Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Seamlessly and Efficiently Integrating DuckDB with Graph Neural Network Libraries

Author: Yiming Wu (2702828)

1st supervisor:	Prof. dr. Peter A. Boncz
2nd reader:	Dr. Gábor Szárnyas (Centrum Wiskunde & Informatica)
daily supervisor:	Daniël ten Wolde (Centrum Wiskunde & Informatica)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 16, 2023

"Luck is what happens when preparation meets opportunity."
by Seneca

Abstract

DuckPGQ is an extension of the DuckDB relational database management system that introduces SQL/PGQ support. This support empowers the definition of graph views over relational tables, facilitating the execution of read-only queries on these views. Notably, DuckPGQ offers a suite of graph analysis capabilities, including graph pattern matching and shortest path finding.

Despite its capabilities, DuckPGQ currently lacks compatibility with graph neural network (GNN) methods for analyzing graphs stored within the DuckDB/DuckPGQ system. In response, this project focuses on establishing seamless and efficient integration between DuckDB/DuckPGQ and two GNN libraries: PyTorch Geometric and Deep Graph Library. The twin objectives of efficiency entail minimizing memory copies whenever feasible, while seamlessness aims to enable the execution of user-defined functions (UDFs) on Python objects akin to UDFs run on DuckDB tables. This integration is executed with minimal user effort.

To achieve this, a set of user-defined functions is developed, serving as a bridge to fetch a DuckPGQ graph as a GNN graph. This GNN graph becomes amenable to downstream training and inference with GNN models. Furthermore, a zero-copy mechanism is implemented to enable direct access to the compressed sparse row (CSR) arrays, pivotal to describing the connectivity structure of a DuckPGQ graph, from within the GNN libraries. An accompanying Python helper library is introduced to facilitate the seamless integration process.

We conducted a series of experiments using the Linked Data Benchmark Council's social network benchmark datasets. These experiments showcase the functional viability of our integration and the effectiveness of our optimization strategies. Moreover, our implementation suggests potential avenues for future research, such as executing inferential tasks with SQL and enabling the training

of GNN models on graphs larger than a single machine's memory capacity. We plan to enhance our project along these lines to achieve more seamless integration between DuckDB/DuckPGQ and GNN libraries.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

Graphs are fundamental data structures representing a set of nodes interconnected by edges (1). They find diverse applications in modeling relationships within social networks (2), protein-protein interactions (3), and knowledge representations (4). The processing of graphs, such as calculating sparse edge representations (5) or finding the shortest path (6), is ubiquitous in both theoretical research and practical applications.

Recent times have witnessed a surge in the use of machine learning methods for graph analysis (7). These methods have demonstrated powerful capabilities in various application scenarios, such as social spammer detection (8), dynamical physical systems (9), protein interactions (10), and power system computations (11). Several graph neural networks, such as graph convolutional networks (12), graph attention networks (13), GraphSAGE (14), and gated graph sequence neural networks (15), have been proposed to handle diverse graph-structured datasets.

To facilitate the development and deployment of graph neural network models, software packages like PyTorch Geometric (16) and Deep Graph Library (DGL) (17) have been introduced. These libraries offer functions to read data from various sources, such as comma-separated value (CSV) files, and represent it using their internal graph classes, typically built on top of PyTorch Tensors (18). This representation enables model training and prediction based on the graph data, utilizing the functionalities provided by the packages.

In addition to the internal graph data structures in PyTorch Geometric or DGL, another approach to representing graphs is the property graph model (19). Property graphs allow vertices and edges to have labels and multiple properties expressed as key-value pairs.

1. INTRODUCTION

Several graph database management systems (GDBMS) support the property graph model, with examples like KÙZU (20).

Property graphs can also be defined over tables in relational database management systems (RDBMS) (21). The SQL:2023 standard introduces a sub-language named property graph queries (PGQ). This extension to the SQL standard enables the creation of graph views over relational tables and supports read-only queries on these views.

DuckPGQ (22), an extension module for the online analytical RDBMS DuckDB (23), adds support for the SQL/PGQ sub-language. It accomplishes this by translating graph queries into SQL statements with user-defined function (UDF) calls.

Property graph databases provide an optimized mechanism for graph storage and efficient graph query execution (20). On the other hand, GNN libraries provide the ability to build predictive models for analyzing graphs and performing inferences. However, due to the gap between the two different models, it is currently inconvenient to leverage the strengths of both systems simultaneously. Integrating one system with the other could prove highly valuable.

In this project, our main objective is to explore the possibility of seamlessly integrating DuckPGQ/DuckDB with two GNN libraries, PyTorch Geometric and DGL. This integration aims to bridge the gap between property graph databases and GNN frameworks, unlocking new opportunities for data modeling and analysis.

This thesis makes several contributions. Firstly, by implementing an integration between DuckPGQ and GNN libraries, we enable the loading of data, GNN model training, and inference from the property graph database. Leveraging the relational nature of DuckDB, this integration opens up opportunities to represent and process relational data as graphs, potentially leading to the discovery of valuable insights, such as capturing the relationships between training samples.

Secondly, our integration allows the execution of graph queries over PyTorch Tensor-based GNN graphs, enhancing the flexibility of querying and analyzing graph-structured data. Additionally, we enable DuckPGQ/DuckDB to store GNN graphs as relational tables, providing users with the capability to utilize both relational and graph-based representations for their data.

Our integration is open-sourced [1], and part of the implementation has been merged in the DuckPGQ extension [2].

[1] https://github.com/vlowingkloude/pyduckpgq
[2] https://github.com/cwida/duckpgq-extension

1.2 Research Questions

1.2.1 Research Goals and Challenges

The primary objective of this project is to establish seamless and efficient integration between DuckDB and graph analysis pipelines within the Python environment. Our goal for efficiency is to minimize the necessity for memory copies wherever feasible, as duplicating data is time-consuming and can result in significantly elevated memory consumption. In terms of seamlessness, we anticipate enabling the execution of user-defined functions on Python objects in a manner akin to their execution on DuckDB tables, thereby demanding no additional effort from users. To achieve this goal, we have identified several key challenges:

1. The schema of a property graph is not directly accessible from outside DuckPGQ, making it challenging to retrieve related data from DuckDB tables using standard `SELECT` queries.

2. The graph connectivity structure of a DuckPGQ property graph is an internal C++ native object, which is not visible or easily accessible from outside clients.

3. Ensuring efficiency in the integration requires minimizing memory copies. However, when DuckDB executes `SELECT` queries, memory copies become necessary.

4. While DuckDB Python APIs offer the capability to run queries over recognized objects, including `Pandas DataFrames`, `PyArrow Tables`, `Polars DataFrames` and `Numpy arrays`, directly, certain functionalities provided by DuckPGQ rely on the existence of a special `rowid` column, which is not present in these Python objects.

5. PyTorch Geometric employs a different format to represent the connectivity structures of a graph, creating a disparity that needs to be addressed during the integration.

1.2.2 Research Questions

Given the above challenges, we have formulated the following research questions to guide our efforts:

1. What information is required to retrieve a DuckPGQ property graph from Python clients?

2. With the required information, what steps are needed to retrieve a DuckPGQ property graph from Python clients?
3. Among these needed steps, is it possible to avoid data copies with memory-sharing?
4. How to construct a PyTorch Geometric graph from CSR representation of connectivity structures?
5. How to run UDFs in DuckPGQ directly over Python objects?
6. How to simplify the process of fetching a property graph as a GNN graph and the use of user-defined functions in DuckPGQ?

1.3 Thesis Structure

The rest of the thesis is structured as follows. In Chapter 2, we present background information on graphs and their representations, graph neural networks, property graphs, SQL/PGQ, DuckDB, and the DuckPGQ extension. Chapter 3 provides a literature study of research projects related to our work, including a discussion of the interactions between database systems and machine learning pipelines from an engineering perspective. In Chapter 4, we describe the design and implementation details of our projects. The experimental results are presented in Chapter 5, where we showcase the findings. Chapter 6 discusses the limitations and future work of this project. Finally, Chapter 7 concludes the thesis.

2

Background

In this section, we concisely present the background information that this project is built on.

2.1 Graphs

Graphs are mathematical structures that depict relationships between objects. They consist of nodes (also known as vertices) and edges (also known as links). Nodes represent entities or data points, while edges represent connections or relationships between them. The edges can be either directed (indicating a one-way relationship) or undirected (indicating a two-way relationship). Additionally, each edge can have an associated weight, which may signify a numerical value or some other relevant information.

Graphs serve as a natural and versatile way to model groups of objects and their interconnections. They find applications in various domains, such as computer networks, social networks, logistics, and more.

In Figure 2.1, we have an illustrative example of a directed graph with six nodes and five edges. This graph demonstrates the money transfer relationships between a group of individuals and companies. Each node represents a unique entity, and each directed edge signifies a money transfer from the source node to the destination node. For instance, the edge connecting `Olivia` and `AWS` denotes that `Olivia` made a money transfer to `AWS`.

2.2 Graph connectivity representations

Consider a graph G with $|n|$ nodes and $|e|$ edges. Each node in the graph is uniquely identified by a number within the range $[0, |n|)$. The edges are represented as pairs of node

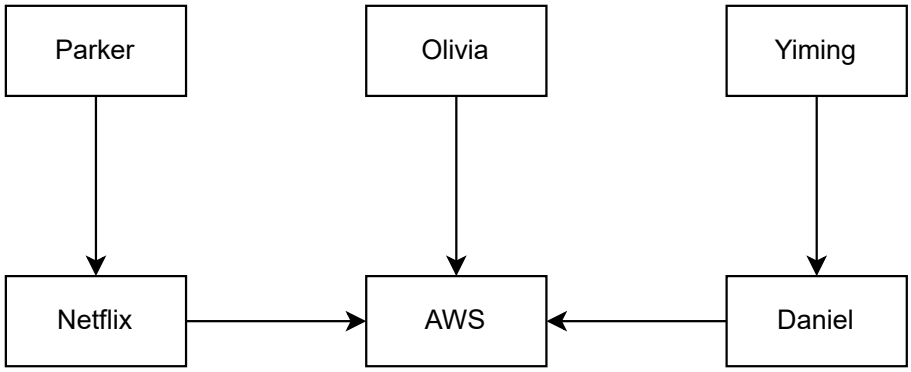

Figure 2.1: Money transfers between a group of entities

IDs. Traditionally, the connectivity of a graph is represented using an adjacency matrix, which is a square matrix of size $|n| \times |n|$. However, for large sparse graphs, employing an adjacency matrix for storage can be inefficient and result in poor data locality.

To address this issue, two alternatives for representing the connectivity of a sparse graph are the Coordinate List (COO) and Compressed Sparse Row (CSR) formats.

The COO representation utilizes two vectors of the same length to store the source and destination nodes for edges, respectively.

On the other hand, the CSR representation also employs two vectors. The first vector, called `v`, consists of $|n| + 1$ elements and maintains the cumulative degree of all vertices, starting from node 0. The last element is the number of edges in a graph. The second vector, named `e`, stores the out-neighbors for each edge.

Additionally, both the CSR and COO representations can include a third vector, denoted as `w` or as a weight array, of the same length as `e` to hold the weight for each edge.

As presented in Table 2.1, an ID is assigned to each entity involved in Figure 2.1. We can represent the connectivity using the COO and CSR formats, as shown in Figure 2.2.

Name	ID
Parker	0
Netflix	1
Olivia	2
AWS	3
Yiming	4
Daniel	5

Table 2.1: Node ID

Source	Destination
0	1
1	3
2	3
4	5
5	3

Table 2.2: COO representation

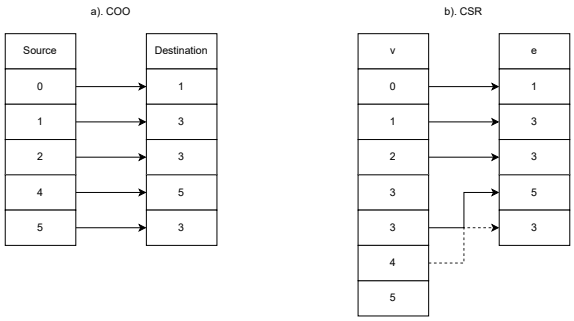

Figure 2.2: COO and CSR Representation

2.3 Property Graphs

While the graph structure is capable of showing relationships or connectivity between a collection of nodes and is useful in applications such as finding shortest paths, it does not inherently allow for storing properties that provide additional information about the entities and relationships they represent.

To address this limitation, the labeled property graph (LPG) (19) model can be employed. In an LPG, each node can have one or more labels associated with it, representing the node's type or category. Similarly, relationships (edges) in an LPG can be assigned labels to describe the type of connection between two nodes. Furthermore, both nodes and edges can have key-value pairs called properties, which characterize their features or attributes.

For example, to enhance the presentation of the money transfer relationships between entities in Figure 2.1, we can label each node and edge and assign attributes to them, as demonstrated in Figure 2.3. Each node is labeled as `Person` or `Company`, while each edge is labeled as `TransferTo`. Additionally, we can attach properties to each node, such as `account` and `name`, providing specific information about individual entities. Similarly, each edge can have a property such as `isRecurring`, which helps identify whether a particular

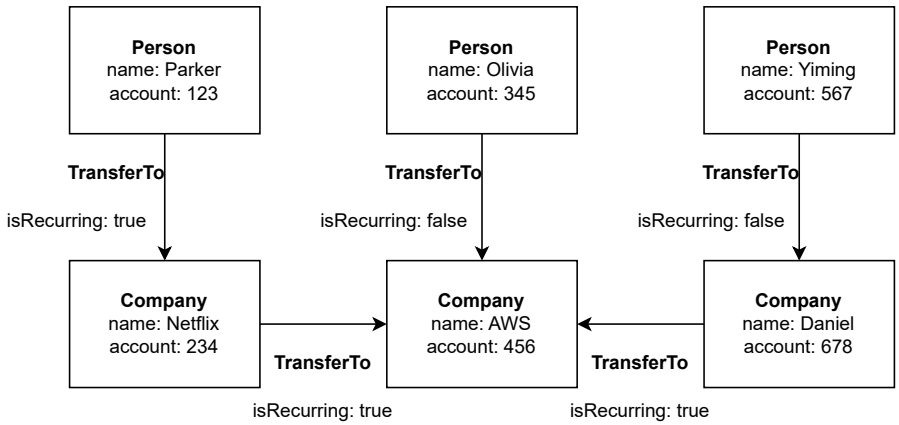

Figure 2.3: Labeled Property Graph of money transfer relationship between a group of entities

money transfer occurs on a recurring basis or not.

2.4 SQL/PGQ

2.4.1 Tabular Representation of an LPG

In some cases, tables stored in a Relational Database Management System (RDBMS) can effectively represent a Labeled Property Graph (LPG) (21). In such situations, each table may correspond to either a node or an edge, and the fields within a table represent attributes for nodes and relationships. The table names themselves serve as labels.

For instance, the nodes depicted in Figure 2.3 can be represented using the table structure shown in Table 2.3 and Table 2.4, while the relationships (edges) can be described using the table structure presented in Table 2.5.

name	account
Parker	123
Olivia	345
Yiming	567
Daniel	678

Table 2.3: Person table

name	account
Netflix	234
AWS	456

Table 2.4: Company table

Source	Destination	isOptional
Parker	Netflix	true
Olivia	AWS	false
Netflix	AWS	true
Yiming	Daniel	false
Daniel	AWS	true

Table 2.5: TransferTo table

2.4.2 Property Graph Queries

SQL/Property Graph Queries (SQL/PGQ) (21) is an extension of the SQL language that enables the creation of graph views, known as property graphs (PGs), over existing relational tables, facilitating the querying and manipulation of PGs within an RDBMS.

