

项目申请书

项目名称：基于 NebulaGraph 的 PyG 远程后端实现

项目主导师：wey-gu

申请人：蔡锋泽

日期：2025.05.26

邮箱：caifengze@hdu.edu.cn, fengzcw@gmail.com

1. 项目背景

- 1.1 项目基本需求
- 1.2 项目相关仓库
- 1.3 快速开始（项目预期成果）

2. 技术方法及可行性

- 2.1 参考项目：KUZU
- 2.2 对比与差异

3. 项目实施细节

- 3.1 仓库结构与开发环境
- 3.2 Connection 扩展
- 3.3 NebulaFeatureStore 实现
- 3.4 NebulaGraphStore 实现
- 3.5 数据导入与 VID 管理

4. 项目计划与时间表

- 第一阶段（2025-07-01 ~ 2025-08-15）
- 第二阶段（2025-08-16 ~ 2025-09-30）
- 期望

1. 项目背景

图神经网络（GNN）已成为处理复杂图结构数据的核心工具，但随着节点与边规模的指数级增长，单机内存已难以容纳完整图。PyTorch Geometric（PyG）自 2.2 版起引入

`FeatureStore / GraphStore` 远程抽象，可以将图结构与特征托管在外部系统。

NebulaGraph 是一款开源、分布式、水平可扩展的原生图数据库，能够支撑千亿级顶点、万亿级边，并在毫秒级完成查询。若能将 NebulaGraph 作为 PyG 的“remote server”，即可在保持 PyG 友好 API 的同时，实现超大规模 GNN 的高效训练与推理。因此项目旨在：

- 实现 NebulaGraph-PyG 远程适配器（`NebulaFeatureStore` & `NebulaGraphStore`）。
- 输出端到端示例、性能基准、中文/英文教程文档，并推动该适配器进入官方生态。

1.1 项目基本需求

设计实现 PyG 的 NebulaGraph remote server，作为其 `FeatureStore` 与 `GraphStore`。

1.2 项目相关仓库

- nebula-python: <https://github.com/vesoft-inc/nebula-python>
- PyG: https://github.com/pyg-team/pytorch_geometric

- PyG远程示例: <https://pytorch-geometric.readthedocs.io/en/latest/advanced/remote.html>
- 示例数据库: <https://ogb.stanford.edu>

1.3 快速开始 (项目预期成果)

用户只需遵循下面 3 步, 就能在 PyTorch Geometric 中无缝使用 NebulaGraph Remote 后端:

1. 安装包

```
pip install nebula-pyg
```

2. 初始化连接与适配器

```
from nebula_pyg import NebulaConnection, NebulaFeatureStore,
NebulaGraphStore

# 1) 建立 Connection
conn = NebulaConnection(
    hosts=[("127.0.0.1", 9669)],
    user="root", password="nebula",
    space="ogbn_products",
)

# 2) 拿到 FeatureStore 和 GraphStore
feat_store, graph_store = conn.get_torch_geometric_remote_backend()
```

3. 正常训练过程

```
# 构造 PyG Data
data = graph_store.as_data(
    num_nodes=...,
    node_attrs={"x": ("product", None, feat_store)},
    edge_attrs={"product", "buys", "product": graph_store}
)

from torch_geometric.loader import NeighborLoader
loader = NeighborLoader(data, num_neighbors=[15,10], batch_size=1024)

for batch in loader:
    # 训练循环完全跟本地内存版一致
    out = model(batch.x, batch.edge_index)
    ...
```

2. 技术方法及可行性

本项目将设计并实现 PyG 的 `FeatureStore` 与 `GraphStore` 远程适配器模块, 分别用于从 NebulaGraph 中加载节点特征与边索引, 以支持 GNN 的训练与采样操作。

2.1 参考项目: KUZU

KuzuDB 是一款专为图分析优化的轻量级嵌入式数据库, 其官方 PyG 后端实现 [KuzuFeatureStore / KuzuGraphStore](#) 结构清晰, 通过以下模块实现了与 PyG 的无缝对接:

- `torch_geometric_feature_store.py` 中的 `KuzuFeatureStore`

- 在 `__get_tensor_by_scan()` 方法中，调用 C++ 接口 `_scan_node_table` 绕过查询引擎，直接读取本地嵌入式数据库的节点属性，并将返回的 NumPy 数组转为 `torch.Tensor`；
- 支持整数索引、切片和列表索引等多种访问模式，并在首次访问时缓存属性元信息。
- `torch_geometric_graph_store.py` 中的 `KuzuGraphStore`
 - 在 `__get_edge_coo_from_database()` 中，通过 C++ 接口 `get_all_edges_for_torch_geometric` 批量拉取所有边的 `(src, dst)` 对，并输出为 PyG 所需的 COO 格式张量；
 - 在初始化时预先读取所有关系表名称，构建边属性元信息表。
- `torch_geometric_result_converter.py` 中的 `TorchGeometricResultConverter`
 - 将通用的 `QueryResult`（节点+边）按 PyG 的 `Data/HeteroData` 数据结构组装，处理属性类型过滤、位置映射等逻辑；
 - 支持多表异构图的自动划分与转换。

