SNAPCHAT

Training Industry-scale GNNs with GiGL

KDD 2025 Hands-On tutorial

Presenters: Yozen Liu, Tong Zhao, Matthew Kolodner, Kyle Montemayor, Shubham Vij, Neil Shah

https://github.com/Snapchat/GiGL/tree/main/examples/tutorial/KDD_2025

Hands-on Tutorial website

SNAPCHAT

Overview

/01 Intro

/02 GNNs and their scale challenges

/03 Overview of GiGL

/04 Hands-on with GiGL - Training and inference with industry-scale GNNs

Break (30m)

/05 Hands-on with GiGL - Customization

SNAPCHAT

Intro

SNAPCHAT

Who are we?

User Modeling and Personalization: A group of scientists and engineers at Snapchat advancing personalization-related algorithms and applications.

We do both fundamental research and support internal technical transfer.

Focus areas:

- graph learning
- classical and generative recommendation systems
- efficient ML at scale
- new applications

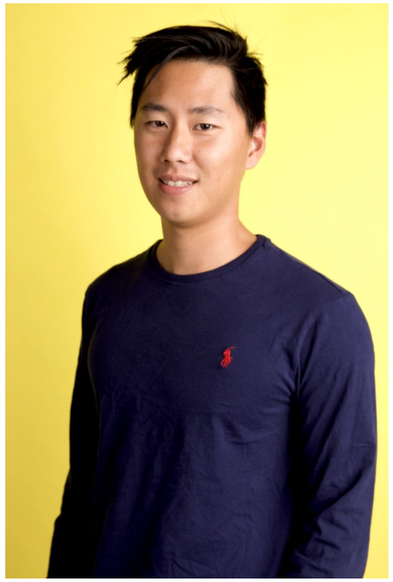
Roles at Snap

Internships

SNAPCHAT

Who are we?

Shubham



Yozen

Matt

Kyle

Neil

SNAPCHAT

Why are we here?

We will showcase **GiGL (Gigantic Graph Learning)**, a library we built at Snapchat to make scaling up and iterating on graph neural networks easy.

By the end of the tutorial, you will…

- Understand fundamentals of GNNs, scaling challenges, and GiGL as a solution
- Gain hands-on experience using GiGL components
- Be able to comfortably train your own large-scale GNNs with GiGL

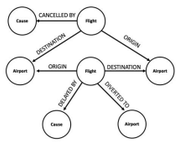
SNAPCHAT

GNNs and their Scale Challenges

SNAPCHAT

Graphs and why they matter

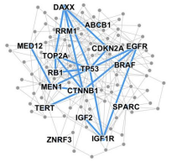
Graphs are a universal language to describe relationships between entities.



Event Graphs

Image credit: SalientNetworks

Computer Networks



Disease Pathways

Image credit: Medium

Social Networks

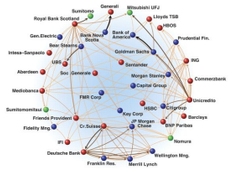


Image credit: Science

Economic Networks

Image credit: Lumen Learning

Communication Networks

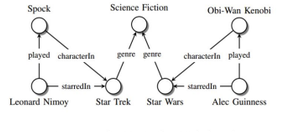


Image credit: Maximilian Nickel et al

Knowledge Graphs

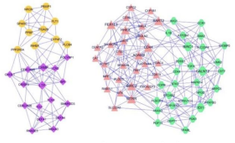


Image credit: ese.wustl.edu

Regulatory Networks

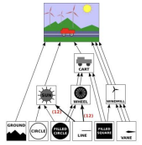


Image credit: math.hws.edu

Scene Graphs

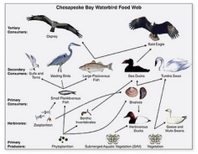


Image credit: Wikipedia

Food Webs

Image credit: Pinterest

Particle Networks

Image credit: visitlondon.com

Underground Networks

Citation Networks

Image credit: Missoula Current News

Internet

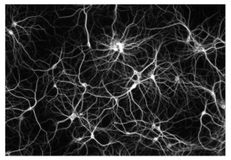


Image credit: The Conversation

Networks of Neurons

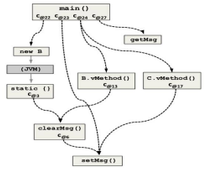


Image credit: ResearchGate

Code Graphs

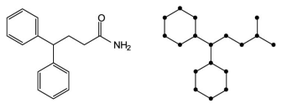


Image credit: MDPI

Molecules

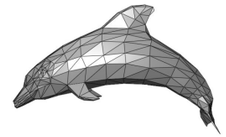


Image credit: Wikipedia

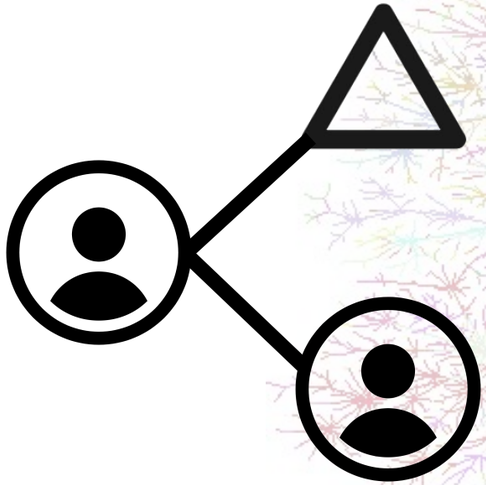
3D Shapes

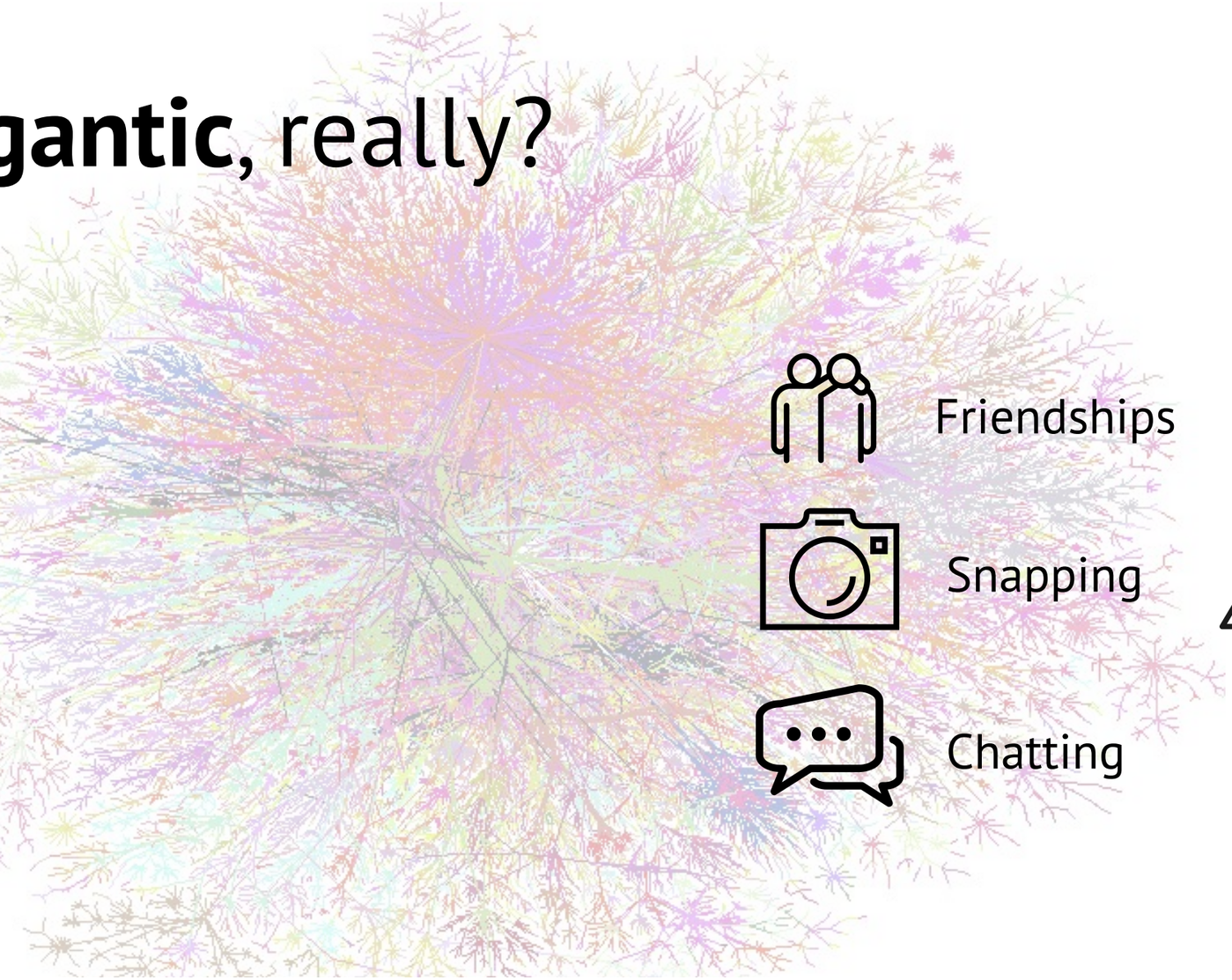
This is immensely valuable context in making predictions.

Figure credit: CS224W Machine Learning with Graphs

SNAPCHAT

What is **gigantic**, really?





Friendships

Snapping

Chatting

Story viewing

Publisher content

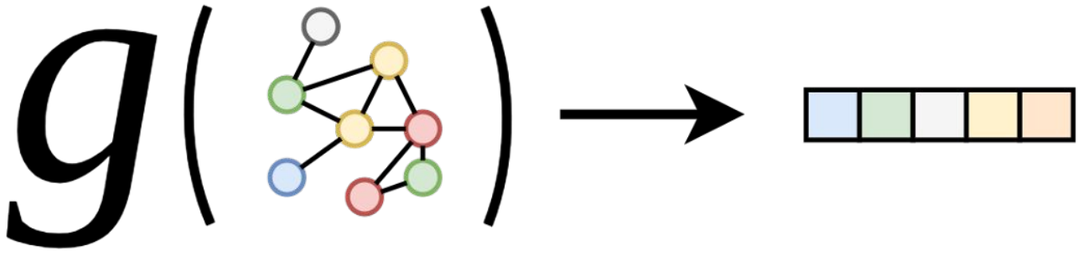
Lenses

O(billion) nodes, O(100B) edges

SNAPCHAT

What are Graph Neural Networks (GNNs)?

GNNs are a modern neural architecture designed to model graph data



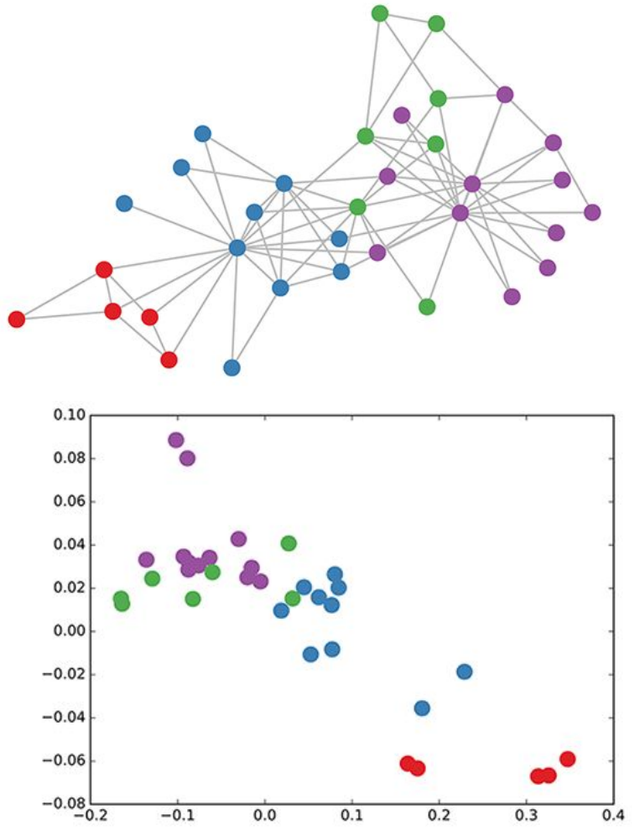
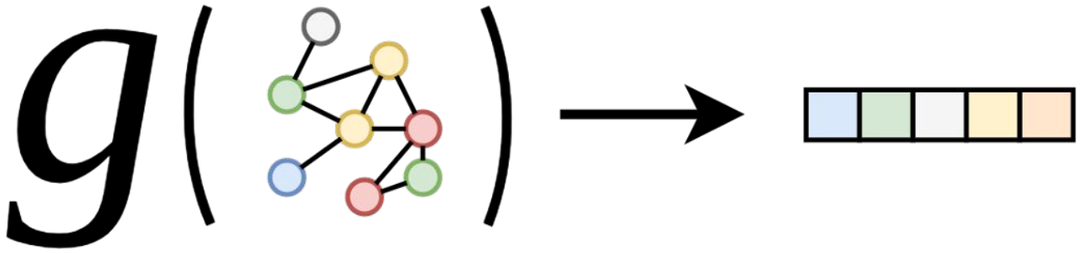


Figure credit: Graph Convolutional Networks (Kipf, 2016)

SNAPCHAT

What can we do with them?

