Helios: Efficient Distributed Dynamic Graph Sampling for Online GNN Inference

Jie Sun[1*], Zuocheng Shi[1*], Li Su[2], Wenting Shen[2], Zeke Wang[1,3]
Yong Li[2], Wenyuan Yu[2], Wei Lin[2], Fei Wu[1,3], Bingsheng He[4], Jingren Zhou[2]
[1] Zhejiang University, China; [2] Alibaba Group; [3] Shanghai Institute for Advanced Study of Zhejiang University, China
[4] National University of Singapore, Singapore

Abstract

Online GNN inference has been widely explored by applications such as online recommendation and financial fraud detection systems, where even minor delays can result in significant financial impact. Real-time dynamic graph sampling enables online GNN inference to reflect the latest graph updates in real-world graphs. However, online GNN inference typically demands millisecond-level latency Service Level Objectives (SLOs) as its performance guarantees, which poses great challenges for existing dynamic graph sampling approaches based on graph databases. The issues mainly arise from two aspects: long tail latency due to imbalanced data-dependent sampling and large communication overhead incurred by distributed sampling. To address these issues, we propose Helios, an efficient distributed dynamic graph sampling service to meet the stringent latency SLOs. The key ideas of Helios are 1) pre-sampling the dynamic graph in an event-driven approach, and 2) maintaining a query-aware sample cache to build the complete K-hop sampling results locally for inference requests. Experiments on multiple datasets show that Helios achieves up to 67× higher serving throughput and up to 32× lower P99 query latency compared to baselines.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; • **Computing methodologies** → **Machine learning**; **Distributed algorithms**.

Keywords: Graph Neural Network, Distributed System, Online Inference

[*] Equal Contribution.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710854

1 Introduction

Graph Neural Networks (GNNs) [24, 27, 39, 47, 69, 89] learn graph vertex representations by aggregating their multi-hop neighbor information, including both the structure and attribute data, which are further utilized in handling inference requests of various machine-learning tasks including node classification and link prediction.

Online GNN inference is increasingly gaining popularity for many applications that depend on real-time decision-making [6, 56, 73, 76, 82]. This is because online GNN inference can reflect the dynamic structure and attributes of real-world graphs. For instance, GNNs are extensively used [6, 55, 56, 73] to identify suspicious scam accounts in financial risk management by aggregating neighborhood information of an account through various relationships, i.e., the transaction between an account and its neighboring accounts. However, training GNNs on large-scale graphs is usually at intervals of hours or days, particularly in production environments [19, 98]. Performing inference using the offline-learned vertex embedding creates a large window of opportunity for fraudsters to escape and causes significant financial loss. Therefore, many works [6, 51, 73, 76] explore online GNN inference to solve this issue by incorporating real-time changes in the graph's structure, as well as the features of its vertices and edges, into the inference representations. For instance, Amazon [6] proposes a fraud detection solution based on online GNN inference to extract the latest transactions involving the target account and features of relevant accounts as inference inputs and feed these inputs into an offline-trained GNN model to assess the likelihood of the target account being involved in a fraud.

Online GNN inference service typically requires a challenging *millisecond-level latency Service Level Objective (SLO)*, e.g., 100ms in a recommendation system for social media service [31]. Neighbor sampling [39] is widely adopted in industrial applications [19, 86, 98] to scale GNN training and inference to large-scale graphs (e.g., graphs with billions edges [98]). For online GNN inference, *real-time* neighbor sampling on the dynamic graph guarantees that the inferred vertex representation can accurately reflect real-time graph updates [6, 50, 51]. Existing approaches [6, 51] leverage graph database systems [5, 8, 28, 50, 77] for dynamic graph storage and real-time neighbor sampling. However, we identify that dynamic graph sampling takes over 90% latency of GNN

```
## Input: a 2-hop query
g.V('User', ID).alias('Seed')
.OutV('Click').sample(2).by('Random')
.OutV('Co-purchase').sample(2).by('TopK').values
## Decomposed into two distinct one-hop queries:
Q1: V('User', ID).outV('Click').sample(2).by('Random')
Q2: V('Item', ID).outV('Co-purchase').sample(2).by('TopK')
```

Figure 1. 2-hop Query Decomposition.

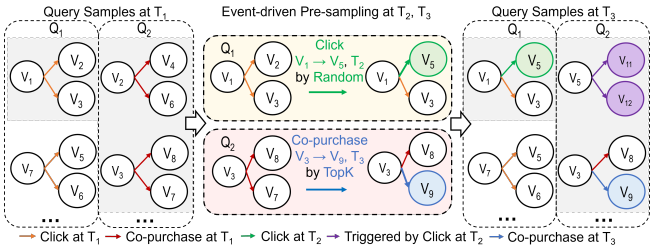

Figure 2. Pre-sampling Driven by Graph Update Events.

inference and consistently surpasses 100ms in existing approaches (see Figure 4(a)), which makes it hard to fulfill the stringent latency SLO due to two severe issues:

(1) Long Tail Latency due to Imbalanced Data-dependent Sampling. Sampling strategies like `TopK` and `EdgeWeight` require traversing all the neighbors of a vertex to obtain the samples. The amount of data access and computation required by such sampling operations can vary significantly among different seed vertices (see Figure 4(c)), due to the inherent distribution skewness in real-world graphs [33, 38]. Consequently, the P99 latency easily surpasses the average latency over 150 ms, as shown in Figure 4(b).

(2) Large Network Communication Overheads. To handle large-scale graphs and high-throughput inference requests, graph databases such as NebulaGraph [77] and TigerGraph [28] support storing and sampling the graphs in a distributed manner. However, distributed multi-hop sampling typically requires repeatedly sampling vertices across multiple machines. The costs of network communication increase with the increasing number of sampling hops (see Figure 4(d)). Although graph databases like Neo4j [8] can utilize the query cache to reuse results of previously executed queries, the continuous updates in dynamic graphs render most query caches unavailable, significantly limiting the cache hit ratio.

In this work, we propose Helios, an efficient distributed dynamic graph sampling service to tackle the challenging latency SLOs for online GNN inference. Our key insight is that *during GNN inference, the pattern of the graph sampling query of a given GNN model, i.e., sampling fan-out number, hop number, and sampling strategy, is determined by how the model is trained* [39, 86, 89, 97]. This is because a GNN model trained with a specific sampling pattern learns from the corresponding sampled data distribution. If the sampling pattern varies significantly during inference, the model's performance can degrade and become unstable[1] due to shifts in data distribution [97]. As such, the key idea is to pre-sample the dynamically updated graphs according to the given sampling query and continuously *push* the complete multi-hop neighborhood samples of each seed vertex to the same machine serving for inference requests. Hence, building a complete sampling result for an inference query only requires a fixed number of local cache lookups, substantially reducing the sampling latency. We achieve these by three key designs:

Event-driven Pre-sampling. By event-driven pre-sampling, Helios transfers the sampling computation from real-time inference to the graph update process. Specifically, Helios decomposes a K-hop sampling query into K one-hop sampling queries, and proactively maintains the results of these one-hop sampling queries following the continuous influx of graph updates. Figure 1 is an example of a 2-hop query used in e-commerce recommendation [86, 98], which is decomposed into two distinct 1-hop queries: Q_1 and Q_2 for the first-hop and second-hop sampling, respectively. As Figure 2 shows, when a new `Click` event arrives, Q_1 takes this edge and the previous sampling results of V_1 as inputs to generate the new sampling results for V_1. Similarly, the sampling results of V_3 are updated by Q_2 with a new `Co-purchase` event. § 5 describes more details on this process.

Query-aware Sample Cache. To scale out the throughput of serving sampling queries and minimize the query latency, Helios slices target inference vertices into multiple serving servers and maintains a query-aware sample cache in each server. Helios tracks dynamically changing one-hop sampling results essential for constructing complete K-hop sampling results for each target vertex. For instance, in Figure 2, once V_5 is selected as a new first-hop sample for V_1, the sampling results of V_5 in Q_2 will be added into the query aware cache serving the inference on V_1. With the query-aware sample cache, generating K-hop sampling results only requires a fixed number of local cache lookups.

Implementing event-driven pre-sampling and query-aware sample cache is non-trivial. In real-world applications, both graph updates and GNN inference often experience sudden workload bursts that do not align, posing great challenges to the scalability of both stages.