As an illustrative example, the script in Listing 2.1 demonstrates the creation of a property graph named `transferrelationship` using data from Table 2.3, Table 2.4, and Table 2.5. In this scenario, the `person` and `company` tables serve as vertex tables, housing properties such as `name` and `account`. On the other hand, the `TransferTo` table functions as an edge table, incorporating the `isRecurring` property.

```
1  CREATE PROPERTY GRAPH transferrelationship
2  VERTEX TABLES (
3      person PROPERTIES ( name, account ) LABEL Person,
4      company PROPERTIES ( name, account ) LABEL Company
5  )
6  EDGE TABLES (
7      TransferTo SOURCE KEY ( Source ) REFERENCES person ( name )
8                  DESTINATION KEY ( Destination ) REFERENCES person ( name
       )
9                  PROPERTIES ( isRecurring ) LABEL TransferTo
10 )
```

Listing 2.1: Creating a property graph in SQL/PGQ

2.5 Graph Neural Network

2.5.1 Overview

A Graph Neural Network (GNN) (24) is a specialized type of neural network designed to analyze and process data organized in the form of graphs. Unlike traditional neural networks that work on sequential or matrix-like data (e.g., texts or images), GNNs are adept at handling complex relationships and dependencies inherent in graph structures.

The fundamental concept behind GNNs is to learn embeddings for each node in a graph. An embedding is a mathematical representation of a node as a dense vector in a continuous vector space. This process involves aggregating information from neighboring nodes through a series of message-passing steps, where nodes exchange and combine information with their connected neighbors. Through iterative updates based on local neighborhoods, GNNs can effectively capture both local and global structural patterns within the graph (25).

GNNs have demonstrated remarkable success across various tasks. Depending on the level of prediction target, these tasks can be broadly categorized into three group (7)s:

- Node Property Prediction: This group includes tasks like node classification and regression, where the GNN aims to predict specific properties or labels associated with individual nodes in the graph.
- Link Property Prediction: This category involves edge classification and regression, as well as link prediction. The GNN focuses on predicting properties related to the edges between nodes or forecasting potential new connections in the graph.
- Graph Property Prediction: This group encompasses tasks such as graph classification, where the GNN's objective is to predict high-level properties or characteristics of the entire graph.

More specifically, common GNN tasks are as follows:

- Node Classification: Inferring a categorical target for a node. For instance, given the features of a node, we can predict if this node is a `Company` or a `Person`.
- Node Regression: Inferring a numerical variable for a node. For example, given the features of a node, we can predict the importance of this node based on the number of transfers that this node is involved in.

- Edge Classification: Inferring a categorical target for an edge.
- Edge Regression: Inferring a numerical target for an edge. For example, we can predict the weight or strength of a particular edge in a graph.
- Link Prediction: Inferring the likelihood of the existence of an edge. For example, we can predict if `Yiming` is likely to transfer money to `Olivia`.

2.5.2 GNN libraries

To foster research and application of Graph Neural Network (GNN) models, several libraries have been developed, two of which are PyTorch Geometric (16) and Deep Graph Library (DGL) (17).

Both PyTorch Geometric and DGL focus on efficient processing and analysis of graph data, providing a wide range of GNN implementations, message-passing functions, and utilities for handling graph datasets. Although they serve similar purposes, they differ in their design philosophies.

PyTorch Geometric is built on top of PyTorch and tightly integrated with the PyTorch ecosystem. DGL, on the other hand, supports various backend frameworks, including PyTorch, MXNet, and TensorFlow, thereby allowing users to choose their preferred deep learning framework. DGL and PyTorch Geometric also diverge in their approaches to representing a graph and defining a GNN model.

In DGL, a high-level class called `DGLGraph` is implemented to store a graph. To define a graph, the function `dgl.graph` or `dgl.heterograph` should be called, with the data parameter set as the graph structure in COO, CSR, or other supported formats for homogeneous or heterogeneous graphs, respectively. Node features and edge features are then filled in the `ndata` and `edata` fields of the `DGLGraph` object, respectively.

An example of how to define a homogeneous graph in DGL using PyTorch as the backend, based on the diagram provided in Figure 2.1, is shown in Listing 2.2.

In contrast to DGL, PyTorch Geometric implements the `Data` class for graphs, where users need to provide the features and edges to create a graph. By default, PyTorch Geometric only recognizes the COO format during initialization. Listing 2.3 shows how to define a homogeneous graph in PyG based on the diagram provided in Figure 2.1.

As shown in Listing 2.2 and Listing 2.3, it is worth noting the difference in how DGL and PyTorch Geometric store node and edge features. In DGL, features are stored separately as 1-dimensional vectors, while in PyTorch Geometric, features are stored as 2-dimensional matrices.

```
import dgl
import torch

coo_src = torch.Tensor([0,1,2,4,5])
coo_dst = torch.Tensor([1,3,3,5,3])

# Construct a DGLGraph with COO
g = dgl.graph((coo_src, coo_dst))

# Set node features
g.ndata['name'] = ...
g.ndata['account'] = ...

# Set edge features
g.edata['isOptional'] = ...
```

Listing 2.2: Creating the DGLGraph for Figure 2.1

```
from torch_geometric.data import Data
import torch

# First create a matrix for COO arrays
coo = torch.Tensor([[0,1,2,4,5],
                    [1,3,3,5,3]])

# Stack all node features as a matrix
node_features = ...

# Stack all edge features as a matric
edge_features = ...

# Construct a graph
g = Data(x = node_features, edge_index = coo, edge_attr = edge_features)
```

Listing 2.3: Creating the PyG graph using `Data` for Figure 2.1

Additionally, PyTorch Geometric introduces two essential classes: `FeatureStore` and `GraphStore` (26). These classes enable the utilization of graph database systems as the backend storage for data. `FeatureStore` defines interfaces for fetching features for nodes and edges from the graph database. On the other hand, `GraphStore` holds the connectivity structure of graphs and facilitates efficient sub-graph sampling. Both feature values and sub-graphs are retrieved from the graph database only when needed, allowing GNN models to be trained on graph data larger than the machine's memory. It's important to note that this functionality is still under development, and the APIs are not yet stable. Furthermore, fetching edge features is currently not supported.

2.6 DuckDB

DuckDB (23) is an in-process analytical RDBMS designed for efficient query execution and analysis of structured data. It offers a C++ API, as well as interfaces for Python and other programming languages. In DuckDB, the engine utilizes the `DataChunk` class as an intermediary representation to execute queries. A `DataChunk` comprises a collection of vectors and represents a subset of a relation.

2.6.1 Replacement Scan

The DuckDB Python package provides a useful functionality known as replacement scan, which allows users to run SQL queries directly over a collection of recognized Python objects. This feature operates as follows: when attempting to read a non-existent table, a pre-registered callback is invoked. This callback generates the table dynamically and then proceeds to execute the SQL queries. The DuckDB Python package currently supports several popular Python data structures, including: `Pandas DataFrames`, `PyArrow Tables`, `Polars DataFrames`, and `NumPy ndarrays`.

2.6.2 Extension and User-Defined Functions

DuckDB offers developers the flexibility to enhance its functionalities without the need to modify its core codebase.

Within an extension module, developers can implement and register user-defined functions (UDFs). These UDFs can take the form of scalar functions or table functions. Scalar functions are designed to carry out computations and return a single value for each record. They are commonly employed within a `SELECT` statement. For example, in the query `SELECT plusone(2);`, the scalar function `plusone` is invoked with the parameter

2 and produces a return value of 3. At the implementation level, any user-defined scalar function requires three parameters: `args`, which represents a `DataChunk` holding a set of input vectors for the UDF; `expr`, providing information about the query's expression state; and `result`, a `vector` used to store the resulting values.

On the other hand, table functions are functions that return tables and are triggered when a non-existent relation is accessed. They are typically utilized in the `FROM` clause of a query. For instance, executing `SELECT * FROM generate_data(100)` invokes the `generate_data` table function to generate 100 records. At the implementation level, any user-defined table function would also require three parameters: the client context, which maintains the state of DuckDB and its extensions; a pointer to the input of the invoked table function; and a result, which is a `DataChunk` used to store the table generated by the table function.

2.7 DuckPGQ

DuckPGQ (22) is an extension module for DuckDB that adds support for the SQL/PGQ sub-language (21). It enables property graph queries to be translated into standard relational queries, with the possibility of using UDF calls for certain queries like path-finding, which helps find the path between a set of nodes. The property graph representation in DuckPGQ, as shown in Figure 2.4, utilizes existing DuckDB tables as vertex or edge tables and allows the use of only a subset of columns in these tables as properties of edges and nodes. This schema information is stored in C++ native structures.

DuckPGQ provides two UDFs, namely `CREATE_CSR_EDGE` and `CREATE_CSR_VERTEX`, for CSR (Compressed Sparse Row) creation. An example of this is given in Listing A.1. All the created CSR objects are stored in an ID-to-CSR mapping, where each ID is a 32-bit integer serving as a key to index a specific CSR object. The representation of a CSR object in DuckPGQ is illustrated in Figure 2.5, and it consists of several member variables. The `v` array is stored as a C-array of 64-bit atomic integers, while the `e` array is stored as a vector of 64-bit integers. The weight array can be either a vector of 64-bit integers or double floating-point numbers, depending on the type of weight used. To conveniently determine the number of elements in the `v` array, the `vsize` variable is employed.

2.8 LDBC

The Linked Data Benchmark Council (LDBC) (27) is an industry consortium dedicated to the development of standardized benchmarks for graph and RDF (Resource Description

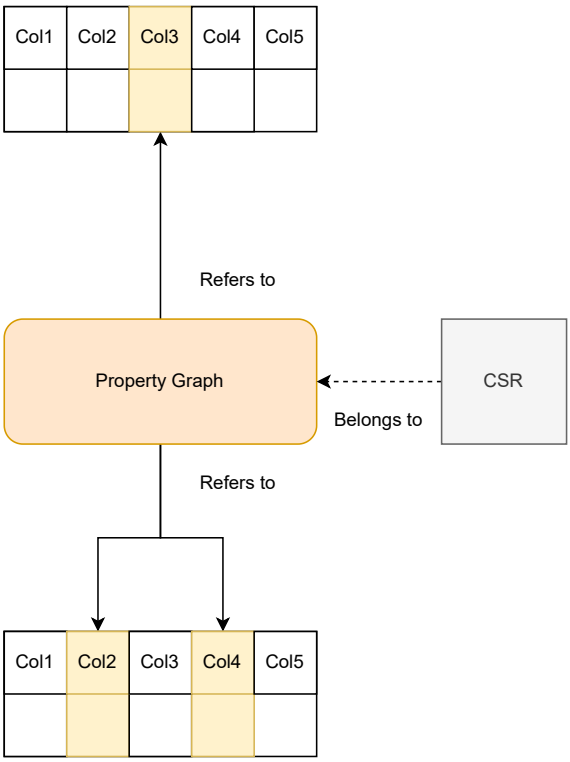

Figure 2.4: DuckPGQ's representation of a property graph

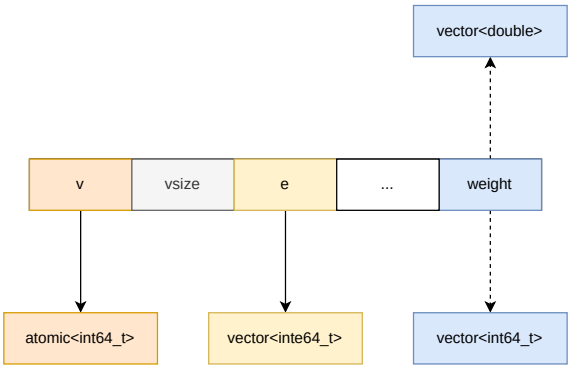

Figure 2.5: DuckPGQ's representation of a CSR object

Nodes

ID
14
16
27
33
47
...

Edges

Source	Destination	Weight
332	2866	9
332	2869	2
376	2783	10
977	1527	5
376	5320	7
...	...	...

Figure 2.6: An example of LDBC SNB data

Framework) database systems (28). The primary objective of LDBC is to address the lack of widely accepted benchmarks for evaluating the performance and scalability of graph database technologies.

One of LDBC's notable achievements is the creation of several benchmark suites, among which is the Social Network Benchmark (SNB) (29). Designed specifically for evaluating the performance of graph database systems in the context of social network applications, SNB provides simulated social network scenarios with realistic data and workloads.

In this project, we utilize the LDBC SNB data. An example of the data schema is illustrated in Figure 2.6. The node data contains a single column representing the unique ID for each node, while the edges data consists of three columns: the source and destination columns, both referring to the node IDs and a weight column. The LDBC SNB datasets vary in size and are quantified by a scale factor, where a larger scale factor indicates larger datasets.

3

Related Work

Retrieving data from database systems has always been a crucial aspect of machine learning pipelines (30). This significance arises because while most machine learning libraries offer built-in structures to handle inputs and outputs (31), the majority of data is not readily available in such formats. Companies and organizations tend to store their data in database systems (32). As a result, transforming the data into a format recognizable by downstream machine learning libraries and models becomes necessary for training and performing inference tasks (33). To bridge the gap between databases and machine learning pipelines, several methods have been developed (30).

In this section, we provide an engineering perspective on the interactions between databases and machine learning pipelines. Firstly, we discuss general approaches for retrieving data from database systems for machine learning pipelines. Next, we focus on the specific interactions between graph neural network pipelines and database systems.

3.1 Interactions between database systems and machine learning pipelines

3.1.1 Transfer data from database to machine learning pipelines

Most database systems, including popular ones like MySQL and PostgreSQL, offer APIs that enable querying and fetching data from Python clients (34). To simplify the data loading process from databases and streamline the conversion from query results to data frames, various data analysis libraries provide procedures that return objects ready for downstream machine learning pipelines. A prominent example is the `read_sql` function in Pandas (35).

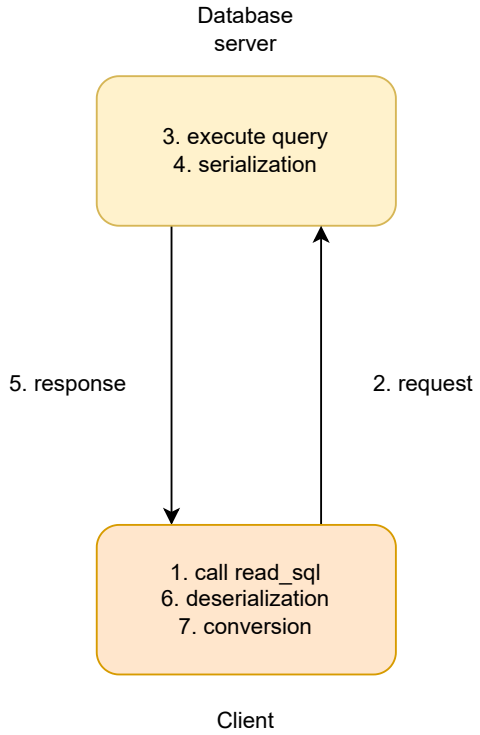

Figure 3.1: `read_sql` requires several steps to retrieve data as DataFrame

Figure 3.1 illustrates how the `read_sql` function operates. First, the client program sends requests to the database server. Upon receiving the request, the server executes the query, fetches the related data, serializes it, and sends the result back to the client. The client then deserializes the response and performs necessary conversions. Finally, the `read_sql` function returns a Pandas DataFrame object to the user (36).

