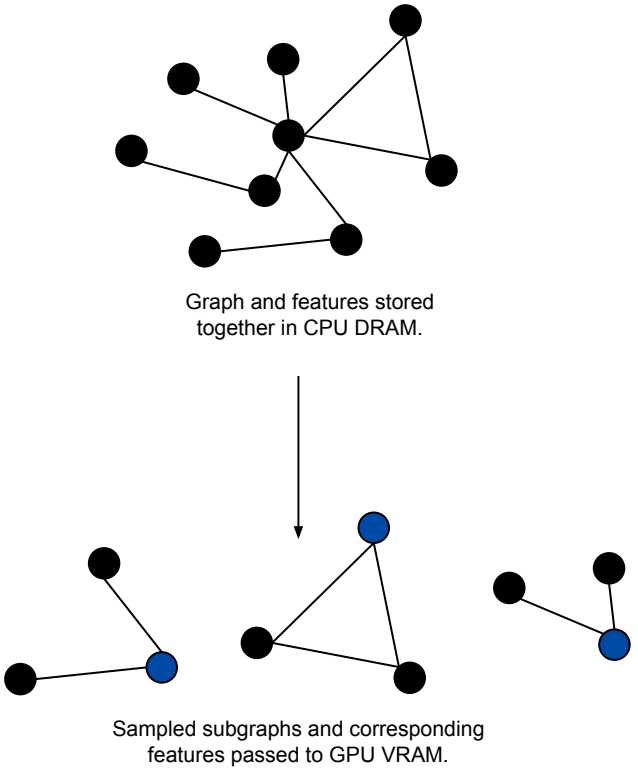
Scaling Up PyG

Manan Shah (manan@kumo.ai)
Dong Wang (dong@kumo.ai)

Graph Learning: An Architectural Blueprint

- Graph Neural Networks (GNNs) operate on a *graph* (nodes and edges) and *features* (tensors) for the elements in the graph
  - Message passing performs scatter/gather operations on feature tensors, between node space and edge space

- Scaling to data larger than GPU VRAM requires training on sampled subgraphs instead of the entire graph
  - Adds stochasticity, but reduces GPU memory requirements to those of the sampled subgraphs



Graph and features stored together in CPU DRAM.

Sampled subgraphs and corresponding features passed to GPU VRAM.

Graph Learning: An Architectural Blueprint

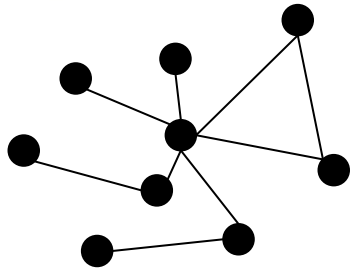
- Training on sampled subgraphs is great! But it's often not enough…
  - Acquiring instances with enough CPU DRAM to store a graph and features is hard
  - Data parallelism requires replicating the graph and features in each compute node
  - Graphs and features can easily be much larger than the memory of a single machine

- Scalability to very large graphs and features requires moving these data structures out-of-core and only processing *sampled subgraphs* on a compute node
  - Features are stored in a key-value feature store, supporting efficient random access
  - Graph information is stored in a graph store, supporting efficient sampling

Graph Learning: An Architectural Blueprint

- Separation of the feature and graph store allows for independent scale-out:
  - We are no longer constrained to single-node, in-memory datasets
  - The graph and feature stores can be independently partitioned, replicated, and stored in optimized formats

- Sampled subgraphs are now computed by *sampling* from the graph store and *joining* samples with features from the feature store
  - Each training node only requires enough memory to store sampled data



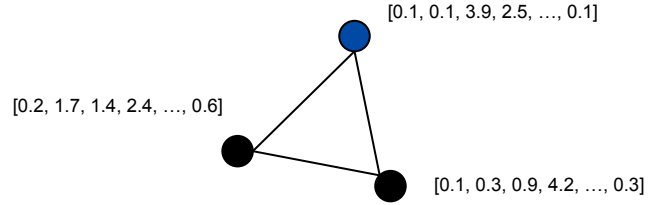
Graph Store: nodes and edges.