Sampling/Serving Separation. To tackle the scalability challenge, Helios adopts a decoupled distributed architecture: sampling workers pre-sample the dynamic graph and push the results to the serving workers; each serving worker maintains a query-aware sample cache and generates sampling

[1] Many online applications like online recommendations [16] require stable service quality, i.e., accuracy.

results for inference requests without incurring communication overhead. As such, the sampling and serving workers can scale independently, enabling Helios to handle both high-concurrency inference requests and high-throughput graph updates. However, the separation architecture makes it challenging to trace the dynamically changing K-hop sampling results for the sample cache. We address this with an efficient event-driven subscription mechanism.

Experiments on multiple datasets show that Helios consistently achieves a P99 latency of subgraph sampling (of a two-hop TopK query) within 50 milliseconds. Each serving worker can handle over 4000 queries per second while exhibiting linear scalability. The pre-sampling throughput of a single sampling worker exceeds $1.49M$ records/s and also demonstrates near-linear scalability. Compared to state-of-the-art baselines, Helios brings up to a 67× increase in serving throughput and up to 32× latency reduction. Helios is implemented and deployed in Alibaba's internal GNN training and inference stack and is open-sourced at: https://github.com/alibaba/graph-learn (as part of Graph-Learn [98]).

The contributions of this paper are:

- We present Helios, the first system that shifts the ad-hoc dynamic graph sampling into the graph update process through event-driven pre-sampling.
- We propose a query-aware sample cache, enabling Helios to meet the strict latency SLOs of graph sampling through efficient local cache lookups.
- We develop a separated sampling/serving distributed architecture that allows Helios to scale almost linearly for both graph update ingestion and GNN inference serving.

2 Preliminaries

2.1 Sampling-based Graph Neural Network

For a graph $G = (V, E)$, GNNs compute compact representations for each target vertex using L neural network layers. At layer $l, l \in L$, the activation of vertex $v, v \in V$ updated by aggregating the features or hidden activations of its neighbors, denoted as $N(v)$:

$$\begin{aligned} a_v^l &= AGGREGATE^l(h_u^{l-1} | u \in N(v)) \\ h_v^l &= UPDATE^l(a_v^l, h_v^{l-1}) \end{aligned} \tag{1}$$

To scale GNNs for large graphs, industrial applications often use mini-batch training [24, 39, 89, 98]. For example, training GraphSAGE [39] involves: (1) selecting a mini-batch of training vertices; (2) uniformly sampling multi-hop neighbors based on specified fan-outs[2]; (3) collecting features of training vertices and their sampled neighbors; (4) applying *AGGREGATE* and *UPDATE* (Equation 1) for forward and backward propagation to update the model.

[2]Graph sampling can introduce gradient bias, especially with limited neighbors [25, 89], but this can be mitigated using techniques like reusing historical activations and feature dropout [25].

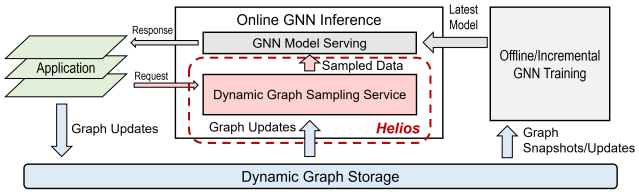

Figure 3. End-to-end Industrial GNN Deployment Workflow on Dynamic Graphs. Helios focuses on the dynamic graph sampling service for online GNN inference.

2.2 Workflow of GNN Deployment

Figure 3 illustrates an example of end-to-end GNN deployment on dynamic graphs in industrial practice. In this example, graph updates (e.g., user clicks) are persisted in dynamic graph storage, which serves as the data source for both training and inference.

GNN Model Training. Training GNNs on large-scale graphs is both resource-intensive and time-consuming [36, 65, 66, 72, 98]. In industrial applications, GNN models are typically updated at intervals of hours, days, or even longer using the approach of offline training on snapshots of the dynamic graph. Incremental training [78, 84] can reduce the training costs by updating the model using the newly arriving data or updates in the graph, rather than retraining the entire model from scratch.

Online GNN Inference. While incremental training can facilitate near-line model deployment (e.g., on an hourly basis), it cannot capture real-time graph updates, which is essential in real-time decision-making scenarios like real-time recommendation [40, 81, 86, 88], traffic prediction [82] and financial fraud detection [56, 73, 87]. This gap necessitates the approach of online inference that can seamlessly integrate real-time updates in graph structures and attributes into the inference representations. The process of online inference involves sampling multi-hop neighbors on dynamic graphs for the vertex indicated in the request and employing the GNN models to infer the vertex representation.

3 Motivation

Stringent millisecond-level latency SLOs [31, 64] of online inference services are very challenging to existing dynamic graph sampling approaches. We validate by deploying TigerGraph [28] and NebulaGraph [77] using a cluster of 10 nodes[3] with the Interactive dataset (see Table 1). The sampling query is a 2-hop TopK query with fan-outs as [25, 10] (see § 7.1 for more cluster and query details). The inference request concurrency is 200 (see more results in § 7.2). Figure 4(a) shows graph sampling accounts for over 90% of the end-to-end latency and consistently surpasses the 100ms latency SLO in both systems. The issues mainly come from two aspects:

[3]The model service is implemented by TensorFlow Serving [14] and deployed on 4 extra nodes.

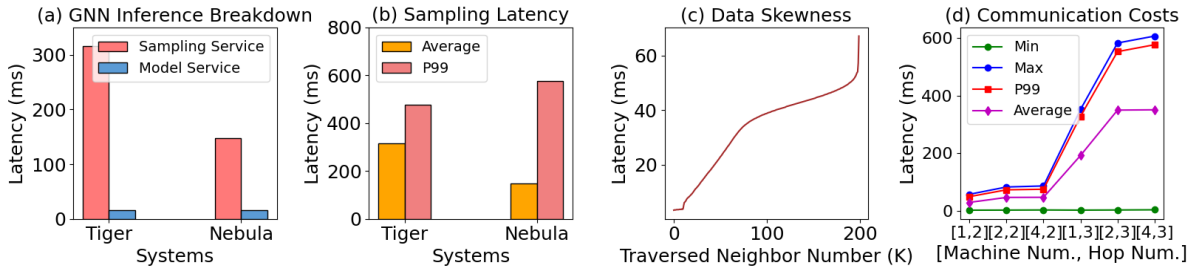

Figure 4. Issues of existing dynamic graph sampling systems motivate the design of Helios. $[x, y]$ in (d) represents the query is evaluated in a x-node cluster and the sampling consists of y hops. The dataset evaluated is Interactive [18].

(1) long tail latency due to imbalanced data-dependant sampling, and (2) large network communication overheads from distributed multi-hop sampling.

3.1 Long Tail Latency

Figure 4(b) shows that the P99 latency of both systems is significantly higher than their average latency. This is because the computation cost required by the ad-hoc sampling query varies significantly based on different sampling seeds due to the skewness of real-world graphs [38, 52]. For instance, in a timestamp-based TopK sampling, to select the K neighbors with the largest timestamps for a vertex V_i, the timestamp of every edge connecting V_i and its neighbor vertex has to be collected and sorted. Moreover, supernodes often exist in real-world graphs, e.g., celebrities in social networks [75] and popular items exist in e-commerce graphs [98]. Traversing and sampling the neighbors of supernodes significantly increase the I/O and computation intensity.

We conduct an experiment on our cluster using TigerGraph to demonstrate the effects of data skewness on sampling query latency. Specifically, we load the Interactive dataset [18] (see details in Table 1) into TigerGraph and randomly select 200,000 vertices as seed vertices of the two-hop timestamp-based sampling with fan-outs as $[25, 10]$. To accurately evaluate the effects of skewed computation on query latency, we set up the database on a single machine and execute the queries sequentially, avoiding any influence from distributed sampling or concurrent query execution. We gather the number of traversed vertices and the execution latency for each query. Figure 4(c) shows that there exists a difference of more than 100× in the number of neighbor vertices accessed by various queries. As the number of traversed vertices grows, the query latency surges by approximately 20×. This result confirms that the skewness of graph distribution significantly impacts the computational workloads of sampling queries, leading to long tail latency.

3.2 Network Communication Overhead

The scale of graphs and the workload of concurrent inference requests in industrial settings often exceed the capacity of a single machine. Distributed graph databases are often used to store graph data and execute sampling queries in a distributed cluster, where each machine stores a partition of the graph. Given that GNN inference typically involves multi-hop sampling, distributed graph storage inevitably incurs communication overhead during the processes of traversing neighboring vertices and fetching the features of neighbors.

