MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks

Roger Waleffe*
University of Wisconsin–Madison

Jason Mohoney
Universtity of Wisconsin–Madison

Theodoros Rekatsinas
ETH Zürich

Shivaram Venkataraman
University of Wisconsin–Madison

DANIEL SCHAFFER

MAY 7, 2024

Introduction

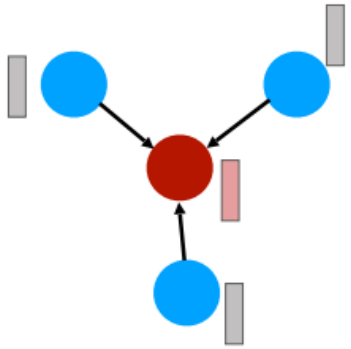
Graph Neural Networks

- Each vertex has an associated feature vector (per layer)
- At each step, at each vertex:
  - Each neighboring vertex sends a message based on its feature
  - Aggregate messages from neighbors (sum, …)
  - **Update the current feature based on the messages**
- Each of these functions (especially the last) can be a neural network
- Starting with the local neighborhood, each iteration implicitly incorporates information about more distant vertices

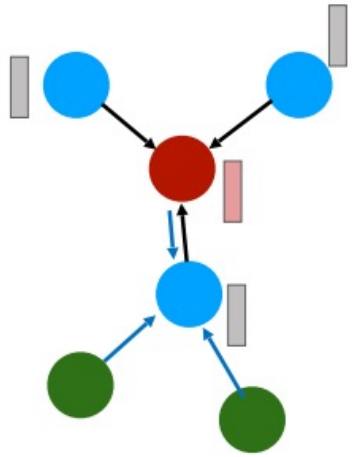
Graph Neural Networks

- Aggregate messages from neighbors (sum, …)
- Update the current feature based on the messages

1



2



feature description of node u in round k-1

(message = identity)

In each round *k*:

Aggregate over neighbors

$$m_{\mathcal{N}(v)}^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$

Update: Combine with current node

$$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, m_{\mathcal{N}(v)}^{(k)}\right)$$

Figure: Regina Barzilay

Applications of GNNs

- The final features are used as inputs to another task
- Depends on what the final task/function/neural network is

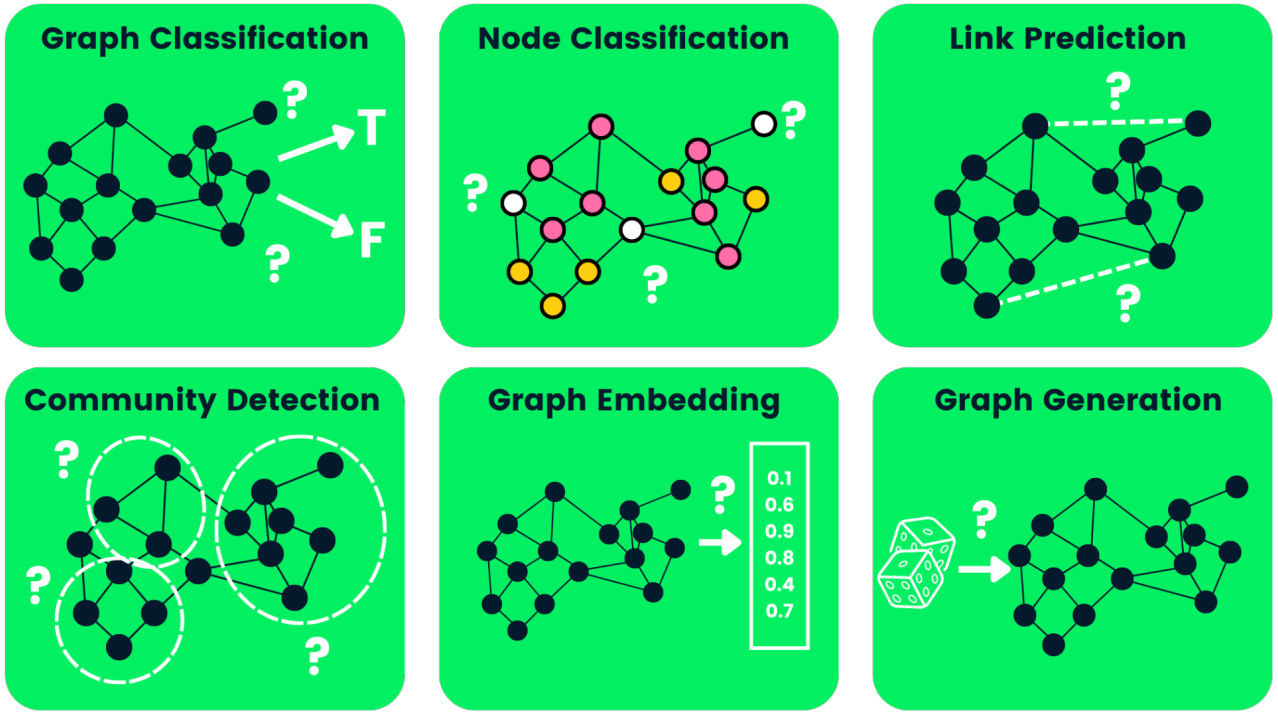


Figure: Datacamp

GNNs are costly to train

- In general, GNNs combine the difficulties of NNs and graphs
- Neural networks: matrix math, many epochs
  - Use a GPU
  - Train in batches
- Graphs: large, random accesses, varying neighborhood sizes
  - Much too large to fit in GPU
  - Batches/neighborhoods may not be efficiently arranged in memory

Training overview

- Run the model on a batch of nodes
  1. Compute features at all layers
  2. Compute final prediction and loss w.r.t. desired output
  3. Use loss and gradients to update model weights
  4. (Possibly) update initial node features
- Repeat

Sampling vertices

- For large graphs, may choose to sample a fraction of the neighbors
- In practice, this can take the most time (on CPU)
- Naïve implementations will access the same vertex multiple times