GNNs are a modern neural architecture designed to model graph data



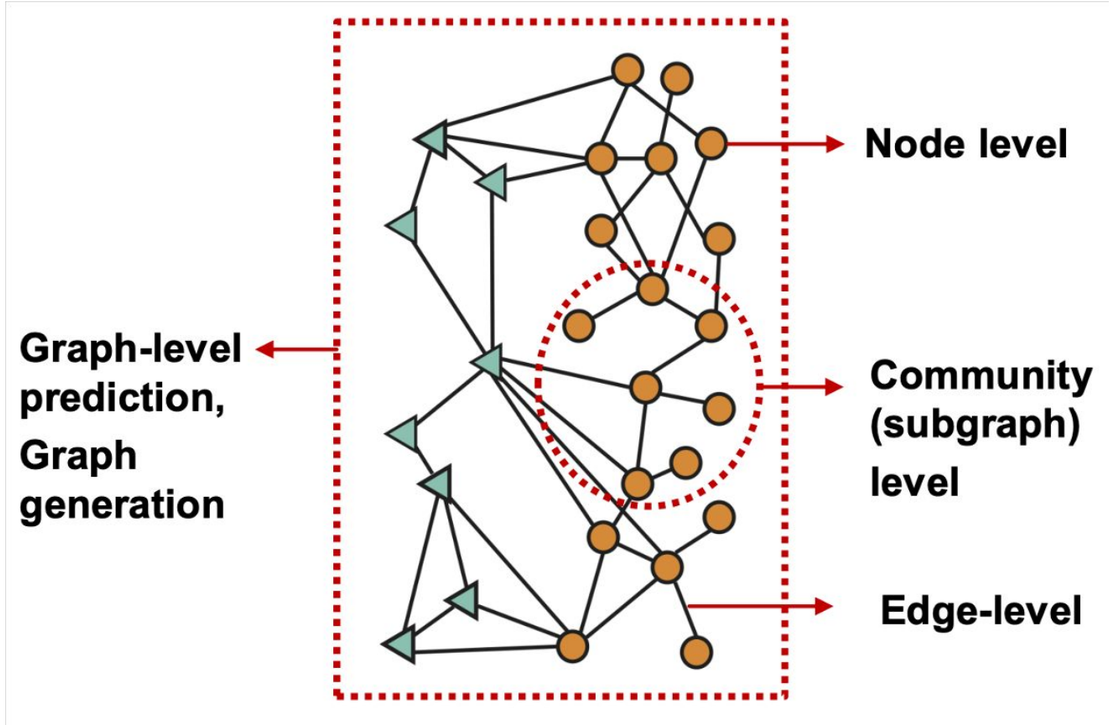
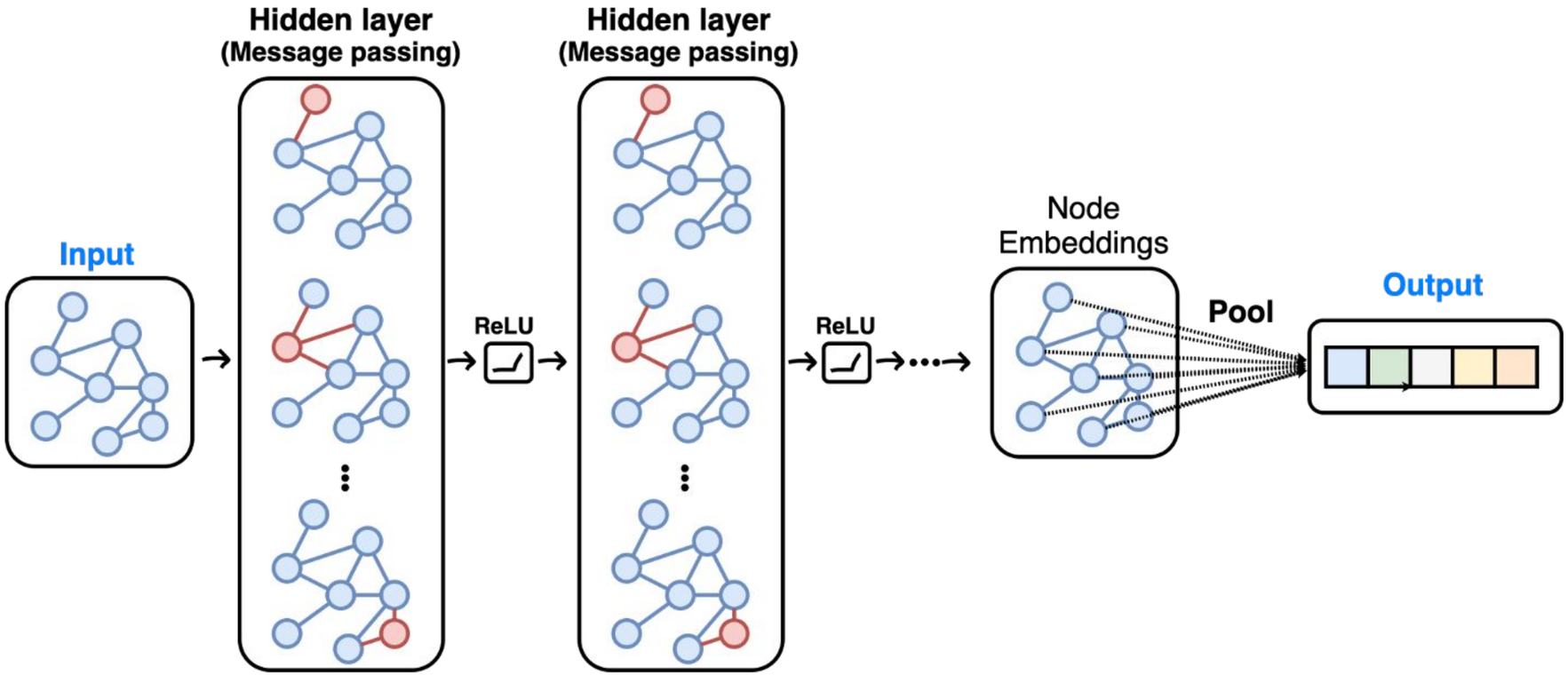


Figure credit: CS224W Machine Learning with Graphs

SNAPCHAT

How do they work?

Key Idea: stack "message passing" layers with nonlinearities



SNAPCHAT

How do they work, **really?**

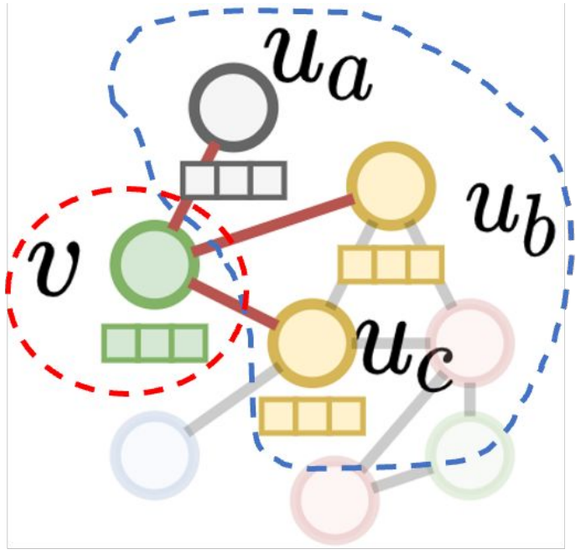
$$h_v^{(t)} = \mathrm{AGG}^{(t)}\left(h_v^{(t-1)}, \left\{\mathrm{MSG}^{(t)}(h_u^{(t-1)}) | u \in \mathcal{N}(v)\right\}\right)$$

Node emb.

Neighbor messages

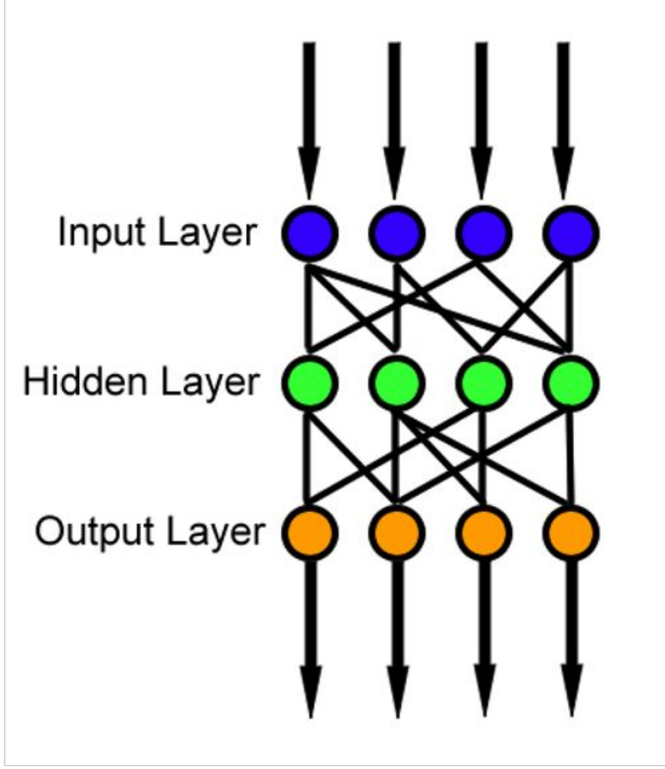
$$H^{(t)} = \sigma\left(AH^{(t-1)}W^{(t-1)}\right)$$

where $H^{(0)} = X$

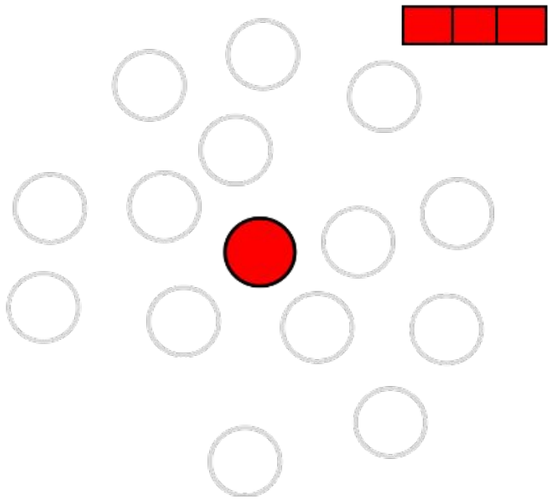


SNAPCHAT

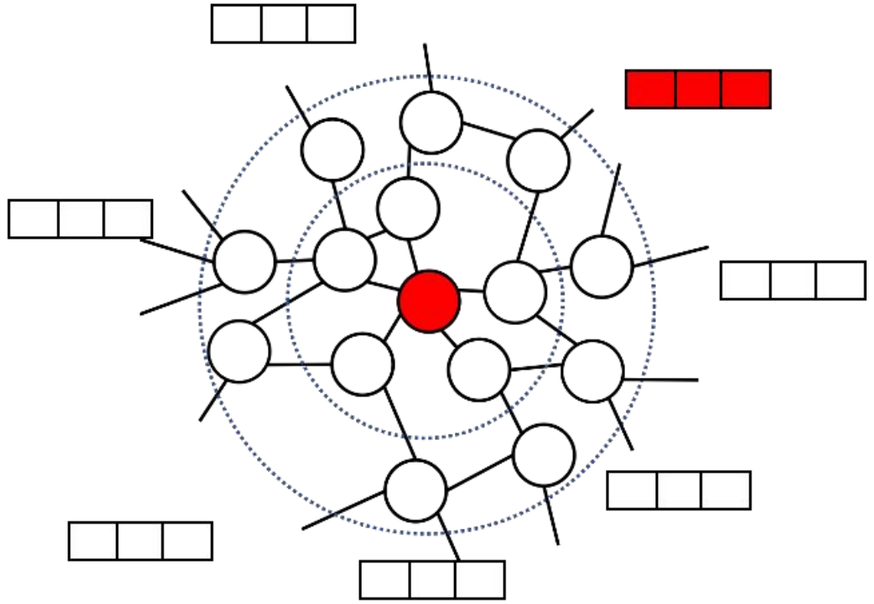
Connections to Multi-Layer Perceptrons



Multi-layer perceptrons
(tabular machine learning)



