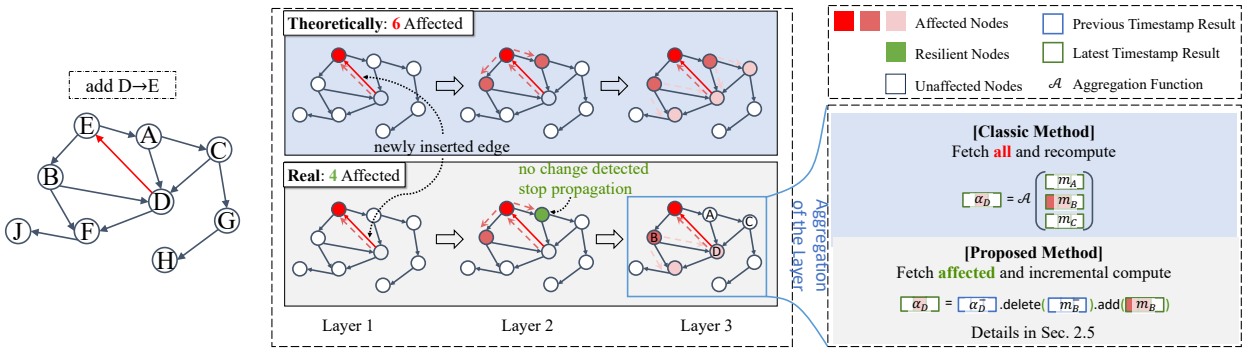


(a) Affected Area vs. Full Graph (b) Real vs. Theoretically Affected

Fig. 1: (a) Ratio of the theoretically affected area to full graph. (b) Ratio of real affected nodes to the theoretically affected area. ΔG: number of changed edges.

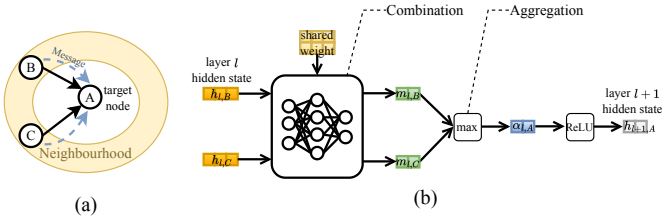
Before delving into these findings, we briefly introduce the general message-passing framework adopted by most GNN models. In a multi-layer GNN model, one layer generates a hidden state for each graph node: an intermediate node embedding vector. One GNN layer can be expressed by two primary execution phases: *Combination* and *Aggregation* (Figure 3). The combination phase acts like neural networks, where each node transforms its embedding with a shared multi-layer perceptron. In the aggregation phase, each node

This research is partially supported by the National Research Foundation, Singapore, under its Competitive Research Programme Award NRF-CRP23-2019-0003. The corresponding author is Zhaoying Li.



(a) Input Graph (b) Inter-Layer: Pruned Propagation (c) Intra-Layer: Minimal Memory Access

Fig. 2: Two-level savings in InkStream. $\mathcal{A}$ for aggregation function. m and α are node embedding before and after aggregation. Superscript $^-$ denotes the result in the previous timestamp.



(a) (b)

Fig. 3: (a) Neighborhood message passing of a GNN layer. (b) Computing abstraction of message-passing mechanism.

receives its neighboring nodes' embeddings and then reduces them into a single vector with a selected *aggregation function*, such as a `mean`, `sum`, `max`, and `min` function [15].

First, we found that *the size of the affected area can be further reduced for GNNs with certain aggregation functions.* When equipped with the `max` or `min` as aggregation function, the aggregation phase of a GNN layer is selective, i.e., a node u is not significant enough to contribute to a neighbor v if it is not selected as max or min for all channels in an embedding vector. In this case, removing the edge between node u and node v will not affect the embedding of v. Similarly, adding an edge with an insignificant neighbor will also not affect a node's embedding. This leads to a reduced affected area. We call it *real affected area*, against the *theoretical affected area* of the full k-hop neighborhood. Figure 1b shows the ratio of real affected nodes versus the theoretically affected area. On average, only 3% nodes in k-hop neighborhood are influenced in Cora, Yelp, and (ogbn-)papers100M datasets with 100 changed edges. In other cases of `mean` and `sum`, there is no such reduction in the affected area as they do not exhibit selective characteristics.

Second, *the embedding of a node can be incrementally updated; however, the method of incremental updating differs across aggregation functions.* As one node's neighborhood often shows minimal changes in a short time, a node can incrementally evolve its embedding by first neutralizing the impact for a changed neighbor's old embedding and then adding the impact of the new embedding, bypassing the computation involving unchanged neighbors. However, the approach differs for *accumulative* (`mean`, `sum`) and *monotonic* (`max`, `min`) aggregation functions. As monotonic functions are selective, a significant neighbor can dominate a node's embedding and make it oblivious of the rest of the neighbors. Deleting this neighbor, the node can experience irrecoverable data loss; so fetching the whole neighborhood and recomputing is necessary. Meanwhile, for accumulative aggregation functions, a node's embedding evolves by adding up the changes in all neighbors' embeddings, and hence the costly recomputation will not happen.

Fully exploiting the aforementioned two findings, we propose a generic method called InkStream with the design principle: "*Propagate only when necessary. Fetch only the necessary.*" **First**, in a GNN model, a changed embedding of one node at a layer will affect its immediate neighbors' embeddings in the next layer, thus forming a propagation tree. In cases of GNNs with monotonic aggregation function and thus reduced affected area, we prune the propagation tree to avoid unnecessary computation. Suppose there is an edge inserted from node D to E, as shown in Figure 2a. Across the layers, InkStream mitigates unnecessary computation by identifying *resilient node*s (green node in the Figure 2b) in the propagation tree and pruning the corresponding subtrees rooted at these nodes (Figure 2b). A node is resilient if it could have been affected but is found to be uninfluenced after updating. We design an *event*-based approach to control the propagation. An event is created and sent along the edges when a node changes its embedding. **Second**, inside each layer, during the aggregation phase, InkStream reuses data from the previous timestamp, incrementally applying the effects of changed neighbors. As shown in Figure 2c, InkStream deletes the impact of changed neighbors' old information and adds the impact for new information, thus incrementally evolving to the latest node embedding. By only accessing prior results and affected neighbors' information without fetching the whole neighborhood, InkStream drastically minimizes memory access and execution time.

InkStream is an acceleration technique ensuring arithmetic

equivalence to classic methods of GNN inference. Particularly, when applied with monotonic aggregation functions, bit-level identical results to classic methods is ensured, thanks to the selective nature of these functions which replicate the outcome once the same choice is made. Moreover, InkStream has good extensibility by providing interfaces to allow the users to easily configure for their own models. On three benchmark GNN models: GCN [16], GraphSAGE [17], and GIN [18], users only need to provide less than 10 lines of additional code. In a nutshell, InkStream is a highly extensible method that minimizes the computation and memory access at the inter-layer level and intra-layer level while bringing no accuracy loss. Its efficient design makes it an ideal solution for HPC environments handling real-time, large-scale dynamic graphs.

Our contributions can be summarized as follows:

1) **New Findings**: We found the gap between the theoretically affected area and the real affected area for GNN inference on dynamic graphs with a monotonic aggregator. We also found that a node embedding can evolve incrementally from an old result in the previous timestamp.

2) **Minimal Overhead**: InkStream requires minimal data access and computation by pruning the effect propagation tree and reusing the previous results.

3) **Generality**: Our method can be applied to a wide range of GNNs through simple configuration, and four mainstream aggregation functions are supported.

4) **No Accuracy Loss**: InkStream accelerates GNN inference by optimizing the redundant computation without changing the model structure, ensuring an arithmetic level equivalence to the class methods.

5) **Large Performance Improvement**: Evaluated on three GNN models and six datasets, our experiment shows that our method can reduce the inference time from hours to milliseconds.