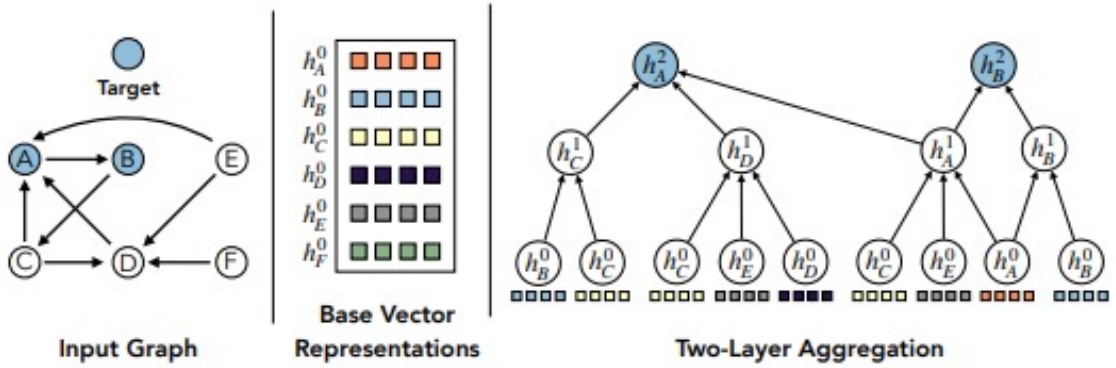


Figure 1. Example two-layer GNN aggregation for nodes {A, B} using a sample of their two-hop incoming neighborhood.

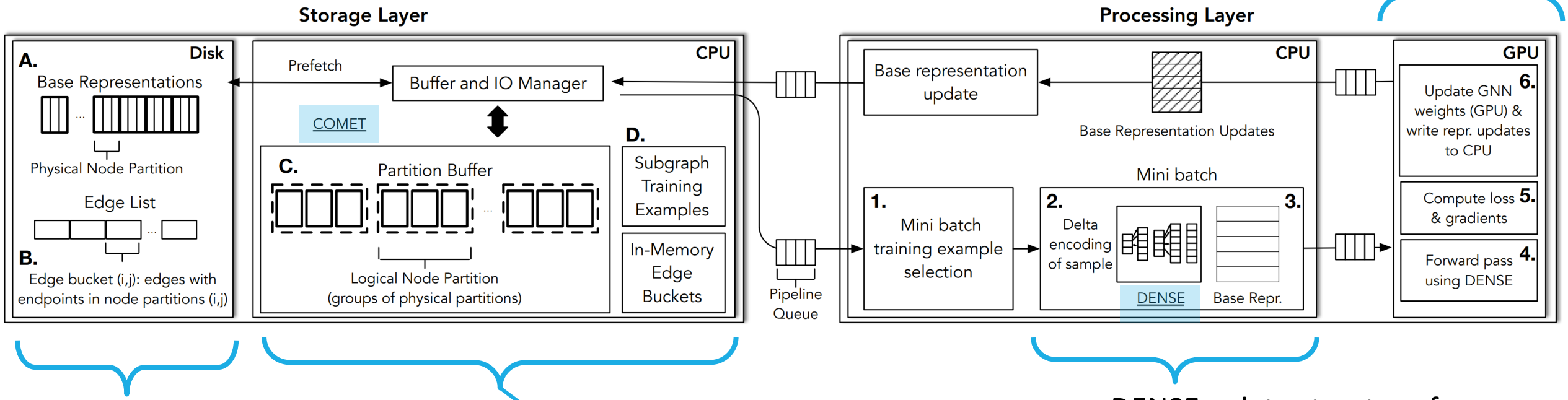
Distributed training

- Ideally, we would magically fit the entire graph in CPU/GPU memory
- Graphs that are too large must be split into partitions that fit in one machine's main memory
- Each batch can be drawn only from a single partition
- Partitions can be:
  - Distributed to multiple machines in parallel
  - Sequentially loaded and sampled, then sent to parallel GPUs
  - **Sequentially loaded/unloaded and used with one GPU**

MariusGNN Overview

- Main idea: it sometimes wastes more time to distribute training to multiple machines than to just keep parts of the graph on a single machine's hard drive
  - Assuming the graph can fit on one SSD
- Also:
  - Efficient sampling of multi-hop neighborhoods
  - A policy for choosing which partitions to load/unload when

Schematic



Permanent graph storage:

- Vertex features (in partitions)
- Edge list (in buckets: edges between nodes in partition i and j)

COMET – partition loading policy

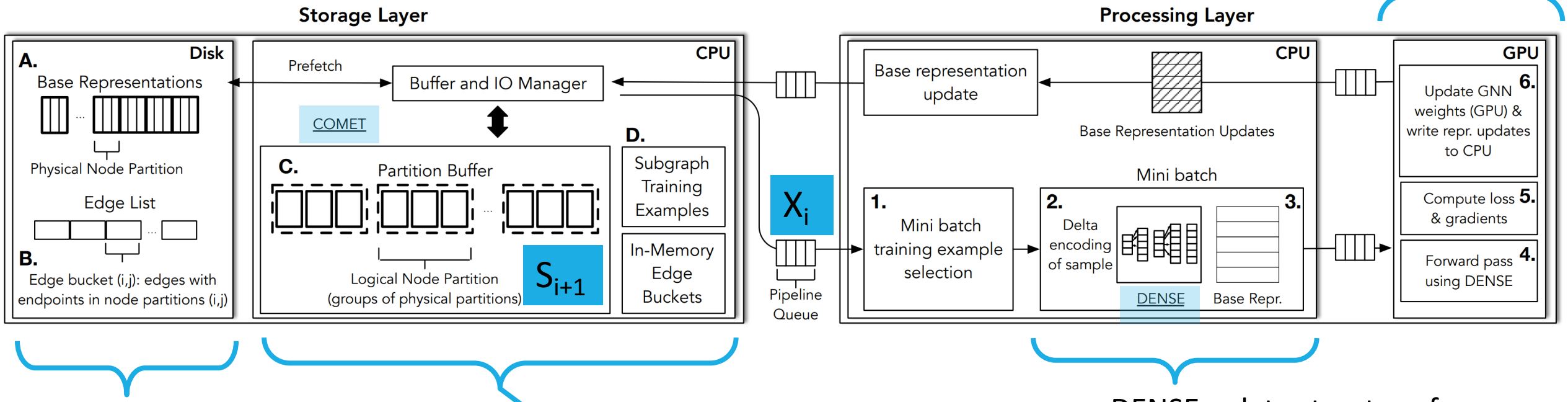
- Graph partitions are periodically loaded into main memory

