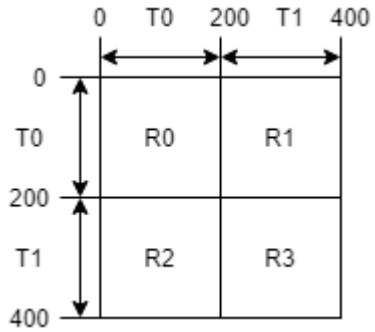
7.5 Heterogeneous Graph Under The Hood

The chapter covers the implementation details of distributed heterogeneous graph. They are transparent to users in most scenarios but could be useful for advanced customization.

In DGL, a node or edge in a heterogeneous graph has a unique ID in its own node type or edge type. Therefore, DGL can identify a node or an edge with a tuple: `(node/edge type, type-wise ID)`. We call IDs of such form as **heterogeneous IDs**. To patition a heterogeneous graph for distributed training, DGL converts it to a homogeneous graph so that we can reuse the partitioning algorithms designed for homogeneous graphs. Each node/edge is thus uniquely mapped to an integer ID in a consecutive ID range (e.g., from 0 to the total number of nodes of all types). We call the IDs after conversion as **homogeneous IDs**.

Below is an illustration of the ID conversion process. Here, the graph has two types of nodes ($T0$ and $T1$), and four types of edges ($R0$, $R1$, $R2$, $R3$). There are a total of 400 nodes in the graph and each type has 200 nodes. Nodes of $T0$ have IDs in [0,200), while nodes of $T1$ have IDs in [200, 400). In this example, if we use a tuple to identify the nodes, nodes of $T0$ are identified as (T0, type-wise ID), where type-wise ID falls in [0, 200); nodes of $T1$ are identified as (T1, type-wise ID), where type-wise ID also falls in [0, 200).



ID Conversion Utilities

During Preprocessing

The steps of Parallel Processing Pipeline all use heterogeneous IDs for their inputs and outputs. Nevertheless, some steps such as ParMETIS partitioning are easier to be implemented using homogeneous IDs, thus requiring a utility to perform ID conversion. The code below implements a simple `IDConverter` using the metadata information in the metadata

JSON from the chunked graph data format. It starts from some node type A as node type 0, then assigns all its nodes with IDs in range $[0, |V_A| - 1)$. It then moves to the next node type B as node type 1 and assigns all its nodes with IDs in range $[|V_A|, |V_A| + |V_B| - 1)$.

```python
from bisect import bisect_left
import numpy as np

class IDConverter:
    def __init__(self, meta):
        # meta is the JSON object loaded from metadata.json
        self.node_type = meta['node_type']
        self.edge_type = meta['edge_type']
        self.ntype2id_map = {ntype : i for i, ntype in enumerate(self.node_type)}
        self.etype2id_map = {etype : i for i, etype in enumerate(self.edge_type)}
        self.num_nodes = [sum(ns) for ns in meta['num_nodes_per_chunk']]
        self.num_edges = [sum(ns) for ns in meta['num_edges_per_chunk']]
        self.nid_offset = np.cumsum([0] + self.num_nodes)
        self.eid_offset = np.cumsum([0] + self.num_edges)

    def ntype2id(self, ntype):
        """From node type name to node type ID"""
        return self.ntype2id_map[ntype]

    def etype2id(self, etype):
        """From edge type name to edge type ID"""
        return self.etype2id_map[etype]

    def id2ntype(self, id):
        """From node type ID to node type name"""
        return self.node_type[id]

    def id2etype(self, id):
        """From edge type ID to edge type name"""
        return self.edge_type[id]

    def nid_het2hom(self, ntype, id):
        """From heterogeneous node ID to homogeneous node ID"""
        tid = self.ntype2id(ntype)
        if id < 0 or id >= self.num_nodes[tid]:
            raise ValueError(f'Invalid node ID of type {ntype}. Must be within range [0,
{self.num_nodes[tid]})')
        return self.nid_offset[tid] + id

    def nid_hom2het(self, id):
        """From heterogeneous node ID to homogeneous node ID"""
        if id < 0 or id >= self.nid_offset[-1]:
            raise ValueError(f'Invalid homogeneous node ID. Must be within range [0,
self.nid_offset[-1])')
        tid = bisect_left(self.nid_offset, id) - 1
        # Return a pair (node_type, type_wise_id)
        return self.id2ntype(tid), id - self.nid_offset[tid]

    def eid_het2hom(self, etype, id):
        """From heterogeneous edge ID to homogeneous edge ID"""
        tid = self.etype2id(etype)
        if id < 0 or id >= self.num_edges[tid]:
            raise ValueError(f'Invalid edge ID of type {etype}. Must be within range [0,
{self.num_edges[tid]})')
        return self.eid_offset[tid] + id

    def eid_hom2het(self, id):
        """From heterogeneous edge ID to homogeneous edge ID"""
        if id < 0 or id >= self.eid_offset[-1]:
            raise ValueError(f'Invalid homogeneous edge ID. Must be within range [0,
self.eid_offset[-1])')
        tid = bisect_left(self.eid_offset, id) - 1
```

```
        # Return a pair (edge_type, type_wise_id)
        return self.id2etype(tid), id - self.eid_offset[tid]
```

