Kùzu: Graph Learning Applications Need a Modern Graph Database Management System

Ziyi Chen
University of Waterloo, Canada
Kùzu, Inc.
chenziyi990424@gmail.com

Xiyang Feng
University of Waterloo, Canada
Kùzu, Inc.
xiyangf1997@gmail.com

Guodong Jin
University of Waterloo, Canada
Kùzu, Inc.
guod.jin@gmail.com

Chang Liu
University of Waterloo, Canada
Kùzu, Inc.
liuc223@gmail.com

Semih Salihoğlu
University of Waterloo, Canada
Kùzu, Inc.
semih.salihoglu@uwaterloo.ca

Abstract

Building a graph learning application requires performing a series of data processing steps, such as extracting data from tabular sources into a graph, cleaning the graph, extracting node/edge features, moving the data into a graph learning library to generate embeddings, and possibly saving these embeddings in a software for further processing. Many of these steps can be performed in an efficient way by database management systems (DBMSs), which come with high-level data models and query languages, and functionalities to export datasets into other formats. However, no current DBMS is tailored for graph learning pipelines. We present Kùzu, an open-sourced graph DBMS that aims to fill this gap. Kùzu is an embeddable system that runs as part of users' applications, implements the property graph data model and the openCypher query language, and a graph-optimized storage structures and join algorithms. Kùzu can ingest data from several tabular raw file formats and export data to popular graph learning libraries. We present Kùzu's design goals, architecture, our ongoing work, and demonstrate how it can be used to train large GNN models that do not fit into main memory. Kùzu is available under a permissive license[1].

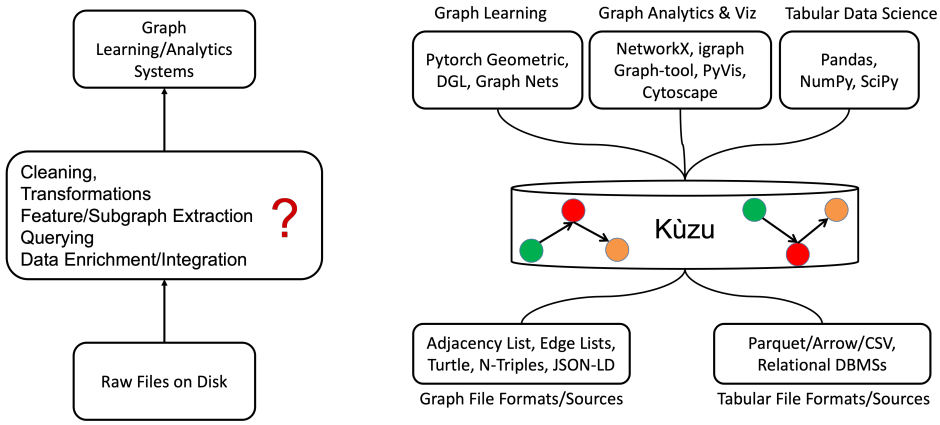
1 Introduction and Motivation

Graphs are one of the most natural and widely adopted data models to model application data that is naturally in the form of a network of entities and their connections. Such applications are ubiquitous in social networks, financial networks, biological networks, amongst many other application domains [1]. Recent developments have shown the effectiveness of graph learning [2], which are techniques to encode the nodes of a graph in low-dimensional embeddings, on many tasks [3, 4]. With the development of several popular graph learning libraries, such as Pytorch Geometric [5] (PyG), Deep Graph Library [6], and Graph Nets [7], as well as extensions of some graph analytics libraries, such as NetworkX [8], these techniques are gaining wider adoption.

Developing graph learning models requires moving the raw files that contain an application's records into a graph learning library. In practice there is often a set of data preparation steps that need to be performed before data can be used by the graph learning library. In many cases, application data is stored in some tabular format, possibly extracted from a relational database management system (DBMS). Further, data often needs a set of data preparation operations as shown in Figure 1a. These include: (i) cleaning, such as filtering certain node and edge records; (ii) transformations, such as

[1]Link to source code: https://github.com/kuzudb/kuzu

Chen et al., Kùzu: Graph Learning Applications Need a Modern Graph Database Management System. *Proceedings of the Second Learning on Graphs Conference (LoG 2023)*, PMLR 231, Virtual Event, November 27–30, 2023.