II. InkStream

InkStream takes the result of an initial full graph inference as input and incrementally updates with a batch of edge removal or insertion requests. During the full graph inference, we save the embedding before and after aggregation for the whole node set in all layers. As the hidden state dimension is usually carefully designed and small (16-256), much smaller than the feature length (100-8710 in our experiments), the memory cost is acceptable and worth the speedup. For clarity, we first introduce the notations used in this paper and the expressiveness of InkStream. Later, we present the overall workflow and each component of the workflow in detail.

Notations. Table I lists the notations used in the paper for streaming graphs and GNNs. A graph is denoted as $G(V,E)$, with vertex set V and edge set E. Between two timestamps, a set of edges ΔG is modified, either inserted or removed. The direct neighborhood of a vertex u is $N(u)$. In the message-passing mechanism, there is a combination phase and aggregation phase, with flexible ordering. The operations in each phase can also be customized for different tasks. We abstract the computation in combination phase into a function $\mathcal{T}()$, and the aggregation function $\mathcal{A}()$. At the end of each layer, there is an element-wise activation function $act()$. In a multi-layer GNN model, we name the embedding of node u before being fed into layer l as $h_{l,u}$, which is the input of layer l. Inside the layer l, we name the input and output of the aggregation phase as the message m_l and the aggregated neighborhood α_l respectively, where $\alpha_{l,u} = \mathcal{A}(m_{l,v} : v \in N(u))$. When aggregation phase is first executed in a layer, $m_l = h_l$, otherwise, $m_l = \mathcal{T}(h_l)$. To distinguish results in previous and current timestamps, a superscript $^-$ is used.

TABLE I: Notations

Notations	Descriptions
$G(V,E)$	graph G with vertices V and edges E
ΔG	modified edges between two timestamps
$N(u)$	direct neighborhood of vertex u
$\mathcal{A}()$	aggregation function
$\mathcal{T}()$	combination function
$act()$	activation function
$h_{l,u}$	hidden state of vertex u in layer l
$m_{l,u}$	message sent from vertex u in layer l
$\alpha_{l,u}$	embedding of u after aggregation in layer l
$\square$	data in current timestamp
$\square^-$	data in previous timestamp

Expressiveness. InkStream can describe any GNNs built on the message-passing framework as long as:

(1) *One node's message in a layer only depends on its message and aggregated neighborhood in the previous layer*: $m_{l+1,u} = act(\mathcal{T}(\alpha_{l,u}, m_{l,u}))$. In this case, a changed edge's impact only propagates along the edges or to the node itself in the next layer. Thus, the changed edge will not influence a significant portion of the graph. An exception to this principle occurs with normalization layers in GNNs, which compute mean and variance across a set of vertices. In dynamic graphs, any modification to the vertex set requires an update to these statistics. To address this challenge, we propose an approximation method for the graph normalization layer, detailed in section II-E.

(2) *The aggregation process is partially or fully reversible*, so the old node embedding can be reversed by canceling a neighbor's old impact. We define fully (partially) reversible as "a node embedding can always (in certain conditions) be reversed by canceling a neighbor's old impact." Mathematically, we say a function g is reversible, if there exists a function f, such that $y^* = f(y, x)$, where $y = g(A), x \in A, y^* = g(A^*)$ with $A^* = A/\{x\}$. For now, our implementation supports four commonly-used aggregation functions (`max`, `min`, `mean`, and `sum`) while irreversible aggregation functions like std are not compatible with our method. Aggregation with weighted sum can also be supported once only graph topology information is used for the weights, like LightGCN [13].

A. InkStream Workflow

InkStream uses *event*s for controlling the update area in a graph as well as applying incremental updates. An *event* is a message carrying the operation, target node, and an embedding

Algorithm 1 InkStream Workflow

Input: GNN model, graph in previous timestamp G^-, modified edges ΔG, messages m^- and aggregated neighborhood α^- in previous timestamp.
Output: Updated node representation, updated messages m and aggregated neighborhood α.

1: **for** each layer l in GNN **do**
2: Create events for ΔG in layer l, push to event queue.
3: Group and reduce events targeting same vertex. ▷ Sec. II-B1
4: **for** each node u in event queue **do**
5: Get reduced event(s) heading to u in event queue.
6: **if** aggregation $\mathcal{A}$ in layer l is monotonic **then**
7: Check eligibility of incremental update. ▷ Sec. II-C1
8: **if** eligible **then**
9: Incremental update $\alpha_{u,l}$. ▷ Sec. II-C1
10: **else**
11: Recalculate $\alpha_{u,l} = \mathcal{A}(m_{l,v}), v \in N(u)$.
12: **end if**
13: **if** event propagation not pruned: $\alpha_{u,l} \neq \alpha^-_{u,l}$ **then**
14: $m_{l+1,u} = act(\mathcal{T}(\alpha_{l,u}, m_{l,u}))$
15: Propagate events to $N(u)$, $N^-(u)$. ▷ Sec. II-B2
16: **end if**
17: **end if**
18: **if** aggregation $\mathcal{A}$ is accumulative **then**
19: Incremental update $\alpha_{u,l}$. ▷ Sec. II-C2
20: Propagate events to $N(u)$, $N^-(u)$. ▷ Sec. II-B2
21: **end if**
22: **end for**
23: **end for**

vector. It tells whether to add or cancel the impact of the vector on the target node's embedding. Following this, in the subsequent layers, every real affected node propagates events to its neighbors, which will be processed in the next layer.

Algorithm 1 shows the overall workflow for InkStream. At the start of each layer, events for each changed edge are generated and added to an event queue (line 2). Events targeting the same node and with the same operation are then grouped and reduced to a single event (line 3). For layers using monotonic aggregation functions, InkStream checks if the target node can be incrementally updated (line 7). If it can, the update is applied; otherwise, the node's embedding is recalculated using all neighbors' information (line 11). If a node's embedding remains unchanged after all events heading to it are processed, the propagation will stop; otherwise, new events are created and propagated to the next layer (lines 14-15). For layers with accumulative aggregation functions (lines 18-21), node embeddings are always affected, and event propagation will not be pruned. With a fully reversible aggregation process, incremental updates can be safely applied to affected nodes with the help of events.

B. Inter-Layer: Event-based Computing Model

We proposed the *event*-based computing model to manage the inter-layer propagation. An event is a message carrying the operation, target node, and embedding vector. The embedding vector is heavy and shared among multiple events (e.g., a node tells all its neighbors to add an impact of its new embedding). Therefore, we separate the lightweight event metadata and heavy node embeddings into two lists to reduce memory consumption. For simplicity, in the paper, we assume that a vector is directly associated with an event, without delving into the underlying data structure.

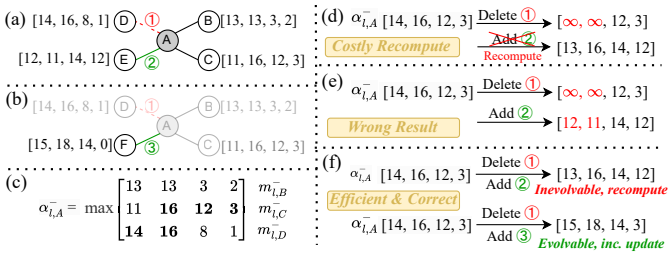


Fig. 4: Illustrative example for the necessity of grouping for `max` aggregation function. (a-b) Example graphs with different newly inserted edges. (c) Derivation of $\alpha^-_{l,A}$. (d-e) Processing events sequentially can be costly or with wrong result. (f) Analyzing all events to determine when efficient incremental updates can be applied ensures correctness while saving computation.

