Innovative Approaches to Provenance Graph Analysis: GDS and LLM

Pasquale Leonardo Lazzaro PAS.LAZZARO@STUD.UNIROMA3.IT
Marialaura Lazzaro MAR.LAZZARO1@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli Studi Roma Tre
Via della Vasca Navale, 79, 00146, Roma, Italy

Abstract

This paper explores the utilization of graph algorithms via the Graph Data Science (GDS) library to analyze provenance data. By leveraging the capabilities of Neo4j, a robust graph database platform, we efficiently managed and synthesized complex provenance graphs generated by the DPDS tool. Our findings reveal that while the detailed documentation of provenance is crucial for transparency and reproducibility, the current level of granularity in the DPDS tool can lead to redundancy, significantly burdening the graph. We demonstrate that integrating GDS algorithms allows for the identification and synthesis of critical information, thereby optimizing the provenance documentation process. Furthermore, we incorporated a Large Language Model (LLM) approach to facilitate natural language interaction with the graph, making the query process more accessible to users unfamiliar with Cypher. This approach not only simplified query generation but also highlighted potential improvements in data provenance management. Our results underscore the transformative potential of combining GDS tools and LLM-based query generation to enhance the efficiency and comprehensibility of provenance data analysis. Our work lays the groundwork and highlights the potential for continued research in this specific domain.

1 Introduction

The realized project leveraging Neo4j to analyze provenance graphs generated by the **DPDS** tool (3) represents a strategic approach to managing complex data systems. Neo4j, as a robust graph database platform, offers a sophisticated means for handling and querying the intricate networks of relationships that characterize provenance graphs.

Provenance, referring to the history and origin of data, plays a critical role across various sectors by ensuring the integrity, reproducibility, and accountability of information. In domains such as scientific research, finance, and healthcare, the ability to trace data modifications, the responsible parties, and the contextual circumstances is crucial for maintaining transparency and trust.
However, the large volume of data involved in tracking provenance can be overwhelming and challenging to interpret. By utilizing the capabilities of Neo4j, the project aims to harness the power of graph database technology to efficiently synthesize and manage this data. Through the application of Cypher queries and the Graph Data Science (GDS) library, Neo4j facilitates the effective analysis and visualization of provenance graphs.

This approach not only enhances the accessibility and comprehensibility of provenance data but also contributes to more rigorous data management practices, fostering better decision-making and policy development in data governance. This research could thus pro-

vide significant insights into the optimization of data provenance management, highlighting the transformative potential of graph databases in contemporary data-driven landscapes.

2 Theoretical Background

2.1 Data Provenance model

The primary objective of data provenance is to facilitate the generation of concise explanations concerning the presence (or absence) of specific data following complex data manipulations. In pursuit of this aim, we have selected a subset of the PROV model (4)—a model developed by the World Wide Web Consortium (W3C). This model is a broadly embraced ontology that formalizes the concept of a provenance document.
Figure 1 graphically depicts the minimal elements of the model, illustrating the foundational components essential for understanding data provenance in this framework.

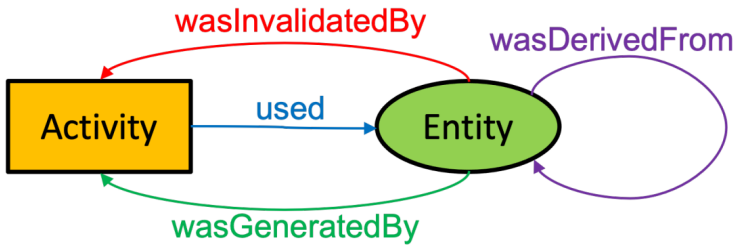

Figure 1: The core W3C PROV model.

2.2 DPDS tool

The "Data Provenance for Data Science" (DPDS) (3) library stands out for its ability to integrate provenance tracking into data preprocessing workflows. By automatically capturing fine-grained provenance data without the need for explicit function calls, DPDS minimizes the overhead typically associated with provenance data collection. This functionality allows data scientists and data engineers to focus more on pipeline development rather than on provenance logistics.
DPDS tracks only Pandas Dataframes.

Once a pipeline is implemented and executed, DPDS generates a detailed provenance graph that records every data transformation within the pipeline. This graph is easily accessible and analyzable through the Neo4j graph database, a powerful tool for managing complex networked data. The use of Neo4j enables users to query and visualize the lineage of any piece of data, offering valuable insights into the data processing steps and aiding in debugging and compliance reporting.

Overall, DPDS enhances the transparency and reproducibility of data science projects by providing a robust framework for tracking data transformations. This provenance tracking is crucial for projects requiring audit trails, rigorous data quality standards, or detailed historical records of data handling.