(a) Overview of graph learning applications. **(b)** Kùzu in the graph learning ecosystem.

Figure 1: Graph learning/data science overview and the gap Kùzu aims to fill.

providing default values on node/edge properties; (iii) data integration, to enrich data with other datasets; (iv) node/edge feature extraction; and (v) querying graph structures for data verification.

To perform such data preparation tasks, graph data scientists often write their own scripts that use a mix of general data wrangling libraries, such as Pandas [9] or NumPy [10], and graph analytics libraries such as NetworkX [8]. This approach has several shortcomings. First, these libraries do not benefit from advanced data processing optimizations that DBMSs have pioneered over decades, such as vectorized execution or intra-query parallelism. Second, these are in memory software whose scalabilities are limited to datasets that fit in memory. Third, using multiple of these libraries require performing expensive data movement and indexing steps. Instead, we argue that graph data scientists can gain significant productivity by using a graph DBMS (GDBMS). GDBMSs implement high-level query languages that can easily express common data preparation steps and are disk-based systems that manage datasets larger than memory. However, existing GDBMSs, such as Neo4j [11], Memgraph [12], and Amazon Neptune [13, 14], have several shortcomings that make them unsuitable to develop graph learning pipelines. For example, they are not libraries and require setting up tedious server and client processes. Further they have known performance problems [1, 15] as they do not implement some of the modern techniques, such as vectorized query execution or columnar storage [16], that have become common wisdom in the last decade in the DBMS community.

This extended abstract presents Kùzu as a new valuable resource for the graph learning community. Kùzu is designed specifically for modern graph learning/data science applications. Figure 1b shows the high-level position of Kùzu in the graph learning/data science ecosystem. Kùzu is an embeddable system that runs as part of user programs. So it does not require setting any server and client processes and users import it simply as a library. Kùzu manages data on disk, so can scale to very large graphs and is architected based on state-of-the-art principles of managing graph-structured databases. Kùzu can easily ingest raw files from popular formats and export data to popular graph learning/data science librariess. Kùzu is being developed as an open-sourced system under the permissive MIT license.

2 Kùzu's Design Goals and Features

We are designing Kùzu with two high-level goals: (i) very easy to use by modern graph learning/data science applications, which require moving data from raw files into a graph learning or data science library; and (ii) highly scalable and performant system that integrates the state-of-the-art knowledge in the field of graph data management. We first review the system's usability features and then the more technical performance and scalability features.

2.1 Usability Features

Data Model and Query Language: Kùzu adopts the property graph data model [17], in which application records are modeled as a set of labeled nodes and edges with key-value properties on these nodes and edges. The system implements the openCypher query language [18], which is the

open source version of the Cypher query language of the popular Neo4j GDBMS [11]. openCypher is a SQL-like query language that has several elegant graph-specific syntax, such as the path syntax for performing joins. As an example, the `MATCH (a:User)-[e:Follows]->(b:User)` statement joins `User` records with the neighbors they "follow" in a social network application.

Programming Language Bindings: Kùzu is an embeddable library and at the time of writing has Python, C/C++, Node.js, Rust, and Java APIs. Data scientists using any of these languages can install and import Kùzu as a library and start creating and querying databases.

Supported Raw Input Files: Currently the system can ingest data from CSV, Apache Parquet [19], Apache Arrow [20], and npy formats [21].

Integrations With Graph Learning and Data Science Libraries: Currently the system allows converting databases or query results into PyG and NetworkX. Integrations to Deep Graph Library [6], Graph Nets [7], and iGraph [22] are currently being implemented. We also have integrations with LlamaIndex [23] and LangChain [24], which are frameworks to build large language model applications. For LlamaIndex, users can store results of knowledge graphs extracted from LlamaIndex directly in Kùzu. We also have an integration with LangChain to provide a natural language interface over Kùzu. Although these are not necessarily graph learning libraries, these integrations could make Kùzu a foundation for AI researchers who are studying topics related to integrating knowledge graphs and graph learning with large language models.