We conducted experiments using TigerGraph with the Interactive dataset [18] to assess the effects of both the number of sampling hops and the cluster size on distributed sampling latency. The fan-out of the 3-hop (2-hop) query is $[25, 10, 5]$ ($[25, 10]$). The query latency for various configurations is presented in Figure 4(d). The results indicate that increasing the number of sampling hops from 2 to 3 significantly elevates the query latency by over 6.52×. This is because 3-hop sampling necessitates an additional round of cross-machine communication compared to 2-hop sampling. Furthermore, when compared to single-machine sampling, distributed sampling introduces a substantial latency spike of up to 1.82×.

4 System Design

To solve the issues in § 3, we present Helios, the first specialized (non-graph-database) dynamic graph sampling service designed to fulfill the stringent latency SLOs for online GNN inference. Inspired by the key insight that *the sampling query pattern of GNN inference is determined by how the model is trained*, Helios shifts ad-hoc sampling query execution to the process of graph updates. Helios adopts a sampling/serving separation architecture: Sampling workers proactively *refresh* the sample results by an event-driven pre-sampling mechanism and *push* the multi-hop pre-sampled results to a designated serving worker, ensuring that GNN inference queries can be completed through a fixed number of local cache lookups in a single serving worker.

4.1 Separation of Sampling & Serving

Figure 5 presents the overall system architecture of Helios. In a Helios deployment with M sampling workers and N serving workers, graph updates are evenly partitioned into M partitions, each of which is designated to a sampling worker. Inference requests are similarly partitioned into N partitions according to the IDs of the seed vertex, with each serving

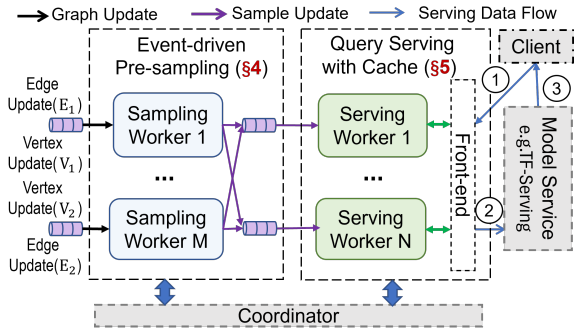

Figure 5. Helios Overview

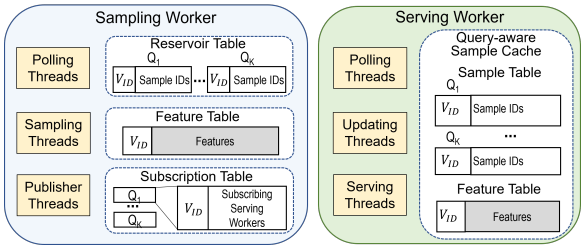

Figure 6. Architecture of Sampling/Serving Worker

worker exclusively handling one partition[4]. Helios adopts Kafka [11] to persistently store and transfer the inputs for sampling and serving workers. Helios allows users to configure their sampling query. The coordinator registers the user-specified query, decomposes the K-hop query into one-hop queries, and initializes these one-hop queries in the sampling and serving workers. The coordinator models the data dependency between one-hop queries as a directed acyclic graph and sends it to all workers to facilitate the management of the subscription table (see § 5.3 for more details) and the construction of complete sampling results. While the service is running, the coordinator monitors the liveliness of all workers via heartbeats and periodically triggers checkpointing for fault tolerance.

The separation of sampling and serving processes enables independent scaling of sampling and serving workers, allowing the system to adapt to dynamically varying workloads. This design also provides physical isolation of the workloads, ensuring stable serving latency during temporary graph update bursts in the sampling workers. We recommend users first assess the throughput of a single sampling or serving worker, and then scale out the sampling and serving workers according to the application's requirements, such as the graph update rates and inference query rates.

4.2 Sampling Worker

Figure 6 shows the detailed architecture of the sampling worker. With the continuously arriving graph updates, the sampling worker proactively updates the results for each one-hop sampling query, which can be further used to construct the complete K-hop sampling results for a query vertex.

Graph Updates. Helios categorizes graph updates into two types: (1) vertex update, denoted to as `VertexUpdate`(V_i), and (2) edge update, referred to as `EdgeUpdate`(E_i). A vertex update corresponds to an insertion of a new vertex or the feature update of a previously observed vertex, and an edge update always denotes the insertion of a new edge E_i. Helios focuses on append-only dynamic graph scenarios, which is common in many logging-based graph systems [1]. For example, when a user clicks on a news article on a website (e.g., Google News [53]), this action is logged and retained. Similarly, in financial networks, once a transfer between two accounts is committed, it cannot be deleted. Helios allows users to configure a time-to-live threshold, according to which the stale data is periodically removed.

Partition Strategies. In a deployment with M sampling workers, graph updates are evenly sliced into M partitions and each sampling worker exclusively handles one partition. A pre-defined hash function is used to determine the partition IDs of vertex updates. For an edge update E_i, if the graph is undirected, E_i is replicated in the partitions of both its source and destination vertices. If the graph is directed, Helios allows users to configure edge storage policy: (1) the `BySrc` policy partitions edge E_i in the ID of the source vertex; (2) the `ByDest` policy determines the partition ID of E_i in the ID of its destination vertex; (3) the `Both` policy will apply the same edge partitioning approach as in an undirected graph.

Components in Sampling Worker. The sampling worker mainly consists of three components: (1) A reservoir table for each one-hop query, where the *key* is the vertex ID and the values are the sampled neighbor vertex IDs for the corresponding vertex. The reservoir capacity is determined by the query fan-out. (2) A feature table that stores the dynamically updated feature for each vertex in the local partition of this sampling worker. (3) A subscription table for each one-hop query, which maintains the list of serving workers subscribing to the feature and sample updates of key vertices in the reservoir table of this query. More details on these components' functionality are explained in § 5.

Execution Engine of Graph Update Ingestion. Helios pipelines IO and computation in sampling workers and minimizes the interference among different types of workloads by isolating them into distinct threads, which are implemented by a distributed actor-based framework. There are three types of threads in sampling workers: (1) Polling threads continuously fetch the latest graph updates from the input queue. (2) Sampling threads execute the one-hop sampling queries and update subscription tables. (3) Publisher threads push the sampled results to the corresponding output Kafka queues according to the subscription table. Helios can prioritize workloads by assigning them to a larger thread pool.

[4]Helios allows rehashing of vertices for sampling workers and replicating the highly loaded serving workers based on the ad-hoc skewness.

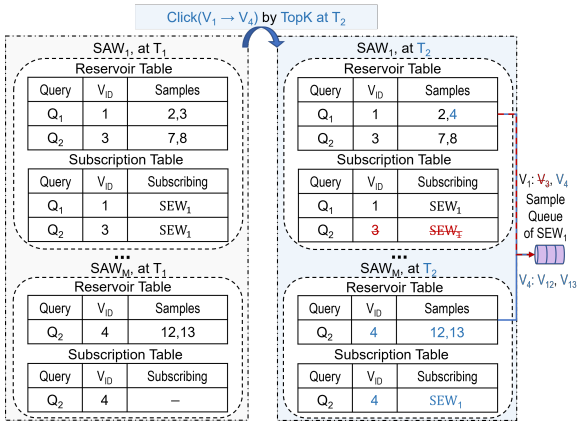

Figure 7. Example of subscription table update with a 2-hop query triggered by sample update changes from timestamp T_1 to T_2. SAW_i represents the sampling worker i and SEW_j represents the serving worker j.

4.3 Serving Worker

Figure 6 shows the detailed architecture of the serving worker. The serving worker subscribes to the outputs of the sampling workers and promptly responds to the sampling queries in inference requests. Each serving worker serves sampling queries of GNN inference requests independently without introducing extra network communication overheads, drastically cutting down the serving latency, while also linearly scaling the serving throughput with more serving workers (resolved the issue in § 3.2). Moreover, the number of local cache lookups required to generate the full sampling result is bounded by the product of the multi-hop sampling fan-outs. This reduces the tail latency even when sampling on highly skewed graphs (resolved the issue in § 3.1).

Components in Serving Worker. Each serving worker maintains a query-aware sample cache that consists of two parts: (1) a sample table for each one-hop sampling query that stores the pre-sampled neighbors of vertices, and (2) a feature table that maintains the latest feature for all the existing vertices in the sample tables, including all the seed and sampled neighbor vertices (see § 6 for more details). Helios also allows users to configure a time-to-live threshold to remove the stale data in the sample cache.