There are four types of operations: `Add` and `Del` for monotonic aggregation functions, `Update` for accumulative functions, and `User` for other non-native user-defined aggregation functions. For example, {`Del`, 5, [10, 1, 0, 2]} means to cancel the contribution of vector [10, 1, 0, 2] to the old aggregated neighborhood of vertex 5, $\alpha^-_{5,l}$, in the current layer with a monotonic aggregation function. For layers with accumulative aggregation functions, to eliminate the influence of an old node embedding of a neighbor, we create an event with operation `Update`, carrying a message of the negative of the previous node embedding, and vice versa.

With the help of our event-based computing model, we can easily manage the computation for the affected area. If one node's embedding remains the same after finishing the impact addition and cancellation in the events heading to it, it will not further propagate events to save computation afterward. The event-based computing model has two primary components: 1) Grouping: group and reduce events heading to the same target node. 2) Propagation: propagate the effect to neighbors.

1) Event Grouping:

Necessity of event grouping. If one node u has multiple neighbors changed in the former layer, it will receive several events, non-continuously stored in the event list. It is inefficient to iterate the event list sequentially and process each event separately, as the old aggregated neighborhood needs to be fetched each time. Instead, we group the events heading to the same node and process all of them at the same time. More importantly, for layers with monotonic aggregation functions, event grouping also improves the evolvability of nodes. For example, as shown in Figure 4a-c, vertex A is connected with B, C, and D in the last timestamp, with an aggregated neighborhood [14, 16, 12, 3]. This node receives two events to update its aggregated neighborhood, one for impact deletion for a dominating neighbor's message [14, 16, 8, 1] and the

other for addition. After the deletion, the first half of the embedding is found affected and needs to be *reset* with a default value ∞. If the two events are processed independently (Figure 4d), since there are two irrecoverable channels after deletion, we must fetch all neighbors' information and recompute in a classic method to get the correct result. For another option, if the incremental update is directly applied without knowing other events (Figure 4e), a wrong result could be derived. However, if all events are processed together, we can leverage them and decide whether the incremental update can be applied while ensuring a correct result (Figure 4f). After grouping by target node, the events with the same operation are reduced into one event to reduce data movement in the following process.

Process of event grouping. To conduct the event grouping, we iterate through the event list and group the events by first the target node and then the operation. Next, when monotonic aggregation functions are applied, we reduce all grouped events in the same group with the aggregation function. In the case of accumulative functions, the events are reduced by summing up.

2) Event Propagation:

When one node has its embedding or connectivity changed in a layer, a set of events will be created for each neighbor, carrying the message whose impact will be canceled or added. Then, these events will be pushed to the event queue for the next layer. In this section, we introduce how these two conditions are handled in our event-based computing model.

Propagate for changed edges. At the *beginning* of processing one layer l, we create events for destinations of changed edges. The events will later be consumed when processing the current layer l. Take the monotonic aggregation function as an example. Suppose the edge (u, v) is removed. To cancel impact of the old message $m^-_{l,u}$, an event is created, containing the operation `Del`, target node v, and the old message $m^-_{l,u}$. For an inserted edge (s, t), the new message $m_{l,s}$ — remains unaffected or updated when processing the former layer — is used to create an event with an `Add` operation. In the case of accumulative aggregation functions, the message $-m^-_{l,u}$ ($m^-_{l,u}$) will be used in events with `Update` operation for edge removal (insertion).

Propagate for affected nodes. At the *end* of processing the layer l, InkStream creates events for the neighbors of those affected nodes in the next layer l+1. These events will later be consumed in the processing of the next layer. The procedure differs from the handling of changed edges, as it necessitates first canceling the effect of the previous message before incorporating the new one. For an affected node u whose aggregated neighborhood changed from $\alpha^-_{l,u}$ to $\alpha_{l,u}$, InkStream first calculates its message in the next layer according to the model configuration, e.g., $m_{l+1,u} = act(\mathcal{T}(\alpha_{l,u}))$. Then InkStream creates events to cancel the impact of $m^-_{l+1,u}$ and add the impact of $m_{l+1,u}$.

Duplicated events exist if one node is the destination of a changed edge and the source node is affected in the former layer. To avoid this situation, InkStream skips the propagation of an affected node once it is the source node of a changed edge.

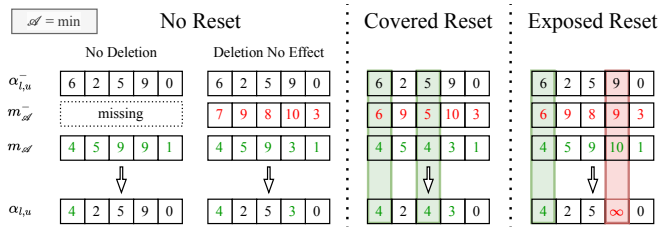


Fig. 5: Events' Effect: Three Conditions.

C. Intra-layer: Incremental Update

Inside each layer, after event grouping, InkStream takes the grouped events of a target node and updates its node embedding. The update approach differs for monotonic aggregation functions and accumulative aggregation functions.

1) Approaches for Monotonic Aggregation Functions:

In the case of monotonic aggregation functions, InkStream first checks whether incremental update can be applied and then updates the node embedding either with incremental update or recomputation in a classic way.

Applicable conditions. The effect of events on the old aggregated neighborhood can fall into three categories, and incremental updates can be applied in two of them. For clarity, we define the notations first. After event grouping without reduction, target node u has a set of messages to be deleted $[m^-_1, m^-_2, \ldots, m^-_p]$ and a set of messages to be added $[m_1, m_2, \ldots, m_q]$. For target node's old result, $\alpha^-_{l,u}$, if there is any channel where the old result equals to the message to be deleted, $\exists\, i \in [0, n], j \in [1, p]$, s.t. $\alpha^-_{l,u}[i] = m^-_j[i]$, where n is the dimension of $\alpha^-_{l,u}$, then $\alpha^-_{l,u}$ need to be *reset* at channel i. Since the aggregation function is monotonic and the messages to be deleted are a subset of the whole neighborhood information $\alpha^-_{l,u}$ encapsulates, a reset channel can only appear in the message whose value at that index is the maximum (minimum) among all messages to be deleted. Therefore, without loss of correctness, these messages are reduced before checking: $m^-_{\mathcal{A}} = \mathcal{A}(m^-_1, m^-_2, \ldots, m^-_p)$, $m_{\mathcal{A}} = \mathcal{A}(m_1, m_2, \ldots, m_q)$.

There are three conditions of how events affect the old result: *no reset*, *covered reset*, and *exposed reset*. An illustrative example is shown in Figure 5. 1) No Reset. When there is no channel that needs to be reset, $\forall\, i \in [0, n],\ \alpha^-_{l,u}[i] \neq m^-_{\mathcal{A}}[i]$, incremental update can be applied. Especially there are two cases in this condition: there is no message to be deleted; or the deletion has no influence on the old result $\alpha^-_{l,u}$. 2) Covered Reset. When the channels that need to be reset are *cover*ed by the message to be added, incremental updates can also be applied. Suppose there is a set of channels that need to be reset, D. We say D is covered by the new message $m_{\mathcal{A}}$, if for any channel i in D, the message to be added is "better" than the message to be deleted, $\forall i \in D$, $\mathcal{A}(m^-_{\mathcal{A}}[i], m_{\mathcal{A}}[i]) = m_{\mathcal{A}}[i]$. Despite the fact that part of the old result is affected and needs to be reset, the correctness of incremental updates can be ensured due to the transitivity of monotonic functions. The

key insight is the fact that messages from unaffected neighbors are dominated by the deleted value. Therefore, if the reduced new message dominates the deleted value, it must dominate the others. 3) Exposed Reset. When channels need to be reset and are not fully covered by the new message, recompute is necessary to ensure a correct result.