Other Features: Kùzu supports all standard primitive and complex data types commonly available in many DBMSs, such as integers, floats, booleans, strings, dates, timestamps, lists, maps, and structs. The system supports a "fixed-list" data type, which is an optimized way for storing vectors of floats. Fixed-list type is designed specifically to store embeddings. The system implements ACID-compliant transactions, so updates are atomic and provide all-or-nothing behavior.

For reference, Listing 1 in Appendix A gives an example Python program that reads raw files into Kùzu, performs several data preparation operations on it, and moves the data into PyG.

2.2 Performance and Scalability Features

We provide a very brief coverage of the system's storage structures and query processor and refer readers to our other resources [25] for details and coverage of other components, such as its buffer manager, query optimizer, and transaction manager.

Storage Structures and Indices: Kùzu is a columnar system similar to modern read-optimized DBMSs [26, 27]. Node properties are stored in vanilla column files. Edges are double indexed and stored in columnar sparse row-based adjacency list indices, which are the core join indices in the system to join node records. Edge properties are similarly stored in separate CSR-based structures. Each node relation has by default a hash index to be able to look up nodes on their primary keys.

Query Processor: Kùzu has a vectorized query processor, i.e., scans produce vectors of tuples (of size 2048) and each operator in the system processes vectors of data. Vectorized query execution is the state-of-the-art query processor architecture for implementing query/read-optimized DBMSs. In addition, the system has a *factorized* query processor. Factorization [28] is a state-of-the-art query processing technique for compressing intermediate results during query processing when joins of records are many-to-many, which is the typical scenario when finding paths in graph-structured datasets. Finally, the processor implements in-query parallelism through a state-of-the-art technique called morsel-driven parallelism in DBMSs [29].

A Note on Performance: Our resources also provide extensive experiments demonstrating that Kùzu is competitive with or outperforms two state-of-the-art RDBMSs DuckDB [26] and Umbra [30] and the Neo4j system. These experiments use a suite of microbenchmark queries and the popular LDBC social network benchmark for graph databases that tests performance on multi-hop "traversal" queries. For example, the comparison of Kùzu with the most competitive baseline system Umbra on the 7 LDBC interactive short read queries were as follows. On 5 of the queries Kùzu outperformed Umbra with 1.3x, 8.8x, 11.3x, 12.6x, and 35.2x runtime improvement factors, while on two queries Umbra was slightly more performant with factors of 1.1x and 1.2x. Factorization plays a particular role for the good performance of Kùzu on these queries. We refer readers to more detailed performance numbers in reference [25].

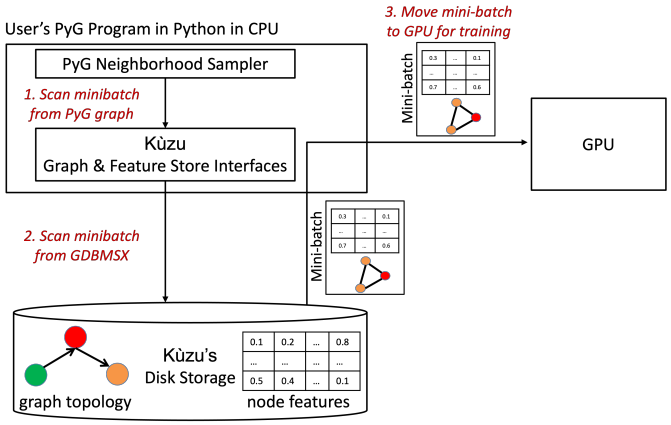


Figure 2: Steps of a single mini-batch training when using Kùzu as a PyG Remote Backend.