There are several advantages to transferring data from database systems to machine learning pipelines:

1. Versatility: This approach is highly versatile and can be applied in most scenarios. For instance, Pandas `read_sql` provides support for reading data from various databases, such as MySQL, PostgreSQL, and Oracle (37).

2. Tight Integration: It provides seamless integration with data analysis libraries, enabling the retrieved data to effortlessly enter the machine learning pipeline (35), without the need for cumbersome data conversion operations.

However, this method also has certain disadvantages, primarily related to performance (33).

3.1 Interactions between database systems and machine learning pipelines

1. Data Transfer Overhead: The data transfer between the database server and the client is unavoidable, even when they are on the same machine and have similar in-memory layout (38), leading to potential time-consuming delays.
2. Conversion Overhead: The conversion from the query result received by the client to data frames consumes significant time and memory resources (30).
3. Additional Overheads: Other substantial overheads include the database connection, serialization, and deserialization processes (39).

To enhance the process of transferring data from database systems to machine learning pipelines, Wang et al. introduced ConnectorX (30). They conducted a performance analysis for each step of the Pandas `read_sql` function when retrieving data from a database system, drawing the following conclusions:

1. Client-Side Time: A significant portion (>85%) of the overall time is spent on the client side.
2. Memory Usage: The peak memory usage is approximately four times larger than the final size of the data frame, as all intermediate results, including raw bytes, temporary Python objects, and data frames, are kept in memory during the execution of the `read_sql` function.

To improve performance and reduce memory usage in `read_sql`, the authors employed several techniques:

1. Metadata Query: They first query the metadata of the query result to determine the size of the final Pandas DataFrame. Using this information, they pre-allocate the required memory.
2. Parallel Execution: A large query is partitioned into smaller queries, each fetching records within a user-specified range. The union of the query results from the smaller queries forms the same result set as the original query, leveraging parallel execution.
3. Efficient Worker Threads: In each worker thread, only a small batch of the query result is fetched and processed. The values are directly written to the pre-allocated memory.

By implementing these optimizations, ConnectorX achieved a runtime reduction of a factor of 13 and a memory usage reduction of three times when compared to the Pandas `read_sql` implementation.

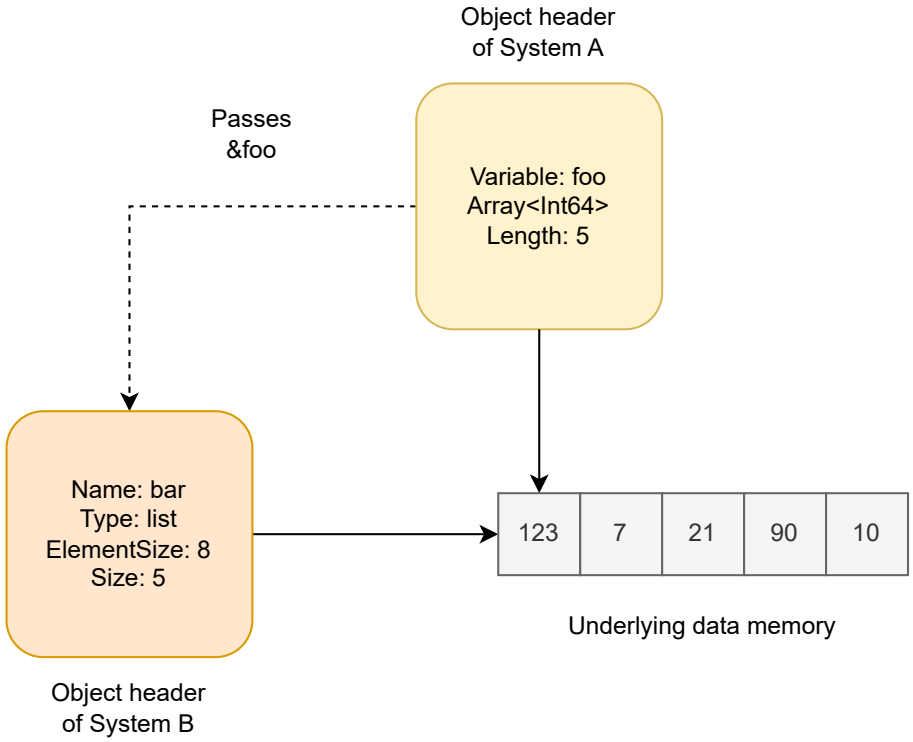

Figure 3.2: An example of zero-copy data sharing between two systems

3.1.2 Zero-copy of data between databases and machine learning pipeline

Another approach to utilizing data stored in databases within a machine learning pipeline involves sharing memory between the two systems, eliminating the need for memory copy operations.

In theory, achieving zero-copy is possible when both systems have the same memory layout for the underlying data of objects. This involves initializing the object headers, which contain metadata and memory references of objects, in both systems and setting the data pointer to the same memory location (40). In practice, C-style arrays, which are essentially pointers to continuous memory sections that store values of native types like integers or doubles, are widely used in various systems, including Numpy arrays and C++ vectors (41). For illustrative purposes, Figure 3.2 provides an example where two systems share part of the memory holding five 64-bit integers.

Even though system A and system B have different object headers, they both use the same C-style array memory layout to store the underlying data. To facilitate system B's access to the array, system A only needs to pass the starting address of the underlying

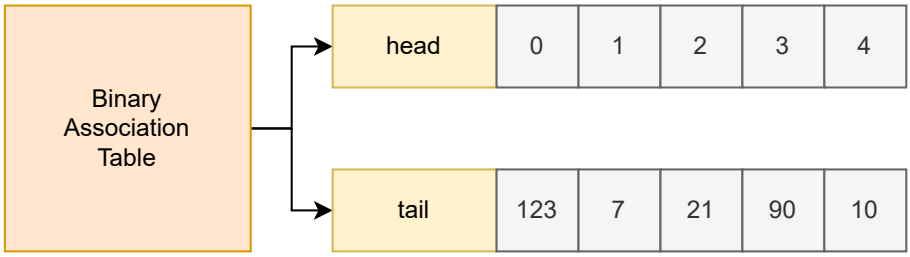

Figure 3.3: Binary Association Table structure of MonetDB

Figure 3.4: INTSXP structure of R

data. Upon receiving the address, system B constructs its object header. As a result, the same section of memory becomes shared between system A and system B. Instead of copying the five 64-bit integers, only the address is transferred. This sharing of memory enables efficient data exchange without unnecessary duplication.

Numerous research projects and prototypes have been developed in this field. For instance, Lajus et al. (41) successfully implemented a zero-copy integration between MonetDB (42), a column-oriented RDBMS that can be embedded into other applications, and R, a software package for statistics and machine learning.

In MonetDB (42), a relation is represented as a set of Binary Association Tables (BATs) (41). Each BAT comprises two arrays: the head array stores the row indices, while the tail array contains the corresponding values. The structure is depicted in Figure 3.3.

On the other hand, R uses a single data type called Symbolic Expression for all values, with various subtypes supported, such as `INTSXP` for integer vectors and `REALSXP` for floating-point number vectors. The structure of an `INTSXP` vector is illustrated in Figure 3.4. Upon comparing Figure 3.3 and Figure 3.4, it becomes evident that a zero-copy integration between MonetDB and R is feasible, as shown in Figure 3.5.

This memory-sharing mechanism allows for deep integration between MonetDB and R, facilitating convenient, efficient, and expressive data analysis functionality using statistical analysis tools (41).

The benefits of memory-sharing between database systems and machine learning pipelines include reducing data transfer and memory duplication (43). However, there are also several limitations. First, data inside different systems may not always have

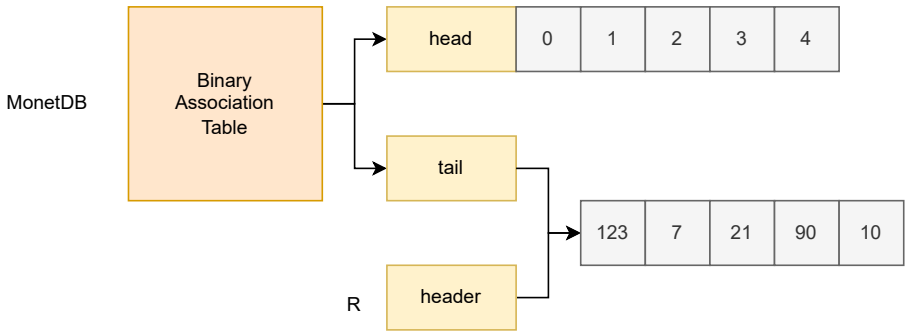

Figure 3.5: Zero-copy between MonetDB and R

identical representations, making zero-copy data sharing impossible in such cases and limiting its applicability. Additionally, both sides of integration can access the shared data, necessitating extra efforts to manage memory and control concurrent accesses. In some systems, like MonetDB and R, only one side is allowed to write data (41).

3.1.3 In-database machine learning

Unlike transferring or memory-sharing data from database systems to machine learning pipelines, in-database machine learning does not export data outside the database systems. Instead, it runs machine learning algorithms and analytics inside the database itself. Several database systems support in-database machine learning nowadays, such as Oracle database (44) and PostgreSQL with Apache MADlib (45).

One way to perform in-database machine learning is to write SQL commands directly or to convert an analysis algorithm implemented in another procedural language into SQL code.

One of the early works on in-database data analysis was conducted by Agrawal et al. (46). In their study, they argued that bringing data from the database to an external analysis environment results in performance degradation due to data copying and context switching. To illustrate this point, they provided an example of implementing the Apriori algorithm, used for mining frequent item sets, in pure SQL. To assess the performance, they compared the SQL version of Apriori with the approach that extracts data from the database first. Their findings showed that the SQL version is more than two times faster, supporting their argument that in-database data analysis generally offers better performance.

To meet the computational power requirements for processing large-scale datasets and enhance the speed of model training and prediction, researchers have explored methods to enable in-database machine learning on distributed systems and GPUs. For instance,

Sandha et al. (47) presented a prototype for training a linear regression model within the Teradata SQL engine. Teradata is a parallel and distributed engine with a share-nothing architecture. On a similar note, Schüle et al. (48) developed a complete in-database machine learning pipeline using the Umbra database system (49). Their implementation supports automatic differentiation and gradient descent operations and is designed to leverage GPUs for computational tasks.

Expressing complex machine learning tasks in SQL manually can be challenging due to the iterative nature of many data analysis algorithms, which is difficult to express in SQL, as pointed out by Passing et al. (50). As a result, researchers have explored methods to convert procedural programs, such as Python or R data analysis programs, into a series of SQL commands. Blacher et al. (51) discussed how to perform such conversions for concepts common in procedural languages, including variables (translated as relations or values in relations in SQL), branch statements (translated as `CASE` statements), loops (expressed as recursive queries), etc. Additionally, due to the power and popularity of the Pandas library, Hagedorn et al. (52) presented an implementation that translates operations on Pandas DataFrames into SQL queries, while Marten et al. (53) translated the Hidden Markov model expressed in R into SQL statements.

One benefit of using a SQL-only approach to express a machine learning pipeline is that the data analysis pipeline is represented as a series of SQL commands, eliminating the need for any changes to the database system itself. This approach provides a convenient way to work with the existing database infrastructure for data analysis tasks.

Raasveldt et al. (54) proposed an alternative approach for in-database machine learning by integrating an external machine learning library with MonetDB. They leveraged MonetDB's capability to invoke Python code through User-Defined Functions (UDFs) and utilized the scikit-learn machine learning library (55) to train models and make predictions. In their approach, machine learning models are treated as Binary Large Objects (BLOBs), enabling efficient storage and retrieval within the database.

This method showcased the possibility of performing in-database data analysis with machine learning models implemented in Python. However, the applicability of this approach is contingent upon the support provided by the database system. The availability of such integration features in the database system will dictate the feasibility of using external machine-learning libraries in conjunction with the database.

In conclusion, in-database machine learning offers the significant advantage of conducting data analysis, model training, and future data prediction directly within the database system. By eliminating the need for data exporting, this approach improves performance

and reduces memory usage. Moreover, the tight integration between SQL and machine learning models facilitates a unified interface for seamless interactions. However, it is worth noting that expressing a machine learning pipeline purely in SQL can be challenging, as pointed out by Passing et al. (50). Additionally, the effectiveness of in-database machine learning heavily relies on the level of support provided by the database vendors. The availability of specific features and capabilities within the database system will impact the feasibility and extent of in-database machine-learning implementations.

3.2 GNN-aware graph database systems

In the previous sections, we explored general research work related to training machine learning models and conducting analysis using data stored in database systems. However, when it comes to fetching graph data from database systems for graph neural networks (GNNs), the approach is somewhat different. In traditional machine learning pipelines, a model is trained using a matrix representation of the training data, where each row represents a data point, and each column corresponds to a feature that quantifies the data point. This matrix is conceptually similar to a table in relational database systems.

However, training a GNN model requires not only the features of nodes and edges but also the graph's connectivity structure (56). This introduces a new dimension of complexity in data representation for GNNs. In this section, we conduct a literature study that focuses on how GNN pipelines interact with databases, addressing the unique requirements for graph-based machine learning tasks.

3.2.1 Neo4j

Neo4j (57) is a graph database management system implemented in Java. It is designed to handle and store graph-structured data efficiently, making it well-suited for applications that involve highly interconnected data and complex relationships between entities. Neo4j uses the Cypher query language (58), which is specifically designed for querying and manipulating graph data. Cypher provides an expressive and intuitive way to interact with the graph, allowing users to create complex queries to traverse, filter, and analyze the graph.

It has several built-in algorithms for graph analysis, including path finding, similarity computation, and community detection (59). Furthermore, it has many algorithms for learning node embeddings and performing link predictions, although not all of them

are production-ready, making it capable of running in-database machine learning on graphs (60).

3.2.2 Kùzu

Kùzu (20) is an in-process property graph database system that utilizes Cypher for querying graphs. It offers a command-line interface and supports language bindings to C++, Java, and Python.

In the Python environment, Kùzu supports exporting a property graph to PyTorch Geometric, leveraging the `FeatureStore` and `GraphStore` mechanisms discussed in Chapter 2. This functionality makes Kùzu a valuable backend graph storage option for GNN model training (61). PyTorch Geometric fetches sub-graphs and feature values from Kùzu, which efficiently stores data on disk, enabling model training on datasets that exceed the machine's memory capacity.

In conclusion, Kùzu provides seamless integration with PyTorch Geometric. However, as detailed in Section 4.4, our approach differs from that of Kùzu.

3.2.3 TigerGraph

TigerGraph (62) is a graph database and analytics platform. It features GSQL (63), its own expressive query language specially designed for querying and manipulating graph data. GSQL offers support for both transactional and analytical queries, making it versatile for various applications. TigerGraph incorporates a module called ML-Workbench (64), which facilitates the exporting of graph data via HTTP, similar to Pandas' `read_sql` function, to different GNN libraries and pipelines, including DGL and PyTorch Geometric. The enterprise edition of TigerGraph further provides support for exporting data via Kafka, an open-source distributed event streaming platform used for data pipelines.