DENSE – data structure for neighborhood sample reuse

Training Scheme

- Train in epochs, each example (labeled vertex) is processed once
- Physical partitions are randomly assigned to logical partitions S_i
  - Each logical partition is small enough to fit into main memory
  - Random order of examples is important for model performance
- Each logical partition is subset into training data X_i
  - If training for link prediction, done at the level of node pairs
- Physical partitions and edge buckets are loaded in the order required by the order of S_i
- Each X_i is passed to the processing component, and minibatches are sampled
- Model is evaluated on each batch, and updates are computed

Schematic



Permanent graph storage:

- Vertex features (in partitions)
- Edge list (in buckets: edges between nodes in partition i and j)

COMET – partition loading policy

- Graph partitions are periodically loaded into main memory

DENSE – data structure for neighborhood sample reuse

Sampling revisited

- We want to compute h^2_A.
- We sample C and D from A's 1-hop neighborhood, using h^1_C and h^1_D
- But, we also need h^1_A, which requires the same sampling

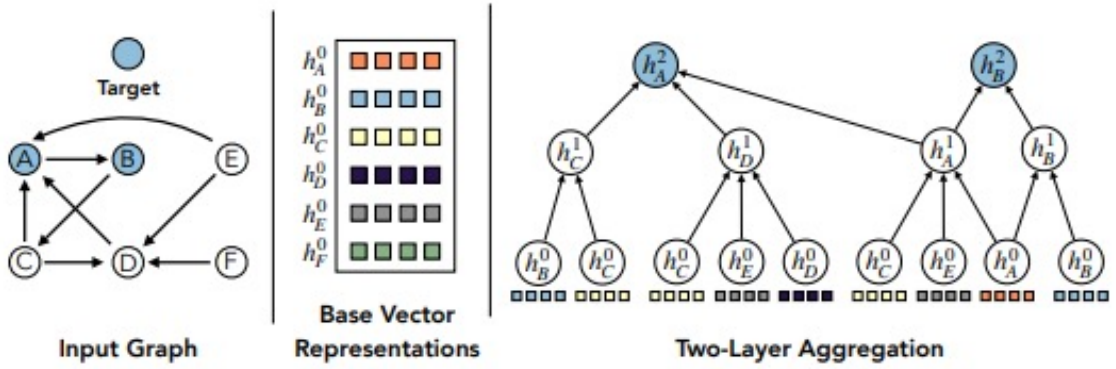
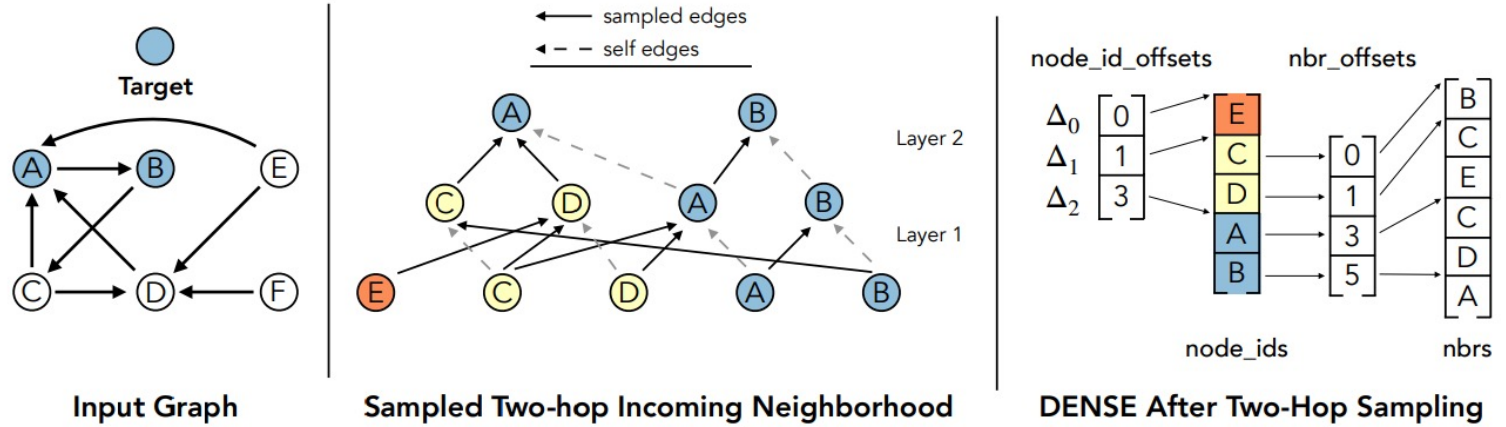


Figure 1. Example two-layer GNN aggregation for nodes {A, B} using a sample of their two-hop incoming neighborhood.