3 Example Graph Learning-Specific Feature: PyG Remote Backend

The disk-based storage of Kùzu and its design as an embeddable library allows it to offer more advanced features for graph data scientists than those mentioned in previous sections. As an example, we next describe a feature that enables Kùzu to be used as a PyG Remote Backend, which are plug-in replacements for PyG's in-memory graph and feature stores [31] that can be used seamlessly with the rest of the PyG interfaces. This allows users to train large-scale graphs that are stored in Kùzu (so in its disk-based storage) and cannot fit into the in-memory default storage of PyG.

Using Kùzu as a PyG Remote Backend consists of two simple steps: (i) loading a graph into Kùzu and obtaining, through 1 function call, the Kùzu `feature_store` and `graph_store` objects that implement PyG's Remote Backend interfaces; and (ii) changing 1 line of code in the PyG model code to pass the Kùzu `feature_store` and `graph_store` objects into the PyG's neighborhood sampler. Figure 2 shows the overall computational and data movement steps when using Kùzu as a PyG remote backend during a single mini-batch of training of a GNN model. PyG's neighborhood sampler calls functions on Kùzu's graph and feature stores to fetch a mini-batch, which use the GDBMX library to fetch a mini-batch from disk. In the last step, a mini-batch is scanned from disk, moved to CPU and then to GPU for training by PyG. Note that Kùzu's design as an embeddable library gives it an important performance advantage for implementing tight integration with external libraries. For example, because Kùzu runs as part of the application code and not as a separate server process, moving data from Kùzu feature and graph stores to PyG can be done with zero-copy data movement. This is not possible in server-client architectures.

Reference [32] presents an experiment demonstrating the memory usage vs per-batch training tradeoff of using Kùzu to train a PyG Remote Backend. Briefly, the experiment uses the ogbn-papers100M dataset from the Open Graph Benchmark [33] to train a 3-layer GNN. The experiment shows an overhead of 1.35x when using as much memory as PyG (110G), which is the controlled setting where Kùzu does not do any I/O since all data fits in memory. The performance overhead increases to 4.23x when using about half the memory (60GB) of PyG, which is when data is primarily scanned from disk but when PyG would fail with an out of memory error.

4 Conclusions and Future Work

We believe developers of graph learning applications can benefit from using a modern performance and scability-focused GDBMS. GDBMSs offer high-level graph-specific query languages that simplify data preparation steps that developers routinely perform in practice as well as state-of-the art data processing optimizations. To this end, we are actively developing Kùzu as the GDBMS of graph learning and data science. One of our major ongoing work is to natively support Resource Description Framework (RDF) graphs [34, 35] . RDF is an alternative and popular graph data model that is specifically suitable for modeling highly heterogeneous datasets, such as large encyclopedic knowledge graphs, are encoded as (subject, predicate, object) triples. We are extending our data model to natively ingest RDF triples so they can be queried natively in Kùzu. As an open-source software released under a permissive license, we hope Kùzu can be a valuable resource for researchers and developers working on graph learning.

References

[1] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDB Journal*, 29(2), 2020. 1, 2

[2] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin*, 40(3), 2017. 1

[3] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1), 2020. 1

[4] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z Sheng, Mehmet A Orgun, Longbing Cao, Francesco Ricci, and Philip S Yu. Graph Learning based Recommender Systems: A Review. *arXiv preprint arXiv:2105.06339*, 2021. 1

[5] Pytorch geometric https://pytorch-geometric.readthedocs.io/, 2023. 1, 6

[6] Deep graph library https://www.dgl.ai/, 2023. 1, 3

[7] Graph nets https://www.deepmind.com/open-source/graph-nets, 2023. 1, 3

[8] NetworkX. NetworkX https://networkx.org/, 2023. 1, 2

[9] Pandas. Pandas. https://pandas.pydata.org/, 2023. 2

[10] NumPy. NumPy. https://numpy.org/, 2023. 2

[11] Neo4j http://neo4j.com, 2023. 2, 3

[12] Memgraph https://memgraph.com/, 2023. 2

[13] Amazon neptune https://aws.amazon.com/neptune/, 2023. 2

[14] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Michael Personick, K. Jeric Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Xiong Wang. Amazon Neptune: Graph Data Management in the Cloud. In *International Workshop on the Semantic Web*, 2018. 2