3.2.4 Amazon Neptune

Amazon Neptune (65) is a managed graph database service provided by Amazon Web Services (AWS).

Amazon Neptune supports data loading from CSV files and allows querying a graph using Gremlin (66), which is the graph traversal language for the Apache TinkerPop graph computing framework (67).

The platform offers a module called Neptune ML (68) designed for graph machine learning tasks, including node/edge level classification/regression and link prediction. The

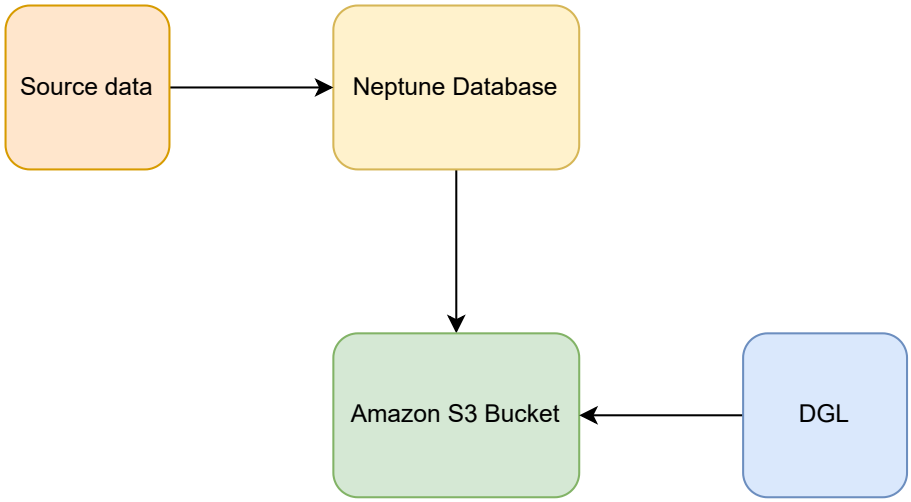

Figure 3.6: How Neptune ML works

model computations are powered by DGL. Figure 3.6 presents an example illustrating the process of training a GNN model and performing inferences using Amazon Neptune.

First, the data needs to be exported from a Neptune cluster to an Amazon S3 storage. This step enables DGL to access the input data. A configuration is required to specify the output path, feature, and target descriptions, along with other parameters. Once the data is exported, the pre-processing and model training can start, which also requires user-specified arguments. After model training and endpoint creation, users can run inferences with Gremlin. Listing 3.1 shows an example of predicting a given name `Parker` is whether a human name or an organization name.

```
g.with("Neptune#ml.endpoint", '${endpoint}').V()
 .has('name', 'Parker').properties("type")
 .with("Neptune#ml.classification").value()
```

Listing 3.1: An example of performing classification with Gremlin

4

Design & Implementation

The aim of this project is to establish an efficient integration between DuckDB/DuckPGQ and GNN libraries. This integration is intended to enable the smooth retrieval of a property graph into GNN pipelines within a Python client. Our central focus is the optimization of memory usage through the reduction of unnecessary data copying. Moreover, we're committed to ensuring a seamless experience, where the execution of a UDF from DuckPGQ on Python objects demands no additional effort. This chapter delves into the significant design details of our implementation.

The initial part of our work concentrates on strategies to facilitate the retrieval of property graphs from DuckPGQ. We emphasize the application of optimization techniques to minimize memory copies, striving for a highly efficient process during this phase.

Transitioning to the subsequent segment, we further enhance our solution to guarantee seamless integration and user-friendliness. Our objective is to establish an intuitive and effortlessly usable connection between these two frameworks.

4.1 Retrieving table and column references and CSR arrays of a property graph

To construct a DGL or PyTorch Geometric graph, it is essential to provide the features of nodes and edges, along with the connectivity structure of a graph, such as CSR arrays. In the case of DuckDB/DuckPGQ, node and edge features are stored in tables, which can be fetched as Numpy arrays or PyTorch Tensors using SQL queries.

However, as illustrated in Figure 2.4, the table reference relationship is an internal state of DuckPGQ and is invisible to the Python client, making it challenging to discern from which table we should collect the required data. Additionally, there is a lack of information

on which columns are utilized in a property graph. Furthermore, the CSR arrays are native C++ data, which is neither directly accessible nor readily retrievable for use in Python.

As a result, our initial step involves implementing several user-defined table functions and scalar functions to facilitate the retrieval of this metadata and the CSR arrays, as demonstrated in Table 4.1.

Name	Function Type	Description
get_csr_v	table function	retrieve the `v` array of a CSR
get_csr_e	table function	retrieve the `e` array of a CSR
get_csr_w	table function	retrieve the weight array of an edge table
csr_get_w_type	scalar function	get the type of the weight array
get_pg_vtablenames	table function	obtain the node table names
get_pg_etablenames	table function	obtain the edge table names
get_pg_vcolnames	table function	obtain the column names used by a node table
get_pg_ecolnames	table function	obtain the column names used by an edge table

Table 4.1: Implemented UDFs descriptions

These UDFs can be categorized into two groups. The first group, which includes `get_pg_vtablenames`, `get_pg_etablenames`, `get_pg_vcolnames`, and `get_pg_ecolnames`, is responsible for obtaining table and column references. The implementation for each UDF in this group is similar. We set the output `DataChunks` to point to the vectors of string names of each table and column, without the need for memory copies, thus enhancing the efficiency of the process. The cardinality is set to the size of corresponding C++ vectors.

The second group of UDFs comprises three table functions: `get_csr_v`, `get_csr_e`, and `get_csr_w`, along with a scalar function `csr_get_w_type`. This group handles the retrieval of CSR arrays. Unlike the reference relationships, each of the three arrays in a CSR exhibits distinct characteristics, as illustrated in Figure 2.5. The `v` array is a C-array of atomic values, the `e` array is a C++ vector containing 64-bit signed integers, and the weight array can either be a vector of 64-bit integers or a vector of doubles.

Retrieval of the `e` array follows a process similar to collecting table and column references in the first group: we configure the output `DataChunk` to point to the underlying data of the vector and set the cardinality to the size of the vector.

For the `v` array, since the number of elements in a C-array is unknown, we extended the CSR class definition in DuckPGQ by introducing a new member variable, `vsize`. During CSR creation, `vsize` is initialized to the number of nodes in a graph, and it is used to accurately set the cardinality of the result. Additionally, a type conversion

from atomic values to integers is necessary for the Python client to recognize the `v` array. As `atomic<int64_t>` shares the same memory layout as `int64_t` in C++, we employ `reinterpret_cast`. This compile-time directive instructs the compiler to treat a variable as if it had a user-specified type. Typically, it does not introduce extra overhead during runtime, as `reinterpret_cast` does not translate into additional CPU instructions. Finally, we configure the output `DataChunk` to be a C-array.

Regarding the array of edge weights, since it can either be a vector of integers or a vector of doubles, an additional function, `get_csr_w_type`, is introduced to determine the weight type. This information is crucial for correctly configuring the type of the result `DataChunk`. Subsequently, the `DataChunk` is configured to point to the underlying data of the weight vector.

Incorporating these UDFs has facilitated the seamless export of a property graph to the Python client. The overall process, as illustrated in Figure 4.1, involves a series of steps:

1. Obtaining referred table and column names. This initial step encompasses the first two stages illustrated in Figure 4.1, focusing on the identification and collection of relevant table and column names essential for subsequent operations. This step incorporates SQL commands integrated with function calls to the first group of UDFs.

2. Retrieving CSR arrays. Corresponding to the third stage in Figure 4.1, this phase entails retrieving CSR arrays as PyTorch Tensors. These tensors serve as the foundational building blocks for constructing a GNN graph. This step involves SQL commands combined with function calls to the second group of UDFs.

3. In the subsequent step, aligned with the fourth stage in Figure 4.1, the process involves retrieving values from related columns within tables. This retrieval is based on the results obtained in step 1. SQL commands are central to this step.

4. Constructing DGLGraph or PyTorch Geometric Data objects. The final step encompasses the construction of either a DGLGraph or a PyTorch Geometric Data object. These objects encapsulate the property graph in a format compatible with downstream operations. Python code forms the core of this step.

It is noteworthy that our implementation of user-defined table functions circumvents the need for memory copies, as the output `DataChunks` are manipulated using pointers. However, it is important to acknowledge that while retrieving query results as Numpy arrays or PyTorch Tensors, DuckDB necessitates the materialization of the required data,

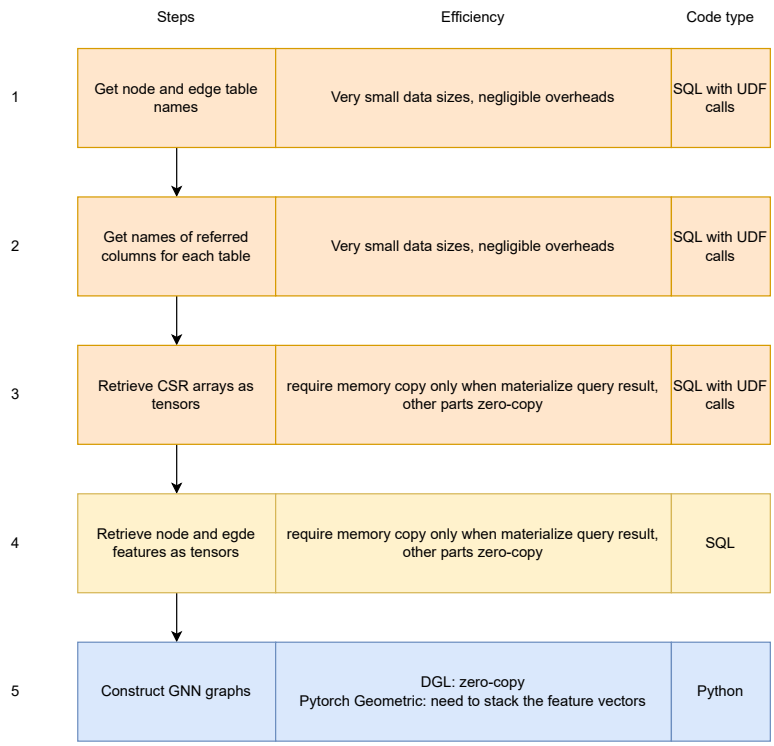

Figure 4.1: Steps to retrieve a property graph from Python clients

which may involve certain memory copies. It is essential to emphasize that once the query results are transformed into the desired Python objects, no additional memory copies are required for this conversion process.

4.2 Zero-copy optimization for retrieving CSR arrays

We first observe that the raw data buffer of a Numpy array shares the same underlying memory layout as a C++ vector (69). Moreover, when constructing PyTorch Tensors from Numpy arrays, no data copy occurs (70). These discoveries present an opportunity to facilitate memory sharing between DuckPGQ CSR arrays and GNN libraries, resulting in reduced memory usage and enhanced overall performance.

However, attaining this zero-copy optimization poses several challenges. Primarily, as an extension, DuckPGQ lacks direct access to Python clients, save for UDF calls, which can solely return data adhering to DuckDB-supported types. Additionally, DuckDB's Python APIs do not directly interact with extension modules. UDFs implemented in extension modules remain invisible and inaccessible (by function calls instead of SQL UDF calls) to Python clients. These challenges render it unfeasible to directly return a Python object

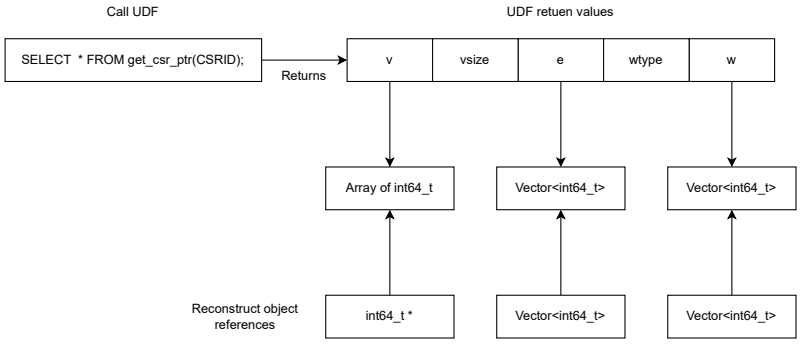

Figure 4.2: Enable access of CSR arrays from outside DuckPGQ

from UDF calls.

To realize the zero-copy optimization, we introduce a novel user-defined table function named `get_csr_ptr`. This function accepts an integer argument and consistently yields five 64-bit integers: the initiation address of the `v` array, the object addresses of the `e` and weight vectors, the `v` array's element count, and the weight vector's type.

Nonetheless, Python clients still cannot access CSR arrays with integer return values solely, owing to the absence of type and size information for each array. Furthermore, manipulating C++ native values proves unwieldy within Python. Consequently, we develop a separate C++ module that operates within the Python client process. This module is invoked after calling `get_csr_ptr`. Leveraging the five return values, it reconstructs object references to the CSR arrays stored within DuckPGQ, as exemplified in Figure 4.2. This reconstruction process first treats the returned address integers as pointers, and subsequently constructs object references of accurate types for each CSR array. This enables accessing CSR arrays from outside DuckPGQ.

To achieve the zero-copy transfer of CSR arrays to Tensors, we need to construct a PyTorch Tensor from the shared memory. To simplify this process without compromising performance, we utilize Numpy arrays as a bridge between the C++ native arrays and PyTorch tensors.

Firstly, capsules are implemented for each C-array. A capsule in Python serves as a mechanism for handling and passing C/C++ generic pointers and the data pointed to. It encapsulates a pointer and the associated destructor for memory deallocation. Subsequently, a Numpy array is formed by configuring the base pointer and size to appropriate values, as demonstrated in Figure 4.3. A capsule is provided during array construction to prevent Numpy from copying our CSR data into the newly formed array.

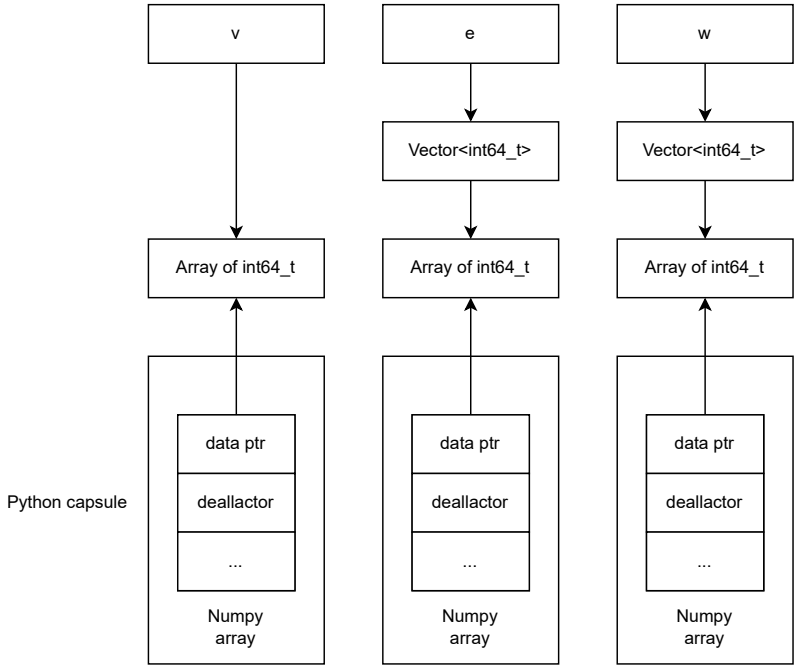

Figure 4.3: Memory sharing between Numpy ndarray and C array

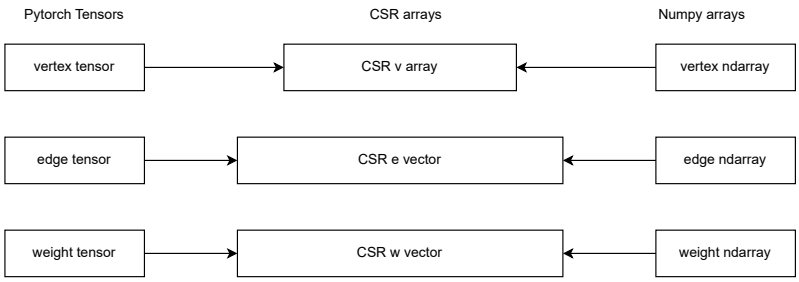

Figure 4.4: Memory sharing between Numpy ndarray and PyTorch tensors

Finally, we create tensors from the Numpy arrays. As depicted in Figure 4.4, both the tensors and the Numpy arrays share the same underlying memory for data, ensuring efficient zero-copy transfer.