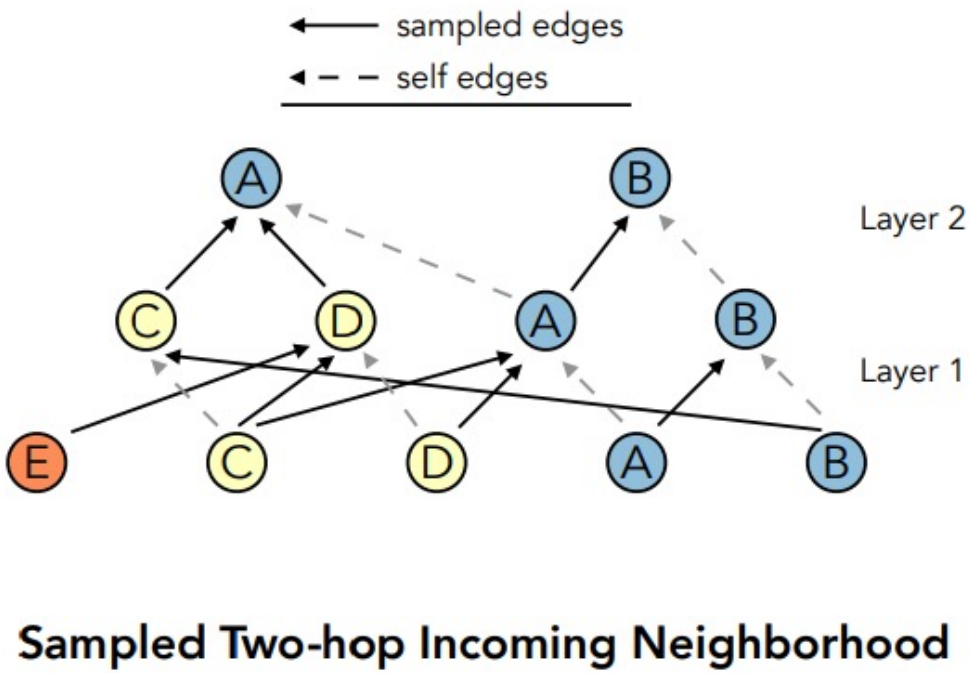
DENSE

- Data structure to save/reuse samples of 1-hop neighborhoods
- We can view *k*-hop sampling as recursive: sample 1-hop neighbors of the *(k-1)*-hop neighborhood
- Dynamic programming approach: save 1-hop samples and only sample unsampled nodes



Input Graph | Sampled Two-hop Incoming Neighborhood | DENSE After Two-Hop Sampling

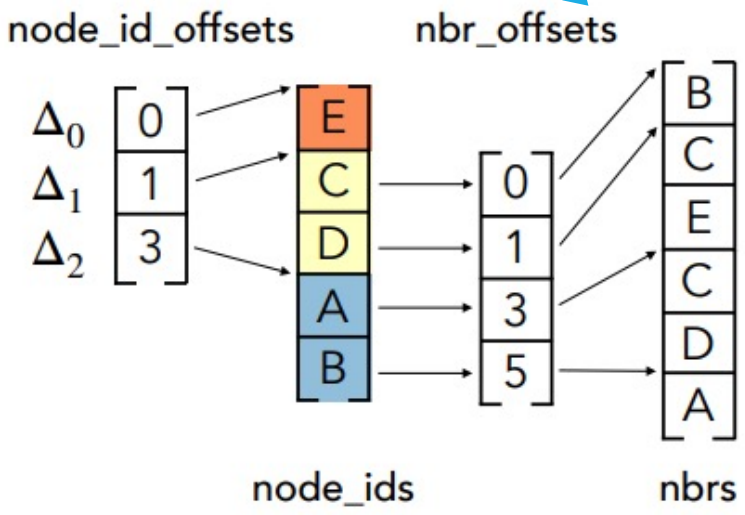
DENSE



Sampled Two-hop Incoming Neighborhood

Groups of nodes first reached after hop size Δ

Offsets into nbrs

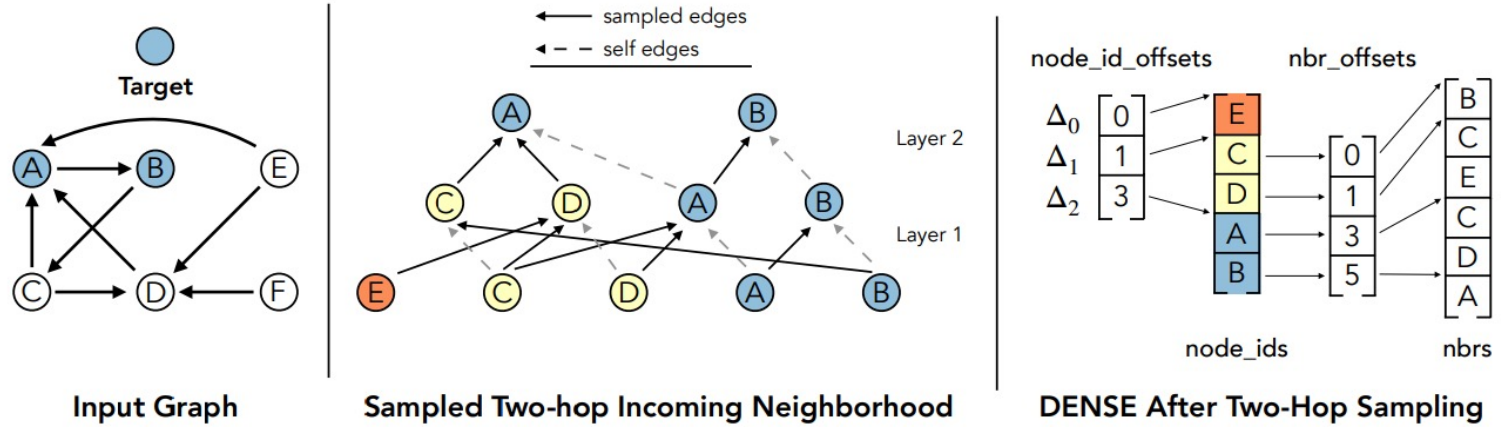


All node IDs in sample

Sampled 1-hop neighbors

DENSE

- 1-hop sampling performed on CPU, in parallel across nodes
  - Number of nodes to sample is a user parameter
- New neighbors are stacked onto the end of the existing arrays
- Trade-off: reuse does reduce randomness a bit