[15] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the Joint International Workshop on Graph Data Management Experiences & Systems*, 2021. 2

[16] Neo4j record formatshttps://neo4j.com/developer/kb/understanding-data-on-disk/, 2023. 2

[17] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Morgan & Claypool, 2018. 2

[18] openCypher. openCypher. https://www.opencypher.org, 2023. 2

[19] Apache parquet https://parquet.apache.org/, 2023. 3

[20] Apache arrow https://arrow.apache.org/, 2023. 3

[21] Numpy npy files https://numpy.org/doc/stable/reference/routines.io.html, 2023. 3

[22] igraph https://igraph.org/, 2023. 3

[23] Llamaindex https://www.llamaindex.ai/, 2023. 3

[24] Langchain https://python.langchain.com/docs/get_started/introduction.html, 2023. 3

[25] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu Graph Database Management System. In *CIDR*, 2023. 3

[26] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *SIGMOD*, 2019. 3

[27] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 2013. 3

[28] Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *TODS*, 40(1), 2015. 3

[29] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014. 3

[30] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *Conference on Innovative Data Systems Research*, 2020. 3

[31] Pytorch remote backend https://pytorch-geometric.readthedocs.io/en/latest/advanced/remote.html, 2023. 4

[32] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Scaling Pytorch Geometric GNNs With Kùzu. https://kuzudb.com/docusaurus/blog/kuzu-pyg-remote-backend, May 2023. 4

[33] 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. *arXiv preprint arXiv:2005.00687*, 2020. 4

[34] W3 Resource Description Framework Overview Wiki https://www.w3.org/RDF/, 2023. 4

[35] Kùzu RDF Support Roadmap. https://github.com/kuzudb/kuzu/issues/1570, 2023. 4

A Appendix: Example Graph Learning Program Using Kùzu

We present a simple example that shows how one could use Kùzu to develop a graph learning-based pipeline. Consider a data imputation application, where we have a dataset of papers and citations between papers where the `category` property of some papers are missing (i.e., NULL). We assume that the datasets are stored in a paper.csv and cites.csv files. The goal is to replace the NULL values with predicted `category` values using a graph convolutional network (GCN) to do the predictions. The final output of the program is a new CSV file called "complete_paper.csv" that contains the category of each paper, so without any NULL values.

Original Data Files:

- paper.csv: for each paper contains an `id`, `title_bools`, and `category` values. `title_bools` is a 1433-size boolean array that describes the words included in each paper's title. If a paper p's title contains word i, p.title_bools[i] is "true". Otherwise it is false. This follows the format of the dataset in the popular Cora datasets in Pytorch Geometric's benchmark datasets [5]. `category` is a numeric value between 0-6 to indicating the category of each paper, e.g., "Reinforcement Learning", in numeric form. Some of the paper records in paper.csv have empty category values.
- cites.csv: a simple edge file storing the ids of each citation between papers in (to, from) format.

Description of the Data Imputation Pipeline: We will take the following 6 high-level steps:

1. Preliminary Database Setup: Connect to Kùzu, prepare the schema of the graph database and load paper.csv and cites.csv into Kùzu.
2. Data Cleaning for Preparing the Training Dataset: We need to train the GCN using the subgraph of the original dataset that does not contain any paper nodes with NULL `category` values. We perform a simple cleaning step to extract that "non-NULL" subgraph from Kùzu as training dataset. In this step, we use Kùzu's `get_as_torch_geometric()` function that converts query results into PyG data objects.
3. GCN Training: This is the longest piece of code in the example and consists only of standard PyG GCN code to define and train a GCN model using the training dataset we obtain in Step 2.
4. Obtain Predictions Using Trained GCN Model: Next, we extract the entire graph, so including also the paper nodes with NULL categories, from Kùzu and predicting all of the category values of all nodes. We can also select only the paper nodes with NULL categories and their neighborhoods but predicting all nodes is simpler code to write in PyG.
5. Set Missing Category Values: For each paper a whose `category` is NULL, we set a's `category` to the prediction we obtained from Step 4.