Execution Engine of Sampling Query Serving. The front-end node routes inference requests to serving servings according to the IDs of their seed vertices. On receiving a sampling query for an inference request, the serving worker constructs the query result by referencing its local sample cache. Similar to sampling workers, serving workers designate different workloads to different physical threads by the distributed actor-based framework: (1) Polling threads continuously fetch the latest samples from the input queue. (2) Data updating threads update the sample and feature table. (3) Serving threads execute the sampling queries received from the front-end nodes and further send sampled results to model services such as TF-serving [14].

5 Event-driven Pre-sampling

As outlined in § 4.1, the pre-sampling process in a sampling worker is triggered by the arrival of graph updates. The pre-sampling of a K-hop sampling query mainly consists of three steps: (1) K-hop query decomposition, (2) event-driven reservoir sampling, and (3) updating the subscription table and publishing samples.

5.1 Query Decomposition

To pre-sample a multi-hop query, the coordinator decomposes a K-hop sampling query into K one-hop queries. Figure 1 shows an example 2-hop query used in an e-commerce recommendation context [86, 98]. This query defines a process of 2-hop sampling that starts from a given `User` vertex. It first randomly samples two neighbors (item vertices) associated with `User` ID via the `Click` edge. Subsequently, it samples two neighbors for these one-hop sampled neighbors through the `Co-purchase` edge, prioritizing edges with larger timestamps. This query is divided into two distinct one-hop queries: Q_1 and Q_2. Specifically, Q_1 performs the first-hop `Random` sampling on the `Click` edge, Q_2 conducts the second-hop `TopK` sampling on the `Co-purchase` edge, and the inputs of Q_2 are the outputs of Q_1.

5.2 Event-driven Reservoir Sampling

On receiving the decomposed queries, sampling workers retrieve the graph updates from their input queues and refresh the sampling results for each one-hop query. Specifically, a reservoir table is maintained for each one-hop query. The keys in each reservoir table are the target vertex IDs of the corresponding one-hop query, e.g., the IDs of `User` vertices for Q_1 in Figure 1. Each value cell stores the IDs of the sampled neighbors corresponding to the key vertex in the reservoir table. Note that, the capacity of value cells is determined by the fan-out of the query. For instance, the cell capacity of Q_1 is 2. The feature table in a sampling worker stores the latest features of vertices in its local partition. As explained in § 4.2, each sampling worker handles a partition of the graph updates, ensuring *no duplication* among all sampling workers for the keys in their reservoir/feature tables.

When processing an edge update, $(E_k : V_i \rightarrow V_j)$, if V_i is the target vertex of a one-hop query Q_k, the sampling results for V_i under Q_k are updated based on the sampling strategy of Q_k. If V_j is chosen as a new sample for V_i, the reservoir table of Q_k will be updated. If the value cell of V_i is full, a previously sampled neighbor of V_i will be replaced by V_j (determined by the sampling algorithm). Otherwise, V_j is added as a new sample for V_i. Upon receiving a vertex update, the sampling worker updates the corresponding entry in the feature table with the latest vertex feature.

Helios can support sampling strategies like `Random`[5] and `TopK`. The data distribution of reservoir sampling is the same

[5]Other sampling strategies like `EdgeWeight` [42] are similar to `Random` [70] which replaces the random generator with the ones considering other probability distributions.

as ad-hoc sampling [42, 70]. When determining if a neighbor V_j of vertex V_i should be chosen as a new sample, `Random` reservoir sampling generates a random number p in range $[1, x]$, where x is the total number of input edge updates to the value cell of V_i at this moment. If p is not larger than the value cell capacity C, the p-th item in the cell will be replaced by V_j. In the timestamp-based `TopK` reservoir sampling, the sampling worker compares the timestamps of the incoming graph update and those of the existing samples and replaces the oldest sample.

5.3 Subscription Update and Sample Publish

As shown in Figure 7, Helios maintains a subscription table in sampling workers for each one-hop query, tracking how sample results are disseminated to serving workers.

After pre-sampling with a graph update, the subscription table will be updated correspondingly. Figure 7 presents an example of updating the subscription table. At time T_1, the serving worker SEW_1 subscribes to the samples of vertex V_1 from Q_1, and vertex V_3 from Q_2, as V_3 is a first-hop sample of V_1. At time T_2, V_4 is chosen as a new sample for V_1 in Q_1, and V_3 is substituted by V_4. If V_1 is the only vertex that triggers the subscription relationship between SEW_1 and V_3, SEW_1 will be deleted from the subscription list of V_3 in Q_2. As V_4 is a new sample for V_1, SAW_1 will send a message to notify SAW_M that SEW_1 should be added to the subscription list of V_4 in Q_2. This will result in SAW_M sending the sample vertices of V_4 in Q_2 and their features to SEW_1.

Sampling workers distribute sample updates to the serving workers' sample queues based on their subscription tables. It's important to note that when vertices are no longer under the subscription of a specific serving worker, the sampling workers also enqueue an `update` message to the sample queues. Serving workers then retrieve messages from the designated sample queues and update the caches accordingly.

5.4 Discussion

Helios is designed for applications that require stringent latency SLO. Thus we adopt a design that trade-offs pre-sampling costs with the performance of real-time serving. Helios performs pre-sampling for every graph update relevant to the sampling query. This will lead to some sampling results that are never accessed by inference requests. However, the ratio of such sampling results is usually quite limited, because: (1) inference requests continuously arrive and the sampled results are accessed not only when they are used as the sampling seeds but also as the multi-hop neighbors of some vertices, and (2) sampling results of vertices that are never used tend to be updated less frequently. We show that these non-accessed sampling results have negligible impact on serving latency/throughput due to the design of sampling/serving separation, as shown in § 7.2.3.

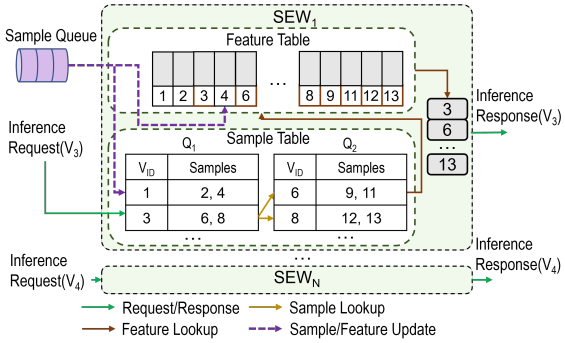

Figure 8. Example of cache updating and query processing with the query-aware sample cache. SEW_j represents the serving worker j.

6 Query-aware Sample Cache

As outlined in § 4.3, each serving worker maintains a query-aware sample cache that holds a partition of vertices with their complete K-hop neighbors and the corresponding features. As depicted in Figure 8, the cache consists of a sample table and a feature table, both of which are key-value (KV) stores and are implemented using the hybrid-memory-disk mode of RocksDB [32].

Serving Sampling Queries. On receiving an inference request, Helios routes the request to a specific serving worker according to the seed vertex ID. For example, in Figure 8, serving worker SEW_1 receives the inference `Request`(V_3). Subsequently, SEW_1 initiates lookup operations in its local sample table and the feature table: Serving worker SEW_1 searches for the samples related to vertex V_3 and gets vertices V_6 and V_8 as the first-hop samples. Subsequently, it continues by iteratively searching for samples corresponding to vertices V_6 and V_8, eventually assembling the complete set of K-hop samples. After obtaining the IDs of sampled vertices, the serving worker then accesses the feature table to retrieve the features of all sampled vertices to compose the final sampling result. Assuming that the fan-outs of the K-hop sampling query are $C_1, C_2, ..., C_K$, the number of lookup operations in the sample table can be calculated as $\prod_{i=1}^{K-1} C_i$, and the number of lookup operations in the feature table can be calculated as $\prod_{i=1}^{K} C_i$. The query-aware sample cache in serving workers optimizes sampling query latency and facilitates linear scaling of serving throughput by eliminating network communications. Constrained by consistent cache lookup costs, Helios effectively minimizes sampling latency, even when handling dynamic graphs with significant skewness.