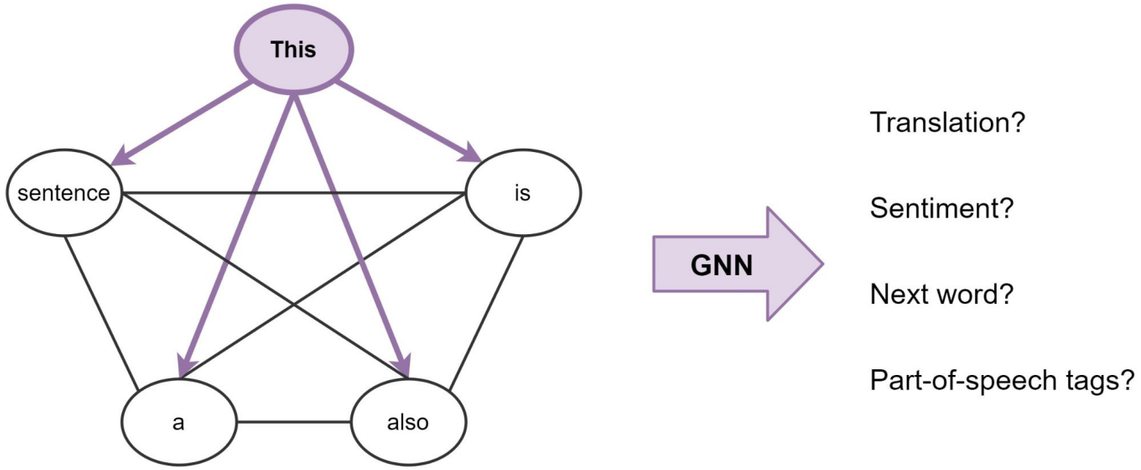
Tabular setting



Relational setting

SNAPCHAT

Connections to Transformers



Transformers are Graph Neural Networks

Chaitanya Joshi
Feb 12, 2020

Project Project Slides Medium

Engineer friends often ask me: Graph Deep Learning sounds great, but are there any big commercial success stories? Is it being deployed in practical applications?

Besides the obvious ones–recommendation systems at Pinterest, Alibaba and Twitter–a slightly nuanced success story is the **Transformer architecture**, which has taken the NLP industry by storm.

Through this post, I want to establish links between Graph Neural Networks (GNNs) and Transformers. I'll talk about the intuitions behind model architectures in the NLP and GNN communities, make connections using equations and figures, and discuss how we could work together to drive progress.

Let's start by talking about the purpose of model architectures–*representation learning*.

Figure Credit: Transformers are Graph Neural Networks (Joshi, 2020)

SNAPCHAT

GNNs are widely successful in practice

Uber AI, Engineering

Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations

PinSage: A new graph convolutional neural network for web-scale recommender systems

LinkSAGE: Optimizing Job Matching Using Graph Neural Networks

Ping Liu, Haichao Wei, Xiaochen Hou, Jianqiang Shen, Shihai He, Kay Qianqi Shen, Zhujun Chen, Fedor Borisyuk, Daniel Hewlett, Liang Wu, Srikant Veeraraghavan, Alex Tsun, Chengming Jiang, Wenjing Zhang
LinkedIn Corporation
Mountain View, CA, USA
{piliu,hawei,xiahou,jershen,she1,qishen,zhuchen,fborisyuk,dhewlett,liawu,atsun,cjiang,wzhang}@linkedin.com

ETA Prediction with Graph Neural Networks in Google Maps

Austin Derrow-Pinion[1], Jennifer She[1], David Wong[2*], Oliver Lange[3], Todd Hester[4*], Luis Perez[5*], Marc Nunkesser[3], Seongjae Lee[3], Xueying Guo[3], Brett Wiltshire[1], Peter W. Battaglia[1], Vishal Gupta[1], Ang Li[1], Zhongwen Xu[6*], Alvaro Sanchez-Gonzalez[1], Yujia Li[1], Petar Veličković[1]
[1]DeepMind [2]Waymo [3]Google [4]Amazon [5]Facebook AI [6]Sea AI Lab *work done while at DeepMind
{derrowap,jenshe,wongda,petarv}@google.com

Personalized Audiobook Recommendations at Spotify Through Graph Neural Networks

Marco De Nadai[1*], Francesco Fabbri[1*], Paul Gigioli[1], Alice Wang[1], Ang Li[1], Fabrizio Silvestri[1,2], Laura Kim[1], Shawn Lin[1], Vladan Radosavljevic[1], Sandeep Ghael[1], David Nyhan[1], Hugues Bouchard[1], Mounia Lalmas-Roelleke[1], Andreas Damianou[1]
[1]Spotify, Denmark, Spain, UK, USA
[2]Sapienza University of Rome, Italy

Embedding Based Retrieval in Friend Recommendation

Jiahui Shi, Vivek Chaurasiya, Yozen Liu, Shubham Vij, Yan Wu, Satya Kanduri
Neil Shah, Peicheng Yu, Nik Srivastava, Lei Shi, Ganesh Venkataraman, Jun Yu
{jshi3,vchaurasiya,yliu2,svij,ywu,satya.kanduri,nshah,peicheng.yu,nsrivastava,lshi3,gvenkataraman,jyu3}@snap.com
Snap Inc.
Santa Monica, CA USA

SNAPCHAT

...especially at Snap!

GiGL: Large-Scale Graph Neural Networks at Snapchat

Tong Zhao*, Yozen Liu*, Matthew Kolodner, Kyle Montemayor, Elham Ghazizadeh†, Ankit Batra†
Zihao Fan, Xiaobin Gao, Xuan Guo, Jiwen Ren, Serim Park, Peicheng Yu, Jun Yu
Shubham Vij*, Neil Shah*
Snap Inc., USA
*{tong,yliu2,svij,nshah}@snap.com

ABSTRACT

Recent advances in graph machine learning (ML) with the introduction of Graph Neural Networks (GNNs) have led to a widespread interest in applying these approaches to business applications at scale. GNNs enable differentiable end-to-end (E2E) learning of model parameters given graph structure which enables optimization towards popular node, edge (link) and graph-level tasks. While the research innovation in new GNN layers and training strategies has been rapid, industrial adoption and utility of GNNs has lagged considerably due to the unique scale challenges that large-scale graph ML problems create. In this work, we share our approach to training, inference, and utilization of GNNs at Snapchat. To this end, we present GiGL (Gigantic Graph Learning), an open-source library to enable large-scale distributed graph ML to the benefit of researchers, ML engineers, and practitioners. We use GiGL internally at Snapchat to manage the heavy lifting of GNN workflows, including graph data preprocessing from relational DBs, subgraph sampling, distributed

1 INTRODUCTION

Graphs are ubiquitous, representing a wide variety of real-world data across domains such as social networks and recommendation systems. In recent years, Graph Neural Networks (GNNs) have emerged as powerful tools for learning from graph data [30, 51, 63].

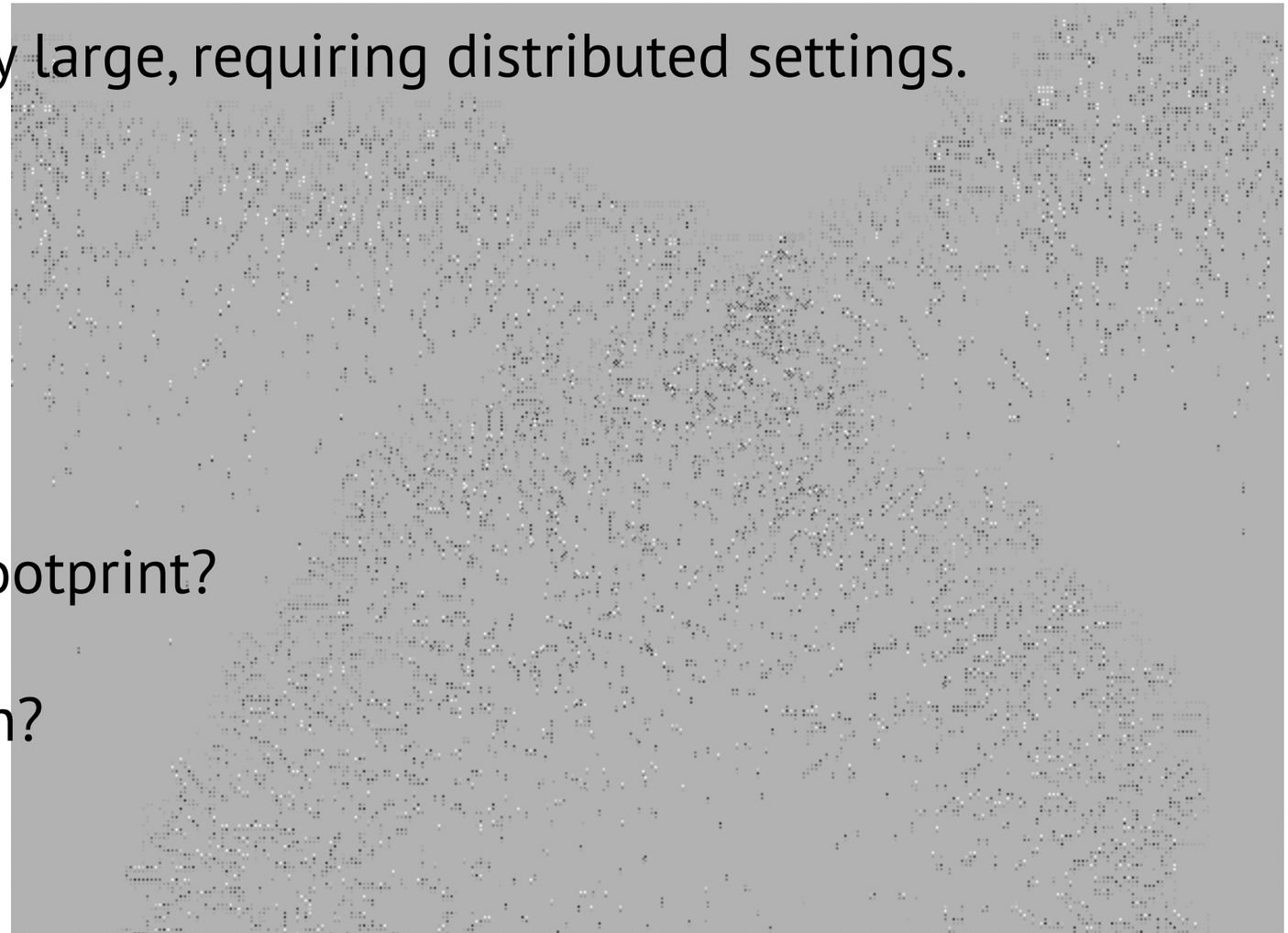
Nevertheless, when deploying GNNs in industrial contexts, such as large-scale user modeling and social recommendation systems, scalability poses a significant challenge. Most commonly used libraries in the research community assume that graph topology, node and edge features, can fit easily within the CPU (or even GPU) memory of a single machine. However, just storing a graph with 100 billion edges as an example (not uncommon in practice [6]) requires 800 GB of memory assuming 32-bit integer IDs, even before accounting for node and edge features or the need to represent graphs with more nodes than 32-bit limits. Such limitations make it difficult to scale GNN models in real-world, billion-scale graphs, which we routinely handle at Snapchat.