1: [0.1, 0.3, 0.9, 4.2, …, 0.3]
2: [0.2, 1.7, 1.4, 2.4, …, 0.6]
3: [0.1, 0.1, 3.9, 2.5, …, 0.1]
…
n: [0.4, 0.5, 0.2, 1.2, …, 0.1]

Feature store: node and edge tensors

Distributed Storage

Training Instance

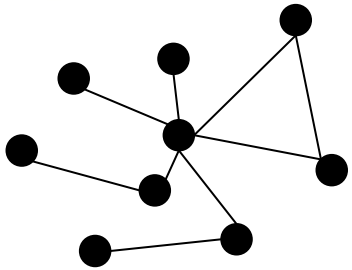


Sampled subgraph, joined with features; all that is necessary for GPU forward/backward.

Graph Learning: An Architectural Blueprint

- PyG 2.2 moves towards an architecture defining a *feature store* and *graph store*, unified behind a common interface

- This architecture is *extensible*, *generalizable*, and integrates seamlessly with the PyG you know and love



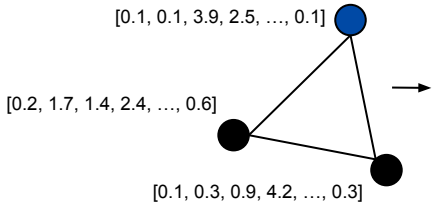
Graph Store: stores all `edge_index` tensors of a graph

```
1: [0.1, 0.3, 0.9, 4.2, …, 0.3]
2: [0.2, 1.7, 1.4, 2.4, …, 0.6]
3: [0.1, 0.1, 3.9, 2.5, …, 0.1]
                …
n: [0.4, 0.5, 0.2, 1.2, …, 0.1]
```

Feature store: stores all node and edge attributes (`x`, `y`, `edge_attr`, etc.)

Distributed Storage

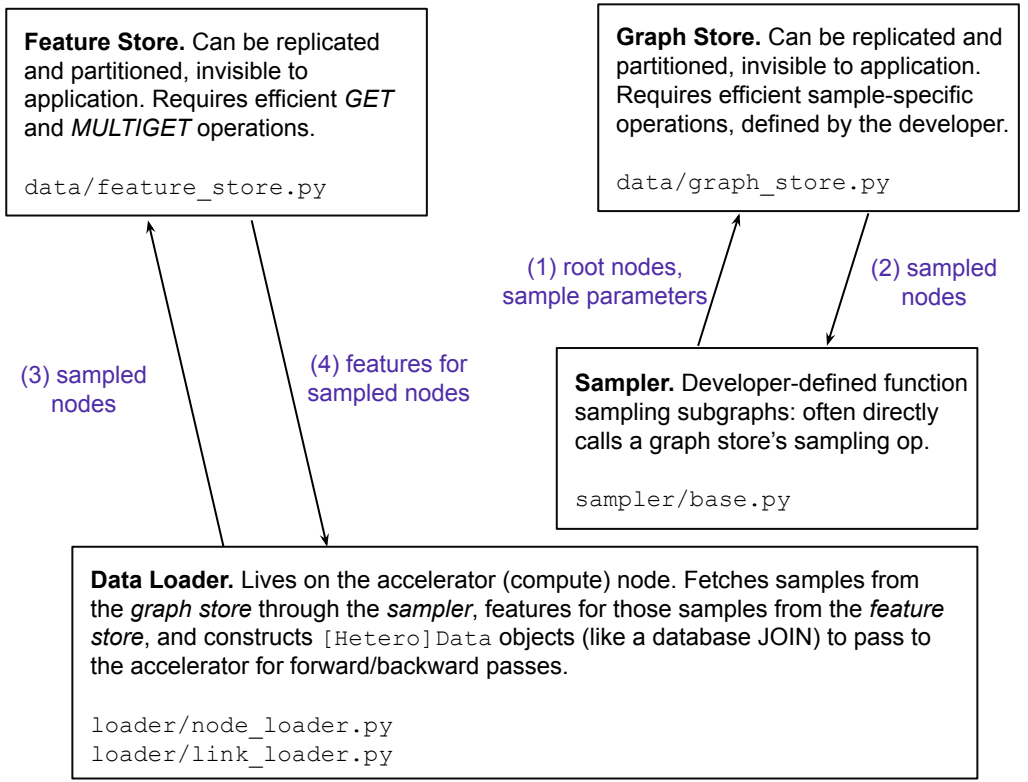
Training Instance



Sampled subgraph: a `Data` or `HeteroData` object containing sampled edges and features