Update and propagate. Once incremental update can be applied to a node u, it will be updated with the reduced addition messages: $\alpha_{l,u} = \mathcal{A}(\alpha^-_{l,u}, m_{\mathcal{A}})$. Specifically, in *no reset* condition, there is also a chance that the new message has no impact on the old result, such that $\alpha_{l,u} = \alpha^-_{l,u}$. Therefore, InkStream compares $\alpha_{l,u}$ against $\alpha^-_{l,u}$ to check whether the node has been affected. If the node is resilient to the changes, it will not propagate events to its neighbors. In other cases, where node 1) has been updated without a reset, 2) has covered reset, or 3) is recomputed, inter-layer propagation will be initiated.

When incremental update is not applicable, all messages from neighbors in the current timestamp are fetched and aggregated to recompute the node: $\alpha_{l,u} = \mathcal{A}(m_{l,v} : v \in N_u)$. Note that no additional computation is required to get $m_{l,v}$. As $m_{l,v}$ is calculated from $\alpha_{l-1,v}$, then $m_{l,v} \neq m^-_{l,v}$ if and only if $\alpha_{l-1,v} \neq \alpha^-_{l-1,v}$. There are only three types of nodes in a layer: unaffected, resilient, and affected. If v is unaffected or resilient in layer l-1, then $\alpha_{l-1,v} = \alpha^-_{l-1,v}$, $m_{l,v} = m^-_{l,v}$. If v is affected in layer l-1, $\alpha_{l-1,v}$ and $m_{l,v}$ will be computed and updated when processing layer l-1.

2) Approaches for Accumulative Aggregation Functions: Consider the `sum` function to exemplify how a node's new aggregated neighborhood is derived from its previous result. The calculation is simplified as follows:

$$\alpha_{l,u} = \alpha^-_{l,u} + \sum \Delta m_{l,v^*} + \sum -m^-_{l,v^+} + \sum m_{l,v'}$$

where $\alpha^-_{l,u}$ represents the old aggregated result, $\Delta m_{l,v^*} = m_{l,v^*} - m^-_{l,v^*}$ denotes the change in messages from nodes in both the old and new neighborhoods ($v^* \in N_u \cap N^-_u$), $\sum -m^-_{l,v^+}$ accounts for messages from deleted edges ($v^+ \in N^-u - N_u$), and $\sum m_{l,v'}$ represents messages from newly inserted edges ($v' \in N_u - N^-_u$). In InkStream, the impact canceling and adding for accumulative aggregation functions share the same event operation `Update`, but different symbols (+/-) for the message. Therefore, to incrementally update the old aggregated neighborhood $\alpha^-_{l,u}$ to the new one $\alpha_{l,u}$, we can simply add it with the sum of all the messages carried by events heading to the same node. That is, $\alpha_{l,u} = \alpha^-_{l,u} + sum(msg)$. In another case of aggregation function `mean`, $\alpha^-_{l,u} = \frac{d^-_u}{d_u}(\alpha^-_{l,u} + sum(msg))$.

D. Flexible User-Defined Functions

To support models with more complex operations, we allow users to define their own event propagation, grouping and apply functions apart from the management of native events in InkStream. Specifically, InkStream provides three interfaces for customization: `user_propagate`, `user_grouping` and `user_apply` functions. `user_propagate` allows users to create and propagate customized events, executed together with InkStream's built-in propagation function. `user_grouping` defines the grouping and processing of customized events during the events grouping process. `user_apply` contains the logic to apply the effect of customized events to the tensor at a phase level. The management of user-defined events is isolated from the native functions. Figure 6 shows the operations in a GraphSAGE [17] layer and its corresponding InkStream implementation. The computation regarding neighborhood aggregation $W_1\mathcal{A}(\{h_{l-1,v} : v \in N_u\})$ is managed by native InkStream events, while the additional logics $W_2 h_{l-1,u}$ is expressed with user-defined events. With a few lines of additional code, InkStream can be customized to support varied model structures.

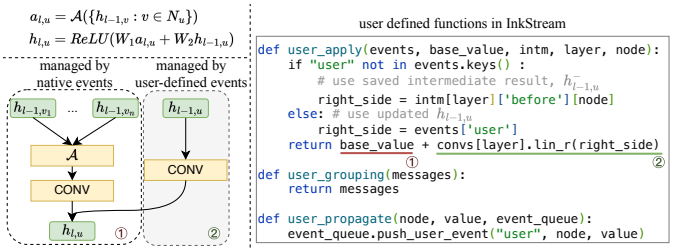


Fig. 6: Equations, illustration and InkStream implementation for GraphSAGE. ReLU is hidden in the figure for clarity.

E. Support for other operators

Graph Normalization. Graph normalization (GraphNorm) layer [19] presents a challenge for real-time inference in dynamic graphs. GraphNorm computes mean and variance across all vertices, which are affected by any modifications to the vertex set. To address this, we design an approximation of GraphNorm, which maintains the model's accuracy while allowing efficient updates. We propose using the mean μ and variance σ^2 values computed during the model's training phase as approximations for these statistics during inference. Note that the training happens periodically to evolve the model with the graph, and the approximated statistics are only used between two training phases. This method is particularly effective for GraphNorm in scenarios with limited vertex removal or insertion, where the feature matrix remains relatively stable.

Sampling. Graph sampling is an important technique in GNN for memory efficiency, model scalability, training/inference speed, and so on. With minimal memory access and millisecond-level inference time, no sampling is used in InkStream by default. However, graph sampling strategies can be seamlessly supported, once the sampled graph structure is known before the inference, like random neighbor sampler [17], and random walk-based sampler [20]. To support such sampling, one needs to cache the structure of the sampled neighborhood from the last timestamp and compare it with the one in the current timestamp. The differences in the two sampled neighborhoods can be represented as a list of edge removal and insertion.

F. Updating Vertices

Vertex operations, including deletion, insertion, and update, are also handled by events in InkStream. To illustrate the process, we'll focus on the example of updating vertex features, which demonstrates the core principles applicable to all vertex operations.

For a vertex u with a new feature x, the update process begins by computing the new message $m_{1,u}$ for the first layer. If aggregation precedes transformation in the GNN architecture, $m_{1,u}$ is simply set to x_u. Otherwise, $m_{1,u}$ is computed as $\mathcal{T}(x_u)$. Next, InkStream propagates the effect of this update by applying effect of new message $m_{1,u}$ and removing effect of saved old message $m^-_{1,u}$ to the node's neighbors. This propagation will then proceed layer by layer and update all the affected nodes.

III. Experiment Evaluation

A. Experiment Setup

Benchmark Datasets. Table II lists the six benchmark graph datasets used in our experiment. The ogbn-products and ogbn-papers100M are referred to as Products and Papers100M in figures. To simulate graph dynamics, we assign random edge creation and deletion times following the work in T-GCN [3]. We use the latest n edges from each dataset to capture a graph's snapshot, excluding overly dated interactions that may introduce noise [1]. n is set as 15M for ogbn-products, 500M for ogbn-papers100M, and 5M for the rest of the datasets. This approach also avoids memory overloads when accessing the whole k-hop neighborhoods of high-degree nodes.

Benchmark GNN models. We choose 2-layer GCN [16], 2-layer GraphSAGE [17], and 5-layer GIN [18] as the benchmark models, following the structure in the original papers. Hidden state dimension is 64 for GIN and 256 for the rest.

Baseline Methods and Implementation. We compare InkStream against the following baselines: *PyG (+SAGE sampler)*: the official implementation from PyTorch Geometric library [21] with neighbor sampler [17] (10 neighbors each layer); *Graphiler* [22]: a state-of-art compiler stack for GNN acceleration; *k-hop*: first calculates the affected area, then inferences for vertices in the affected area with neighbor loader. We design *k-hop* following the core idea of DyGNN [1] that updates only for the affected area. Since DyGNN deploys a customized model architecture mainly targeting model accuracy with no other optimization for efficiency, it cannot be directly used as a baseline.