GiGL: Large Scale Graph Neural Networks at Snapchat (Zhao et al, KDD 2025)

SNAPCHAT

Practical Challenges in Scaling GNNs

Graphs in practice can be extremely large, requiring distributed settings.



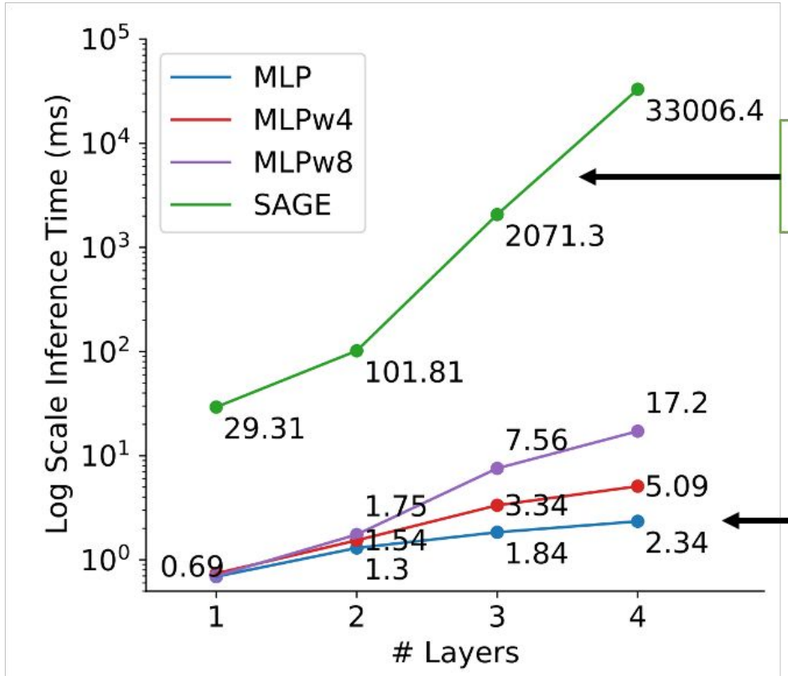
- How to represent graph data?
- How to manage memory usage?
- How to minimize computation footprint?
- How to minimize communication?

SNAPCHAT

Practical Challenges in Scaling GNNs

Neighbor explosion in GNNs creates scaling overheads.

Operation	OGB-arXiv		
#Nodes	169343		
#Edges	1166243		
	Forward	Backward	Total
$Z^l = W^l H^{l-1}$	0.32	1.09	1.42
$H^l = AZ^l$	1.09	1028.08	1029.17
			724×



MLPInit: Embarrassingly Simple GNN Acceleration with MLP Initialization (Han et al, ICLR 2023
Graph-less Neural Networks: Teaching old MLPs new Tricks via Distillation (Zhang et al, ICLR 2022)

SNAPCHAT

Large-Scale Graph Neural Networks:
Navigating the Past and Pioneering New Horizons

Rui Xue Haoyu Han Tong Zhao Neil Shah Jiliang Tang Xiaorui Liu

8:30 am – 12:30 pm, February 21th, AAAI 2024

GNN Scaling: Research

Approximations: node and layer-wise sampling, historical embeddings

Decoupling: pre-computing and post-computing with the graph

Distributed processing: efficient algorithms for parallelization and tabularization

Training paradigm: lazy propagation, alternating opt, pre-training, coarsening

Inference paradigm: cross-model distillation

Data-centric: graph condensation, subgraph sketching

Please check out our team's past work on this space @ https://research.snap.com/team/user-modeling-and-personalization.html

SNAPCHAT

GNN Scaling: Practice

In our experience, we want tooling which enables…

- Flexible model specification and iteration
- Familiar imperatives for ML practitioners (e.g. PyG)
- Orchestration for batch training and inference pipelines
- Elastic horizontal scaling to large graphs
- Minimal cost and e2e latency

SNAPCHAT

Overview of GiGL

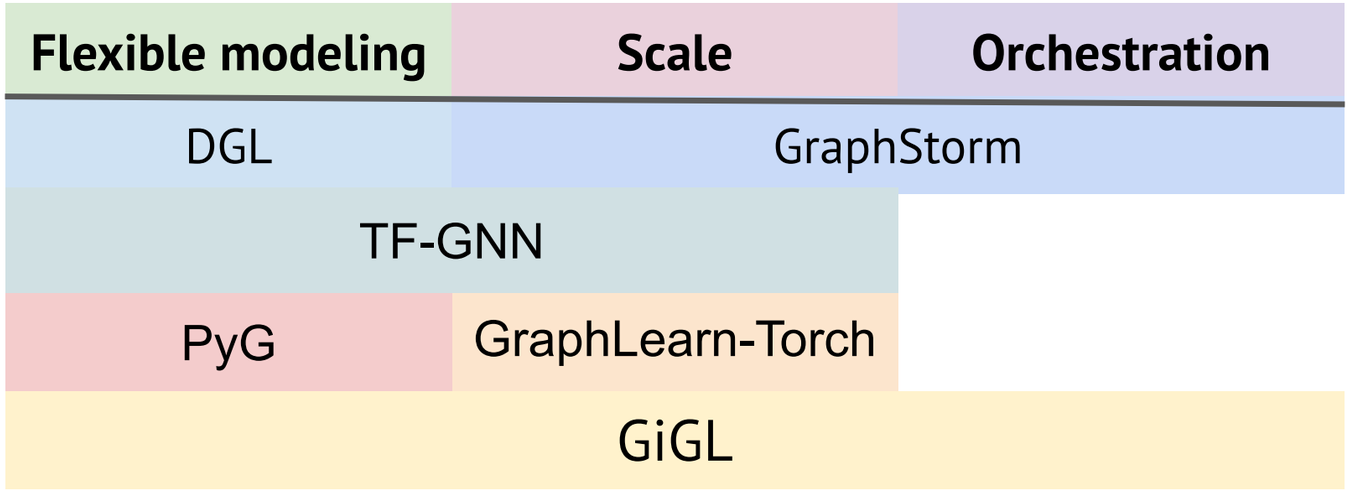
SNAPCHAT

What is GiGL?

- GiGL is a graph learning library built for training **industry-scale graphs**
- GiGL provides **end-to-end pipeline and orchestration** handling all pre/post-processing, training and inference steps
- GiGL provides easy onboarding, extensive code customization, and is compatible with popular graph learning frameworks like **PyTorch Geometric**.

SNAPCHAT

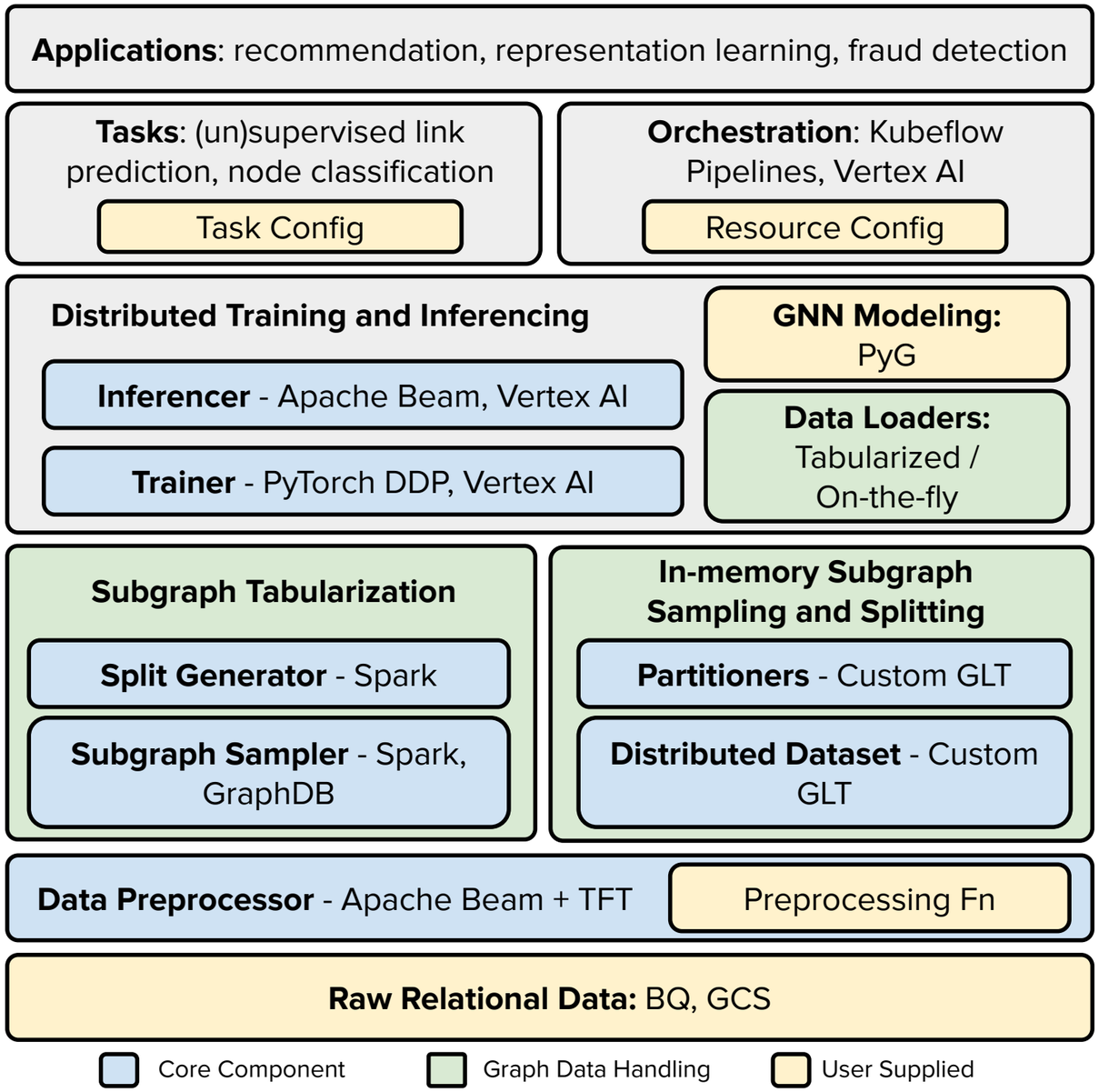
Why build GiGL?