Cache Update. Each serving worker continuously polls the sample queues to get the updated sample data, which are used to update the sample and feature table. Helios only caches the sampled graph topology and feature data in serving workers, which are much smaller compared to the original graph. When scaling out serving workers, the cache size of each single worker decreases with more serving workers deployed, though the caches could partially overlap among

Table 1. Dataset Statistics.

Dataset	Vertex Number	Edge Number	Feature Dim.	Out Degree (Max/Min/Ave.)
BI	1.9B	2.4B	10	8,525/ 0/ 1.26
INTER	40M	3.8B	10	3,632/ 0/ 95
FIN	2M	2.2B	10	9,831/ 0/ 5.5
Taobao	1.8M	8.6M	128	3,726/ 2/ 4.8

Table 2. Sampling Queries

Dataset	Query Pattern	#Hops	Fan-outs
BI	`Person-Knows-Person-Likes-Comment`	2	[25,10]
INTER	`Forum-Has-Person-Knows-Person`	2	[25,10]
FIN	`Account-TransferTo-Account-TransferTo-Account`	2	[25,10]
Taobao	`User-Click-Item-CoPurchase-Item`	2	[25,10]
INTER	`Forum-Has-Person-Knows-Person-Knows-Person`	3	[25,10, 5]

different serving workers. As shown in Figure 16, the cache size on each serving worker reduces from 62% to 19% of the original dataset size by increasing the number of serving workers from 1 to 4. In real-world online GNN inference workloads, such as online recommendation and fraud detection, graph updates prompted by individual user actions typically occur at intervals of several seconds or even longer granularity [4, 9, 18, 29, 60, 91, 93]. Helios can achieve second-level ingestion latency under input rates of millions of updates per second, which is sufficient to reflect the latest graph updates in online GNN inference scenarios (see experiments in § 7.4).

Consistency. We compare the impact of different consistency guarantees between graph updates and query serving on inference accuracy. Assuming that there are new graph updates and concurrent inference requests. For sampling queries on dynamic graphs, a single graph update can result in multiple neighbor updates to K-hop sampling results for relevant vertices. In the following, we discuss four consistency cases.

Under a strong consistency guarantee, when the graph updates are completely ingested (**Case 1**), query serving can observe 100% updates, which is an optimal case where all the latest graph updates are immediately visible to serving requests. When the graph updates are not completely ingested (**Case 2**), the query serving can observe 0% new updates. Under eventual consistency, when graph updates are completed (**Case 3**), query serving can observe all updates like case 1. When graph updates are not completed (**Case 4**), query serving can partially observe, e.g., 50%, updates. Because GNN can aggregate multi-hop neighbor information, eventual consistency (Cases 3 and 4) allows Helios to observe as many sampling updates as possible during query serving, thereby enabling Helios to reflect graph updates more rapidly. Therefore, Helios guarantees eventual consistency. We show in § 7.4 that inference accuracy of Helios can be close to the optimal case (Case 1) where all the latest graph updates are immediately visible to serving requests.

7 Evaluation

We study the performance of Helios in the following aspects:

E1, Overall System Performance. We study the graph update ingestion and query serving performance of Helios by comparing with state-of-the-art graph databases, and also the impact of sampling/serving design in Helios (§ 7.2).

E2, System Scalability. We study how well pre-sampling and serving in Helios can scale up in a many-core server and scale out in a distributed cluster (§ 7.3).

E3, Microbenchmarks. We design microbenchmarks to study the impacts of sampling hop numbers, and the scale of the sample cache in serving workers and study the data ingestion latency and inference accuracy of Helios. (§ 7.4).

E4, Online Deployment Performance. We deploy an online GNN inference service with Helios to evaluate the inference performance (§ 7.5).

7.1 Experiment Setup

Platform. The experiments are conducted using a Kubernetes cluster with 10 nodes. Each node has an Intel(R) Xeon(R) Platinum 8269CY (2 × 16 threads) CPU @ 2.50GHz, 128 GB memory, 200GB cloud-based ESSD and 10Gbps network.

Datasets. We conduct experiments on three dynamic graph datasets from the LDBC graph benchmark [18, 60] and an industrial dynamic graph dataset from Taobao [29]. Table 1 shows the dataset characteristics. The LDBC-Interactive (INTER), LDBC-Business (BI), and LDBC-finbench (FIN) datasets are from the LDBC social network benchmark [18] and financial benchmark [60]. We replay the four datasets to simulate continuously arriving dynamic graph updates. For the FIN dataset, we replay the edge updates 200 times with increasing timestamps to scale the number of edges to 2 billion.

Sampling Queries. Existing approaches and GNN benchmarks [13, 39, 46, 54] mainly evaluate two-layer or three-layer GNNs, as deeper GNNs are less used in practice due to the over-squashing effects [17] and over-smoothing effects [21]. Table 2 summarizes the queries used in the experiments[6]. Unless explicitly explained, we use a two-hop query for each dataset by default. In the microbenchmark experiment (**E3**), we implement a three-hop query on the INTER dataset to stress Helios[7]. For each query, we implement two different sampling strategies: (1) `TopK` sampling and (2) `Random` sampling. When evaluating the serving performance, the results are collected by an average of 10 times experiments, and each time we randomly select 10,000 vertices as seed nodes of the sampling queries.

Baselines and System Configurations. To the best of our knowledge, Helios is the first specialized (non-graph-database) system that supports real-time graph sampling on dynamic graphs. GNN frameworks like DGL [72] and PyG [34] are mainly used for model training and do not

[6]More details of sampling queries are in the supplementary material.

[7]Note that Helios can support arbitrary hops, according to user's specification

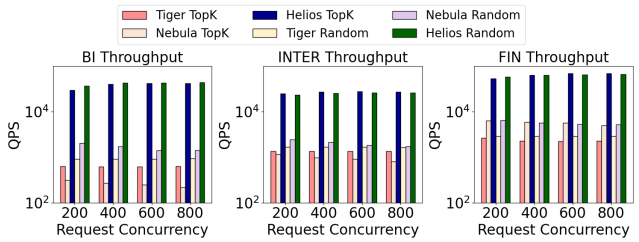

Figure 9. End-to-end Throughput of Helios and the Baselines with TopK and Random queries.

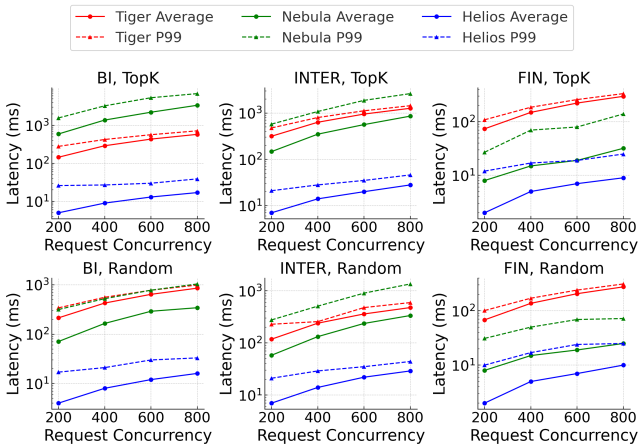

Figure 10. End-to-end Latency of Helios and the Baselines with TopK and Random Queries.

support dynamic graph updates. Graph databases support dynamic graph storage and ad-hoc sampling query execution, and thus are commonly adopted in online GNN inference services. E.g., Amazon adopts Neptune [5], a graph database service, to support GNN inference on dynamic graphs [6]. We choose two state-of-the-art distributed graph databases, NebulaGraph [77] and TigerGraph [28], as baseline systems[8].

For TigerGraph, we run the experiments under the officially recommended regular query mode [12] instead of the distributed query mode, since a GNN inference query starts from a single vertex and usually traverses a small subgraph. For all the compared systems, the number of threads utilized in each node is set to 32 by default (the CPU can hyper-thread up to 32 threads). For Helios, we deploy a Kafka cluster of 6 nodes, and the number of input Kafka partitions for both sampling and serving workers is set to 24. We set a TTL threshold in Helios to ensure no graph data are expired.

7.2 Overall System Performance

We compare the serving and graph update performance of Helios with baselines on all billion-scale benchmarks (BI, INTER, FIN). TigerGraph and NebulaGraph use all 10 nodes as their servers. Helios uses 4 nodes as sampling workers and 6 nodes as serving workers. In § 7.2.1 and § 7.2.2, we measure the performance of graph update and serving processes separately, i.e., by freezing one and measuring the other. In § 7.2.3, we study how Helios can physically isolate the workload of graph updates and GNN inference.