After Partition Loading

After the partitions are loaded into trainer or server processes, the loaded `GraphPartitionBook` provides utilities for conversion between homogeneous IDs and heterogeneous IDs.

- `map_to_per_ntype()`: convert a homogeneous node ID to type-wise ID and node type ID.
- `map_to_per_etype()`: convert a homogeneous edge ID to type-wise ID and edge type ID.
- `map_to_homo_nid()`: convert type-wise ID and node type to a homogeneous node ID.
- `map_to_homo_eid()`: convert type-wise ID and edge type to a homogeneous edge ID.

Because all DGL's low-level distributed graph sampling operators use homogeneous IDs, DGL internally converts the heterogeneous IDs specified by users to homogeneous IDs before invoking sampling operators. Below shows an example of sampling a subgraph by `sample_neighbors()` from nodes of type `"paper"`. It first performs ID conversion, and after getting the sampled subgraph, converts the homogeneous node/edge IDs back to heterogeneous ones.

```
gpb = g.get_partition_book()
# We need to map the type-wise node IDs to homogeneous IDs.
cur = gpb.map_to_homo_nid(seeds, 'paper')
# For a heterogeneous input graph, the returned frontier is stored in
# the homogeneous graph format.
frontier = dgl.distributed.sample_neighbors(g, cur, fanout, replace=False)
block = dgl.to_block(frontier, cur)
cur = block.srcdata[dgl.NID]

block.edata[dgl.EID] = frontier.edata[dgl.EID]
# Map the homogeneous edge Ids to their edge type.
block.edata[dgl.ETYPE], block.edata[dgl.EID] = gpb.map_to_per_etype(block.edata[dgl.EID])
# Map the homogeneous node Ids to their node types and per-type Ids.
block.srcdata[dgl.NTYPE], block.srcdata[dgl.NID] = gpb.map_to_per_ntype(block.srcdata[dgl.NID])
block.dstdata[dgl.NTYPE], block.dstdata[dgl.NID] = gpb.map_to_per_ntype(block.dstdata[dgl.NID])
```

Note that getting node/edge types from type IDs is simple – just getting them from the `ntypes` attributes of a `DistGraph`, i.e., `g.ntypes[node_type_id]`.

Access distributed graph data

The `DistGraph` class supports similar interface as `DGLGraph`. Below shows an example of getting the feature data of nodes 0, 10, 20 of type $T0$. When accessing data in `DistGraph`, a user needs to use type-wise IDs and corresponding node types or edge types.

```
import dgl
g = dgl.distributed.DistGraph('graph_name', part_config='data/graph_name.json')
feat = g.nodes['T0'].data['feat'][[0, 10, 20]]
```

A user can create distributed tensors and distributed embeddings for a particular node type or edge type. Distributed tensors and embeddings are split and stored in multiple machines. To create one, a user needs to specify how it is partitioned with `PartitionPolicy`. By default, DGL chooses the right partition policy based on the size of the first dimension. However, if multiple node types or edge types have the same number of nodes or edges, DGL cannot determine the partition policy automatically. A user needs to explicitly specify the partition policy. Below shows an example of creating a distributed tensor for node type $T0$ by using the partition policy for $T0$ and store it as node data of $T0$.

```
g.nodes['T0'].data['feat1'] = dgl.distributed.DistTensor(
    (g.num_nodes('T0'), 1), th.float32, 'feat1',
    part_policy=g.get_node_partition_policy('T0'))
```

The partition policies used for creating distributed tensors and embeddings are initialized when a heterogeneous graph is loaded into the graph server. A user cannot create a new partition policy at runtime. Therefore, a user can only create distributed tensors or embeddings for a node type or edge type. Accessing distributed tensors and embeddings also requires type-wise IDs.