- Existing Graph ML tooling does not support our requirements on -
  - The scale of graph we operate on
  - End-to-end orchestration that seamlessly integrates into our ecosystem
  - Flexible modeling and subgraph sampling customization

SNAPCHAT

Bird's Eye View



GiGL

SNAPCHAT

Overview of GiGL

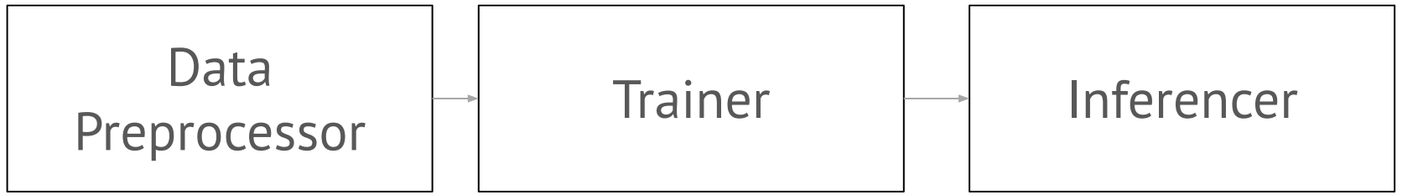
Solving scaling challenges

SNAPCHAT

How does GiGL solve scale? - Two paradigms

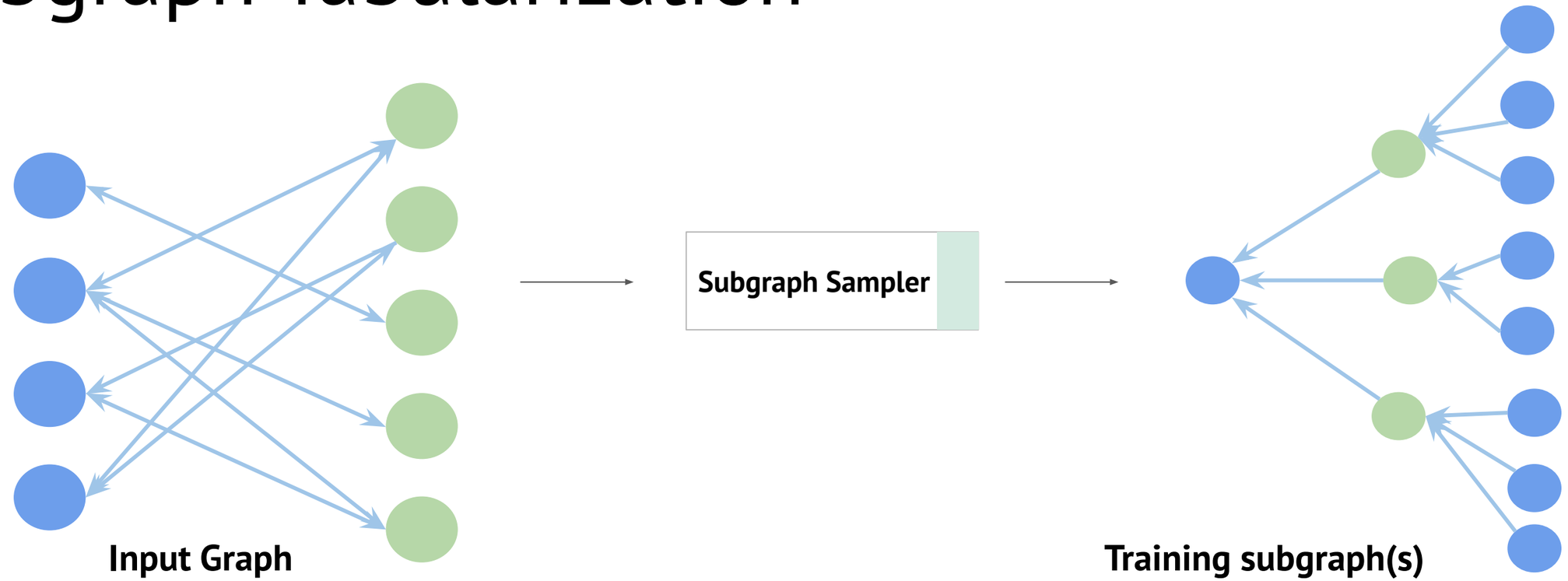
Subgraph Tabularization (Powered by Spark)

In-memory Subgraph Sampling (Powered by GraphLearn-for-PyTorch [GLT])



SNAPCHAT

Subgraph Tabularization



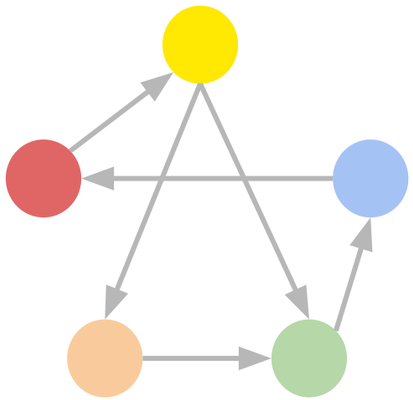
Enables training GNNs at scale by generating training subgraphs stored on disk in a distributed processing (Spark) job prior to training

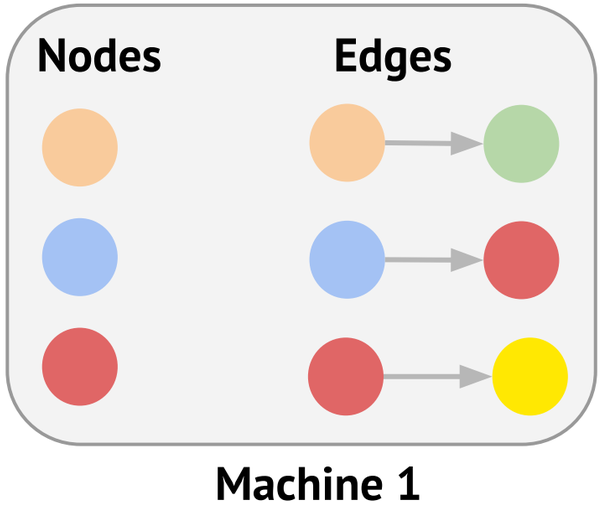
SNAPCHAT

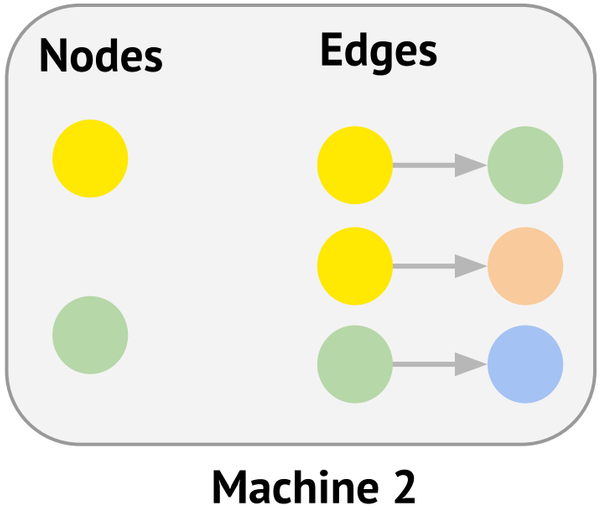
In-memory Subgraph Sampling

Graph is stored in-memory

- Nodes & edges partitioned into multiple machines
- Training subgraphs sampled during training process







...

In-memory subgraph sampling is powered by GraphLearn-for-pytorch https://github.com/alibaba/graphlearn-for-pytorch

SNAPCHAT

Tabularized vs In-memory - When to use which?

Tabularization

- Offline batch jobs
- Multiple tuning/experimental runs benefit from pre-sampled training subgraphs

In-memory

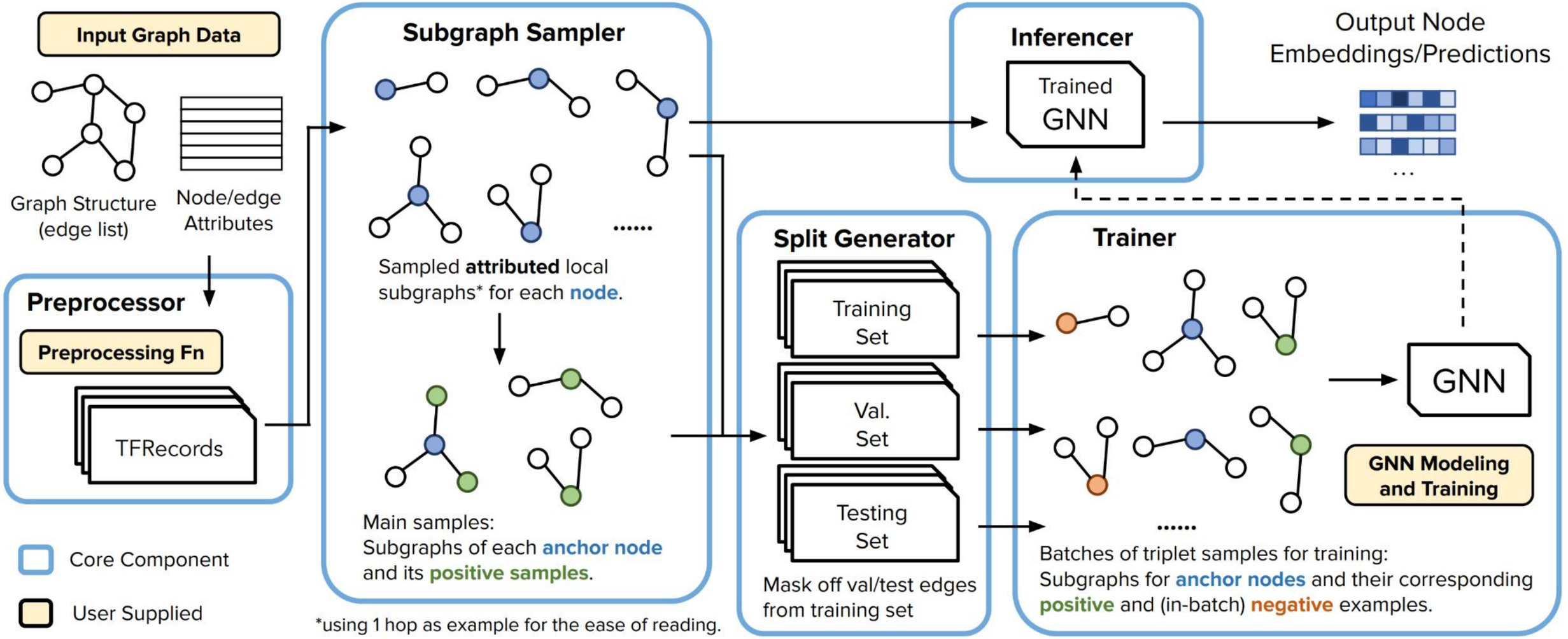
- Offline and Online workloads and real-time inference
- Needs machines with large memory
- Efficient in-memory subgraph sampling