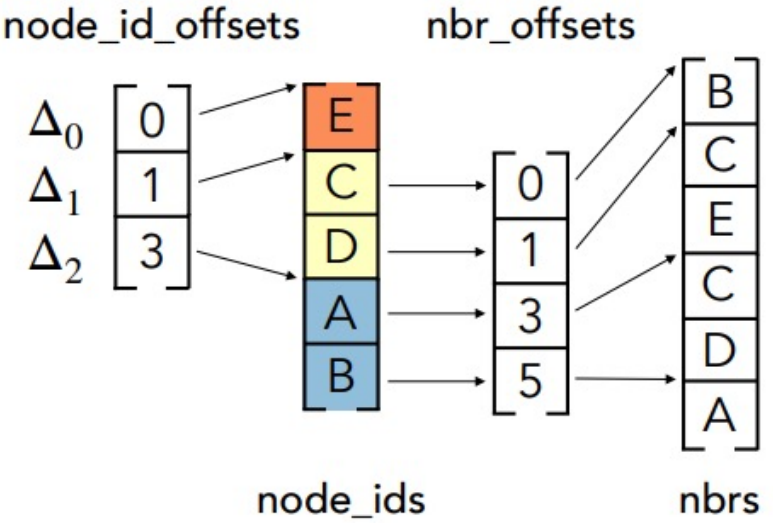


Multi-hop sampling

Algorithm 1: Multi-hop Neighborhood Sampling

Input: target_nodes: unique node IDs for k-hop sampling;
fanouts: max # of neighbors to sample per hop

```
node_id_offsets = [0]; node_ids = target_nodes
nbr_offsets = [ ]; nbrs = [ ]; Δ_k = target_nodes
for i in [k . . . 1] do
    Δ_i_nbrs, Δ_i_offsets = oneHopSample(Δ_i, fanouts[i])
    nbr_offsets = cat(Δ_i_offsets, nbr_offsets + len(Δ_i_nbrs))
    nbrs = cat(Δ_i_nbrs, nbrs)
    Δ_{i-1} = computeNextDelta(Δ_i_nbrs, node_ids)
    node_id_offsets = cat([0], node_id_offsets + len(Δ_{i-1}))
    node_ids = cat(Δ_{i-1}, node_ids)
return DENSE(node_id_offsets, node_ids, nbr_offsets, nbrs)
```



DENSE After Two-Hop Sampling

DENSE on GPU

- Sent to GPU:
  - DENSE data structure after sampling
  - Base features/representation for each node in DENSE.nbrs
- Forward pass on GPU:
  - Iterate over each layer
  - Compute the output for layer i according to the GNN
  - Remove nodes and neighbors that will not be needed in later layers
- Keeping only nodes that are needed allows dense GPU kernels

Algorithm 3: k^{th} GNN Layer Additive Aggregation

Input: DENSE; H^{k-1}: layer input vector representations

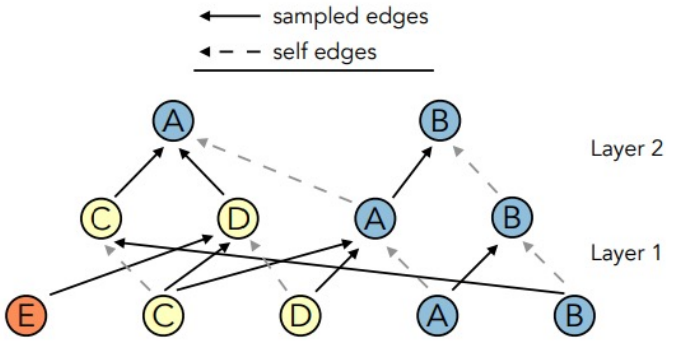
1 nbr_repr = H^{k-1}.index_select(DENSE.repr_map)
2 nbr_aggr = segment_sum(nbr_repr, DENSE.nbr_offsets)
3 self_repr = H^{k-1}[DENSE.node_id_offsets[1] :]
4 H^k = nbr_aggr + self_repr
5 **return** H^k

Updating DENSE during forward pass

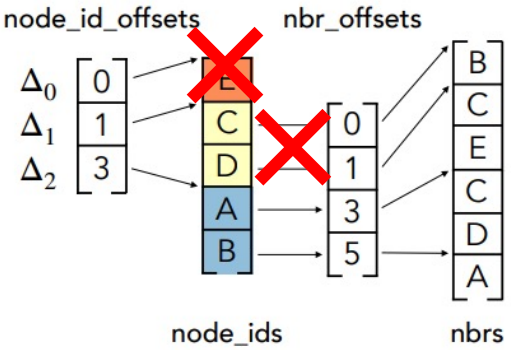
Algorithm 2: On GPU DENSE Update After Layer i

Input: node_id_offsets, node_ids, nbr_offsets, nbrs, repr_map

1 Δ_{i-1} = node_ids[: node_id_offsets[1]]
2 Δ_i = node_ids[node_id_offsets[1] : node_id_offsets[2]]
3 Δ_i_nbrs = nbrs[: nbr_offsets[len(Δ_i)]]
4 nbrs = nbrs[len(Δ_i_nbrs):]
5 repr_map = repr_map[len(Δ_i_nbrs):] - len(Δ_{i-1})
6 nbr_offsets = nbr_offsets[len(Δ_i):] - len(Δ_i_nbrs)
7 node_ids = node_ids[node_id_offsets[1]:]
8 node_id_offsets = node_id_offsets[1:] - len(Δ_{i-1})
9 **return** node_id_offsets, node_ids, nbr_offsets, nbrs, repr_map



Sampled Two-hop Incoming Neighborhood

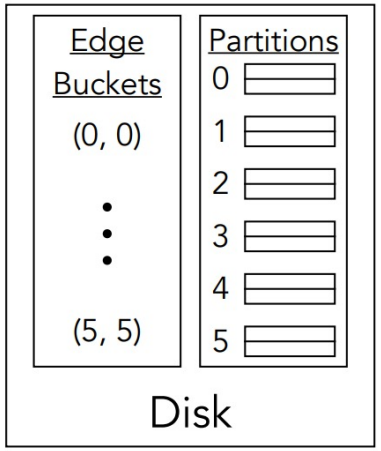


DENSE After Two-Hop Sampling

Partition Replacement Policy

- Recall: this determines when each partition is loaded into memory
  - Focus on the link prediction case, so examples are vertex pairs
- Simple idea: greedily minimize IO
  - swap partitions such that new partitions maximize # of new training examples
  - e.g., BETA brings in one partition and uses edges between new & existing partitions
  - all training examples are correlated because they all have one vertex in that partition
- Randomness of training examples is assumed by the SGD/batch training framework and is empirically important for model performance

Greedy replacement example



One Training Epoch: Greedy Policy

Partitions In Memory	Training Examples: (all edges in the following edge buckets)
S_1: {0, 1, 2, 3}	X_1: {0, 1, 2, 3} x {0, 1, 2, 3}
S_2: {0, 1, 2, **4**}	X_2: (0,**4**) (1,**4**) (2,**4**) (**4**,0) (**4**,1) (**4**,2) (**4**,**4**)
S_3: {0, 1, 2, **5**}	X_3: (0,**5**) (1,**5**) (2,**5**) (**5**,0) (**5**,1) (**5**,2) (**5**,**5**)
S_4: {1, 2, **3**, 5}	X_4: (**3**,5) (5,**3**)
S_5: {2, 3, **4**, 5}	X_5: (3,**4**) (**4**,3) (**4**,5) (5,**4**)

Figure 4. Greedy sequence of partitions in memory S and training examples X that are correlated. E.g., the examples in X_2 all come from edge buckets containing partition four.