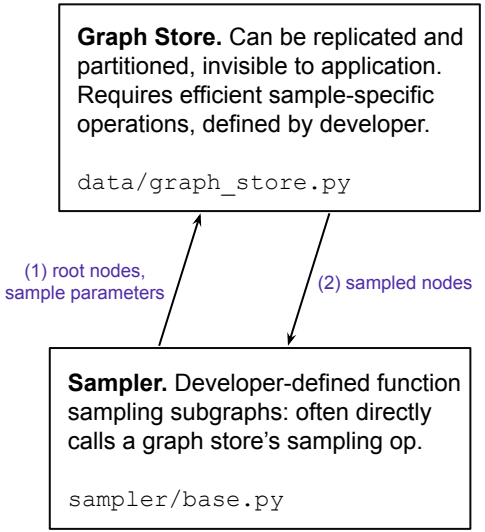
Graph Learning: An Architectural Blueprint

- There are four key abstractions that enable this effort: the feature store, graph store, sampler, and data loader
- The data loader orchestrates feature fetching and sampled subgraph generation by querying relevant stores



The Graph Store and Sampler

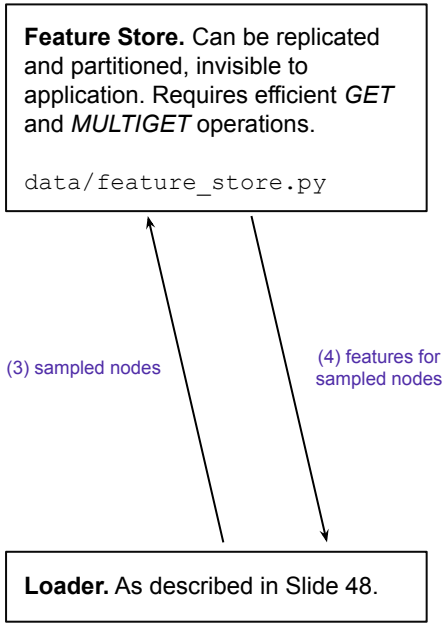
- A graph store stores graph edge indices in a manner conducive to efficient sampling
  - Requires GET, PUT, and sample-specific ops
- A graph sampler operates on a graph store to sample a subgraph from root nodes and related parameters
  - Implements logic to sample from a graph store (e.g. calling sample-specific ops).
  - We do not define a DSL
- The physical location, replication, partitioning, and other implementation details of the store are abstracted from PyG.
  - Ex: METIS-partitioned graphs, stores with pre-generated sampled subgraphs, etc.



The Feature Store

- A feature store stores features for the nodes and edges of a graph
  - Feature storage is often a primary storage bottleneck in graph learning applications, as storing a graph edge index is relatively cheap
  - Requires PUT and *efficient random access* GET

- Like the graph store, implementation details of the feature store are abstracted from PyG through a CRUD-like interface



The Data Loader

- A PyG data loader is built upon a sampler, graph store, and feature store. It:
  - Generates a batch of input nodes and requests sampled subgraphs from the sampler, and in turn the graph store
  - Collates sampled nodes and requests associated node and edge features from the feature store
  - Constructs PyG-native data objects from these data to perform forward/backward and weight updates

- The PyG data loader *handles all interactions between PyG and remote backends*.
  - This includes multi-process data loading, prefetching within each process, and communication with the feature and graph stores.
  - PyG end users only need to care about the data loader interface; once a feature store, graph store, and sampler are specified, PyG behaves just as it would on a single node!