Overview of GiGL

GiGL Components & Orchestration

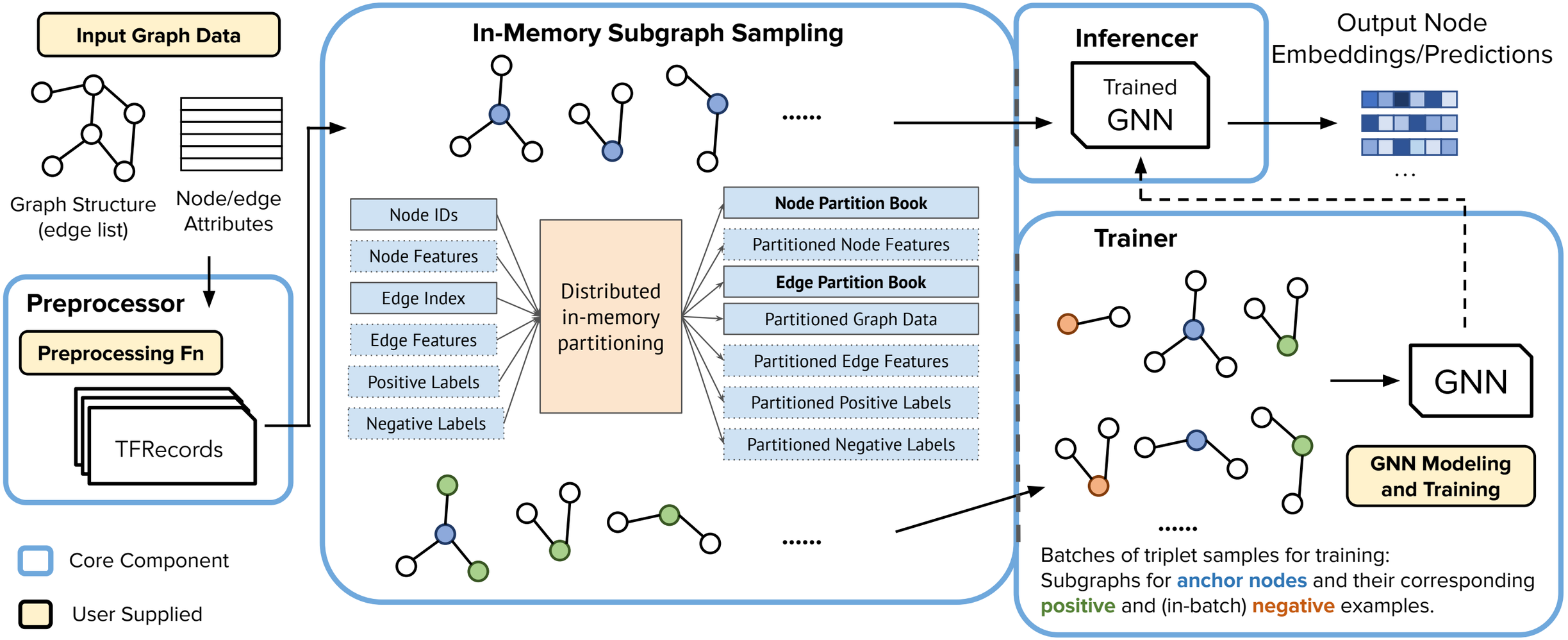
SNAPCHAT

GiGL components - tabularization



SNAPCHAT

GiGL components - In-memory

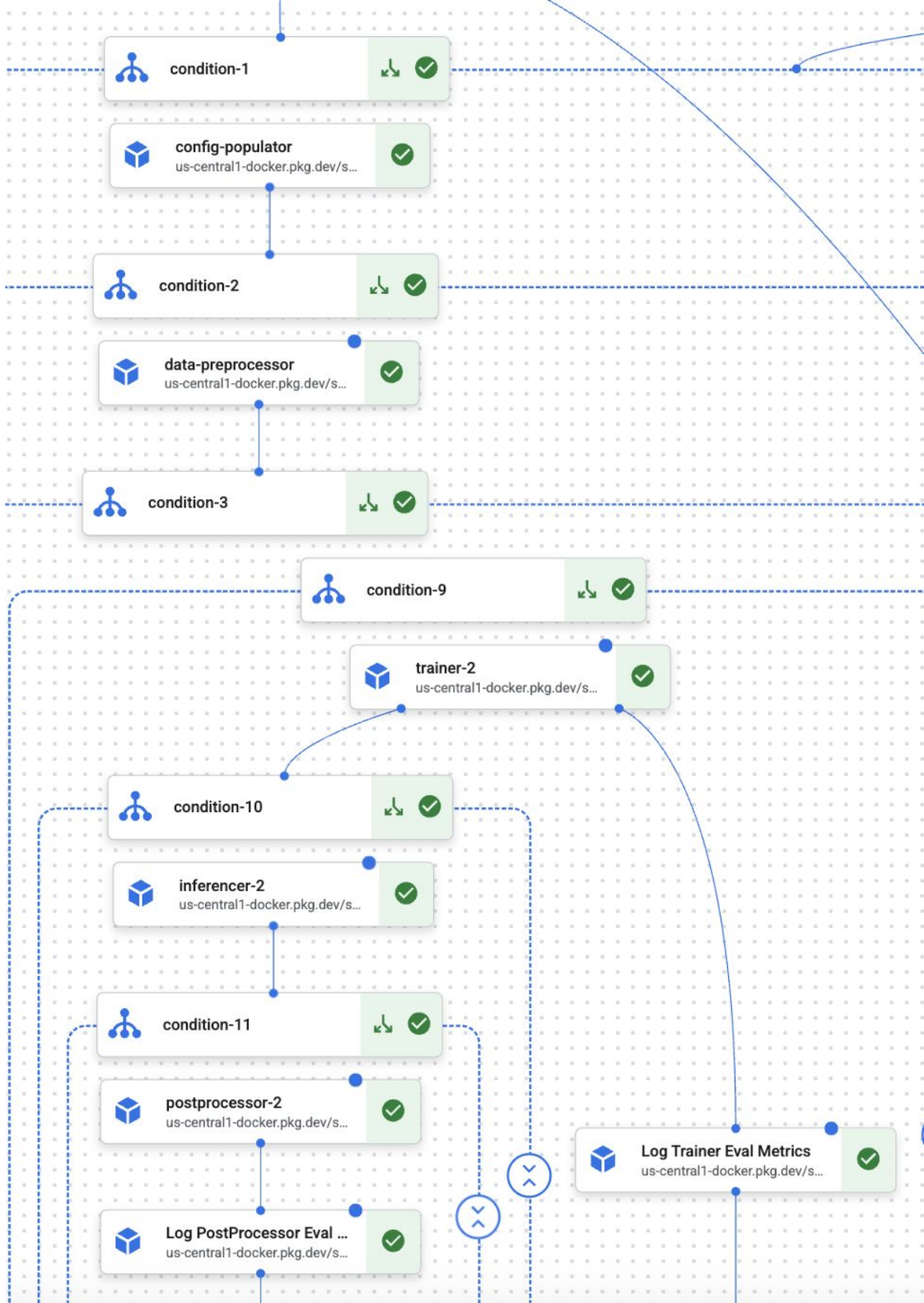


SNAPCHAT

GiGL Orchestration

GiGL is orchestrated with Kubeflow pipelines (KFP)

Rightside example: KFP deployed on GCP Vertex AI pipelines



Kubeflow https://www.kubeflow.org/
Vertex AI pipelines https://cloud.google.com/vertex-ai/docs/pipelines/introduction

SNAPCHAT

Where to use GiGL?

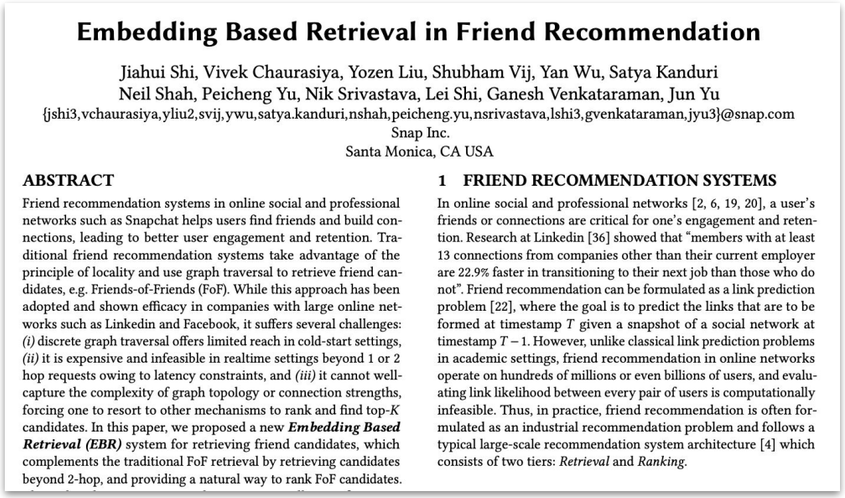
- Train graph ML models from small to large scale

- Modeling relational (graph) data with GNNs
  - Target Prediction (node classification, link prediction)
  - Embedding generation (supervised, self-supervised)
  - Yes graphs provides additional information to sequence models

- Recommendation, anomaly detection and more!

SNAPCHAT

Multiple impact verticals in product



Embedding Based Retrieval in Friend Recommendation

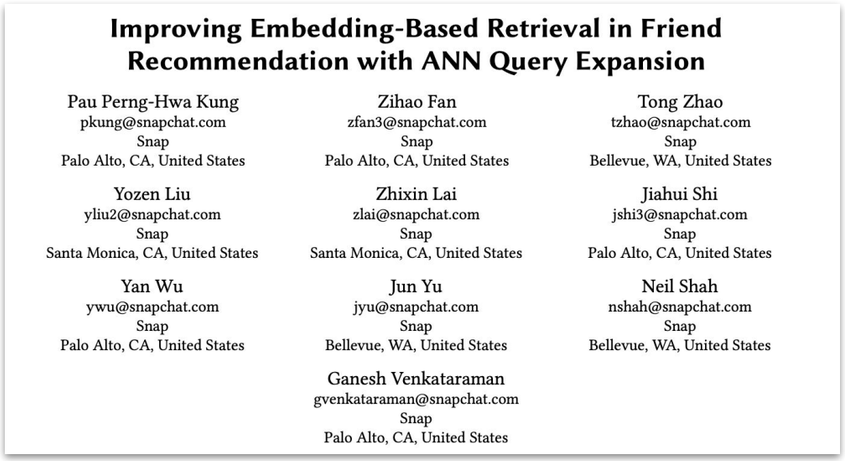
Jiahui Shi, Vivek Chaurasiya, Yozen Liu, Shubham Vij, Yan Wu, Satya Kanduri
Neil Shah, Peicheng Yu, Nik Srivastava, Lei Shi, Ganesh Venkataraman, Jun Yu
{jshi3,vchaurasiya,yliu2,svij,ywu,satya.kanduri,nshah,peicheng.yu,nsrivastava,lshi3,gvenkataraman,jyu3}@snap.com
Snap Inc.
Santa Monica, CA USA

ABSTRACT

Friend recommendation systems in online social and professional networks such as Snapchat helps users find friends and build connections, leading to better user engagement and retention. Traditional friend recommendation systems take advantage of the principle of locality and use graph traversal to retrieve friend candidates, e.g. Friends-of-Friends (FoF). While this approach has been adopted and shown efficacy in companies with large online networks such as Linkedin and Facebook, it suffers several challenges: *(i)* discrete graph traversal offers limited reach in cold-start settings, *(ii)* it is expensive and infeasible in realtime settings beyond 1 or 2 hop requests owing to latency constraints, and *(iii)* it cannot well-capture the complexity of graph topology or connection strengths, forcing one to resort to other mechanisms to rank and find top-K candidates. In this paper, we proposed a new ***Embedding Based Retrieval (EBR)*** system for retrieving friend candidates, which complements the traditional FoF retrieval by retrieving candidates beyond 2-hop, and providing a natural way to rank FoF candidates.

1 FRIEND RECOMMENDATION SYSTEMS

In online social and professional networks [2, 6, 19, 20], a user's friends or connections are critical for one's engagement and retention. Research at Linkedin [36] showed that "members with at least 13 connections from companies other than their current employer are 22.9% faster in transitioning to their next job than those who do not". Friend recommendation can be formulated as a link prediction problem [22], where the goal is to predict the links that are to be formed at timestamp T given a snapshot of a social network at timestamp $T-1$. However, unlike classical link prediction problems in academic settings, friend recommendation in online networks operate on hundreds of millions or even billions of users, and evaluating link likelihood between every pair of users is computationally infeasible. Thus, in practice, friend recommendation is often formulated as an industrial recommendation problem and follows a typical large-scale recommendation system architecture [4] which consists of two tiers: *Retrieval* and *Ranking*.