It is crucial to emphasize that we assume memory management is handled within DuckPGQ, implying that PyTorch Tensors become invalidated when an explicit call to the CSR deletion function is made.

We acknowledge that memory sharing can be susceptible to errors, as any slight change in data by one side will impact the other side. To address this concern, both the original method and the optimized approach for obtaining the CSR arrays are made available in the final implementation. This flexibility allows users to choose the approach that best

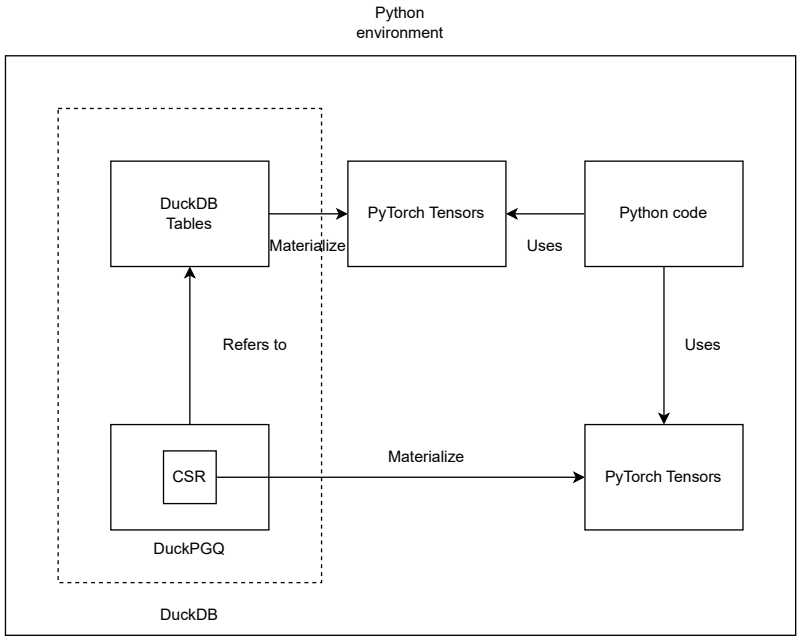

Figure 4.5: Architectural overview of fetching a property graph as a GNN graph with copying CSR arrays

aligns with their specific use cases and requirements.

4.3 An overview of the architectural design

In this section, we present an overview of our methods for converting a DuckPGQ property graph into a GNN graph. We discuss the architecture of our implementation in the following paragraphs.

Figure 4.5 illustrates the process of extracting a property graph, where CSR arrays are acquired through user-defined table functions. All components are situated within the Python environment. The box outlined with a dashed line represents a DuckDB connection. Within this connection, DuckPGQ is fully loaded and retains the CSR arrays it generated, along with references to DuckDB tables. Prior to constructing a GNN graph, the CSR arrays are obtained using our user-defined table functions. The relevant data from DuckDB tables is retrieved using SQL `SELECT` commands. After retrieval, the query results are materialized, necessitating memory copies. Subsequently, PyTorch Tensors are constructed based on the materialized results. The Python code encompasses the user program responsible for creating GNN graphs from these acquired Tensors.

Figure 4.6 outlines our optimized approach to property graph retrieval. While several

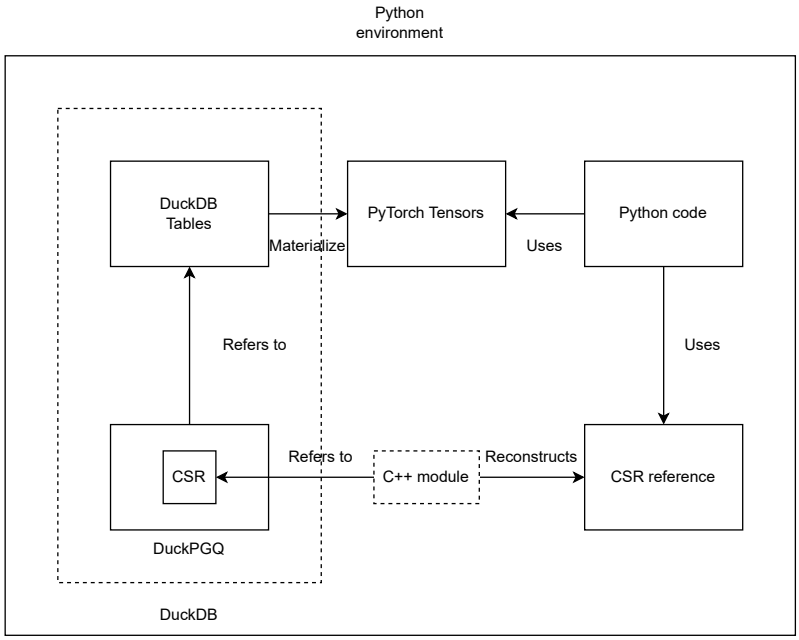

Figure 4.6: Architectural overview of fetching a property graph as a GNN graph without copying CSR arrays

steps resemble those in Figure 4.5, such as fetching DuckDB tables as PyTorch Tensors, there is a notable distinction in how CSR arrays are obtained. In this approach, we introduce a C++ module that enables access to CSR arrays within DuckPGQ from outside DuckDB by reconstructing the array references in the Python environment. No memory copy is involved in this process.

4.4 Construction of GNN graphs

4.4.1 DGL

The code snippets for building a DGL graph from the retrieved values can be found in Listing 4.1. In this snippet, we initialize a graph object by providing the CSR representation of the edge connectivity structure. Then, we proceed to set the node and edge features of the graph, establishing a complete representation of the property graph for further analysis and processing.

4.4.2 PyTorch Geometric

Creating a PyTorch Geometric graph involves a different approach. As discussed in a previous chapter, PyTorch Geometric requires the retrieved 1-dimensional node and edge

```
import dgl
import torch
import duckdb

# Setup DuckPGQ and collect necessary data
con = duckdb.connect()
csrv, csre, node_features, edge_features = ...
# Initialize a graph object
g = dgl.graph(('csr', (csrv, csre, [])))
# Set the node features, reshaping is necessary
for feature_name, feature in node_features:
    g.ndata[feature_name] = feature.reshape((feature.shape[0], 1))
# Set the edge features, reshaping is necessary
for feature_name, feature in edge_features:
    g.edata[feature_name] = feature.reshape((feature.shape[0], 1))
```

Listing 4.1: Creating a DGLGraph object from retrieved values

features to be stacked as 2-dimensional feature matrices for each type of nodes or edges, which may involve unavoidable memory copies. Additionally, PyTorch Geometric does not natively support creating a graph from CSR arrays. We devised two methods to address these challenges and represent a property graph as a PyTorch Geometric graph.

The first approach involves utilizing the PyTorch Sparse library, an optional dependency for PyTorch Geometric. This library introduces the `SparseTensor` class that supports the CSR format. We provide code snippets for constructing a PyTorch Geometric graph with `SparseTensor` in Listing 4.2.

The second approach involves converting the CSR arrays to a COO representation. The conversion algorithm is detailed in Listing 4.3. This conversion can be efficient, as it requires constructing only the source array.

Moreover, this algorithm lends itself to straightforward parallelization, which can further enhance performance. An illustration of the parallelized version of the CSR to COO conversion is provided in Listing 4.4.

4.5 The helper library duckpgq.py

As shown in Figure 4.1, the process of retrieving a property graph as a GNN graph involves several crucial steps. However, using only these UDFs for obtaining a property graph through the optimized zero-copy method is not feasible due to the necessity of reconstructing object references and creating Numpy arrays.

```
import torch
from torch_geometric.data import Data
from torch_geometric.typing import SparseTensor
import duckdb

# Setup DuckPGQ and collect necessary data
con = duckdb.connect()
csrv, csre, node_features, edge_features = ...
# Construct a SparseTensor object
adj_t = SparseTensor(rowptr = csrv, col = csre)
# we stack the 1-dimensional node features as a two-dimensional feature
    matric
datax = torch.stack(list(node_features.values()), dim = 1)
# we stack the 1-dimensional edge features as a two-dimensional feature
    matric
edge_attr = torch.stack(list(edge_features.values()), dim = 1)
# Construct a PyTorch Geometric graph
data = Data(x = datax, edge_attr = edge_attr)
# Specify the connectivity structure
data.adj_t = adj_t
```

Listing 4.2: Creating a PyTorch Geometric graph object from retrieved values

```
int64_t* convert_csr_to_coo(int64_t *v, int64_t size) {
    // v is the vertex array of a CSR, size is the number of nodes plus 1
    // The last element of the v array is the number of edges
    // allocate the result source array
    int64_t* source = (int64_t *)malloc(sizeof(int64_t) * v[size-1]);
    // indexing variables
    int64_t i = 0;
    int64_t z = 0;
    int64_t c = 0;
    // Fill the source array
    for(i = 0; i < size-1; i++) {
        int64_t num_edges_for_node_i = v[i+1] - v[i];
        for(z = 0; z < num_edges_for_node_i; z++) {
            source[c++] = i;
        }
    }
    return source;
}
```

Listing 4.3: Convert CSR to COO

```
void worker(void *args) {
    // ... prepare arguments
    int64_t *source, int64_t i, int64_t start, int64_t end = ...;
    for(int64_t z = start; z < end; z++) {
        source[z] = i;
    }
}
int64_t* convert_csr_to_coo(int64_t *v, int64_t size) {
    // v is the vertex array of a CSR, size is the number of nodes plus 1
    // The last element of the v array is the number of edges
    // allocate the result source array
    int64_t* source = (int64_t *)malloc(sizeof(int64_t) * v[size-1]);
    int64_t i = 0;
    int64_t start = 0;
    int64_t end = 0;
    // ... fill the source array in parallel
    pthread_t threads[NUM_THREADS];
    for(int i = 0; i < NUM_THREADS; i++) {
        //Prepare the arguments
        void *args = ...;
        pthread_create(&threads[i], NULL, worker, args);
    }
    // ... other code
    return source;
}
```

Listing 4.4: Convert CSR to COO with multi-threading

Additionally, some UDFs in DuckPGQ assume the existence of the pseudo-column `rowid`. Consequently, these user-defined functions cannot directly operate on Python objects like Pandas `DataFrames`. One possible solution to address this issue is to add the `rowid` column during the scan. However, this approach might lead to different results when users query a Python object without specifying the column names, such as `SELECT * FROM DataFrame`, since an unexpected column appears in the query result.

To ensure seamless integration and eliminate the need for additional effort when calling a UDF on Python objects, as well as to simplify the interface and reduce the amount of code required for calling a UDF or retrieving a property graph, we have developed the Python helper library `duckpgq.py`.

`duckpgq.py` provides the following functionalities:

1. It sets up the DuckPGQ extension module and ensures its complete loading.

2. A class `Connection` that mimics the original DuckDB Python API, serving as a drop-in replacement for DuckDB with additional features.

3. Several high-level functions streamline the process of obtaining CSR arrays, property graphs, and constructing GNN graphs. A complete list of implemented functions and their descriptions are presented in Table 4.2.

4. A mechanism based on function decorating that ensures a seamless integration. The `rowid` column is automatically added and removed on the fly as needed. Different Python objects have varying types of the `rowid` column, as illustrated in Table 4.3. An example of calling the CSR creation UDFs over Pandas DataFrames, without any extra effort, is provided in Listing 4.5.

5. Functions to save a GNN graph as DuckDB tables. For node features and edge features, a table is created for each type of graph element. To store edge information, the COO representation of the graph connectivity structure is transformed into DuckDB tables.

Function	Description
`get_csr`	Retrieve a CSR representation from DuckPGQ as PyTorch Tensors through zero-copy.
`get_pyg`	Fetch a property graph and the CSR arrays as PyTorch Geometric graph.
`get_dgl`	Fetch a property graph and the CSR arrays as `DGLGraph`.
`create_csr`	Create a CSR representation of a given graph.
`save_dgl`	Save a `DGLGraph` as DuckDB tables.

Table 4.2: Functionalities provided by the `duckpgq.py` Python helper library.

Table 4.3: rowid types for different Python objects

Object Type	Rowid Type
Polars DataFrame	Polars row count
Numpy arrays	Numpy.arange
PyArrow table	PyArrow array
Pandas DataFrame	Pandas.index

```
# We use the helper library
import duckpgq
import pandas as pd

# Connect to an in-memory database
con = duckpgq.connect()

# Prepare the DataFrames
edgedf, nodedf = ...
# Prepapre node tables
con.sql("CREATE TABLE nodes ...")
# Prepapre edge tables
con.sql("CREATE TABLE edges ...")

# Create CSR over DuckDB Tables
con.create_csr("edges", "src", "dst", "nodes", "id")

# Create CSR over Pandas DataFrame
con.create_csr(edgedf, "src", "dst", nodedf, "id")
```

Listing 4.5: CSR creation over DuckDB tables and Pandas DataFrame

5

Evaluation

In this section, we present a series of experiments conducted to evaluate the performance of our implementation. We begin by describing the experimental environment, datasets, and parameter setup. Subsequently, we present and discuss the benchmark tasks and their results.

5.1 Experimental Setup

In our experiments, we utilized the LDBC Social Network Benchmark (LDBC SNB) data. LDBC SNB offers datasets with different scale factors, quantifying the size and scale of each dataset. For our benchmarks, we employed scale factors 1, 3, 10, 40, and 100. The number of nodes and edges for each dataset is shown in Table 5.1.

Scale factor	Number of nodes	Number of edges
1	10620	219450
3	25870	668431
10	70800	2304951
30	175950	6880584
100	487700	23116805

Table 5.1: Number of nodes and edges for different scale factors

Each benchmark task was executed within a container environment. The details of the container and host machine configurations can be found in Table 5.2 and Table 5.3, respectively. Additionally, Table 5.4 provides insights into the parameter setups used in the experiments.