COMET Replacement Policy

- Add sources of **randomness**
  - Random grouping of physical partitions into logical partitions
    - Small physical partitions reduce # of nodes that must stay together
    - Large logical partitions improve turnover/reduce IO per epoch
  - Random selection of training examples
    - Pairs of physical partitions are randomly assigned to be trained on during any memory state when both are loaded
- Sequence of logical partitions is greedy – one partition is swapped at each step until all pairs have co-occurred
  - Sequence of examples/physical partitions is not greedy

COMET Example

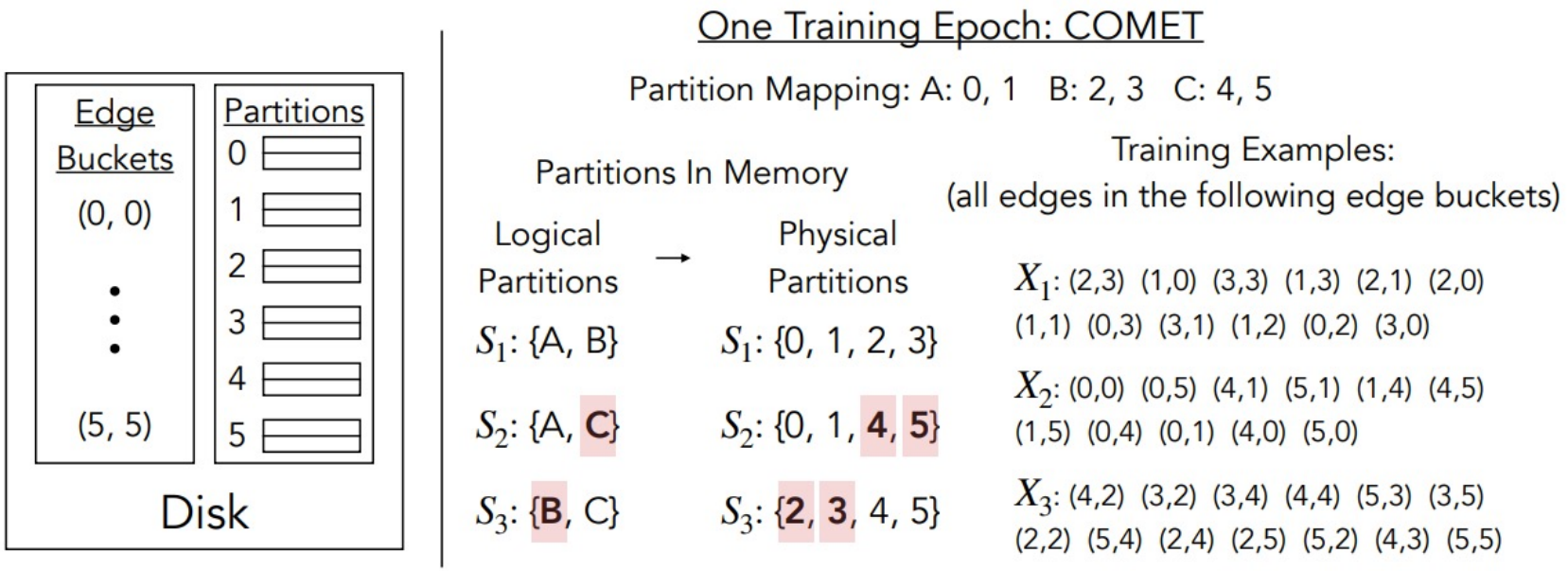
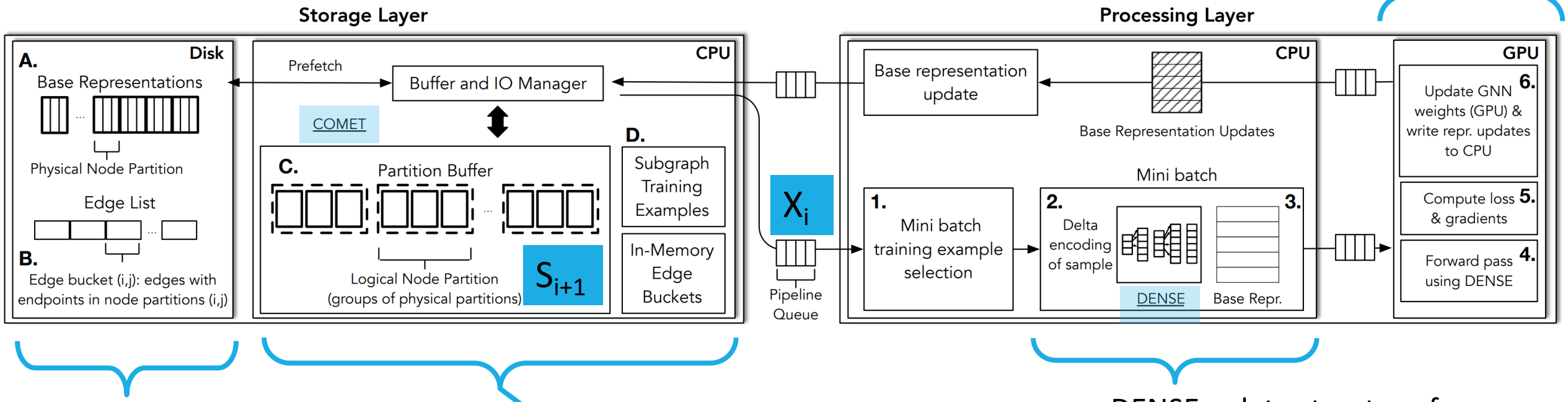


Figure 5. Partition and training example sequences generated by COMET to minimize training example correlation.

Replacement Policy for Node Classification

- No issue of correlated incident vertices, so a simple policy is empirically fine
- All training (labeled) nodes are grouped in physical partitions.
- If they can all fit in memory, load those and a few random partitions, up to the buffer capacity.
- All training nodes are assigned to create mini batches .
- Otherwise, load some partitions and randomly swap (without replacement) logical partitions until all have appeared in memory.