Improving Embedding-Based Retrieval in Friend Recommendation with ANN Query Expansion

Pau Perng-Hwa Kung
pkung@snapchat.com
Snap
Palo Alto, CA, United States

Zihao Fan
zfan3@snapchat.com
Snap
Palo Alto, CA, United States

Tong Zhao
tzhao@snapchat.com
Snap
Bellevue, WA, United States

Yozen Liu
yliu2@snapchat.com
Snap
Santa Monica, CA, United States

Zhixin Lai
zlai@snapchat.com
Snap
Santa Monica, CA, United States

Jiahui Shi
jshi3@snapchat.com
Snap
Palo Alto, CA, United States

Yan Wu
ywu@snapchat.com
Snap
Palo Alto, CA, United States

Jun Yu
jyu@snapchat.com
Snap
Bellevue, WA, United States

Neil Shah
nshah@snapchat.com
Snap
Bellevue, WA, United States

Ganesh Venkataraman
gvenkataraman@snapchat.com
Snap
Palo Alto, CA, United States

Launches	online improv.
GraphSAGE on friend graph (initial launch of EBR)	+10%, +11.6%, +7%
Graph definition update: engagement graph	+8.9%, +5.8%, +10.8%
Graph definition update: graph augmentation	+1.0%, +2.1%
Model update: GAT	+6.5%, +2.92%
Loss update: retrieval loss	+6.2%, +8.8%
Loss update: multi-task training	+1.2%, +1.3%
Task update: supervised LP with user-defined labels	+2.2%, +1.8%
EBR scheme update: Stochastic EBR	10.2%, 13.9%

Embedding-based Retrieval in Friend Recommendation (Shi et al, 2023)
Improving Embedding-based Retrieval in Friend Recommendation with ANN Query Expansion (Kung et al, 2024)

SNAPCHAT

Multiple impact verticals in product

Launches	online improv.
Spotlight EBR with GNN embeddings	+0.55%, +1.54%, +2.02%
Discover EBR with GNN embeddings	+0.35%, +1.02%, +3.17%

Leveraging GNN embeddings for Content Retrieval

Applications	online improv.
Community guideline violation detection	+10.9%, +6.9%, +36%
Bad actor detection	+25.9%, +27.1%
EBR for Display Name Search (DNS)	+6.23%, +4.65%
Social Graph Embeddings in DNS Heavy Ranker	+1.01%, +0.98%, +0.75%
Social Graph Embeddings in Lens Ranking	+0.54%, +0.51%, +0.62%

Multiple applications across Safety, Search and Lens Experience

Setup	MRR	HR@1
Base model: 2 layer GAT, sample 15 neighbors per layer	0.39	0.24
+ feature normalization	0.54	0.40
+ further feature eng.	0.59	0.46
+ increase to 3 layers, double training data-	0.64	0.53
+ # of neighbor sampling to 20	0.68	0.57
+ # of neighbor sampling to 30, half hidden dim.	0.70	0.59

Offline iterations for Content Retrieval

Setup	# edges	neighbors size	MRR	Precision	Recall
Control (graph embed.)	60B	-	-	0.1523	0.3220
2 layer GAT (v1)	3.36B	15	0.871	0.1138	0.3239
2 layer GAT (v2)	6.46B	15	0.959	0.1505	0.3819
2 layer GAT (v3)	6.46B	40	0.980	0.1523	0.4109

Offline iterations for Ad Ranking

SNAPCHAT

Future of GiGL

- GiGL is actively being developed
- We plan to support and open-source more graph embedding methods and graph tooling in the future
- We welcome open-source contributions!

SNAPCHAT

Want to learn more?

Check out our **KDD ‘25 ADS track paper** online

KDD ‘25 ADS track paper

GiGL is also **open-sourced** for public use;
please enjoy, reach out and contribute

Open-source repo (active!)

We have ample documentation to get
started building your own pipelines

Documentation and resources

SNAPCHAT

Acknowledgements

Google Labs for Sales Team

GraphScope

GraphLearn-for-PyTorch team

SNAPCHAT

Hands-on with GiGL

SNAPCHAT

Showcase

Let us showcase an End-to-End pipeline run with a large dataset

(MAG240M - billion scale edges)

SNAPCHAT

Please follow lab instructions

https://github.com/Snapchat/GiGL/tree/main/examples/tutorial/KDD_2025

SNAPCHAT

Break (30m)

We'll be back at 15:30

SNAPCHAT

Hands-on with GiGL

In-Memory SGS Deep Dive + Customizations

SNAPCHAT

Extending Toy Examples to Large Scale

SNAPCHAT

Resource Config Changes for large-scale runs

Toy Resource Config

```yaml
preprocessor_config:
  edge_preprocessor_config:
    num_workers: 1
    max_num_workers: 2
    machine_type: "n2-standard-16"
    disk_size_gb: 300
  node_preprocessor_config:
    num_workers: 1
    max_num_workers: 2
    machine_type: "n2-standard-16"
    disk_size_gb: 300
trainer_resource_config:
  vertex_ai_trainer_config:
    machine_type: "n1-highmem-8"
    gpu_type: ACCELERATOR_TYPE_UNSPECIFIED # CPU Training
    gpu_limit: 0  # CPU Training
    num_replicas: 1
inferencer_resource_config:
  vertex_ai_inferencer_config:
    machine_type: "n1-highmem-8"
    gpu_type: ACCELERATOR_TYPE_UNSPECIFIED # CPU Inference
    gpu_limit: 0  # CPU Inference
    num_replicas: 1
```

MAG240M Resource Config

```yaml
preprocessor_config:
  edge_preprocessor_config:
    num_workers: 1
    max_num_workers: 256
    machine_type: "n2d-highmem-64"
    disk_size_gb: 300
  node_preprocessor_config:
    num_workers: 1
    max_num_workers: 128
    machine_type: "n2d-highmem-64"
    disk_size_gb: 300
trainer_resource_config:
  vertex_ai_trainer_config:
    machine_type: n1-highmem-96
    gpu_type: NVIDIA_TESLA_T4
    gpu_limit: 4
    num_replicas: 20
inferencer_resource_config:
  vertex_ai_inferencer_config:
    machine_type: n1-highmem-96
    gpu_type: NVIDIA_TESLA_T4
    gpu_limit: 4
    num_replicas: 20
```

examples/tutorial/KDD_2025/resource_config.yaml

SNAPCHAT

Data Preprocessor – Prepare for Pipeline

- We can use the prepare_for_pipeline function of the DataPreprocessorConfig class to perform any operation needed prior to running the pipeline. This can include
  - Generating the labeled edge types for supervision edges
  - Filtering existing tables based on some criteria
  - Transforming existing graph tables

```python
def prepare_for_pipeline(
    self, applied_task_identifier: AppliedTaskIdentifier
) -> None:
    """
    This function is called at the very start of the pipeline before enumerator and datapreprocessor.
    This function does not return anything. It can be overwritten to perform any operation needed
    before running the pipeline, such as gathering data for node and edge sources

    Args:
        applied_task_identifier (AppliedTaskIdentifier): A unique identifier for the task being run. This is usually
            the job name if orchestrating through GiGL's orchestration logic.
    Returns:
        None
    """
    return None
```

examples/tutorial/KDD_2025/preprocessor_config.py

SNAPCHAT

Data Preprocessor – Prepare for Pipeline

- For MAG240M, we use this to generate features for the author and institution node types by taking the average of their 1-hop neighbors with features.

```python
def prepare_for_pipeline(
    self, applied_task_identifier: AppliedTaskIdentifier
) -> None:

    logger.info(
        f"Preparing for pipeline with applied task identifier: {applied_task_identifier}",
    )
    bq_utils = BqUtils(project=self._resource_config.project)

    author_table = (
        f"{self._resource_config.project}.{self._resource_config.temp_assets_bq_dataset_name}."
        + f"{applied_task_identifier}_author_feature_table"
    )

    average_author_query = query_template_compute_average_features.format(
        feature_table=self._paper_table,
        edge_table=self._author_writes_paper_table,
        join_identifier=self._paper_node_type,
        group_by_identifier=self._author_node_type,
        average_feature_query=self._average_feature_query,
    )
    bq_utils.run_query(
        query=average_author_query,
        labels={},
        destination=author_table,
        write_disposition=WriteDisposition.WRITE_TRUNCATE,
    )
```

```python
query_template_compute_average_features = """
WITH joined_table AS (
  SELECT
    *
  FROM `{feature_table}` AS feature_table
  JOIN
    `{edge_table}` AS edge_table
  ON
    feature_table.{join_identifier} = edge_table.{join_identifier}
)
SELECT
  {group_by_identifier},
  {average_feature_query}
FROM
  joined_table
GROUP BY
  {group_by_identifier};
    """
```

examples/MAG240M/preprocessor_config_heterogeneous.py - examples/MAG240M/queries.py

SNAPCHAT

Data Preprocessor – Adding Features

- The build_ingestion_feature_spec_fn in our preprocessing config is used to identify the input columns and their types from the source, in this case BigQuery.
- For our toy example, we provide our node identifiers in the fixed_int_fields and the features in fixed_float_fields, which are the same across both node types.

```
feature_spec_fn = build_ingestion_feature_spec_fn(
    fixed_int_fields=[node_identifier],
    fixed_float_fields=self._node_float_feature_list,
)
```

examples/tutorial/KDD_2025/preprocessing_config.py

SNAPCHAT

Data Preprocessor – Adding Features

- For more complex datasets, we can specify any typed combination of fields, and can vary this for each node and edge type
- Each node and edge type can have their own preprocessing functions for their features

```
def build_ingestion_feature_spec_fn(
    fixed_string_fields: Optional[list[str]] = None,
    fixed_string_field_shapes: dict[str, list[int]] = {},
    fixed_float_fields: Optional[list[str]] = None,
    fixed_float_field_shapes: dict[str, list[int]] = {},
    fixed_int_fields: Optional[list[str]] = None,
    fixed_int_field_shapes: dict[str, list[int]] = {},
    varlen_string_fields: Optional[list[str]] = None,
    varlen_float_fields: Optional[list[str]] = None,
    varlen_int_fields: Optional[list[str]] = None,
) -> Callable[[], FeatureSpecDict]:
```

gigl/src/data_preprocessor/lib/data_preprocessor_config.py

SNAPCHAT

Data Preprocessor – Preprocessing Functions

Passthrough Function

```python
def preprocessing_fn(inputs: TFTensorDict) -> TFTensorDict:
    return inputs
```

Normalizing Function

```python
def preprocessing_fn(inputs: TFTensorDict) -> TFTensorDict:
    outputs = inputs.copy()
    for feature_name in numerical_features:
        # outputs[feature] = tft.scale_to_z_score(inputs[feature]) would be used normally,
        # but this is not executable in a python environment, so we can calculate z-score manually below
        feat = inputs[feature_name]
        mean, variance = tf.nn.moments(feat, axes=0)
        stddev = tf.sqrt(variance)
        outputs[feature_name] = (feat - mean) / stddev
    for feature_name in categorical_features:
        feat = inputs[feature_name]
        # Extract unique city names
        unique_count, _ = tf.unique(feat)

        # Create a lookup table mapping each unique city to a unique integer
        lookup_table = tf.lookup.StaticHashTable(
            initializer=tf.lookup.KeyValueTensorInitializer(
                keys=unique_count,
                values=tf.range(tf.size(unique_count, out_type=tf.int64), dtype=tf.int64)
            ),
            default_value=-1  # This is the value returned for any key not in the table
        )

        # Use the lookup table to convert city names to integers
        encoded_cities = lookup_table.lookup(feat)
        outputs[feature_name] = tf.reshape(encoded_cities, tf.shape(feat))
    return outputs
```

examples/tutorial/KDD_2025/heterogeneous_walkthrough.ipynb

SNAPCHAT

Adding Modeling Levers

SNAPCHAT

Task Definition - Self-Supervised Link Prediction

- Link prediction is specified in the taskMetadata field

```yaml
taskMetadata:
  nodeAnchorBasedLinkPredictionTaskMetadata:
    supervisionEdgeTypes:
    - dstNodeType: story
      relation: to
      srcNodeType: user
```

- Self-supervised edges can be selected as an argument in trainerArgs field

```yaml
trainerConfig:
  trainerArgs:
    ssl_positive_label_percentage: "0.7"
  command: python -m examples.tutorial.KDD_2025.heterogeneous_training
```

examples/tutorial/KDD_2025/task_config.yaml

SNAPCHAT

Task Definition - Supervised Link Prediction

Alternatively, we can specify positive and negative edges in the DataPreprocessorConfig class for the link prediction task.

If we have two BigQuery tables which contains positive and negative supervision edges. We can preprocess these edges in get_edges_preprocessing_spec()

```
main_edge_ref = BigqueryEdgeDataReference(
    reference_uri=table,
    edge_type=edge_type,
    edge_usage_type=EdgeUsageType.MAIN,
)
```

Main Edge Reference

```
positive_edge_data_ref = BigqueryEdgeDataReference(
    reference_uri=self.positive_edge_table,
    edge_type=labeled_edge_type,
    edge_usage_type=EdgeUsageType.POSITIVE,
)

negative_edge_data_ref = BigqueryEdgeDataReference(
    reference_uri=self.negative_edge_table,
    edge_type=labeled_edge_type,
    edge_usage_type=EdgeUsageType.NEGATIVE,
)
```

Labeled Reference

```
return {
    main_edge_data_ref: edge_data_preprocessing_spec,
    positive_edge_data_ref: edge_data_preprocessing_spec,
    negative_edge_data_ref: edge_data_preprocessing_spec
}
```

EdgeDataPreprocessingSpec

examples/tutorial/KDD_2025/preprocessor_config.py

SNAPCHAT

Task Definition - Supervised Link Prediction

This will populate our YAML at the preprocessedMetadataURI with fields specifically for positive and negative edges for that edge type.