All baseline methods only take the latest snapshot of graph structure as input without knowledge of previous timestamps. Graphiler has an official implementation for GCN but lacks support for the other two benchmark models. We implemented the other two models following its original interface. We also replaced the datasets with our graph snapshots and changed model parameters for alignment. *InkStream*, *PyG (+SAGE sampler)* and *k-hop* are implemented with PyTorch Geometric library [21] without additional kernel level optimization.

InkStream Implementation. We showcase the performance of InkStream with two variants: *InkStream-m* for GNN models with monotonic aggregation function `max`, and *InkStream-a* for those with accumulative aggregation function `mean`.

Evaluation. As the location of changed edges between two timestamps influences the affected area and the execution time, for *k-hop*, *InkStream-m* and *InkStream-a*, we evaluate the performance across a set of graph changing scenarios, and report the average. For 2-layer GCN and GraphSAGE, there are 100/100/10/10/1 saved scenarios for ΔG=1/10/100/1k/10k separately. For 5-layer GIN, there are 10 graph-changing scenarios for ΔG=1. By default, we set the number of changed edges to 100 for GCN and GraphSAGE, 1 for GIN to ensure the theoretical affected area is around 10% of the full graph across all six datasets. The changed edges are evenly distributed for edge insertion and deletion.

Platforms. We test the performance of InkStream on a CPU-GPU platform. Table III lists the system configurations.

TABLE II: Datasets

Dataset	$\|V\|$	$\|E\|$	Feat. Len.	Scale
PubMed (PM) [23]	20K	89K	500	Small
Cora (CA) [24]	20K	127K	8710	Small
Yelp (YP) [25]	717K	114M	300	Medium
Reddit (RD) [24]	233K	14M	602	Medium
ogbn-products (PD) [26]	2.45M	124M	100	Medium
ogbn-papers100M (PP) [26]	111M	1.62B	172	Large

TABLE III: System Configurations

CPU Units	Intel Xeon 6230	Host Memory	264 GB
CPU Count	104	GPU Memory	48.3 GB
GPU Units	NVIDIA A6000	CPU-Memory Bw.	6.0 GB/s
GPU Count	1	CPU-GPU Bw.	12.3 GB/s

B. Performance

Table IV shows the comparison of execution time for the five methods. Despite the use of sampling to enhance efficiency, *PyG (+SAGE sampler)* still requires seconds to hours to perform inference on the entire graph. Both *k-hop* and InkStream, which focus on only the affected areas, achieve speedups of up to several orders of magnitude (greater than 1,000x). Consequently, we compare the speedup of InkStream primarily against the *k-hop* baseline. Compared to *k-hop*, InkStream demonstrates 2-282x speedup across various benchmarks, effectively reducing inference time from hours to seconds and from seconds to milliseconds. The variability in speedup trends across different datasets and models is influenced by a combination of factors, including graph density, model depth, and model complexity.

Comparing across models, InkStream shows better speedup on GCN than GraphSAGE and GIN in most cases. First, in GCN, the effect of changed edge only propagates along the graph edges, while in GraphSAGE and GIN, the effect also propagates to the node itself in the next layer. Moreover, GCN has simpler dense computation inside a layer than GraphSAGE and GIN, making it more memory-bounded and also benefit more from our memory-efficient solution.

Comparing across datasets, there is no constant trend due to the interaction of graph size, graph density and model depth.

TABLE IV: Inference time (in ms) comparison. k-*hop* refers to the baseline of inference only for the k-hop neighborhood. 'OOM' indicates Out of Memory instances.

	PubMed	Cora	Yelp	Reddit	ogbn-products	ogbn-papers100M
GCN (k=2, ΔG = 100)						
PyG (+SAGE sampler)	6.9E+03	9.5E+03	8.6E+04	9.5E+04	2.1E+04	5.8E+08
k-hop	676 (1x)	3,064 (1x)	4,218 (1x)	3,564 (1x)	2,188 (1x)	27,602 (1x)
Graphiler [22]	**1.34 (504x)**	**2.96 (1035x)**	**70.59 (60x)**	346 (10x)	525 (4x)	12,546 (2x)
InkStream-m	128 (5x)	108 (28x)	144 (29x)	**61 (59x)**	**203 (11x)**	**233 (118x)**
InkStream-a	115 (6x)	120 (28x)	106 (40x)	**77 (46x)**	**313 (7x)**	**426 (65x)**
GraphSAGE (k=2, ΔG = 100)						
PyG (+SAGE sampler)	5.0E+3	8.8E+03	1.3E+05	5.2E+04	5.1E+05	8.0E+08
k-hop	682 (1x)	3,183 (1x)	4,324 (1x)	3,605 (1x)	1,866 (1x)	30,518 (1x)
Graphiler [22]	**1.67 (408x)**	**4.96 (641x)**	**83.35 (52x)**	449 (8x)	OOM	OOM
InkStream-m	381 (2x)	410 (8x)	393 (11x)	**384 (9x)**	**710 (3x)**	**1,617 (19x)**
InkStream-a	371 (2x)	391 (8x)	306 (14x)	683 (5x)	**796 (2x)**	**667 (46x)**
GIN (k=5, ΔG = 1)						
PyG (+SAGE sampler)	3.0E+03	3.6E+04	2.7E+06	2.0E+06	1.2E+06	8.5E+08
k-hop	3,425 (1x)	43,248 (1x)	22,833 (1x)	34,929 (1x)	738,118 (1x)	528,217 (1x)
Graphiler [22]	**3.47 (987x)**	**5.22 (8285x)**	OOM	OOM	OOM	OOM
InkStream-m	1,756 (2x)	1,694 (26x)	**5,579 (4x)**	**1,502 (23x)**	**6,488 (114x)**	**10,305 (51x)**
InkStream-a	1,726 (2x)	1,637 (26x)	**3,812 (6x)**	**2,805 (12x)**	**2,619 (282x)**	**71,297 (7x)**

For example, *InkStream-a* shows significant speedup on the ogbn-products dataset with the GIN model (k=5) compared to the GCN model (k=2). This is because, in the baseline k-*hop* method, the entire $2k$-hop neighborhood data is fetched for computation when a vertex is updated, leading to inefficiencies, particularly in dense graphs. *InkStream-a* improves upon this by restricting data fetching to the immediate k-hop neighborhood, reducing redundant memory access. In contrast, for the Yelp dataset, *InkStream-a*'s speedup on GIN is much lower than that on GCN. This is because the Yelp dataset has a much smaller graph size with a high graph density. Hence, even with just one changed edge, the 5-hop neighborhood can cover over 70% of the entire graph, reducing the benefits of restricted data fetching.

The baseline Graphiler stands out due to its compiler-level optimizations, including expert-engineered primitives, GPU kernels, and data flow graph optimization techniques. Graphiler compiles model architectures into a message passing data flow graph abstraction, implementing operations with GPU kernels. However, it overlooks necessary CPU-based pre-processing techniques like graph sampling and minibatch data loading, which are crucial for model scalability. Consequently, Graphiler can only handle full-graph GNN inference on the GPU, leading to out-of-memory issues for large graphs and deep models. Moreover, it is designed for static graphs and cannot exploit the redundancy in dynamic graph scenarios. Therefore, in cases of medium and large-size datasets, where the real affected area only takes a tiny portion of the whole graph, InkStream outperforms Graphiler despite its highly optimized code.

Table V shows the reduced memory access and computation of InkStream, compared with k-*hop*, for GCN, ΔG=100. Both *InkStream-m* and *InkStream-a* take advantage of efficient intra-layer incremental updates and save 68%-97% of the memory access than the classic method of k-*hop*. With monotonic aggregation functions, 4% to 68% nodes in the theoretically affected area are bypassed for computation in *InkStream-m* thanks to inter-layer propagation. Note that a resilient node can also be visited once its neighbor has been affected in the former layer. So this value will be larger than those reported in Figure 1b. Although *InkStream-a* cannot enjoy any saving on inter-layer effect propagation, the efficient incremental update can always be applied, making it still efficient.