[8]We do not compare with Neptune as a fully managed service that can only be deployed on AWS. TigerGraph outperforms Neptune in terms of data ingestion throughput and query execution latency [15].

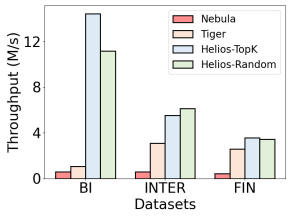

Figure 11. Comparison of Graph Update Ingestion Throughput in Helios and Baselines.

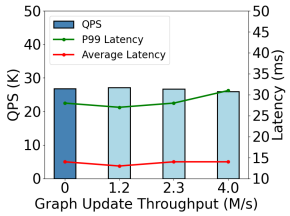

Figure 12. Impact of Helios's Sampling/Serving Separation on the INTER Dataset.

7.2.1 Serving Performance

Figures 9 and 10 show the end-to-end serving throughput and latency. We increase the request concurrency, i.e., the number of clients sending inference requests concurrently, to stress the systems.

Serving Throughput (QPS). Figures 9 present the serving throughput. Compared to baselines, Helios achieves an up to 184× throughput improvement with the TopK query and up to 47× improvement with Random query. For both baselines, the throughput of the TopK query is lower than that of the Random query because executing TopK sampling requires traversing all the neighbors of accessed vertices to compare the timestamps, thus incurring higher computation costs. In contrast, processing a K-hop sampling query in Helios only requires a fixed number of local cache lookups, which is independent of the degrees of sampled vertices. Thus Helios can maintain a stable serving throughput for queries with different sampling strategies.

Serving Latency. Figures 10 present the serving latency. The latency of baselines grows significantly with the increase of request concurrency and even reaches second-level under high-concurrency workloads. Moreover, the P99 latency of TigerGraph and NebulaGraph is over 150ms higher than their average, indicating a long tail latency. In contrast, Helios maintains a P99 latency of less than 50ms on all the queries and datasets, and the gap between the P99 latency and the average latency is within 20ms. Helios achieves up to 32× (24×) lower P99 latency on TopK (Random) query latency reduction compared to state-of-the-art baselines. These results show that Helios can achieve the latency SLO even under highly concurrent workloads by only local cache lookup.

7.2.2 Graph Update Throughput

Figure 11 illustrates the update ingestion throughput, which is defined as a million records per second (M/s). We evaluate Helios's throughput with two different sampling strategies: TopK and Random.

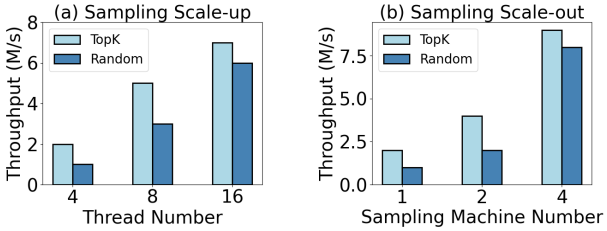

Figure 13. Scalability of Sampling.

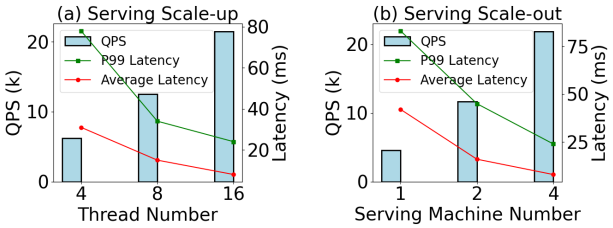

Figure 14. Scalability of Serving

Compared to baselines, Helios has an ingestion throughput improvement over 1.32×. This is because TigerGraph and NebulaGraph ingest graph updates with a strong consistency guarantee, introducing extra overheads. As discussed in § 6, Helios guarantees eventual consistency, which facilitates high-throughput data ingestion while maintaining inference quality, as proven later in § 7.4. Helios achieves a throughput of $11.17M/s$ on the BI dataset because the BI dataset has a large number of vertices, which can be directly ingested into the feature table without pre-sampling computation.

7.2.3 Impact of Sampling/Serving Separation

We examine the impact of the sampling/serving separation design by evaluating Helios's serving throughput and latency under different graph update ingestion rates. Figure 12 shows that the serving throughput and average latency remain almost stable with the increase of ingestion workloads. As discussed in § 4, this is because Helios physically isolates the pre-sampling computation and inference serving using different types of workers and threads.

7.3 Scalability

In this set of experiments, we study how Helios can efficiently scale up/out to handle fluctuating workloads from graph updates and concurrent GNN inference requests. We use the INTER dataset in all the following experiments.

7.3.1 Scalability of Sampling

In this experiment, we evaluate the sampling scalability of Helios. We record the throughput of pre-sampling as a million records per second (M/s) with both `TopK` and `Random` sampling strategies.

Scale-up. First, we examine the scale-up performance of the pre-sampling under a cluster setting including 4 nodes for sampling and 4 nodes for serving, and vary the number of sampling threads used in each sampling worker. Figure 13(a) shows that Helios can achieve a near-linear scale-up on the pre-sampling throughput.

Scale-out. In the scale-out experiment of pre-sampling, we fix the number of serving nodes at 4 and set the thread number used in each sampling node to 16. Then we vary the number of sampling nodes from 1 to 4. Figure 13(b) illustrates that, when increasing the node number from 1 to 4, pre-sampling in Helios can linearly scale out.

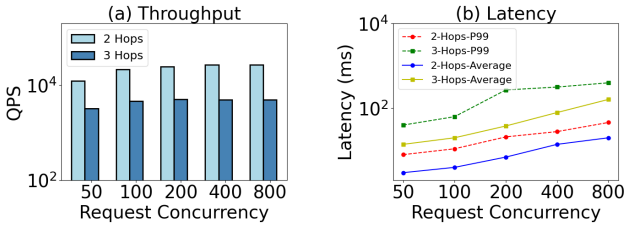

Figure 15. Impact of Sampling Hop Numbers

7.3.2 Scalability of Serving

We set the concurrency of inference requests to 200 and the sampling strategy to `Random`[9].

Scale-up. We set both the numbers of sampling and serving nodes to 4 and vary the number of serving threads used in each serving worker from 4 to 16. As shown in Figure 14(a), the serving throughput grows near-linearly with the increase in thread numbers. On the other hand, the P99 (average) serving latency can be significantly reduced from 78ms (31ms) to 24ms (8ms) by increasing the thread number from 4 to 16.

Scale-out. We fix the number of sampling nodes to 4, set the thread number in each serving node to 16, and increase the number of serving nodes from 1 to 4. Figure 14(b) illustrates that, with an increase in the number of serving nodes, the serving throughput increases linearly, and the P99 (average) latency decreases from 83ms (42ms) to 24ms (8ms). This experiment highlights that the serving performance of Helios scales out well, as processing K-hop sampling queries only requires local cache lookups in serving workers and incurs no network communication cost.

7.4 Microbenchmark

In this set of experiments, we stress Helios's serving performance with more sampling hops and study the scale of sample caches in serving workers. We also examine the ingestion latency and inference accuracy of Helios. We use the `Random` sampling in this set of experiments. Helios is deployed with 4 sampling nodes and 6 serving nodes.

Sampling Hops. We compare the serving performance when deploying a two-hop query and a three-hop query (see Table 2) on the dataset INTER. The three-hop query increases the serving workload by 5× over the two-hop query. Thus we can observe in Figure 15 that serving throughput under a three-hop query is lower than that of a two-hop query

[9]The serving process only involves local cache lookups, thus the serving performance is independent of the sampling strategy.

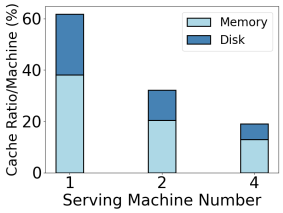

Figure 16. Evaluation of Sample Cache Ratio.

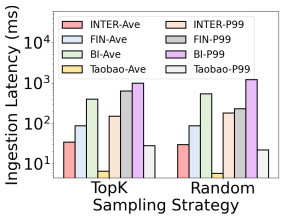

Figure 17. Helios's Ingestion Latency.

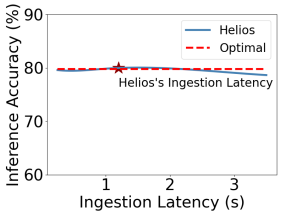

Figure 18. Inference Accuracy.