6. Final Output: We export all paper nodes from Kùzu to "complete_paper.csv" file.

The program, written in Python, is shown below. We assume the user has executed "pip install kuzu" before running the below program to easily install Kùzu.

```
import kuzu

# Step 1.1: Obtain a connection to the database. "./cora" is the
# directory where Kuzu will store the database files.
conn = kuzu.Connection(kuzu.Database("./cora"))
# Step 1.2: Define the nodes, edges and the properties on nodes/edges
conn.execute("CREATE NODE TABLE paper(id INT64, title_bools BOOLEAN[],
              category INT64, PRIMARY KEY (id))")
conn.execute("CREATE REL TABLE cites(FROM paper TO paper)")
# Step 1.3: Load paper nodes and cites edges from CSV files
conn.execute("COPY paper FROM ./paper.csv")
conn.execute("COPY cites FROM ./cites.csv")
# Step 2: Data Cleaning Step: Extract the entire graph from Kuzu
# but exclude nodes whose category is missing, i.e., is NULL.
# get_as_torch_geometric() converts query results into PyG Data
# objects
train_data = conn.execute(
             "MATCH (a:paper)-[r:cites]->(b:paper)
              WHERE a.category IS NOT NULL and b.category IS NOT NULL
              RETURN *").get_as_torch_geometric()
# Set the x (node feature matrix) and y (node-level ground truth
# labels) of the training data. title_bools is converted to float as
# the GCN model requires node features to be floating point numbers.
train_data.x = train_data.title_bools.float()
train_data.y = train_data.category

# Step 3: Define and train a GCN model. Note this is standard PyG code
# that is independent of Kuzu and is presented for
# completeness. We will put ...'s in parts to simplify presentation
# Step 3.1: Prepare test/train split as 60-40 split
total_count = len(train_data.title_bools)
train_ids, test_ids = torch.utils.data.random_split(
    range(total_count), (0.6 * total_count, 0.4 * total_count),
    generator=torch.Generator().manual_seed(42)
)
# Step 3.2: Add masks to identify train and test nodes (Omitted)
# Step 3.3: Define the GCN Model
from torch_geometric.nn import GCNConv
model = GCN(input_channels=train_data.x.shape[1], hidden_channels=32,
            output_channels=train_data.category.max().item() + 1)
...

# Step 3.4: Define the train/predict/test functions (code related to
# the optimizer to use is omitted)
def train(data):
      model.train()
      ...
      return loss
def predict(data):
      model.eval()
      ...
      return pred
def test(data):
      predictions = predict(data)
      ...
      return test_accuracy
# Step 3.5: Perform the actual GCN model training for 100 epochs
for epoch in range(1, 101):
    loss = train(train_data)
    test_acc = test(train_data)
```

```
61
62 # Step 4: Obtain and store the predicted category of each node
63 pred_data = conn.execute("MATCH (a:paper)-[r:cites]->(b:paper)
64                           RETURN *").get_as_torch_geometric()
65 pred_data.x = pred_data.title_bools.float()
66 pred_all = predict(pred_data).tolist()
67 # Step 5: For each node a with a NULL category, set a's category value
68 # to a's predicted category. Kuzu's get_as_df() function returns
69 # query results as a Pandas data frame
70 null_paper_ids = conn.execute("MATCH (a:paper)
71                                WHERE a.category IS NULL
72                                RETURN a.id").get_as_df()
73 for paper_id in null_paper_ids:
74     conn.execute("MATCH (a:paper)
75                   WHERE a.id = $pid
76                   SET a.category = $pred",
77                   [("pid", paper_id), ("pred", pred_all[paper_id])])
78 # Step 6: Finally output all paper nodes as complete_paper.csv
79 conn.execute("COPY (MATCH (a:paper) RETURN *) TO complete_paper.csv")
```

Listing 1: Using GDBMX in a simple graph learning pipeline