Operating system	Fedora Linux 38
CPU	Intel Core i7-10710U
Number of cores	6
Number of threads	12
RAM	16GB

Table 5.2: Host machine specification

Container engine	Podman
Guest operating system	Fedora Linux 38
Compiler	GCC 13.1.1 20230426

Table 5.3: Container environment specification

5.2 Evaluation of CSR creation over different Python objects and DuckDB tables

In this section, we conduct a performance evaluation of CSR creation using various Python objects, including Pandas DataFrame with Numpy backend, Polars DataFrame, and Pyarrow Table, while also comparing the results with running the UDFs over DuckDB tables. The addition of the `rowid` column to these Python objects has been made to ensure smooth evaluation. The CSR creation query is generated from Listing 5.1, wherein template variables, enclosed in curly braces, are replaced with appropriate table, data frame, or column names. The results for scale factors 1, 3, 10, 30, and 100 are shown in Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, and Figure 5.5, respectively.

Based on the above results, we draw the following conclusions:

First, for CSR creation, Pandas DataFrames exhibit similar performance to DuckDB tables.

Second, when dealing with small-sized datasets, such as when the scale factor is 1 or 3, creating a CSR representation over DuckDB tables or Pandas DataFrames significantly

DuckDB version	0.8.2-dev
Build parameters	release_python
In-memory database	true
Allow unsigned extensions	true

Table 5.4: DuckDB and DuckPGQ specification

```
SELECT  CREATE_CSR_EDGE(
            {csr_id},
            (SELECT count(srctable.{src_ref_col}) FROM {src_ref_table} srctable),
            CAST (
                (SELECT sum(CREATE_CSR_VERTEX(
                            {csr_id},
                            (SELECT count(srctable.{src_ref_col}) FROM {src_ref_table} srctable),
                            sub.dense_id,
                            sub.cnt)
                            )
                FROM (
                    SELECT srctable.rowid as dense_id, count(edgetable.{src_key}) as cnt
                    FROM {src_ref_table} srctable
                    LEFT JOIN {edge_table} edgetable ON edgetable.{src_key} = srctable.{src_ref_col}
                    GROUP BY srctable.rowid) sub
                )
            AS BIGINT),
            srctable.rowid,
            dsttable.rowid,
            edgetable.rowid) as temp
        FROM {edge_table} edgetable
        JOIN {src_ref_table} srctable on srctable.{src_ref_col} = edgetable.{src_key}
        JOIN {dst_ref_table} dsttable on dsttable.{dst_ref_col} = edgetable.{dst_key} ;
```

Listing 5.1: Template for generating CSR creation code

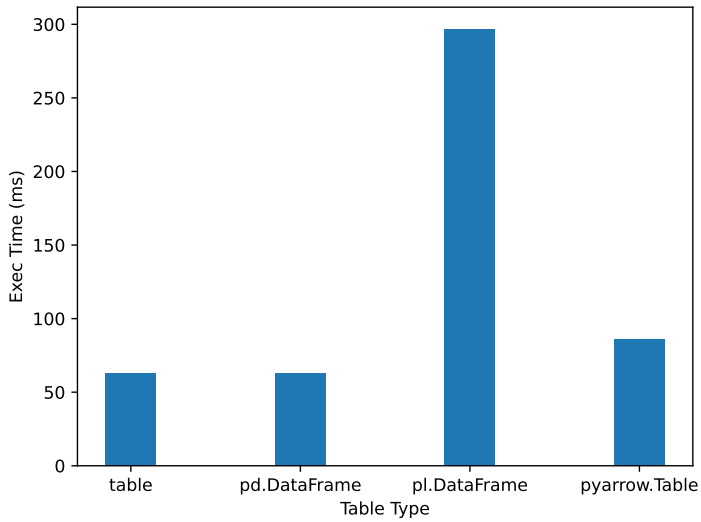

Figure 5.1: Execution time of CSR creation over different Python objects and DuckDB tables, SF = 1

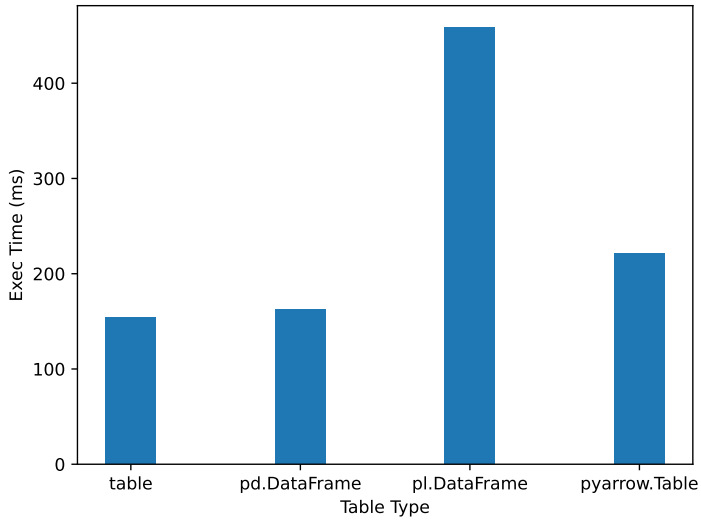

Figure 5.2: Execution time of CSR creation over different Python objects and DuckDB tables, SF = 3

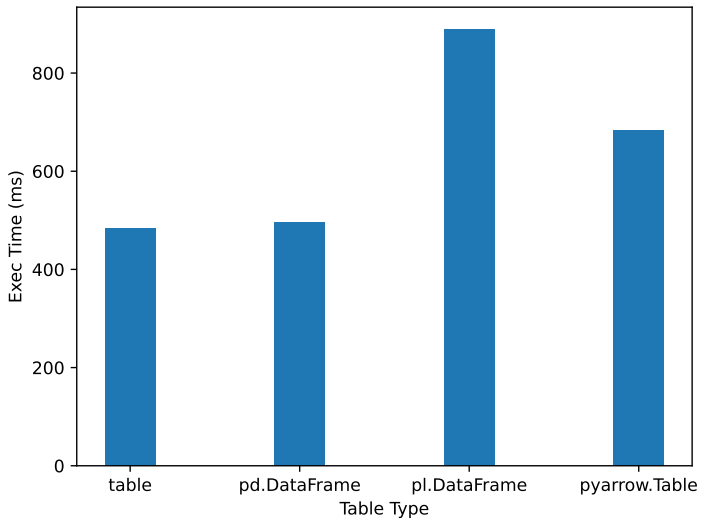

Figure 5.3: Execution time of CSR creation over different Python objects and DuckDB tables, SF = 10

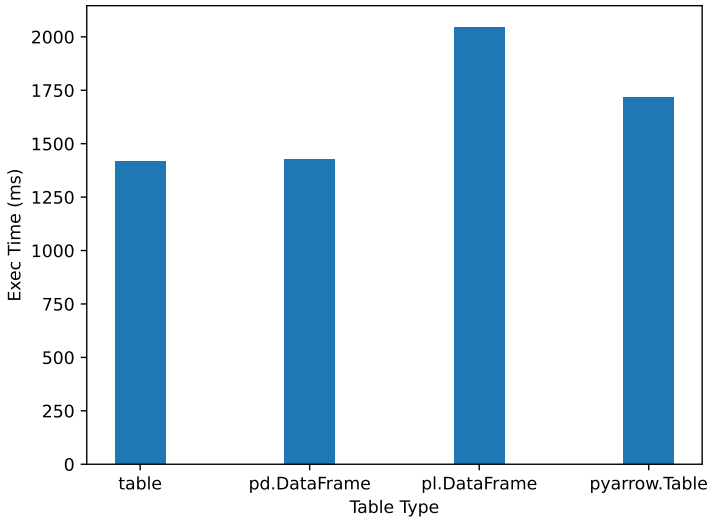

Figure 5.4: Execution time of CSR creation over different Python objects and DuckDB tables, SF = 30

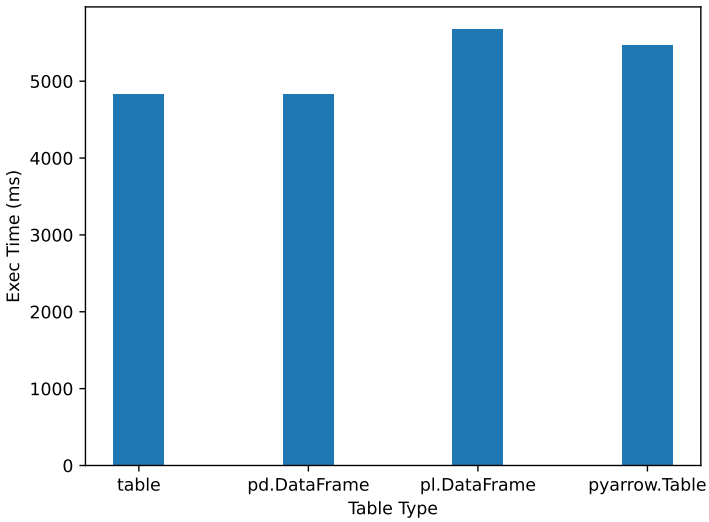

Figure 5.5: Execution time of CSR creation over different Python objects and DuckDB tables, SF = 100

outperforms Polars DataFrames. However, as the scale factor increases, the performance difference becomes less pronounced.

5.3 Evaluation of exporting a DuckPGQ property graph as a DGL graph

In Figure 5.6, we present the execution time for retrieving property graphs from DuckPGQ as DGL graphs, considering various scale factors.

To gain deeper insights into the time-consuming steps, we conducted an extensive performance analysis with a scale factor of 100, as shown in Figure 5.7. As discussed in Section 4.1, the steps involving the retrieval of table and column names incur negligible overhead.

The two most time-consuming steps are the retrieval of CSR arrays and the fetching of edge data. For unweighted edges, the CSR retrieval involves two steps, with each step responsible for retrieving either the `v` or `e` arrays. As highlighted in Figure 5.8, the collection of the `e` array constitutes the majority of the time.

As discussed in Section 4.1, DuckDB engages in the process of materialization and the construction of Numpy arrays before fetching query results as PyTorch Tensors. The breakdown of execution times for each step can be found in Table 5.5, where it becomes

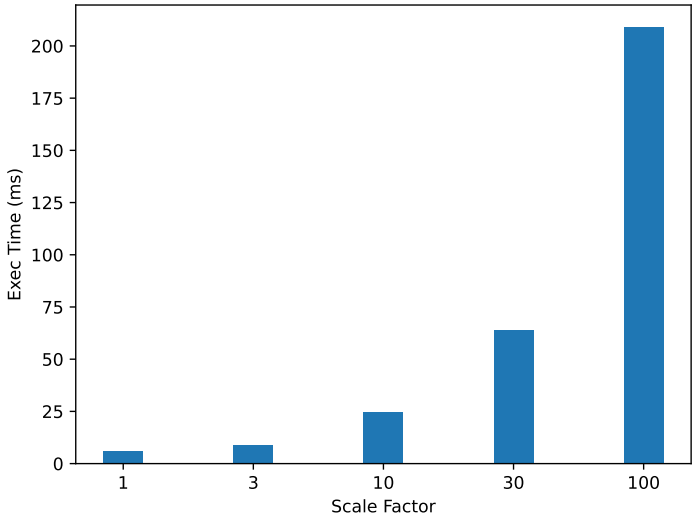

Figure 5.6: Execution time of retrieving property graphs as DGL graphs with various scale factors

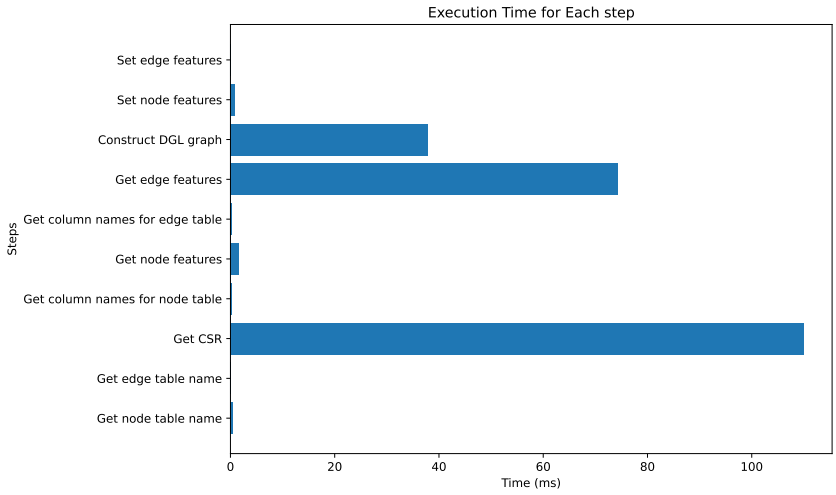

Figure 5.7: Execution time of each step for retrieving property graph as GNN graph

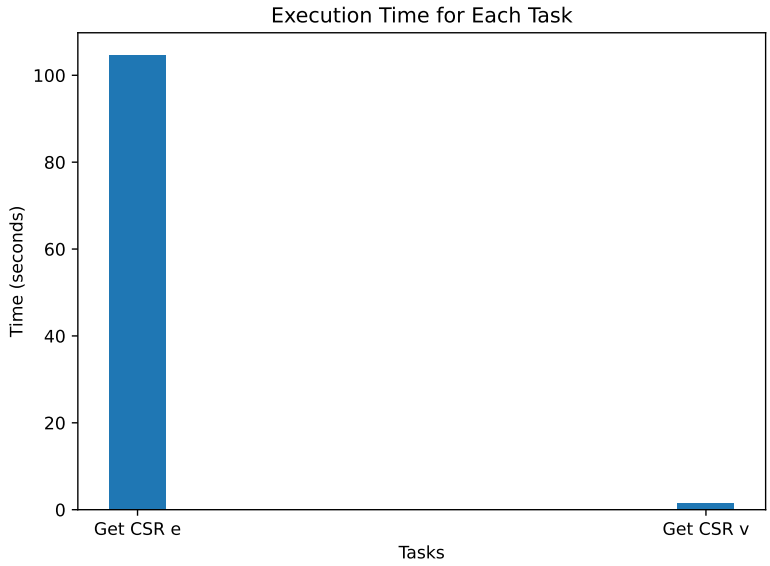

Figure 5.8: Execution time of getting the `v` and `e` array of CSR

evident that the materialization phase consumes the majority of the time. This is shown in the `Total time (ms)` column of Table 5.5, which measures the execution time for the entire task of obtaining the `e` array or edge features. It's important to note that this task encompasses additional steps, including SQL command parsing, which contributes to the `Total time (ms)` being larger than the sum of the time required for materialization and array construction.

Table 5.5: Execution Times for each step to fetch query result as PyTorch tensors

Task	Total time (ms)	Materialization (ms)	Construct Array (ms)
Get CSR e	112.73	44.02	0.083
Get edge features	75.16	48.63	0.087

To enhance the performance of exporting property graphs, as discussed in Section 4.2, we proposed an approach that involves memory sharing between DuckPGQ and the Python client for CSR arrays, eliminating the need for data copying. As seen in Figure 5.9, the evaluation time is significantly improved, and Figure 5.10 demonstrates that the overhead of fetching a CSR is now negligible.