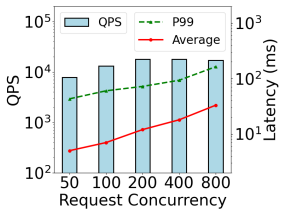

Figure 19. Online GNN Inference with Helios.

but still maintains 5000 QPS. Figure 15 also demonstrates an increase in query latency of the three-hop query. When the concurrency is relatively low (e.g., 100), the P99 latency of the three-hop query remains under 100ms. To effectively manage highly concurrent requests and more than three layers, it is advisable to deploy Helios with more serving workers to reduce serving latency, as verified in § 7.3.2.

Sample Cache. We examine the size of serving caches with the growth of serving node numbers on the INTER dataset. We record the average cache size including both memory and disk utilization (hybrid mode with Rocksdb) across all nodes and divide the average cache size by the original dataset size to get the cache ratio per node. Figure 16 illustrates that the cache ratio decreases from 62% to 19% with 1 to 4 serving nodes. This is because Helios only caches the sampled graph and feature in serving workers and slices the samples and features across partitions. Helios can scale out serving workers to handle more sampling hops or larger datasets.

Ingestion Latency. Figure 17 demonstrates the ingestion latency of Helios across four datasets. The results show that Helios can achieve as low as 1.2 seconds of P99 ingestion latency, which is recorded under millions of incoming graph updates per second, demonstrating that Helios can effectively capture dynamic graph updates in near real-time.

To study the impact of ingestion latency on sampling results, we simulate a worst-case read-after-write workload for Helios: an inference request on V_i is made immediately after an update within any of $V_i's$ two-hop neighboring subgraph is detected. We measure the percentile of relevant graph updates that should be considered when sampling V_i but missed due to the ingestion latency. We randomly select 1% of the vertices and use their updates to trigger inference requests. The results show that, on the four datasets in Table 1, only 0.03%, 0.02%, 1.90%, and 0.01% updates of the sampling subgraphs are invisible under the ingestion latency.

Effect of Consistency to Inference Accuracy. We select a *real-world* Taobao dataset to study Helios's inference accuracy using a GraphSAGE model for User-to-Item link prediction. By manually varying the ingestion latency in Helios from 0.25 seconds to 3.5 seconds, we generate the sampling results and use them as the inputs of the model serving. We compare the inference accuracy of Helios (where the sampling results are impacted by the ingestion latency) with the optimal case (case 1, see § 6), which allows each inference request to see all the writes during sampling. The results in Figure 18 show that Helios that relies on eventual consistency guarantee achieves similar inference accuracy as the implementation that adopts the optimal case, where the typical ingestion latency is marked with a star.

7.5 Online GNN Inference with Helios

We examine the online GNN inference performance with Helios. We deploy Helios with 4 sampling nodes, 6 serving nodes, and 4 nodes for TensorFlow Serving [14]. We use 4 extra client nodes to send concurrent GNN inference requests. The inference query is a two-hop query on the INTER dataset (illustrated in Table 2). Figure 19 demonstrates that GNN inference with Helios achieves a serving throughput up to 17000 QPS and maintains the P99/average latency below 100ms in most cases. When the concurrency level is high, e.g. 800, the P99 serving latency slightly exceeds 100ms, mainly because the client nodes are overloaded with receiving and sending sample results. With more client nodes, e.g., 8 nodes, we can still observe a 94ms P99 latency.

8 Related Work

Graph Learning on Dynamic Graphs. Different from traditional graph learning on static graphs [24, 27, 39, 47, 69, 89], many recent works [48, 55, 56, 59, 61, 62, 68, 71, 73, 74, 76, 83, 96] highlight the necessity to incorporate the dynamically updated graph structure and attribute into GNN representations to address the issues such as concept drift [35] in recommendation systems [50, 51], real-time fraud detection in financial management systems [73]. Temporal GNN models [48, 59, 61, 62, 73, 83] encode the time-domain information into the graph representation. Others [6, 51, 55, 56, 98] incrementally update static GNN model and leverage online inference to facilitate real-time model serving. Helios aims to accelerate dynamic graph sampling for online GNN inference, which is orthogonal to the above works. There are dynamic graph approaches besides GNN to mine dynamic graph updates. Systems like Kineograph [26], Graphbolt [57], and Tripoline [43] focus on non-learning-based algorithms such as shortest paths and connected components. In contrast, Helios primarily focuses on sampling and serving for GNN inference.

GNN Systems. GNN frameworks [34, 66, 72, 79, 80] mainly focus on GNN training. Although they can also be used for inference on static graphs with temporal information, they don't support dynamic graph updates required in online inference scenarios. Some GNN inference systems [44, 49, 67, 85, 90] only focus on inference with static graphs. Other GNN inference systems [63, 92, 94, 95] build hardware accelerators, e.g., FPGA, to support inference with dynamic graph updates. Besides, there are some GPU-based sampling systems [37, 41, 58] that can provide high graph sampling throughput but also limit the graph size within GPU/CPU memory capacity in a single machine. In contrast, Helios supports distributed graph sampling on dynamic graphs.

Graph Databases. Graph databases [2, 3, 5, 7, 8, 10, 20, 22, 23, 28, 30, 45, 77, 99] are inherently designed to store, manage, and query graph data. However, when employed as graph sampling services for online GNN inference, graph databases must perform multi-hop graph sampling queries in real time, which introduces substantial graph traversal and computational overhead, thereby hindering them from meeting millisecond-level SLOs. In contrast, Helios shifts the graph sampling to the graph update stage and only executes local cache lookups during the online GNN inference process, thereby sufficiently guaranteeing the latency SLO.

9 Conclusion

We propose Helios, the first system that shifts the ad-hoc dynamic graph sampling into the graph update process through event-driven pre-sampling. Helios maintains the query-aware sample cache to serve inference queries locally, enabling to meet the stringent latency SLOs for online GNN inference. Helios separately allocates distributed machines for pre-sampling and inference serving, allowing scaling near linearly to handle both high-concurrency inference requests and high-throughput graph updates. Experiments show that Helios achieves up to 67× improvements in serving throughput and up to 32× reduction in P99 query latency, compared to existing dynamic graph sampling approaches.

Acknowledgments

The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), the National Natural Science Foundation of China under the grant numbers (62472384, 62441605, 62441236, U24A20326), the Fundamental Research Funds for the Central Universities, Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (SN-ZJU-SIAS-0010). Zeke Wang and Li Su are the corresponding authors.

References

[1] 2013. The Log: What every software engineer should know about real-time data's unifying abstraction. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.
[2] 2021. ArangoDB. https://www.arangodb.com/.
[3] 2021. JanusGraph. https://janusgraph.org/.
[4] 2021. User Behavior Data from Taobao for Recommendation. https://tianchi.aliyun.com/dataset/649.
[5] 2022. AWS Neptune. https://aws.amazon.com/neptune/.
[6] 2022. AWS SAGEMaker. https://aws.amazon.com/cn/blogs/machine-learning/build-a-gnn-based-real-time-fraud-detection-solution-using-amazon-sagemaker-amazon-neptune-and-the-deep-graph-library/.
[7] 2022. Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/graph/graph-introduction.
[8] 2022. Neo4j. https://neo4j.com.
[9] 2022. Rec-Tmall. https://tianchi.aliyun.com/dataset/140281.
[10] 2023. Alibaba GDB. https://www.aliyun.com/product/gdb/.
[11] 2023. Apache Kafka. https://kafka.apache.org/.
[12] 2023. Issues of Distributed Query Mode. https://dev.tigergraph.com/forum/t/obvious-reasons-why-distributed-mode-would-slow-down-query/1335.
[13] 2023. Leaderboards for Node Property Prediction. https://ogb.stanford.edu/docs/leader_nodeprop/.
[14] 2023. TensorFlow Serving. https://github.com/tensorflow/serving.
[15] 2023. TigerGraph Compare to Neptune. https://www.tigergraph.com/blog/amazon-neptune/.
[16] Gediminas Adomavicius and Jingjing Zhang. 2012. Stability of recommendation algorithms. *In TOIS* (2012).
[17] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *ICLR*.
[18] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, et al. 2020. The LDBC social network benchmark. *In Arxiv* (2020).
[19] Fedor Borisyuk, Shihai He, Yunbo Ouyang, Morteza Ramezani, Peng Du, Xiaochen Hou, Chengming Jiang, Nitin Pasumarthy, Priya Bannur, Birjodh Tiwana, et al. 2024. LiGNN: Graph Neural Networks at LinkedIn. In *SIGKDD*.
[20] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *SIGMOD*.
[21] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and Relieving the Over-Smoothing Problem for Graph Neural Networks from the Topological View. In *AAAI*.
[22] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: a high performance distributed graph database with a decentralized architecture. In *VLDB*.
[23] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High performance distributed OLAP on property graphs with grasper. In *SIGMOD*.
[24] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR*.
[25] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *ICML*.
[26] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Eurosys*.
[27] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *SIGKDD*.
[28] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. Tigergraph: A native MPP graph database. *In Arxiv* (2019).
[29] Zhengxiao Du, Xiaowei Wang, Hongxia Yang, Jingren Zhou, and Jie Tang. 2019. Sequential scenario-specific meta learner for online recommendation. In *SIGKDD*.