Kuzu 的核心优势在于：

1. 本地嵌入式部署、零网络开销；
2. 通过 PyBind11 暴露 C++ 扫描接口精准高效地拉取列向数据；（这个和未来优化PyO3的思路类似）
3. 入口统一：在 `Database.get_torch_geometric_remote_backend()` 中一次性返回 `(FeatureStore, GraphStore)`，用户只需将其传入 `NeighborLoader` 即可完成远程训练。

2.2 对比与差异

对比维度	KuzuDB	本项目（NebulaGraph-PyG 远程后端）
部署方式	嵌入式单机库	分布式服务（graphd + storaged）
数据扫描	C++ 内存映射 + 扫描	Thrift/GRPC RPC + <code>GraphStorageClient</code>
网络开销	无	存在网络延迟
ID 体系	连续偏移量（offset）	多分区 VID，需要预处理/映射
并发模型	本地多线程	多连接池 + 异步 RPC 并发
依赖体积	< 10 MB（仅 C++ 库）	<code>nebula3-python</code> (~ 1 MB) + <code>torch/pyg</code> (~ > 2 GB)

基于以上差异，本项目的核心工作包括：

1. Connection 扩展：在 `nebula-python` 的 `Connection` 类中，除原有的 graphd Thrift 客户端外，再初始化一个 `GraphStorageClient`，并暴露 `storage()` 方法，供 PyG 适配层直接调用。预期情况下：

```
# 上层只需：
storage = conn.storage()
buf = storage.scan_vertex(tag, prop, ids, dtype="float32", num_threads=4)
tensor = torch.frombuffer(buf, dtype=torch.float32).view(len(ids), dim)
```

2. 批量扫描 API：为 `nebula-python` 增加 `scan_vertex_async(tag, prop, ids, ...)` 与 `scan_edge_async(edge_type, ...)`，实现多线程/异步 RPC 下的高吞吐列扫描。

3. PyG 适配层：实现 `NebulaFeatureStore.get_tensor()` 与 `NebulaGraphStore.get_edge_index()`，分别调用底层批量扫描 API，并把返回的二进制缓冲区零拷贝转换成 `torch.Tensor`。
4. VID 映射：设计或借助导入工具确保图中 VID 连续 / 可控，或在适配层对外部 VID 做一次映射，使其符合 PyG 的从 0 开始、连续索引要求。