Schematic



Permanent graph storage:

- Vertex features (in partitions)
- Edge list (in buckets: edges between nodes in partition i and j)

COMET – partition loading policy

- Graph partitions are periodically loaded into main memory

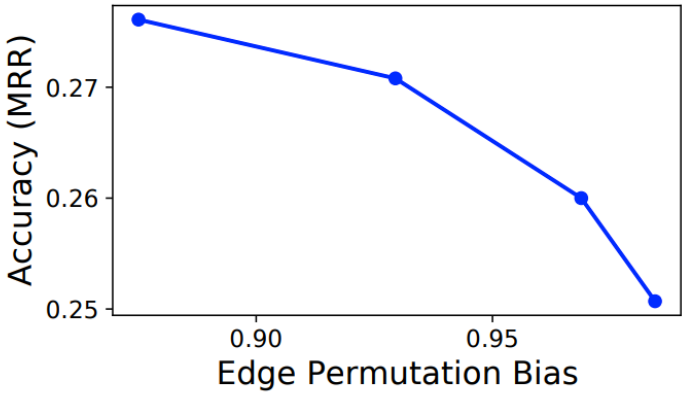
DENSE – data structure for neighborhood sample reuse

Parameters

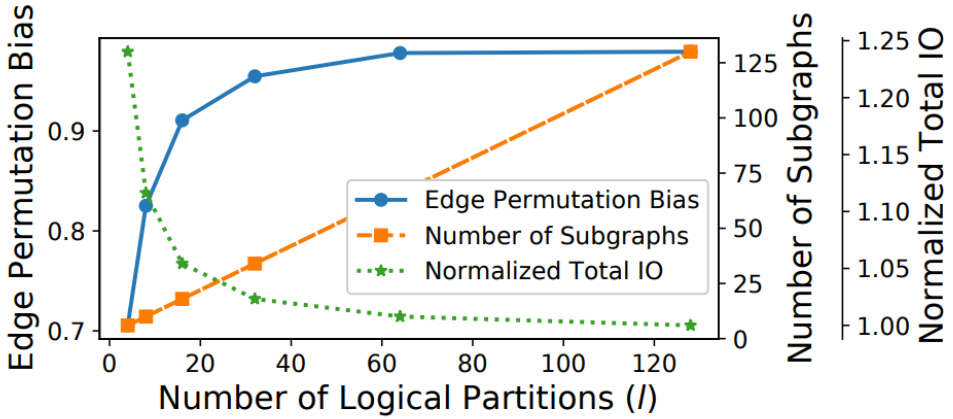
- Three parameters affecting memory size:
  - *p*, number of physical partitions
  - *l*, number of logical partitions
  - *c*, buffer capacity
- Edge Permutation Bias measures training on correlated examples (bad)
  - Decreased by increasing *p* and decreasing *l*
- Training time is dominated by I/O on logical sets (with prefetch)
  - Linear in *l*; linear in *p* only when *p* is very large
- Always want to maximize *c* given hardware constraints