[30] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. In *VLDB*.

[31] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *WWW*.

[32] Facebook. 2023. Rocksdb. https://github.com/facebook/rocksdb.

[33] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* (1999).

[34] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning With PyTorch Geometric. *In ArXiv* (2019).

[35] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *In CSUR* (2014).

[36] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *OSDI*.

[37] Ping Gong, Renjie Liu, Zunyao Mao, Zhenkun Cai, Xiao Yan, Cheng Li, Minjie Wang, and Zhuozhao Li. 2023. gSampler: General and efficient GPU-based graph sampling for graph learning. In *SOSP*.

[38] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*.

[39] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*.

[40] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. Tencentrec: Real-time stream recommendation in practice. In *SIGMOD*.

[41] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Eurosys*.

[42] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura, and David P Woodruff. 2019. Weighted reservoir sampling from distributed streams. In *SIGMOD*.

[43] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Eurosys*.

[44] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *MLsys*.

[45] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD*.

[46] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2024. IGB: Addressing The Gaps In Labeling, Features, Heterogeneity, and Size of Public Graph Datasets for Deep Learning Research. In *SIGKDD*.

[47] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[48] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *SIGKDD*.

[49] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. 2022. HolisticGNN: Geometric Deep Learning Engines for Computational SSDs. In *NVMW*.

[50] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. In *VLDB*.

[51] Dandan Lin, Shijie Sun, Jingtao Ding, Xuehan Ke, Hao Gu, Xing Huang, Chonggang Song, Xuri Zhang, Lingling Yi, Jie Wen, et al. 2022. PlatoGL: Effective and Scalable Deep Graph Learning System for Graph-enhanced Real-Time Recommendation. In *CIKM*.

[52] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling Gnn Training on Large Graphs via Computation-aware Caching. In *SoCC*.

[53] Jiahui Liu, Peter Dolan, and Elin Rønby Pedersen. 2010. Personalized news recommendation based on click behavior. In *In IUI*.

[54] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. In *NSDI*.

[55] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. 2021. Pick and choose: a GNN-based imbalanced learning approach for fraud detection. In *WWW*.

[56] Mingxuan Lu, Zhichao Han, Susie Xi Rao, Zitao Zhang, Yang Zhao, Yinan Shan, Ramesh Raghunathan, Ce Zhang, and Jiawei Jiang. 2022. BRIGHT-Graph Neural Networks in Real-Time Fraud Detection. In *CIKM*.

[57] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Eurosys*.

[58] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC*.

[59] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *AAAI*.

[60] Shipeng Qi, Heng Lin, Zhihui Guo, Gábor Szárnyas, Bing Tong, Yan Zhou, Bin Yang, Jiansong Zhang, Zheng Wang, Youren Shen, et al. 2023. The LDBC Financial Benchmark. *In Arxiv* (2023).

[61] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *In Arxiv* (2020).

[62] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *WSDM*.

[63] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2022. Flowgnn: A dataflow architecture for universal graph neural network inference via multi-queue streaming. *In Arxiv* (2022).

[64] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, et al. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *OSDI*.

[65] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *ATC*.

[66] Jie Sun, Mo Sun, Zheng Zhang, Jun Xie, Zuocheng Shi, Zihan Yang, Jie Zhang, Fei Wu, and Zeke Wang. 2025. Hyperion: Optimizing SSD Access is All You Need to Enable Cost-efficient Out-of-core GNN Training. In *ICDE*.

[67] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *In Arxiv* (2023).

[68] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *ICLR*.

[69] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *In Arxiv* (2017).

[70] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *In TOMS* (1985).

[71] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: pipelined and parallel dynamic GNN training on GPUs. In *PPoPP*.

[72] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLRW*.

[73] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *SIGMOD*.

[74] Yufeng Wang and Charith Mendis. 2023. Tgopt: Redundancy-aware optimizations for temporal graph attention networks. In *PPoPP*.

[75] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna PN Puttaswamy, and Ben Y Zhao. 2009. User interactions in social networks and their implications. In *Eurosys*.

[76] Dan Wu, Zhaoying Li, and Tulika Mitra. 2023. InkStream: Real-time GNN Inference on Streaming Graphs via Incremental Update. *In Arxiv* (2023).

[77] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *In Arxiv* (2022).

[78] Yue Wu, Yinpeng Chen, Lijuan Wang, Yuancheng Ye, Zicheng Liu, Yandong Guo, and Yun Fu. 2019. Large Scale Incremental Learning. In *CVPR*.

[79] Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2024. Scaling New Heights: Transformative Cross-GPU Sampling for Training Billion-Edge Graphs. In *SC*.

[80] Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2023. Redundancy-Free High-Performance Dynamic GNN Training with Hierarchical Pipeline Parallelism. In *HPDC*.

[81] Xu Xie, Fei Sun, Xiaoyong Yang, Zhao Yang, Jinyang Gao, Wenwu Ou, and Bin Cui. 2021. Explore User Neighborhood for Real-time E-commerce Recommendation. In *ICDE*.

[82] Yi Xie, Yun Xiong, and Yangyong Zhu. 2020. SAST-GNN: a self-attention based spatio-temporal graph neural network for traffic prediction. In *DASFAA*.

[83] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *In Arxiv* (2020).

[84] Yishi Xu, Yingxue Zhang, Wei Guo, Huifeng Guo, Ruiming Tang, and Mark Coates. 2020. Graphsail: Graph structure aware incremental learning for recommender systems. In *CIKM*.

[85] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *SIGKDD*.

[86] 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 *SIGKDD*.

[87] Zhongbao Yu, Jiaqi Zhang, Xin Qi, and Chao Chen. [n. d.]. Application Research of Graph Neural Networks in the Financial Risk Control. ([n. d.]).

[88] Quan Yuan, Gao Cong, and Aixin Sun. 2014. Graph-based point-of-interest recommendation with geographical and temporal influences. In *CIKM*.

[89] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2021. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *ICLR*.

[90] Dalong Zhang, Xianzheng Song, Zhiyang Hu, Yang Li, Miao Tao, Binbin Hu, Lin Wang, Zhiqiang Zhang, and Jun Zhou. 2023. InferTurbo: A Scalable System for Boosting Full-graph Inference of Graph Neural Network over Huge Graphs. In *ICDE*.

[91] Yuyu Zhang, Liang Pang, Lei Shi, and Bin Wang. 2014. Large scale purchase prediction with historical user actions on B2C online retail platform. *arXiv preprint arXiv:1408.6515* (2014).

[92] Kai Zhong, Shulin Zeng, Wentao Hou, Guohao Dai, Zhenhua Zhu, Xuecang Zhang, Shihai Xiao, Huazhong Yang, and Yu Wang. 2023. CoGNN: An Algorithm-Hardware Co-Design Approach to Accelerate GNN Inference with Mini-Batch Sampling. *In TCAD* (2023).

[93] Wenliang Zhong, Rong Jin, Cheng Yang, Xiaowei Yan, Qi Zhang, and Qiang Li. 2015. Stock constrained recommendation in tmall. In *SIGKDD*.

[94] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. In *VLDB*.

[95] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. 2022. Model-architecture co-design for high performance temporal gnn inference on fpga. In *IPDPS*.

[96] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: a general framework for temporal GNN training on billion-scale graphs. In *VLDB*.

[97] Qi Zhu, Natalia Ponomareva, Jiawei Han, and Bryan Perozzi. 2021. Shift-robust gnns: Overcoming the limitations of localized graph training data. In *NeurIPS*.

[98] 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. In *VLDB*.

[99] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. In *VLDB*.