C. Impact of Number of Changed Edges

Using GCN as a representative example, as it is a typical message-passing model, we investigate the impact of the number of changed edges on InkStream's performance. Figure 7 shows the speedup against the classic method of affected area inference with the different number of changed edges between two timestamps. When there is only one changed edge (ΔG=1), the inference time of InkStream(-a/-m) is only 1ms to 51ms across all datasets, while k-*hop* takes 277ms to 3.8s. When refreshing for every 10 changed edges, InkStream(-a/-m) takes tens of milliseconds, while k-*hop* consumes 300ms to 20s. Regardless of the model, the speedup shows a decreasing trend when the number of changed edges ΔG increases. This is because InkStream monitors the changes in nodes to avoid unnecessary computation and memory access. When ΔG increases, more nodes in the neighborhood of a node can be changed, thus increasing the probability of the node being affected. When the affected area is too large and a non-trivial portion of the nodes changes, the savings brought by redundant computation reduction will not be enough to offset the overhead of monitoring the node changes. Result on dataset ogbn-papers100M doesn't fit this trend due to the randomness introduced by the location of changed edges in a graph. This randomness is reduced when ΔG is large.

	PM	CA	YP	RD	PD	PP
RNVV InkStream-m (%)	12	12	26	4	20	68
RMC InkStream-m (%)	68	69	70	69	88	97
RMC InkStream-a (%)	80	81	90	88	94	92

TABLE V: Comparison of InkStream-m and InkStream-a towards k-hop across various datasets, evaluated for GCN, ΔG=100. RNVV: Reduction in the number of visited nodes. RMC: Reduction in memory cost.

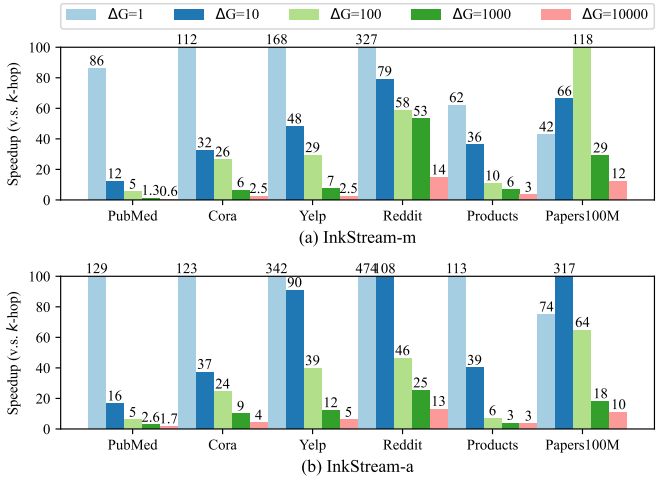


Fig. 7: The effect of the number of changed edges ΔG on the speedup of *InkStream-m* and *InkStream-a* against k-*hop*, evaluated for GCN.

D. Distribution of Pruning and Incremental Update Conditions

If a node can be incrementally updated, we can bypass accessing the whole neighborhood from memory and use the newly generated events in the previous layer for computation. Figure 8 shows the distribution of different applicable conditions for incremental update at the nodes, introduced in section II-C1, as well as pruned propagation. In GCN and GIN, over 70% of the nodes in the affected area are either pruned for computation or can be incrementally updated, resulting in a significant reduction of inference time. In GraphSAGE, the exposed reset constitutes a non-negligible fraction because the model suffers from sensitive node embedding due to self-impact. Moreover, the shallow network provides less opportunity for pruned propagation compared with the deeper model of GIN.

E. Memory Cost

In this section, we analyze the additional memory cost for saving intermediate results of a previous timestamp in InkStream. To avoid over-smoothing, GNNs are usually shallow (k=2 for most cases), with a carefully tuned bandwidth [15]. Compared with the input feature vector (8710 in Cora), the intermediate result of node embedding is light (usually set to 16/32/64/128/256). Moreover, in one layer, InkStream only sets two checkpoints for saving the intermediate result:

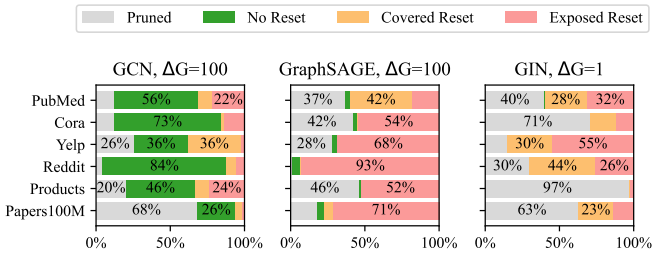


Fig. 8: Distribution of evolvable conditions for nodes in the affected area, evaluated for *InkStream-m*.

TABLE VI: Effect of each component in *InkStream-m* (GCN, ΔG=100). 1: Intra-layer incremental update. 2: Inter-layer pruned propagation.

Time (ms)	*k-hop*	*InkStream-m* (1)	*InkStream-m* (1&2)
PubMed	676 (1×)	219 (3×)	128 (5×)
Cora	3,064 (1×)	261 (12×)	108 (28×)
Yelp	4,218 (1×)	217 (19×)	144 (29×)
Reddit	3,564 (1×)	100 (36×)	61 (59×)
Products	2,188 (1×)	475 (5×)	203 (11×)
Papers100M	27,602 (1×)	361 (76×)	233 (118×)

immediately before and after the aggregation phase. Therefore, the saved intermediate results will not bring exploding memory costs. In our experiments, the additional memory overhead is 0.12-10× the size of the dataset for GCN. However, this is because we are using a relatively large hidden state dimension (256), which is longer than the node feature (100 and 172) for ogbn-products and ogbn-papers100M. When a hidden state dimension of 32 is used, the additional memory cost is reduced to only 0.015-1.28× the size of the dataset.

F. Ablation Study

Our ablation study on *InkStream-m* for the GCN model reveals the impact of each component on inference time. The study, summarized in Table VI, compares the performance of the system with only the intra-layer incremental update (component 1) against the full implementation incorporating both intra-layer incremental updates and inter-layer pruned propagation (components 1&2). When only component 1 is enabled, *InkStream-m* accelerates by only fetching the changed neighbors without visiting the whole neighborhood, similar to *InkStream-a*, resulting in 3-36x speedup. When component 2 is enabled, the inference time is further reduced by 1.5-2.4x, due to shrinking affected area. The combined approach significantly enhances efficiency, underscoring the importance of both components in achieving rapid GNN inference on dynamic graphs.

G. Accuracy Analysis

In this section, we analyze the accuracy of the InkStream method. For **accumulative functions**, incremental methods have been studied in prior works [14], [27]–[29], and proved to be effective with negligible impact on accuracy. For **monotonic functions**, the InkStream method (section II-C1) is equivalent to a well-established method of incremental update

in graph processing algorithms, e.g., single-source shortest path (SSSP). In InkStream-m, for a layer l with a `min` aggregator, the aggregated neighborhood of a vertex u is given by

$$\alpha_{\mathbf{l,u}} = \min(\mathbf{m_{l,v}} : v \in N(u)).$$

Without loss of generality, when looking into a single channel, we have a scalar calculation of

$$\alpha_{l,u} = \min(m_{l,v} : v \in N(u)).$$

This is equivalent to SSSP's distance calculation:

$$d_u = \min(d_v + w_{u,v} : v \in N(u)),$$

which reduces to

$$d_u = \min(d_v : v \in N(u))$$

when edge weights are zero. Incremental updates for SSSP have been extensively studied in works [7], [8], [30], establishing its correctness.

H. Effect of GraphNorm Approximation

Normalization layers play a crucial role in improving the training and inference stability of graph neural networks. However, in the dynamic graph scenario, any change in the vertex set will disturb the mean and variance in a batch of vertices, causing other vertices to be scaled differently as before. This disturbance requires other unrelated vertices to be updated as well. Therefore, there is a need for an approximation of the normalization layer that reduces this disturbance while maintaining the model accuracy.