Putting it all together in PyG

```python
feature_store = MyFeatureStore()  # or HeteroData, if in-memory

paper_features = ... # [num_papers, num_features_paper]
author_features = ... # [num_authors, num_features_author]

# Add the features, specifying node type and attribute name. This
# calls MyFeatureStore.put_tensor(...):
feature_store['paper', 'x', None] = paper_features
feature_store['author', 'x', None] = author_features
```

- Adding features to a custom feature store is easy: just define the store, and let PyG handle the syntactic sugar
- Internally, `put_tensor` may make an RPC call to store data remotely, invisible to the user

Putting it all together in PyG

```python
graph_store = MyGraphStore()  # or HeteroData, if in-memory

paper_to_author = ...  # [2, num_edges_writes]
paper_to_paper = ...  # [2, num_edges_cites]

graph_store.put_edge_index(
    paper_to_author,
    edge_type=('author', 'writes', 'paper'),
    layout='coo'
)

graph_store.put_edge_index(
    paper_to_paper,
    edge_type=('paper', 'cites', 'paper'),
    layout='coo'
)
```

- Adding edges is just as simple: specify the edge tensor, type, and layout, and the custom graph store's implementation details are abstracted away.

Putting it all together in PyG

```
feature_store = ... # as previous
graph_store = ... # as previous

# MyGraphSampler knows how to sample on MyGraphStore:
graph_sampler = MyGraphSampler(num_neighbors=[10, 20])

loader = NodeLoader(
    data=(feature_store, graph_store),
    node_sampler=graph_sampler,
    batch_size=20,
    input_nodes='paper',
)

# Train:
for batch in loader:
    ...
```

- Training brings it all together: `NodeLoader` manages sampling (with a custom sampler), communication with the feature and graph store, and other implementation details

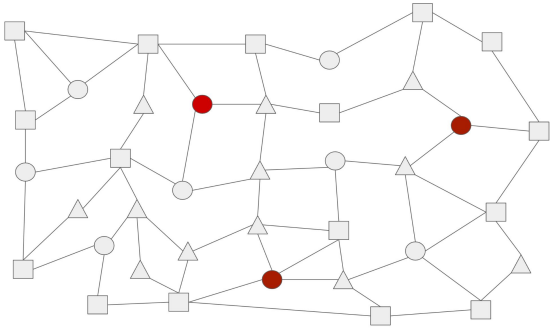
Putting it all together in PyG

- PyG's remote backend abstractions make working with a distributed feature and graph store no different from single-node modeling:
  - It's easy to store a graph and features out-of-core, and to switch between in-memory and remote representations
  - These abstractions support a wide variety of extensions to PyG (e.g. ahead-of-time sampling), and enable other axes of parallelism (e.g. multi-node DDP without data replication)
  - Stay tuned: all these features will be publicly launched in PyG 2.2!

- However: performance is critical! Be sure to monitor throughputs of feature fetching, sampling, and forward/backward when looking for bottlenecks

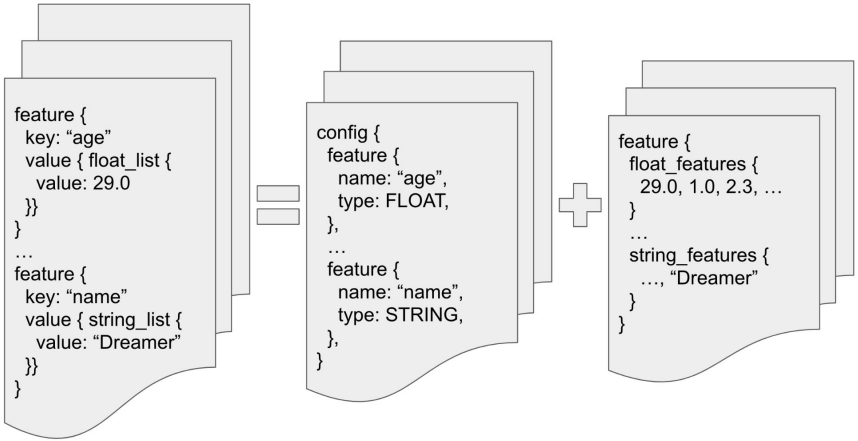
Building a Feature and Graph Store at Scale

- In Memory Graph Store + On Disk Feature Store
  - Fast access
  - Random lookup
  - High data volume
  - High throughput