These fields are automatically compatible with our training and inference data loaders and will load these labeled edges instead of the self-supervised edges.

```
condensedEdgeTypeToPreprocessedMetadata:
    '0':
        mainEdgeInfo:
          dstNodeIdKey: dst
          mainEdgeInfo:
            featureDim: 2
            featureKeys:
            - f0
            - f1
            schemaUri: …
            tfrecordUriPrefix: …
        negativeEdgeInfo:
            featureDim: 0
            schemaUri: …
            tfrecordUriPrefix: …
        positiveEdgeInfo:
            featureDim: 0
            schemaUri: …
            tfrecordUriPrefix: …
    …
```

SNAPCHAT

Task Definition - Node Classification

- We have enabled node classification support for tabularized SGS jobs and are in the process of enabling it for in-memory SGS training and inference.

- Node Classification can be specified in the taskMetadata field

```yaml
taskMetadata:
  nodeBasedTaskMetadata:
    supervisionNodeTypes:
      - user
```

examples/tutorial/KDD_2025/task_config.yaml

SNAPCHAT

Trainer – Hyperparameter Tuning

- Any training loop can be specified by the command field of the trainerConfig
- Arguments can be provided through trainerArgs

```yaml
trainerConfig:
  trainerArgs:
    fanout: "[10, 10]"
    learning_rate: "0.0005"
    batch_size: "256"
  command: python -m examples.link_prediction.heterogeneous_training
```

examples/tutorial/KDD_2025/task_config.yaml

SNAPCHAT

Inferencer – Hyperparameter Tuning

- Similarly, inference loops can be specified by the command field of the inferencerConfig

- Arguments can be provided through inferencerArgs
  - Batch size can alternatively be provided through inferenceBatchSize

```yaml
inferencerConfig:
  inferencerArgs:
    fanout: "[10, 10]"
  inferenceBatchSize: 512
  command: python -m examples.link_prediction.heterogeneous_inference
```

examples/tutorial/KDD_2025/task_config.yaml

SNAPCHAT

Post Processor

- To allow flexibility for any post-inferencer operations, we also have an PostProcessor component which can be optionally specified in the task config

- Handles operations such as asset cleanup, outputting metrics, etc

```yaml
postProcessorConfig:
  postProcessorArgs: {}
  postProcessorClsPath: path.to.custom.postProcessor
```

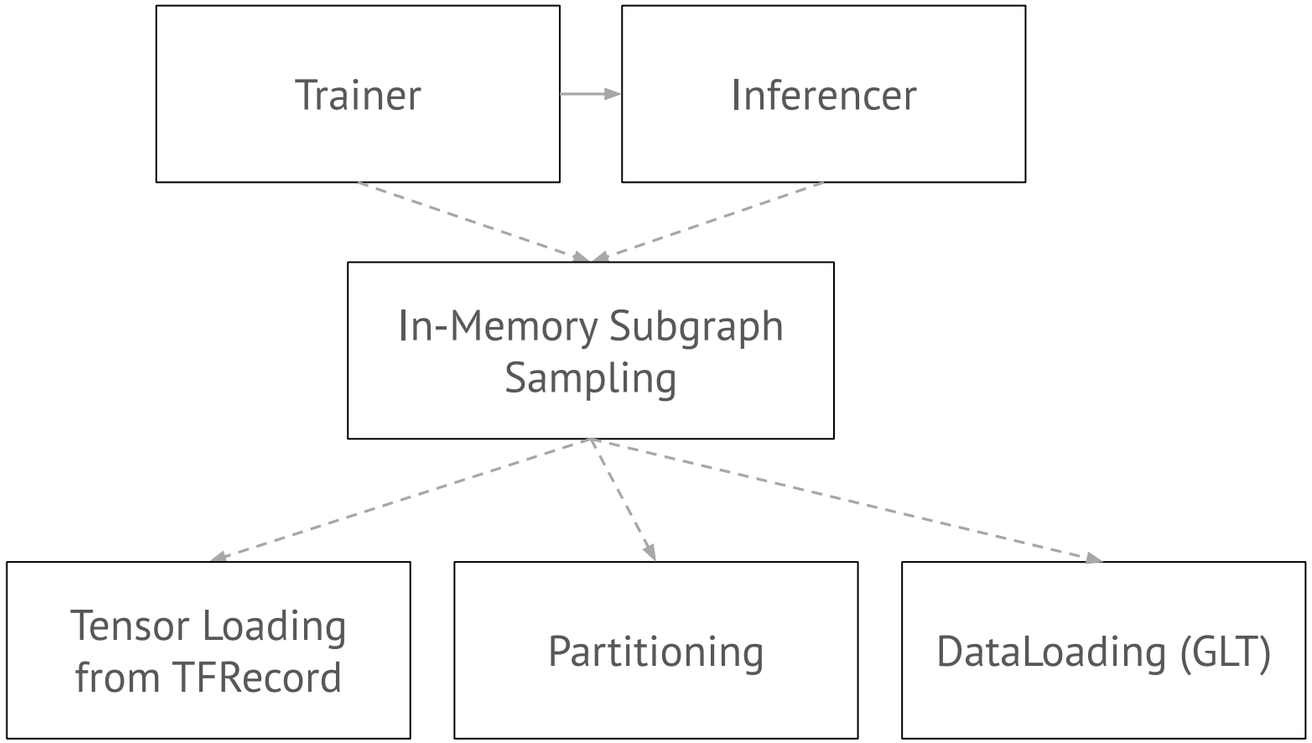
```python
class BasePostProcessor(ABC):
    """
    Post processor does all operations required after inferencer.
    Ex. persist inferencer output assets to text files, or run checks on output metrics etc.
    """

    @abstractmethod
    def run_post_process(
        self, gbml_config_pb: gbml_config_pb2.GbmlConfig
    ) -> Optional[EvalMetricsCollection]:
        raise NotImplementedError
```

gigl/src/post_processor/lib/data_preprocessor_config.py

SNAPCHAT

In-Memory SGS Deep Dive and Tuning

In-Memory Subgraph Sampling Overview



SNAPCHAT

In-Memory Subgraph Sampling Overview

TFRecord Loading + Partitioning

Dataloading

Training

```
fanout = [10, 10]

dataset = build_dataset_from_task_config_uri(...)

main_loader = DistABLPLoader(
    dataset=dataset, input_nodes=dataset.train_node_ids, fanout=fanout
)
random_loader = DistNeighborLoader(
    dataset=dataset, input_nodes=dataset.node_ids, fanout=fanout,
)

model: torch.nn.parallel.DistributedDataParallel = ...
loss_fn = ...
optimizer = ...

for batch_idx, (main_data, random_data) in enumerate(zip(main_loader, random_loader)):
    optimizer.zero_grad()
    loss = compute_loss(main_data, random_data, model, loss_fn)
    loss.backward()
    optimizer.step()
```

SNAPCHAT

In-Memory Subgraph Sampling Components

In this step, we load the TFRecords from DataPreprocessor into memory across multiple machines. Each machine will house a subset of the data as torch tensors containing information such as:

- Node Features
- Edge Indices
- Edge Features
- Positive and Hard Negative Labels

SNAPCHAT

In-Memory Subgraph Sampling Components

Two options for how we can load TFRecords:

- **Sequential**: Loads all entities (Node, Edge, Labels) sequentially. Slower, but has smaller peak memory usage. Recommended for cases where memory is a bottleneck
- **Parallel**: Loads all entities (Node, Edge, Labels) in parallel. Faster but larger peak memory usage. Recommended for smaller datasets

Can be determined by setting the **should_load_tensors_in_parallel** flag, defaults to True in our examples.

```
trainerConfig:
  trainerArgs:
    should_load_tensors_in_parallel: True
```

SNAPCHAT

In-Memory Subgraph Sampling Components

In this step, we shuffle the graph data across the machines so that each machine's edges live on the same machine as its source node (if the edge direction is outward) or destination node (if the edge direction is inward)

Generates *partition_books* which store information about what machine each edge index, edge feature, node feature, and labels live on

The partitioned data is subsequently stored in a **DistLinkPredictionDataset** class to be used for data loading.

SNAPCHAT

In-Memory Subgraph Sampling Components

Two options for how we can partition the data

- **Tensor-based Partitioning**: The partition book is a large tensor equal in size to the number of entities (nodes, edges). This has higher memory usage, but results in faster lookup for identifying which rank an item lives on. Recommended for small datasets.

1	0	2	0	3	3	1	2

Length = Number of Nodes

Partition_book[5] = 3

SNAPCHAT

In-Memory Subgraph Sampling Components

Two options for how we can partition the data

- **Range-based Partitioning**: Keeps track of *ranges* of IDs. This is *much* more memory efficient. Recommended for larger datasets.

2	4	6	8

Length = Number of Machines

Partition_book[5] = 2

SNAPCHAT

In-Memory Subgraph Sampling Components

Two options for how we can partition the data

- **Tensor-based Partitioning**: The partition book is a large tensor equal in size to the number of entities (nodes, edges). This has higher memory usage, but results in faster lookup for identifying which rank an item lives on. Recommended for small datasets.
- **Range-based Partitioning**: Keeps track of *ranges* of IDs. This is *much* more memory efficient. Recommended for larger datasets.

Can be determined by setting the **should_use_range_partitioning**, defaults to True in our examples.

```
trainerConfig:
  trainerArgs:
    should_use_range_partitioning: True
```

SNAPCHAT

In-Memory Subgraph Sampling Components

- Dataloaders produce subgraph samples on the fly using the partitioned data. Sampling occurs asynchronously – subgraph batches are pre-fetched to improve throughput for training/inference.
- The GiGL anchor-based link prediction (ABLP) Dataloader fans out around anchor nodes and labeled nodes when sampling subgraphs.

SNAPCHAT

In-Memory Subgraph Sampling Tuning

- **Sampling_workers_per_process**
  - Number of workers per process for sampling.
  - Can be tuned to speedup sampling operations – defaults to 4.
- **Sampling_worker_shared_channel_size**
  - The shared memory buffer size allocated for the channel during sampling
  - Can be tuned to decrease memory usage from dataloaders – defaults to `4GB`
- **Process_start_gap_seconds**
  - Spaced out data loader initialization across processes to reduce memory spike
  - Default to 0, should be set to 30 in large-scale settings.

```
trainerConfig:
  trainerArgs:
    sampling_workers_per_process: "4"
    sampling_worker_shared_channel_size: "4GB"
    process_start_gap_seconds: "30"
```

SNAPCHAT

Revisiting our Large-scale run

Let us take a look at our completed End-to-End pipeline run

SNAPCHAT

Conclusions

SNAPCHAT

You now have:

- A good understanding of GNN scale challenges and how GiGL solves them
- Hands-on experience and familiarity with GiGL and its components
- Ability to train your own GNNs at large-scale
- We’re excited to see how GiGL can be leveraged in your use case!

SNAPCHAT

FAQ

What scale does GiGL support? Internally we have tested GiGL up to *1e9* nodes and *1e10* edges.

What's the cost of GiGL? Our internal inference pipelines cost <$100 for our scale. This however will depend on your GCP cost, and feature size.

Can GiGL handle temporal graphs? No, we find that Graph Snapshots work well, sequence features can be embedded as node/edge features.

Can GiGL handle real time inference? GiGL can be customized to handle real-time inference.

Does GiGL support graph methods other GNNs? We will be introducing more types of Graph ML methods in the future (ex. Graph embedding based methods etc.)

Can you integrate GiGL with graph databases? We find that in-memory solutions work best for scale, and we're internally moving towards that.

SNAPCHAT

Thank you for attending!

For more questions related to GiGL, please check the resources below.

GiGL documentation

Open source Github repo