Given the extensive combinations of models and datasets available, we focus on showcasing the impact of our proposed approximated GraphNorm layer using the Cora and Reddit datasets with a 2-layer GCN model. In the model, a GraphNorm layer comes after each GCNConv layer. Our experimental setup involves initially training the model with a given train set and caching the calculated mean and variance used in the GraphNorm layer for the final inference. Subsequently, we either remove or add a percentage of vertices from the train set and evaluate the model accuracy on the test set.

As shown in Figure 9, the models using the GraphNorm layer with accurate mean and variance, and those with the approximate mean and variance, exhibit minimal accuracy differences (<0.1%). This demonstrates the effectiveness of our approximation method. Moreover, in the application scenario of InkStream, the model is expected to be retrained frequently, and changes in the graph are typically minor between the two retraining phases (<1%). In such cases, the accuracy drop caused by the approximated mean and variance in the GraphNorm layer is negligible.

IV. Related Works

In this section, we discuss the existing literature related to GNNs taking dynamic input graphs.

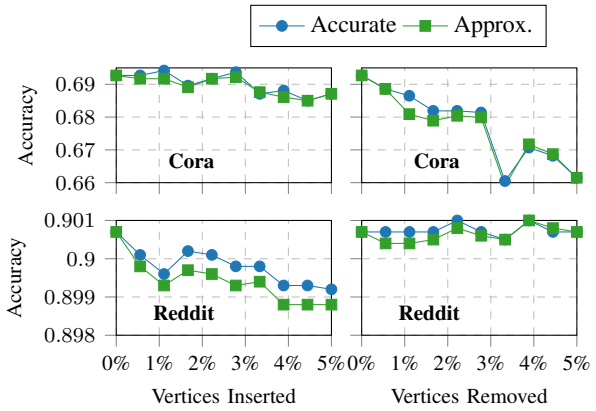


Fig. 9: Model accuracy for 2-layer GCN with accurate and approximate GraphNorm on Cora and Reddit datasets.

A. Dynamic Graphs

Dynamic graphs add the temporal dimension to static graphs. At time t, the graph is represented as $G(V_t, E_t, X_t)$ for vertices, edges and features at time t. To maintain such information, there are two mainstream representations: *discrete-time dynamic graph*s (D-TDG) and *continuous-time dynamic graph*s (C-TDG) [32]–[34]. The D-TDG takes the dynamic graph as a set of snapshots captured at discrete timestamps, while C-TDG treats the dynamic graph as an initial graph with a series of events, e.g., node insertion and edge removal.

Processing dynamic graphs typically involves either *static GNNs* or *temporal GNNs*. Static GNNs are applied to individual snapshots of the graph and serve as the target of InkStream. In contrast, temporal GNNs explicitly incorporate time information to capture temporal dependencies.

B. Static Graph Neural Networks

Real-time inference of static GNN is critical for large-scale graph systems and latency-sensitive applications, requiring rapid adaptation to dynamic and evolving data. BRIGHT [31] pioneered real-time GNN inference on fraud transaction detection tasks by deploying two networks: Batch Net and Real-time Net (RT Net). The 1-layer RT Net achieves rapid inference by reusing previously computed node representations from Batch Net. In object detection using event cameras, SlideGCN [28] and AEGNN [14] focus on evolving graphs constructed from point clouds. These methods incrementally update the k-hop neighborhood of newly observed or outdated points while leaving unaffected areas unchanged. This approach significantly improves efficiency in handling dynamic point cloud data. Also equipped with incremental updates, InstantGNN [29] targets linear systems with infinite-hop propagation rather than traditional GNN architectures. Taking advantage of the linear system, InstantGNN further saves computation by deferring the update of vertices with small changes in node representation. From a hardware perspective, DeltaGNN [27] presents an accelerator implementation for incremental approaches. On the compiler level, Graphiler [22]

TABLE VII: Qualitative comparison of InkStream to prior works of real-time inference. InkStream is adaptable to general GNN architectures with minimal restrictions on model architecture and aggregation function; meanwhile, it enjoys minimal computation brought by reduced update area and incremental update.

Method	Scenario	Graph	Aggregation	Networks	Reduced Updated Area	Inc. Update
BRIGHT-RTNet [31]	Fraud Detection	D-TDG	**Any**	1-layer GConv	No	No
SlideGCN [28]	Object Detection	C-TDG	Accumulative	two proposed architectures	No	**Yes**
AEGNN [14]	Object Detection	C-TDG	Accumulative	GCN + Max Pooling	No	**Yes**
InstantGNN [29]	**General**	C-TDG	Accumulative	Linear System-based GNN	**Yes**	**Yes**
DeltaGNN [27]	**General**	C-TDG	Accumulative	**General**	No	**Yes**
Graphiler [22]	**General**	Static	**Any**	**General**	No	No
InkStream (Ours)	**General**	C-TDG	**Accumulative & Monotonic**	**General**	**Yes**	**Yes**

compiles GNNs with user-defined functions into the proposed message passing data flow graph abstraction and applies optimizations on it to get efficient execution plans. Benefiting from the optimized execution plan with carefully engineered primitives and GPU kernels, Graphiler successfully reduces the inference time to a millisecond level.

While these methods have made significant contributions, they often face limitations in terms of generalizability or efficiency under dynamic graph scenarios. All existing incremental approaches are specialized for certain tasks or model structures, and restricted to using `sum` as the aggregation function. The general approach of Graphiler, however, is designed for static graphs and overlooks the redundancy in the dynamic graphs. Table VII shows the qualitative comparison of InkStream with prior works focusing on instant GNN inference. This comparison highlights the unique features and advantages of InkStream in the context of existing real-time GNN inference methods.

C. Temporal Graph Neural Networks

Temporal Graph Neural Networks extend the capabilities of static GNNs by incorporating time-dependent dynamics, enabling effective modeling of evolving graph structures. In the context of D-TDGs, the graph structure changes at fixed time intervals. DynamicTriad [35] proposed a method that captures both structural and temporal information by modelling the triadic closure process. GCRN [36], DySAT [37] and A3TGCN [38] stacks the RNN layer after the GNN layer, each capturing one kind of information. EvolveGCN [2] introduced an approach where the GNN parameters are evolved using an RNN. EPNE [39] extended node embeddings to dynamic graphs by incorporating temporal random walks and network snapshots.

C-TDGs present a more challenging scenario, as they model interactions that occur at arbitrary time points. JODIE [40] proposed a coupled recurrent neural network architecture to learn time-dependent node embeddings for bipartite graphs. TGAT [41] leveraged self-attention mechanisms to aggregate temporal neighbor information, while other approaches like TGN [42] introduced memory modules to capture long-term dependencies in dynamic graphs.

Recent advancements also focus on improving the efficiency of temporal GNNs. DyGNN [1] enhances computational efficiency by restricting the update area around newly inserted edges and nodes. APAN [43] introduces an efficient attention mechanism that adaptively selects important historical interactions. TGL [44] unifies various temporal GNN models and incorporates a learnable time encoding method. DistTGL [45] improves convergence rate on distributed GPU clusters through novel model architecture and training algorithm. RTGA [46] accelerates temporal GNN inference by reducing redundant computation and memory access through temporal tree structures and temporal-aware data caching. TGLight [47] accelerates runtime performance by providing C-TDG-specific optimizations such as deduplication, memorization, and precomputation.

These approaches have significantly advanced the field of dynamic graph representation learning, enabling more accurate modeling of time-evolving networks across various domains. While temporal GNNs offer superior representation quality, they incur substantially higher computational costs compared to static GNNs, as they must process both temporal and structural information. This computational burden impacts their scalability — for instance, the largest dataset processed by a SOTA temporal GNN acceleration framework, TGLight [47], is merely 1% of the size handled by InkStream. Furthermore, temporal GNNs exhibit higher inference latency, demonstrated by 2-14x longer inference times reported in TGLight [47] on the Reddit dataset, even when using more powerful GPU hardware. This performance gap highlights the need for methods specifically designed for real-time inference in dynamic graphs, precisely where our proposed method, InkStream, aims to contribute.