3. 项目实施细节

本章详细说明各功能模块的设计思路、关键接口、数据流程。

3.1 仓库结构与开发环境

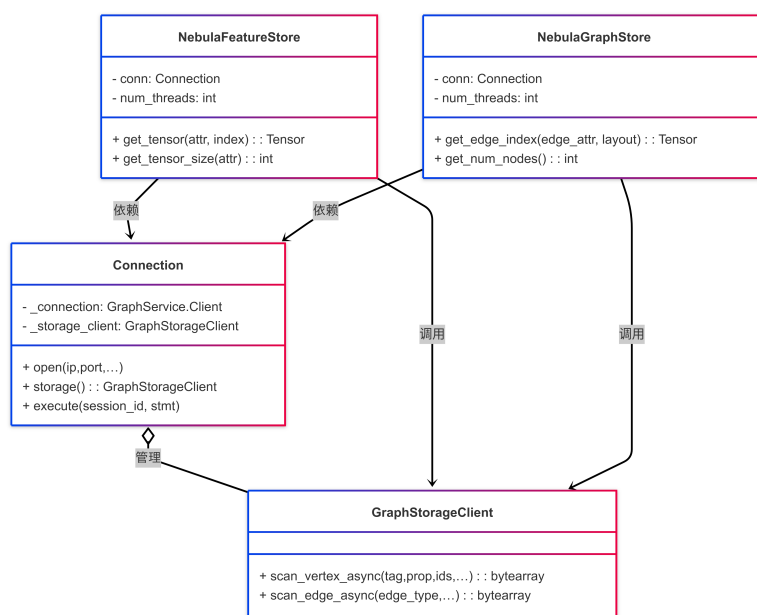
- 代码仓库

- `nebula-pyg/`
 - `nebula_pyg/connection.py`：扩展 `Connection` 实现 `storage()` 与批量扫描接口
 - `nebula_pyg/feature_store.py`： `NebulaFeatureStore` 实现 `FeatureStore` 接口
 - `nebula_pyg/graph_store.py`： `NebulaGraphStore` 实现 `GraphStore` 接口
 - `nebula_pyg/result_converter.py`： `Result` → `PyG Data` 转换器（KUZU中实现）
 - `examples/train_ogbn_products.py`：端到端示例脚本
 - `tests/`：单元测试代码
 - `docker-compose.yml`：一键起 `NebulaGraph` 集群 + 运行示例

- 开发环境

- Python ≥ 3.6.2（与 `nebula-python` 最低兼容版本一致）
- 依赖：`nebula3-python` ≥ 3.8.3、`torch` ≥ 2.2、`torch-geometric` ≥ 2.5、`numpy`、`pytest` 等
- 包管理：pdm

- 类图说明



总体设计如上图所示（方法只做简单示例，图目的为了展示类关系），其中包含三个类，`NebulaFeatureStore`、`NebulaGraphStore` 和 `GraphStorageClient`，其中 `GraphStorageClient` 被聚合在 `Connection` 内，并由其管理生命周期，在 `Connection` 的创建时产生，在 `Connection` 关闭时回收，主要包括了 `scan_vertex_async(tag, prop, ids, ...)` 与 `scan_edge_async(edge_type, ...)` 方法。`NebulaFeatureStore` 和 `NebulaGraphStore` 则是依赖于 `Connection`，并能够直接调用 `GraphStorageClient` 中的方法。

3.2 Connection 扩展

- 在原生 `nebula3.gclient.net.Connection` 基础上：
 - `open()` / `_reopen()` 时，初始化并保存 `GraphStorageClient`
 - 新增 `def storage(self) -> GraphStorageClient`，直接返回已配置好的客户端，这个 `GraphStorageClient` 目的是直接与 `storage` 进行通信，绕过 `graphd` 层，来执行 `scan_vertex` 和 `scan_edge` 等高吞吐工作。
- `get_torch_geometric_remote_backend()` 方法

大致实现如下：

```
def get_torch_geometric_remote_backend(
    self, num_threads: int | None = None
) -> tuple[NebulaFeatureStore, NebulaGraphStore]:

    if self._conn is None:
        self.open()

    feat_store = NebulaFeatureStore(self, num_threads)
    graph_store = NebulaGraphStore(self, num_threads)
    return feat_store, graph_store
```

只传入一个可选的 `num_threads`，用于 `StorageClient` 并发扫描。

3.3 NebulaFeatureStore 实现

- 继承： `torch_geometric.data.FeatureStore`
- 核心方法：
 - `get_tensor(self, attr: TensorAttr, index: Optional[Sequence[int]] = None) -> Tensor`
 - 可以先从 `TensorAttr` 对象中解析出对应的 `NebulaTag` (`attr.group_name`) 和对应属性列 (`attr.attr_name`)。其中，如果出现 `index is None` 的情况，则可以先从 `get_tensor_size(attr)` 拿到节点总数 `N`，将 `ids = list(range(N))`
 - 通过 `self.conn.storage().scan_vertex_async(tag, prop, ids, ...)` 拉取原始 buffer，此处可以先查一次属性的列类型，计算每行所需占用的字节数，并且要注意 `scan_vertex_async` 中的数据类型一致性
 - 在得到 buffer 以后，`torch.frombuffer(buf, dtype)` 零拷贝创建一个 1D Tensor，再做 `view(len(ids), feat_dim)`。
 - `get_tensor_size(self, attr: TensorAttr) -> int`：可以先执行一次获取总量，并将结果缓存到内存中，在后续查询中可以直接调用。
- 设计细节
 - 能够支持 `index=None` 的全量拉去，或者根据列表机型索引索引。
 - 当拉数量过于庞大的时候，可以做一个分批处理，避免 RPC 的超时或者数据包过大。
 - 由于 NebulaGraph 的节点属性可以有很多类型，对于非数值型的特征 PyG 并不能直接使用，所以需要做一个 embedding 的过程，以 vector 的形式呈现（这个类型应该是在做，我们用 string 暂时性的替代）。
 - LRU 缓存 `vid->tensor slice`，这个是做一个缓存层，对可能出现的需要反复重复方位的节点做一个缓存，减少重复 RPC。