Benchmarking

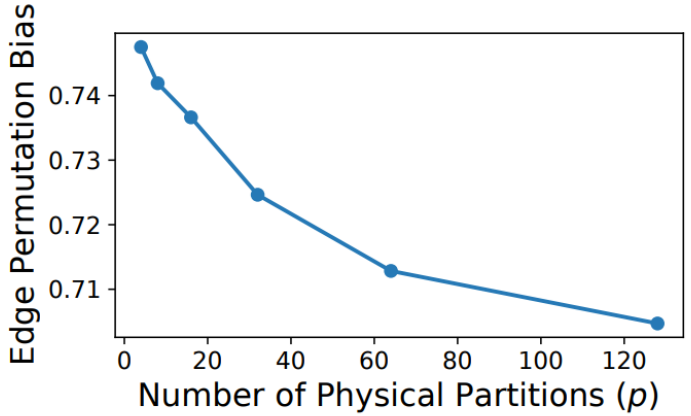
Effect of *p* and *l*



(a) Model Accuracy vs. Bias *B*



(b) Effect of logical partitions

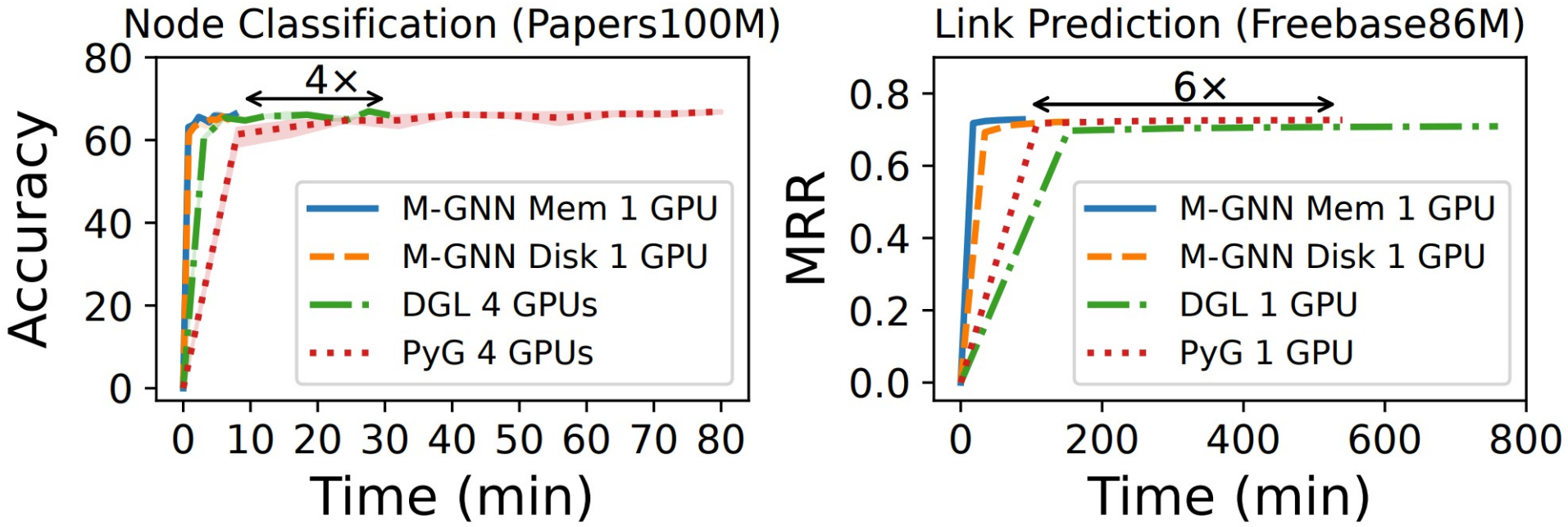


(c) Effect of physical partitions

Choice of parameters

- Set *p* such that each partition is the size of one disk read
  - Minimize bias without affecting time
- Set *c* to be as large as possible leaving space for working memory
- Set *l = 2p/c*, which is the minimizer subject to constraints
  - Minimizing l minimizes both time and bias
  - Constraints imposed by COMET:
    - Logical partitions in buffer c_l is at least 2
    - $p/c = l/c_l$

Time-to-accuracy comparison



- DENSE (shared sampling) gives a big speedup with a small cost in accuracy (due to correlated samples within each minibatch)
- These graphs fit in memory. Training with the graph stored on disk takes similar time but requires a cheaper machine

Table version

Table 3. MariusGNN, DGL, and PyG for node classification on large-scale graphs using a GraphSage GNN. Using a single GPU, MariusGNN can reach the same level of accuracy as multi-GPU baselines 3-8× faster and up to 64× cheaper.

	Epoch (min.)		Accuracy		Cost ($/epoch)	
Dataset	Papers	Mag	Papers	Mag	Papers	Mag
M-GNN$_{Mem}$	**0.77**	2.57	66.38	63.17	0.16	1.05
M-GNN$_{Disk}$	0.83	**0.94**	66.03	62.53	**0.04**	**0.05**
DGL	3.07	7.83	66.98	63.73	0.63	3.19
PyG	8.01	19	66.93	63.47	1.63	7.75

Sampling breakdown

	CPU Sampling Time (ms)				
#Layers	1	2	3	4	5
M-GNN	**1.4**	**18**	**103**	**401**	**1.8k**
DGL	5.7	28	376	5.4k	49k
PyG	2.2	59	1227	19k	96k

GPU Computation Time (ms)				
1	2	3	4	5
4	**6.1**	**21**	**153**	OOM
4.7	29	215	1231	OOM
3.2	13	168	OOM	OOM

Number of Nodes/Edges Sampled Per Mini Batch				
1	2	3	4	5
12k/13k	136k/181k	1M/2M	6M/17M	23M/91M
13k/20k	182k/278k	2M/4M	9M/37M	33M/222M
13k/20k	178k/258k	2M/4M	9M/32M	31M/174M

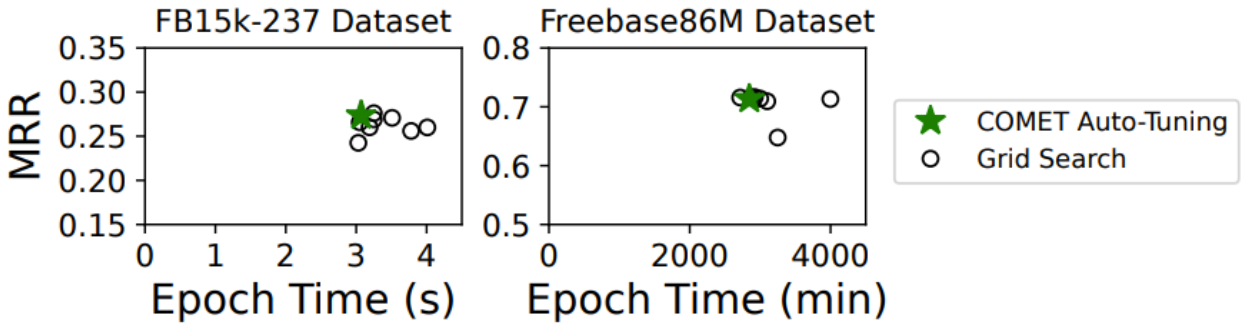
Some Larger Graphs and Models

	Epoch (min.)		MRR		Cost ($/epoch)	
Dataset	FB	Wiki	FB	Wiki	FB	Wiki
M-GNN$_{Mem}$	**17.5**	**46.6**	.7285	.4655	3.57	9.38
M-GNN$_{Disk}$	34.2	69.9	.7216	.4156	**1.74**	**3.56**
DGL	152	844	.7091	OOT	31.0	172
PyG	108	312	.7267	.4683	22.0	63.6

	Epoch (min.)		MRR		Cost ($/epoch)	
Model	GS	GAT	GS	GAT	GS	GAT
M-GNN$_{Mem}$	**17.5**	**52.6**	.7285	.7331	3.57	10.7
M-GNN$_{Disk}$	34.2	56.9	.7216	.7251	**1.74**	**2.90**
DGL	152	151	.7091	.6516	31.0	30.8
PyG	108	107	.7267	.7252	22.0	21.8

COMET vs BETA policies

Model	Graph	Mem MRR	Disk-Based MRR		Epoch (min.)	
			COMET	BETA	COMET	BETA
DM	237	.2533	**.2659**	.2431	**1.78**	1.95
DM	FB	.7249	**.7220**	.7189	**13.73**	17.51
DM	Wiki	.3941	**.4071**	.3951	**22.54**	27.75
GS	237	.2825	**.2736**	.2369	**3.07**	3.28
GS	FB	.7342	**.7123**	.6976	**47.45**	50.08
GS	Wiki	.4658	.4078	.4080	**76.66**	82.34
GAT	237	.2869	**.2341**	.2076	**3.51**	3.90
GAT	FB	.7418	**.7053**	.6860	**42.01**	46.02



Conclusions

- Can train certain GNNs really efficiently with only one GPU/machine
  - Also offers a straightforward time/memory tradeoff
- General principle: try to optimize before reaching for more compute
- Future work?
  - Replacement policies for other tasks, e.g. whole-graph classification
  - Use with multiple GPUs anyways