V. Conclusion

InkStream introduces a novel, efficient approach for real-time GNN inference on dynamic graphs, specifically targeting the HPC environment with large-scale graphs. By minimizing computational overhead through selective propagation and incremental embedding updates, InkStream significantly accelerates processing times without sacrificing accuracy. Our results demonstrate that InkStream reduces inference time from hours to milliseconds across multiple GNN models and datasets. Unlike previous works, which have strong restrictions on model architecture and only one supported aggregation function, InkStream is general enough to express most GNN architectures and all common aggregation functions; it provides a scalable, versatile solution for real-time inference in dynamic settings.

REFERENCES

[1] J. Huang, Y. Chang, X. Cheng, J. Kamps, V. Murdock, J.-R. Wen, Y. Liu, Y. Ma, Z. Guo, Z. Ren, J. Tang, and D. Yin, "Streaming Graph Neural Networks," *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020.

[2] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. Schardl, and C. Leiserson, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5363–5370.

[3] C. Huan, S. L. Song, Y. Liu, H. Zhang, H. Liu, C. He, K. Chen, J. Jiang, and Y. Wu, "T-gcn: A sampling based streaming graph neural network system with hybrid architecture," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 69–82.

[4] S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang, "Streaming recommender systems," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 381–389.

[5] D. Yang, B. Qu, J. Yang, L. Wang, and P. Cudre-Mauroux, "Streaming graph embeddings via incremental neighborhood sketching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 5, pp. 5296–5310, 2022.

[6] P. Bielak, K. Tagowski, M. Falkiewicz, T. Kajdanowicz, and N. V. Chawla, "Fildne: a framework for incremental learning of dynamic networks embeddings," *Knowledge-Based Systems*, vol. 236, p. 107453, 2022.

[7] V. Salapura, M. Zahran, F. Chong, L. Tang, J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang, and H. Yu, "TDGraph: a topology-driven accelerator for high-performance streaming graph processing," *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 116–129, 2022.

[8] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator," *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1091–1105, 2021.

[9] M. Afarin, C. Gao, S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "CommonGraph: Graph Analytics on Evolving Data," *ACM Transactions on Storage*, vol. 15, no. 4, pp. 1–40, 2019.

[10] T. K. Rusch, M. M. Bronstein, and S. Mishra, "A survey on oversmoothing in graph neural networks," *arXiv preprint arXiv:2303.10993*, 2023.

[11] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.

[12] X. Gao, W. Zhang, J. Yu, Y. Shao, Q. Nguyen, B. Cui, and H. Yin, "Accelerating scalable graph neural network inference with node-adaptive propagation," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDE60146.2024.00236

[13] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, "Lightgcn: Simplifying and powering graph convolution network for recommendation," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, 2020, pp. 639–648.

[14] S. Schaefer, D. Gehrig, and D. Scaramuzza, "Aegnn: Asynchronous event-based graph neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12 371–12 381.

[15] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[17] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[18] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[19] T. Cai, S. Luo, K. Xu, D. He, T.-y. Liu, and L. Wang, "Graphnorm: A principled approach to accelerating graph neural network training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1204–1215.

[20] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.

[21] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[22] Z. Xie, M. Wang, Z. Ye, Z. Zhang, and R. Fan, "Graphiler: Optimizing graph neural networks with message passing data flow graph," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 515–528. [Online]. Available: https://proceedings.mlsys.org/paper/2022/file/a87ff679a2f3e71d9181a67b7542122c-Paper.pdf

[23] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*. PMLR, 2016, pp. 40–48.

[24] A. Bojchevski and S. Günnemann, "Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking," *arXiv preprint arXiv:1707.03815*, 2017.

[25] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[26] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[27] C. Yin, J. Jiang, Q. Wang, Z. Mao, and N. Jing, "Deltagnn: Accelerating graph neural networks on dynamic graphs with delta updating," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[28] L. Yijin, Z. Han, Y. Bangbang, C. Zhaopeng, B. Hujun, and Z. Guofeng, "Graph-based asynchronous event processing for rapid object recognition," in *International Conference on Computer Vision (ICCV)*, October 2021.

[29] Y. Zheng, H. Wang, Z. Wei, J. Liu, and S. Wang, "Instant graph neural networks for dynamic graphs," in *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*, 2022, pp. 2605–2615.

[30] K. Vora, R. Gupta, and G. Xu, "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*, 2017, pp. 237–251.

[31] M. Lu, Z. Han, S. X. Rao, Z. Zhang, Y. Zhao, Y. Shan, R. Raghunathan, C. Zhang, and J. Jiang, "Bright-graph neural networks in real-time fraud detection," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 3342–3351.

[32] Z. Feng, R. Wang, T. Wang, M. Song, S. Wu, and S. He, "A comprehensive survey of dynamic graph neural networks: Models, frameworks, benchmarks, experiments and challenges," *arXiv preprint arXiv:2405.00476*, 2024.

[33] L. Yang, C. Chatelain, and S. Adam, "Dynamic graph representation learning with neural networks: A survey," *IEEE Access*, vol. 12, pp. 43 460–43 484, 2024.

[34] A. Gravina and D. Bacciu, "Deep learning for dynamic graphs: models and benchmarks," *IEEE Transactions on Neural Networks and Learning Systems*, 2024.

[35] L. Zhou, Y. Yang, X. Ren, F. Wu, and Y. Zhuang, "Dynamic network embedding by modeling triadic closure process," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[36] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *Neural Information Processing: 25th International Conference, ICONIP 2018, Siem Reap, Cambodia, December 13-16, 2018, Proceedings, Part I 25*. Springer, 2018, pp. 362–373.

[37] A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang, "Dysat: Deep neural representation learning on dynamic graphs via self-attention networks," in *Proceedings of the 13th international conference on web search and data mining*, 2020, pp. 519–527.

[38] J. Bai, J. Zhu, Y. Song, L. Zhao, Z. Hou, R. Du, and H. Li, "A3t-gcn: Attention temporal graph convolutional network for traffic forecasting," *ISPRS International Journal of Geo-Information*, vol. 10, no. 7, p. 485, 2021.

[39] J. Wang, Y. Jin, G. Song, and X. Ma, "Epne: Evolutionary pattern preserving network embedding," in *ECAI 2020*. IOS Press, 2020, pp. 1603–1610.

[40] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1269–1278. [Online]. Available: https://doi.org/10.1145/3292500.3330895

[41] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan, "Inductive representation learning on temporal graphs," *arXiv preprint arXiv:2002.07962*, 2020.

[42] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *arXiv preprint arXiv:2006.10637*, 2020.

[43] X. Wang, D. Lyu, M. Li, Y. Xia, Q. Yang, X. Wang, X. Wang, P. Cui, Y. Yang, B. Sun *et al.*, "Apan: Asynchronous propagation attention network for real-time temporal graph embedding," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 2628–2638.

[44] H. Zhou, D. Zheng, I. Nisa, V. Ioannidis, X. Song, and G. Karypis, "Tgl: A general framework for temporal gnn training on billion-scale graphs," *arXiv preprint arXiv:2203.14883*, 2022.

[45] H. Zhou, D. Zheng, X. Song, G. Karypis, and V. Prasanna, "Disttgl: Distributed memory-based temporal graph neural network training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–12.

[46] H. Yu, Y. Zhang, A. Tan, C. Lu, J. Zhao, X. Liao, H. Jin, and H. Liu, "Rtga: A redundancy-free accelerator for high-performance temporal graph neural network inference," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

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