5. 错误重试机制，可以写一个重传机制，比如遇到超时，存储节点异常等问题，使用指数退避式对具体的批次（可能是子批次）重试

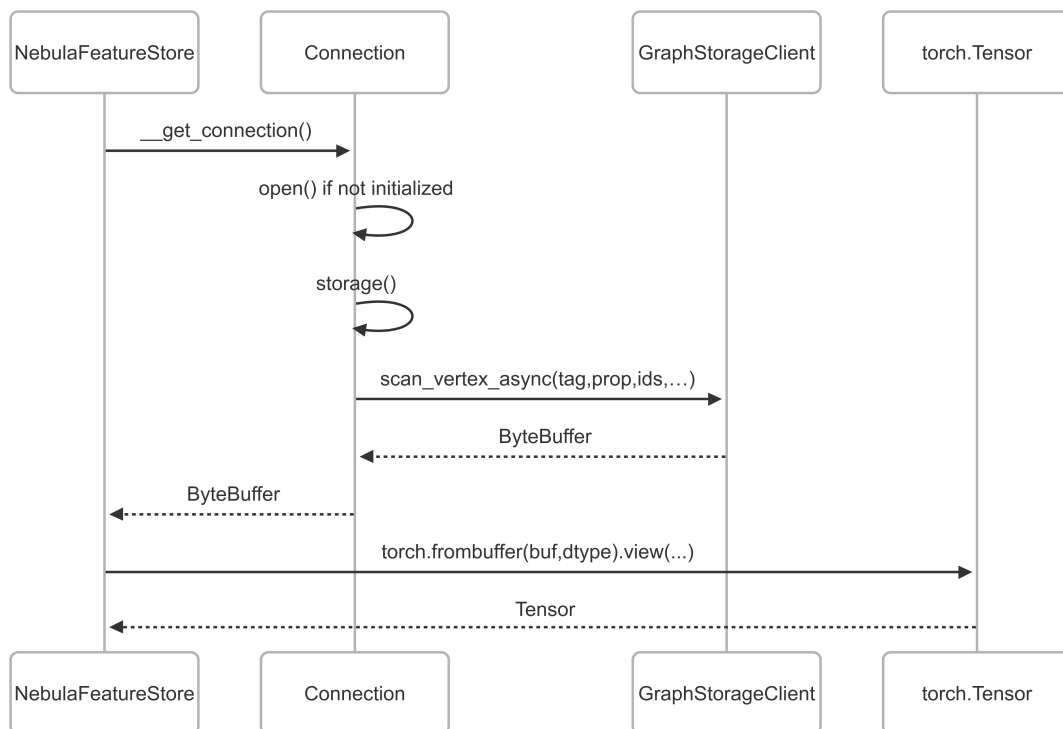
3.4 NebulaGraphStore 实现

- **继承：** `torch_geometric.data.GraphStore`
- **核心方法：**
 - `get_edge_index(self, *, edge_attr: EdgeAttr, layout, is_sorted=False) -> Tensor`
 1. 首先要解析单条的关系类型，即 `edge_type`，格式形如 `((src_tag, relation, dst_tag))`。将其作为参数调用 `scan_edge_async(edge_type, ...)` 批量边数据。
 2. 得到数据后，可以使用 `numpy.frombuffer(..., dtype=np.int64).reshape(-1, 2)` 的方法转为 `(num_edges, 2)` 的 NumPy 数组，之后通过构造两个一维的 tensor 拼接成 `[2, num_edges]` 的形式，满足 PyG 要求。
 - `get_num_nodes(self) -> int`：可复用 `FeatureStore.get_tensor_size`，并且实现缓存，仅需一次调用。
- **设计细节**
 - 本类 `NebulaGraphStore` 针对一个边类型（`edge_type`），执行一次全量扫描，如果是一个图存在多种关系的话。则是可以在上层通过多个示例传入不同的 `edge_type`，再在 PyG 构造 `HeteroData` 中将 `edge_index` 合并。
 - 对于大图，可能一次拉取的边过多，也会类似上面所提到的 RPC 超时或数据包过大的问题。可以在 `scan_edge_async` 进行拆分，然后进行并发调用，最后再进行重新拼接。
 - 是不是可以针对无向训练提供一个额外的参数 `directed: bool` 进行控制。
 - 重传机制和 `FeatureStore` 中所提到的类似实现即可。