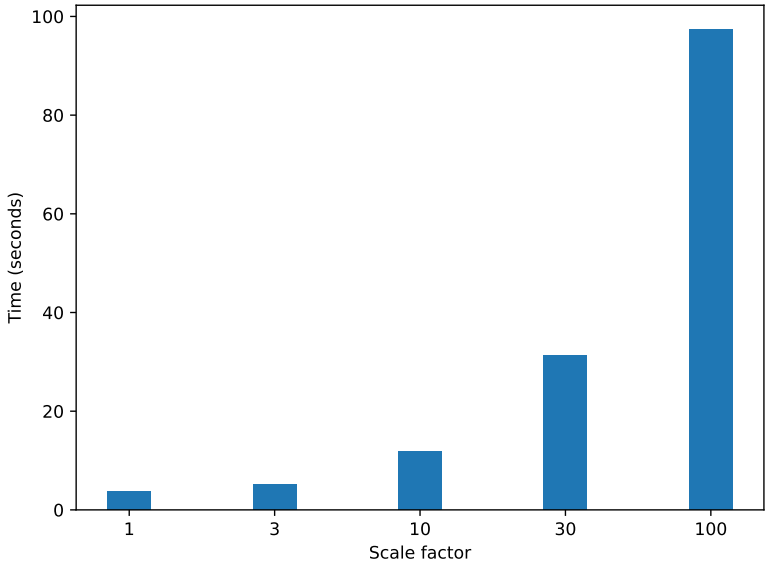

Figure 5.9: Execution time of retrieving property graphs as DGL graphs

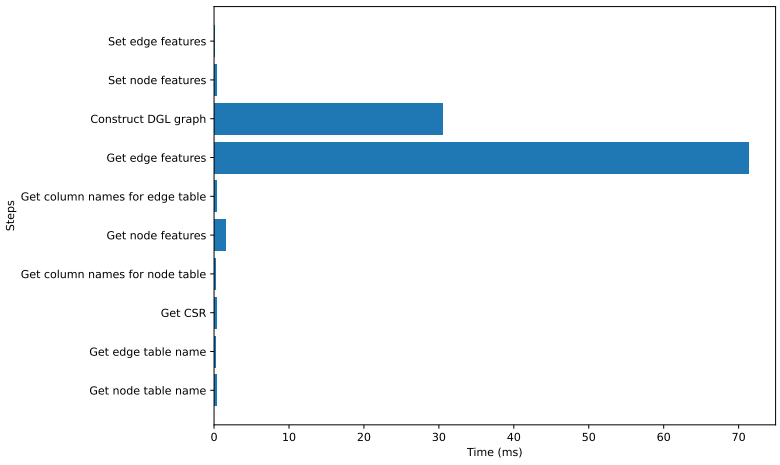

Figure 5.10: Execution time of each step for retrieving property graph as DGL graph

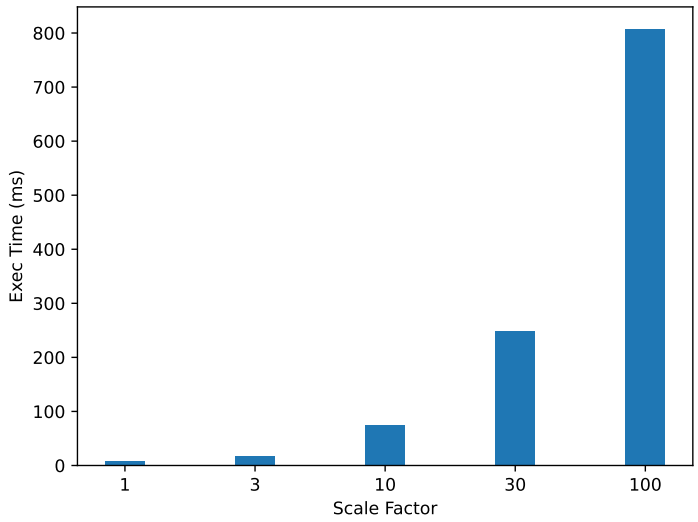

Figure 5.11: Execution time of retrieving property graphs as PyTorch geometric graphs using SparseTensor

5.4 Evaluation of exporting a DuckPGQ property graph as a PyTorch geometric graph

In this section, we evaluated the performance of retrieving a DuckPGQ property graph as a PyTorch tensor graph. As discussed in Section 4.4, we devised two methods for achieving this goal. The first method utilized the `SparseTensor` class, while the second method involved converting the CSR arrays to a COO representation. The evaluation results of both approaches are presented in Figure 5.11 and Figure 5.12, respectively. For these experiments, we employed the zero-copy implementation to enhance performance. Additional insights into the performance analysis can be found in Figure 5.13 and Figure 5.14.

Significantly, the second method, which leveraged an optimized and parallelized conversion algorithm, demonstrated notably faster execution times compared to the first method that relied on the `SparseTensor` class.

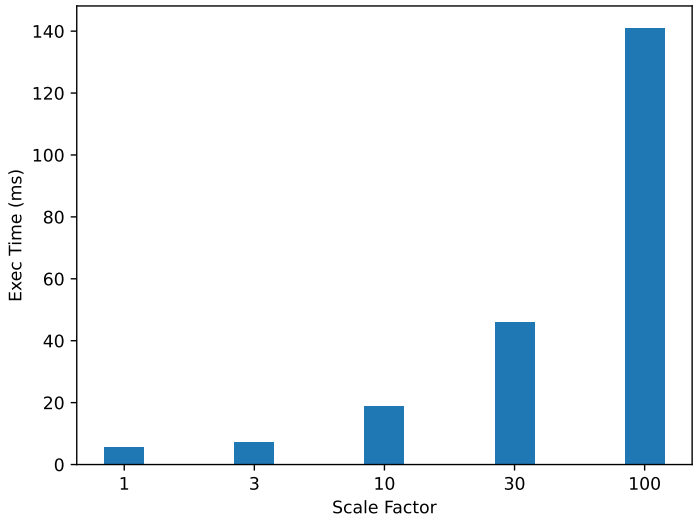

Figure 5.12: Execution time of retrieving property graphs as PyTorch geometric graphs using COO

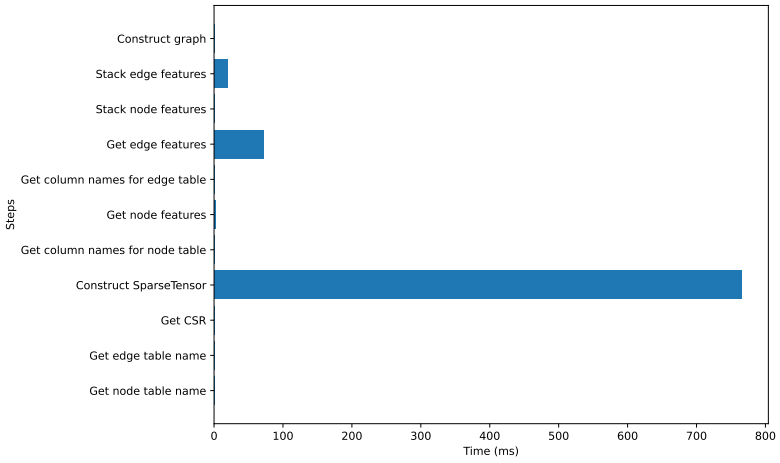

Figure 5.13: Execution time of each step for retrieving property graph as PyTorch geometric graph using SparseTensor

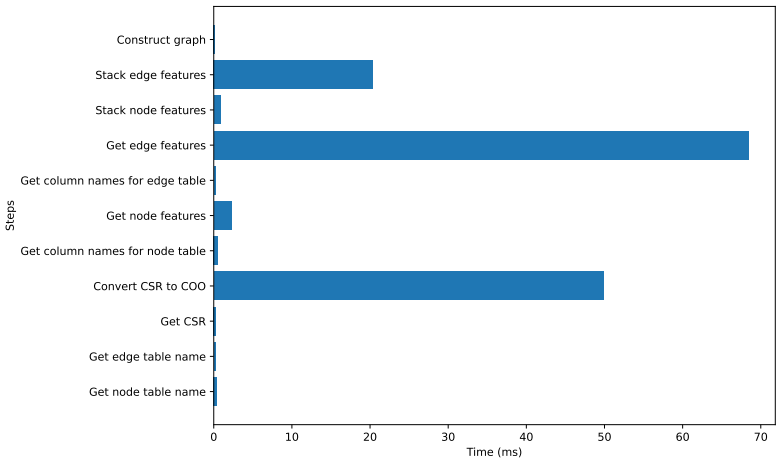

Figure 5.14: Execution time of each step for retrieving property graph as PyTorch geometric graph using COO

6

Future Work

This project enables DuckDB/DuckPGQ to efficiently and seamlessly connect with graph neural network pipelines. The implemented process is functional, and several optimizations have been made to enhance performance. However, there remain possibilities for further improving both performance and functionality. In this chapter, we explore potential areas for future work and discuss potential improvements that can be made to enhance the integration even further.

6.1 A tighter integration with GNN libraries

By utilizing the implemented UDFs and the helper library, we can efficiently retrieve property graphs and seamlessly use them in GNN pipelines. Additionally, we provide functions to save a GNN graph as DuckPGQ tables. During these processes, several optimizations are possible to further enhance the integration.

1. Enable appending an existing CSR to the CSR list in DuckPGQ: While the memory of each CSR array is shared between DuckPGQ and GNN libraries, modifying the internal ID-to-CSR mapping of DuckPGQ from Python clients is currently not feasible. Consequently, when storing a GNN graph in DuckPGQ, the edges are saved as tables, and the CSR arrays are re-created when the related UDFs are called, even though these sparse representations already exist in our GNN graphs. Implementing a mechanism to directly utilize the existing CSR arrays in DuckPGQ would significantly improve efficiency.

2. Add support for COO sparse representation of graphs: Presently, DuckPGQ only includes CSR creation UDFs. However, incorporating COO arrays instead of CSR

arrays, as seen in the PyTorch Geometric graph construction, leads to more compact code and potentially higher performance. Moreover, to store the graph connectivity structure in DuckDB tables, a COO format is required due to the need for the cardinality of each column in a table. Therefore, adding COO sparse representation creation functionality will simplify the conversions between property graphs and GNN graphs.

3. Interact with GNN models and perform inferences using SQL statements with potential UDF calls: While our implementation focuses on an efficient export of DuckPGQ property graphs to GNN graphs, further processes and analysis are outside our current scope. However, the ability to train models and make predictions based on provided configurations and data using SQL statements with UDF calls offers several advantages. Firstly, it unifies the interfaces, as all operations can be expressed with SQL. Additionally, deeper integration between GNN pipelines and SQL enables querying and relational analysis over inference results, enhancing the overall usability and versatility of the solution.

6.2 A more complete helper library

DuckPGQ is a relatively new extension to DuckDB, and it is actively being developed with a focus on continuously enhancing its graph analysis capabilities. As part of this ongoing development, new features and UDFs are regularly being added to further enrich its functionality. Presently, DuckPGQ supports various graph-related operations, such as finding shortest paths, checking reachability, and graph pattern matching, among others.

However, in our current implementation, only the high-level function for CSR creation is included in `duckpgq.py`, which somewhat limits the applicability of our helper library. To maximize the usefulness and completeness of our library, we aim to provide additional high-level functions that encompass a broader range of graph analysis tasks. By expanding the functionality, users will have access to a more comprehensive set of tools and capabilities, further simplifying and streamlining the process of working with property graphs and GNN pipelines in the context of DuckDB and DuckPGQ.

6.3 `FeatureStore` and `GraphStore` for PyTorch Geometric

As discussed in Chapter 2, PyTorch Geometric introduced two abstract classes, `FeatureStore` and `GraphStore`, starting from version 2.2. These classes enable the

utilization of graph databases as remote storage backends, facilitating the training of GNN models on graphs that exceed the main memory capacity of a machine. This presents a new opportunity for integrating DuckDB/DuckPGQ with PyTorch Geometric.

However, despite the advantages offered by `FeatureStore` and `GraphStore`, we opted for a different approach in implementing our integration. One primary reason is the lack of support for homogeneous graphs. Additionally, the absence of support for edge features further limits the suitability of these classes for our specific use case. Furthermore, the APIs of the two abstract classes are still under development and not yet stable.

As a result, we have decided to postpone enabling DuckDB as PyTorch Geometric remote backends until this functionality becomes stable and ready for practical use.

7

Conclusion

In this project, we have successfully developed an efficient and seamless integration between DuckDB/DuckPGQ and the GNN libraries PyTorch Geometric and Deep Graph Library. The implementation includes a group of user-defined table functions and scalar functions, enabling the smooth export of a property graph as GNN graphs. To enhance performance, several optimizations have been applied, such as memory-sharing of CSR arrays to avoid unnecessary data copying. Additionally, a conversion from CSR to COO representation has been implemented to improve the efficiency of constructing PyTorch Geometric graphs. Furthermore, we have created a helpful companion library to ensure a seamless integration experience.

Overall, this project has effectively addressed our pre-defined research questions, achieving our objectives of enabling efficient graph analysis with DuckDB/DuckPGQ and seamless integration with popular GNN libraries.

7.1 RQ1: What information is required to retrieve a DuckPGQ property graph from Python clients?

To construct a GNN graph from a property graph, several pieces of information are required. First, we need the features of each node and edge, which are stored as tables in DuckDB. Additionally, the connectivity structure, represented as CSR arrays, is a crucial component and part of the internal state of DuckPGQ. This internal state also includes essential information about table and column references within the property graph.

However, due to the restricted access of the Python client to the internal data of DuckPGQ, we require the implementation of several User-Defined Functions (UDFs) to facilitate the retrieval of the necessary information. These UDFs play a vital role in the

process, including table functions responsible for collecting the referred table and column information, as well as functions designed to obtain the CSR arrays needed for constructing the GNN graph effectively.

7.2 RQ2: With the required information, what steps are needed to retrieve a DuckPGQ property graph from Python clients?

Retrieving a DuckPGQ property graph from Python clients involves several essential steps. First, we initiate the process by calling user-defined table functions with the property graph name as input. These functions are responsible for acquiring the table names that serve as edge tables and nodes, respectively, in the property graph. Additionally, we need to identify which columns of each table are used in the property graph.

Next, we retrieve the node and edge features from the corresponding tables and columns and obtain the CSR arrays associated with the property graph.

Finally, armed with all the collected data, we construct the GNN graph.

7.3 RQ3: Among these needed steps, is it possible to avoid data copies with memory-sharing?

When retrieving feature values of edges and nodes from DuckDB tables, it is not feasible to avoid memory copies. This is primarily because DuckDB tables store their data in vectors of fixed length, which may not be contiguous in memory. As our performance analysis demonstrated, a materialization step is required before we can fetch the query result as Python objects, leading to memory copies and potential performance overhead.

Conversely, each CSR array in DuckPGQ is stored in continuous memory. This layout allows for sharing data with GNN libraries, as they also utilize a C-array-like memory structure. To achieve CSR arrays without memory copying, we implemented additional functions, as discussed in Chapter 4. Through memory sharing, we successfully attained higher performance levels and reduced memory usage.

7.4 RQ4: How to construct a PyTorch Geometric graph from CSR representation of connectivity structures?

PyTorch Geometric constructs graphs using COO arrays by default. To utilize the CSR representation, we transformed the CSR arrays into `SparseTensor` objects.

Alternatively, we can adopt a different approach by converting the CSR arrays to COO format. As demonstrated in Chapter 5, implementing a parallel CSR-to-COO algorithm significantly improves performance.

7.5 RQ5: How to run UDFs in DuckPGQ directly over Python objects?

DuckDB offers a functionality called replacement scan, which allows SQL queries with UDF calls to be executed over Python objects. However, some UDFs implemented in DuckPGQ rely on the existence of the `rowid` column, which is available in DuckDB tables but not in Python objects.