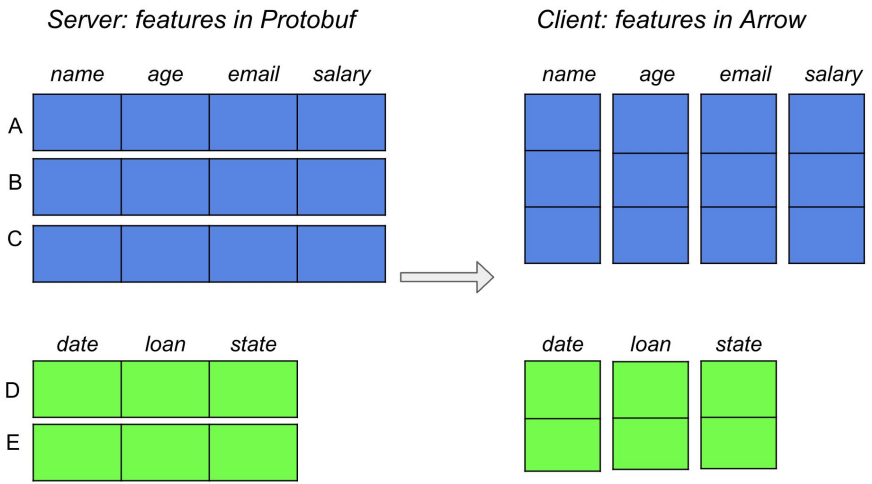
Building a Feature and Graph Store at Scale

- Optimize Feature Store
  - Features are stored in protobuf.
  - Shared feature configuration.
  - Compact feature representation.



Building a Feature and Graph Store at Scale

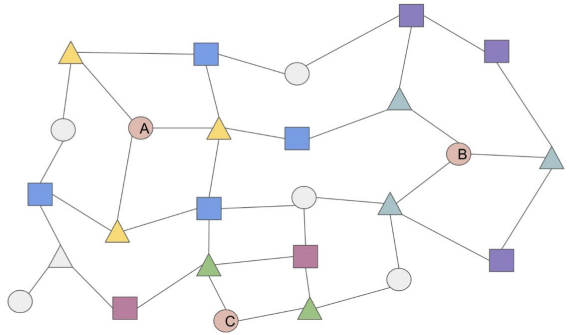
- Store as a Service
  - Communicate via gRPC.
  - Protobuf to Arrow for columnize access and feature encoding.
  - No N/A values for zero-copy to tensors.
  - C++ implementation.



Building a Feature and Graph Store at Scale

- Improve Feature Locality

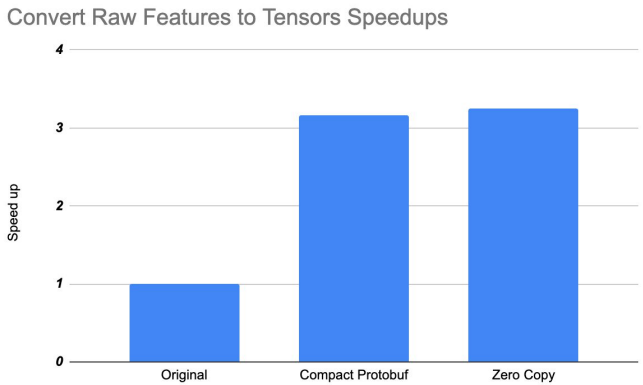
Features are re-ordered based on neighbors.

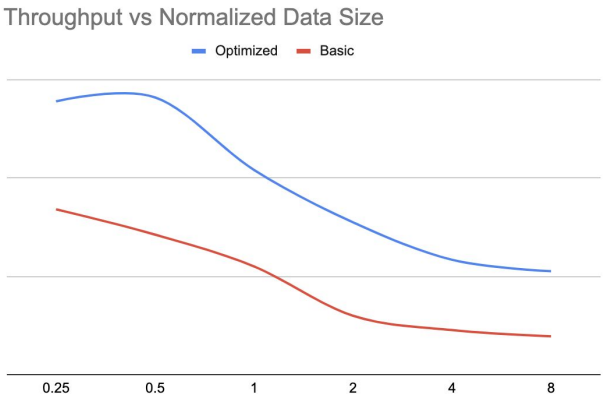


Order in the feature store.

Building a Feature and Graph Store at Scale

- Result





Thank you for listening!