3.5 数据导入与 VID 管理

- **OGB 数据导入工具**
 - 脚本 `nebula_pyg/cli/import_ogb.py`：
 1. 读取 OGBN-Products 边列表及特征文件
 2. 生成连续 VID 映射表
 3. 用 `nebula3-python` 批量写入 `INSERT VERTEX` / `INSERT EDGE`
- **VID 映射策略**
 - 在导入阶段用新编号做 VID，即让数据进入 Nebula 之前，就完成了将原始节点映射到 0 到 N-1。
 - 如果没有，该项目进行一定的自动化处理，运行时扫描所有 VID，生成 `vid_to_idx` 映射字典（用一个 KV 数据库持久化），供 `FeatureStore` / `GraphStore` 使用。

基本实现如上，具体关系可见如下时序图：



4. 项目计划与时间表

第一阶段（2025-07-01 ~ 2025-08-15）

目标：完成核心接口设计与骨架实现，并在 `nebula-python` 上提交批量扫描 PR。

☐ 环境搭建与 PoC

- ☐ 部署本地 NebulaGraph 集群（`nebula-up` / Docker Compose）
- ☐ 安装并验证 `nebula3-python`、`torch`、`torch-geometric` 环境
- ☐ 运行随机 Demo 脚本，验证接口调用链完整

☐ 接口设计与仓库骨架

- ☐ 明确定义 `Connection.storage()`、`scan_vertex_async` / `scan_edge_async` API（产出 `DESIGN.md`）
- ☐ 搭建 `nebula-pyg` 仓库结构、模块声明及接口签名
- ☐ 编写单元测试骨架，保证 CI（GitHub Actions + nebula 容器）能跑通

☐ NebulaFeatureStore 实现 & PR

- ☐ Fork `vesoft-inc/nebula-python`，开发并提交 `feature/async-scan` 分支 PR
- ☐ 完成 `NebulaFeatureStore.get_tensor()`：RPC → buffer → `torch.frombuffer`
- ☐ 单元测试覆盖全量与切片两种调用场景

☐ NebulaGraphStore 实现 & PR

- ☐ 在 `nebula-python` 上提交 `feature/async-edge` 分支 PR
- ☐ 完成 `NebulaGraphStore.get_edge_index()`：批量拉边 → COO 张量
- ☐ 单元测试覆盖边索引加载场景

第二阶段（2025-08-16 ~ 2025-09-30）

目标：完成端到端 Demo、性能基准、文档与社区贡献，发布 `nebula-pyg` 1.0。

☐ 端到端导入工具 & Demo

- ☐ 编写 `import_ogb.py`：OGBN-Products 边列表 + 连续 VID 映射 → NebulaGraph
- ☐ 完善 `examples/train_ogbn_products.py`：从数据导入到 GraphSAGE 训练全链路演示
- ☐ 跑通全流程并记录日志与模型输出
- ☐ 编写 `Dockerfile` 与 `docker-compose.yml`：
 - 一键部署本地 NebulaGraph + `nebula-pyg` 环境
 - 镜像中预装依赖、导入脚本、示例代码，使用户只需 `docker-compose up` 即可运行 Demo
- ☐ 撰写 Jupyter Notebook 示例：
 - 在 notebook 中展示全过程，从创建连接、导入数据、加载 `FeatureStore` / `GraphStore` 到执行简单训练

☐ 性能基准与优化

- ☐ 在 OGBN-Products、LiveJournal 数据集上测量吞吐 (edges/sec) 与延迟 (ms)
- ☐ 对比不同 `num_threads`、RPC 批大小、LRU 缓存策略
- ☐ 撰写性能对比报告，生成图表

☐ 文档完善与社区贡献

- ☐ 完善 `README.md`、用户手册、FAQ 文档
- ☐ 撰写 blog，介绍项目，便于更多用户了解使用
- ☐ 向 `pyg-team/pytorch_geometric` 提交 RFC，申请纳入 `torch_geometric.contrib`

☐ 收尾发布

- ☐ 根据导师与社区反馈迭代改进代码与文档
- ☐ 在 PyPI 发布 `nebula-pyg` v1.0
- ☐ 发布项目总结，以及可能有的演示视频

期望

其实对于开源社区很早就有了了解，主要来自好朋友 iyear，每次和他聊到技术相关的，也总想为社区做点什么，可能因为自己比较弱也很害怕，总是会被自己的能力劝退。近期，也就是升学这个空档期，每天逛逛社区，总能发现很有意思的东西，尝试着帮别人解决 issue。发现这个过程真的很开心，当然，也能发现自己很多的不足。因此，也想借助 OSPP 这个平台来提升自己，也恰好遇上了 nebula 这个数据库（没想到自己跑 PyG 整天爆内存唯一的途径就是加量，其实开源社区中肯定是有好的工具的，这个地方真的很神奇！）想着自己能帮助其他人解决我曾经遇到的困惑。