To ensure seamless integration, wherein UDFs can be executed over Python objects without requiring extra effort, and to avoid altering the behavior of the replacement scan, we adopted a mechanism that dynamically adds and deletes the `rowid` column to Python objects as needed. The efficacy of our approach was evaluated in Chapter 5.

7.6 RQ6: How to simplify the process of fetching a property graph as a GNN graph and the use of user-defined functions in DuckPGQ?

To enhance the ease of use and provide a more user-friendly interface for our integration, we have developed the helper library `duckpgq.py`. This library offers several high-level functions that simplify the process of retrieving a property graph, as demonstrated in Appendix A. By utilizing this helper library, the number of lines of code required to retrieve a property graph from Python clients is significantly reduced.

References

[1] Jonathan L. Gross, Jay Yellen, Lowell W. Beineke, and Robin J. Wilson. **Introduction to Graphs**. In Jonathan L. Gross and Jay Yellen, editors, *Handbook of Graph Theory*, Discrete Mathematics and Its Applications, pages 1–55. Chapman & Hall / Taylor & Francis, 2003. 1

[2] Shazia Tabassum, João Gama, Paulo J. Azevedo, Mário Cordeiro, Carlos Martins, and Andre Martins. **Social network analytics and visualization: Dynamic topic-based influence analysis in evolving micro-blogs**. *Expert Syst. J. Knowl. Eng.*, **40**(5), 2023. 1

[3] Preya Shah, Arian Ashourvan, Fadi Mikhail, Adam Pines, Lohith Kini, Kelly Oechsel, Sandhitsu R Das, Joel M Stein, Russell T Shinohara, Danielle S Bassett, et al. **Characterizing the role of the structural connectome in seizure dynamics**. *Brain*, **142**(7):1955–1972, 2019. 1

[4] Zhe Chen, Yuehan Wang, Bin Zhao, Jing Cheng, Xin Zhao, and Zongtao Duan. **Knowledge Graph Completion: A Review**. *IEEE Access*, **8**:192435–192456, 2020. 1

[5] Brian Wheatman and Helen Xu. **Packed Compressed Sparse Row: A Dynamic Graph Representation**. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7. IEEE, 2018. 1

[6] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh M. Basalamah. **A Survey of Shortest-Path Algorithms**. *CoRR*, **abs/1705.02044**, 2017. 1

[7] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. **Graph**

neural networks: A review of methods and applications. *AI Open*, **1**:57–81, 2020. 1, 10

[8] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. **Graph Convolutional Networks with Markov Random Field Reasoning for Social Spammer Detection**. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1054–1061. AAAI Press, 2020. 1

[9] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin A. Riedmiller, Raia Hadsell, and Peter W. Battaglia. **Graph Networks as Learnable Physics Engines for Inference and Control**. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, **80** of *Proceedings of Machine Learning Research*, pages 4467–4476. PMLR, 2018. 1

[10] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. **Protein Interface Prediction using Graph Convolutional Networks**. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6530–6539, 2017. 1

[11] Wenlong Liao, Birgitte Bak-Jensen, Jayakrishnan Radhakrishna Pillai, Yuelong Wang, and Yusen Wang. **A Review of Graph Neural Networks and Their Applications in Power Systems**. *CoRR*, **abs/2101.10025**, 2021. 1

[12] Thomas N. Kipf and Max Welling. **Semi-Supervised Classification with Graph Convolutional Networks**. *CoRR*, **abs/1609.02907**, 2016. 1

[13] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. **Graph Attention Networks**. *CoRR*, **abs/1710.10903**, 2017. 1

[14] Jihun Oh, Kyunghyun Cho, and Joan Bruna. **Advancing GraphSAGE with A Data-Driven Node Sampling**. *CoRR*, **abs/1904.12935**, 2019. 1

[15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. **Gated Graph Sequence Neural Networks**. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. 1

[16] Matthias Fey and Jan Eric Lenssen. **Fast Graph Representation Learning with PyTorch Geometric**. *CoRR*, **abs/1903.02428**, 2019. 1, 11

[17] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. **Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks**. *arXiv preprint arXiv:1909.01315*, 2019. 1, 11

[18] Pytorch Geometric. **Data Handling of Graphs**. `https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html#data-handling-of-graphs`. Accessed on August 2, 2023. 1

[19] Renzo Angles. **The Property Graph Database Model**. In Dan Olteanu and Barbara Poblete, editors, *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, **2100** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 1, 7

[20] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. **KÙZU Graph Database Management System**. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. 2, 25

[21] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. **Graph Pattern Matching in GQL and SQL/PGQ**. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022. 2, 8, 9, 14

[22] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A. Boncz. **DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS**. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. 2, 14

[23] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019. 2, 13

[24] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. **A Comprehensive Survey on Graph Neural Networks**. *IEEE Trans. Neural Networks Learn. Syst.*, **32**(1):4–24, 2021. 10

[25] Maciej Besta and Torsten Hoefler. **Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis**. *CoRR*, **abs/2205.09702**, 2022. 10

[26] Pytorch Geometroc. **Scaling Up GNNs via Remote Backends**. `https://pytorch-geometric.readthedocs.io/en/latest/advanced/remote.html`. Accessed on August 2, 2023. 13

[27] LDBC Council. **The Linked Data Benchmark Council**. `https://ldbcouncil.org/`. Accessed on August 2, 2023. 14

[28] Peter A. Boncz. **LDBC: benchmarks for graph and RDF data management**. In Bipin C. Desai, Josep Lluís Larriba-Pey, and Jorge Bernardino, editors, *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013*, pages 1–2. ACM, 2013. 16

[29] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. **The LDBC Social Network Benchmark: Business Intelligence Workload**. *Proc. VLDB Endow.*, **16**(4):877–890, 2022. 16

[30] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng

Wang, Eugene Wu, and Qingqing Zhou. **ConnectorX: Accelerating Data Loading From Databases to Dataframes**. *Proc. VLDB Endow.*, **15**(11):2994–3003, 2022. 17, 19

[31] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. **Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey**. *Artif. Intell. Rev.*, **52**(1):77–124, 2019. 17

[32] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. **Towards Scalable Dataframe Systems**. *Proc. VLDB Endow.*, **13**(11):2033–2046, 2020. 17

[33] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. **Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats**. *Proc. VLDB Endow.*, **14**(4):534–546, 2020. 17, 18

[34] Jason Myers and Rick Copeland. *Essential SQLAlchemy: Mapping Python to Databases*. " O'Reilly Media, Inc.", 2015. 17

[35] Wes McKinney. **Data Structures for Statistical Computing in Python**. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. 17, 18

[36] Pandas developers. **pandas.read_sql**. `https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html`. Accessed on August 2, 2023. 18

[37] Wes McKinney. **Data Structures for Statistical Computing in Python**. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. 18

[38] Mark Raasveldt. **MonetDBLite: An Embedded Analytical Database**. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1837–1838. ACM, 2018. 19

[39] Mark Raasveldt and Hannes Mühleisen. **Don't Hold My Data Hostage - A Case For Client Protocol Redesign**. *Proc. VLDB Endow.*, **10**(10):1022–1033, 2017. 19

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. **PyTorch: An Imperative Style, High-Performance Deep Learning Library**. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. 20

[41] Jonathan Lajus and Hannes Mühleisen. **Efficient data management and statistics with zero-copy integration**. In Christian S. Jensen, Hua Lu, Torben Bach Pedersen, Christian Thomsen, and Kristian Torp, editors, *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 12:1–12:10. ACM, 2014. 20, 21, 22

[42] Peter A. Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution**. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005. 21

[43] Kurt Hornik, Friedrich Leisch, Achim Zeileis, et al. **Statistical computing and databases: Distributed computing near the data**. In *Proceedings of DSC*, page 2, 2003. 21

[44] Oracle. **Machine Learning in Oracle Database**. `https://www.oracle.com/artificial-intelligence/database-machine-learning/`. Accessed on August 2, 2023. 22

[45] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. **The MADlib Analytics**

Library or MAD Skills, the SQL. *Proc. VLDB Endow.*, **5**(12):1700–1711, 2012. 22

[46] Rakesh Agrawal and Kyuseok Shim. **Developing Tightly-Coupled Data Mining Applications on a Relational Database System**. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 287–290. AAAI Press, 1996. 22

[47] Sandeep Singh Sandha, Wellington Cabrera, Mohammed Al-Kateb, Sanjay Nair, and Mani B. Srivastava. **In-database Distributed Machine Learning: Demonstration using Teradata SQL Engine**. *Proc. VLDB Endow.*, **12**(12):1854–1857, 2019. 23

[48] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. **In-Database Machine Learning with SQL on GPUs**. In Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar, editors, *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*, pages 25–36. ACM, 2021. 23

[49] Thomas Neumann and Michael J. Freitag. **Umbra: A Disk-Based System with In-Memory Performance**. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. 23

[50] Linnea Passing, Manuel Then, Nina C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. **SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases**. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Bress, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 84–95. OpenProceedings.org, 2017. 23, 24

[51] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. **Machine Learning, Linear Algebra, and More: Is SQL All You Need?**

In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. 23

[52] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. **Putting Pandas in a Box**. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. 23

[53] Dennis Marten and Andreas Heuer. **Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements**. *Open J. Databases*, **4**(1):22–42, 2017. 23

[54] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. **Deep Integration of Machine Learning Into Column Stores**. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 473–476. OpenProceedings.org, 2018. 23

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. **Scikit-learn: Machine Learning in Python**. *Journal of Machine Learning Research*, **12**:2825–2830, 2011. 23

[56] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. **Open Graph Benchmark: Datasets for Machine Learning on Graphs**. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. 24

[57] Jim Webber. **A programmatic introduction to Neo4j**. In Gary T. Leavens, editor, *SPLASH'12 - Proceedings of the 2012 ACM Conference on Systems, Programming, and Applications: Software for Humanity, Tucson, AZ, USA, October 21-25, 2012*, pages 217–218. ACM, 2012. 24

[58] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. **Cypher: An Evolving Query Language for Property Graphs**. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018. 24

[59] Victor Chang, Yeshwanth Kumar Songala, Qianwen Ariel Xu, and Ben Shaw-Ching Liu. **Scientific Data Analysis using Neo4j**. In Mitra Arami, Patricia Baudier, and Victor Chang, editors, *Proceedings of the 4th International Conference on Finance, Economics, Management and IT Business, FEMIB 2022, Online Streaming, April 24-25, 2022*, pages 75–84. SCITEPRESS, 2022. 24

[60] Neo4j, Inc. **Neo4j Graph Data Science Documentation: Machine Learning**. `https://neo4j.com/docs/graph-data-science/current/machine-learning/machine-learning/`. Accessed on August 2, 2023. 25

[61] Chang Liu, Semih Salihoğlu. **Scaling Pytorch Geometric GNNs With Kùzu**. `https://kuzudb.com/docusaurus/blog/kuzu-pyg-remote-backend`. Accessed on August 2, 2023. 25

[62] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. **TigerGraph: A Native MPP Graph Database**. *CoRR*, **abs/1901.08248**, 2019. 25

[63] Alin Deutsch. **Querying Graph Databases with the GSQL Query Language**. In Bernadette Farias Lóscio, Carina F. Dorneles, and Maria Camila Nardini Barioni, editors, *XXXIII Simpósio Brasileiro de Banco de Dados, SBBD 2018, Rio de Janeiro, RJ, Brazil, August 25-26, 2018*, page 313. SBC, 2018. 25

[64] Tigergraph. **TIGERGRAPH MACHINE LEARNING WORKBENCH**. `https://www.tigergraph.com/ml-workbench/`. Accessed on August 2, 2023. 25

[65] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov,

Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Wang. **Amazon Neptune: Graph Data Management in the Cloud**. In Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, **2180** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 25

[66] Marko A. Rodriguez. **The Gremlin Graph Traversal Machine and Language**. *CoRR*, **abs/1508.03843**, 2015. 25

[67] Marko A. Rodriguez. **Tales from the TinkerPop**. `https://www.datastax.com/blog/tales-tinkerpop`. Accessed on August 2, 2023. 25

[68] Amazon Neptune ML. **Amazon Neptune ML**. `https://aws.amazon.com/neptune/machine-learning/`. Accessed on August 2, 2023. 25

[69] Numpy. **Numpy repository**. `https://github.com/numpy/numpy/blob/main/numpy/core/include/numpy/ndarraytypes.h#L671`. Accessed on August 2, 2023. 30

[70] PyTorch. **PyTorch.from_numpy**. `https://pytorch.org/docs/stable/generated/torch.from_numpy.html`. Accessed on August 2, 2023. 30

Appendix A

`duckpgq.py` reduces the lines of code to perform various operations

A. DUCKPGQ.PY REDUCES THE LINES OF CODE TO PERFORM VARIOUS OPERATIONS

```
# We use the helper library
import duckpgq

# Connect to a in-memory database
con = duckpgq.connect()

# Prepapre node tables
con.sql("CREATE TABLE nodes ...")
# Prepapre edge tables
con.sql("CREATE TABLE edges ...")

# CSR creation with SQL
con.sql("""
    SELECT  CREATE_CSR_EDGE(
            0,
            (SELECT count(srctable.id) FROM nodes srctable),
            CAST (
                (SELECT sum(CREATE_CSR_VERTEX(
                            0,
                            (SELECT count(srctable.id) FROM nodes srctable),
                            sub.dense_id,
                            sub.cnt)
                            )
                FROM (
                    SELECT srctable.rowid as dense_id, count(edgetable.src) as cnt
                    FROM nodes srctable
                    LEFT JOIN edges edgetable ON edgetable.src = srctable.id
                    GROUP BY srctable.rowid) sub
                )
            AS BIGINT),
            srctable.rowid,
            dsttable.rowid,
            edgetable.rowid) as temp
        FROM edges edgetable
        JOIN nodes srctable on srctable.id = edgetable.src
        JOIN nodes dsttable on dsttable.id = edgetable.dst ;"""
)

# Create CSR with \texttt{duckpgq.py}
con.create_csr("edges", "src", "dst", "nodes", "id")
```

Listing A.1: Comparison of CSR creation with SQL and with duckpgq.py

```
# We use the helper library
import duckpgq

# Connect to a in-memory database
con = duckpgq.connect()

# Prepapre node tables
con.sql("CREATE TABLE nodes ...")
# Prepapre edge tables
con.sql("CREATE TABLE edges ...")

# CSR is required before we can fetch a graph
con.create_csr("edges", "src", "dst", "nodes", "id")

# Fetch a property graph with SQL
# Fetch node table names
con.sql("SELECT vtables FROM get_pg_vtablenames(...);").fetchall()
# Fetch column names for node tables
con.sql("SELECT colnames FROM get_pg_vcolnames(...);").fetchall()
# Fetch node features
con.sql("SELECT ... FROM ...").torch()

# Fteching edge features is similar
....

# Fetch CSR arrays
# Here we have to use the non-zero-copy way, as it's impossible to have a
    memory-sharing in pure Python
con.sql("SELECT csrv FROM get_csr_v(...);").torch()
con.sql("SELECT csre FROM get_csr_e(...);").torch()
con.sql("SELECT csrw FROM get_csr_w(...);").torch()

# Construct GNN graphs
...

# ----------------------------
# Fetch a property graph with \texttt{duckpgq.py}
con.get_dgl(graphname, csrid)
# or
con.get_pyg(graphname, csrid)
```

Listing A.2: Comparison of fetching a property graph with SQL and with `duckpgq.py`