Category	Function	Description
Data Reduction	Feature Selection	One or more features are removed.
Data Reduction	Instance Drop	One or more records are removed.
Data Augmentation	Feature Augmentation	One or more features are added.
Data Augmentation	Instance Generation	One or more records are added.
Space Transformation	Dimensionality Reduction	Features and records are added/removed. The overall number of removed features and records is greater than those added.
Space Transformation	Space Augmentation	Features and records are added/removed. The overall number of added features and records is greater than those removed.
Space Transformation	Space Transformation	Features and records are added/removed. In this case, there can be a reduction in dimensionality for one axis and a space augmentation for the other.
Data Transformation	Value Transformation	The values of one or more features are transformed.
Data Transformation	Imputation	Missing values in one or more features are filled with estimated values.
Feature Manipulation	Feature Rename	One or more features are renamed.
Data Combination	Join	Two or more datasets are combined based on a common attribute or key.
Data Combination	Cartesian Product	Two datasets are combined resulting in a Cartesian product.

Table 1: Data Operations and Their Descriptions.

The types of functions captured by DPDS are shown in Table 1

2.3 Neo4j Overview

Neo4j is a graph database, known for its open-source accessibility while offering full commercial support. As a NoSQL system developed in Java, Neo4j is schema-optional and boasts high scalability, making it particularly adept at managing intricate data relationships. It is widely recognized as the leading enterprise-ready graph database in today's market, due to its robust capabilities in handling complex data structures.

Notably, Neo4j includes the Graph Data Science (GDS) library (5), which offers efficiently implemented, parallel versions of common graph algorithms, accessible through Cypher procedures. Furthermore, the GDS library extends its functionality by incorporating machine learning pipelines that enable the training of predictive supervised models, aimed at addressing graph-specific challenges, such as the prediction of missing relationships. This blend of advanced analytics and flexible database architecture underscores Neo4j's exceptional utility in modern data environments.

3 Graph structure

In the Neo4j Graph modeling approach, nodes represent distinct data entities and processes, while the relationships between them map the dynamic interactions that govern data transformations.

This section begins by detailing how relationships between activity nodes, which symbolize distinct operations or actions, serve a critical function. These relationships not only connect activities but also crucially denote the sequence in which these operations are executed. This sequencing is essential to understand how data evolves and is manipulated over time, ensuring transparency in data processing.

The subsequent subsections will delve deeper into the specifics of these nodes and their interconnections. We will explore how each type of node and relationship contributes to building a comprehensive and traceable data provenance framework.

3.1 Node Labels

3.1.1 Acitivity

Activity node represent distinct operations or actions performed during data processing and analysis workflows. These nodes are essential for tracing the modifications, computations, and transformations applied to datasets over time.
Each Activity node is structured to encapsulate key details about the operations it represents, including the followings properties:

- **<id>:** A unique identifier that distinguishes each activity within the graph, facilitating easy reference and retrieval.
- **code:** The actual code line that was executed, providing direct insight into the operation's nature and implementation.
- **code_line:** The specific line number in the source code where the operation is implemented, aiding in debugging and review processes.
- **description:** A brief description of the operation, offering an at-a-glance understanding of what the activity involves.
- **function_ame:** The name of the function or method in which the operation occurs, which helps in mapping the activity within larger codebases.
- **operation_number:** An ordinal number that may represent the sequence of the operation within a workflow, useful for tracking execution order.
- **tracker_id:** An identifier that links the activity to a broader tracking system, which is crucial for monitoring and auditing the data processing lifecycle.
- **used_features:** A list of features (columns) that are involved or affected by the operation.

3.1.2 Entity

Each entity node represent an entity of the Dataframe in the graph and has the following properties, which are key to understanding the data structure and the operations performed:

- **id**: A unique identifier for the entity. This is an alphanumeric value, uniquely distinguishing one entity from another.
- **feature_name**: The name of the feature (column) this entity represents. This could be a variable, column name, or any other descriptor relevant to the data.
- **instance**: The instance number that represents the chronological order of the instance generation.
- **index**: A numeric value representing the entity's position.
- **type**: The data type of the entity, which could be integer, string, boolean, etc., defining the kind of data the entity holds.
- **value**: The actual value of the entity at a given point in the process or data workflow.

3.2 Relationship Types

Five types of relationship are supported by the DPDS tool, and they are listed below.

- **NEXT**: It is a relationship between activities, and the purpose is to indicate the order of execution of the latter.
- **USED**: it represents the usage, it is the beginning of utilizing an entity by an activity. Before usage, the activity had not begun to utilize this entity and could not have been affected by the entity.
- **WAS_DERIVED_FROM**: A derivation is a transformation of an entity into another, an update of an entity resulting in a new one, or the construction of a new entity based on a pre-existing entity.
- **WAS_GENERATED_BY**: Generation is the completion of production of a new entity by an activity. This entity did not exist before generation and becomes available for usage after this generation.
- **WAS_INVALIDATED_BY**: Invalidation is the start of the destruction, cessation, or expiry of an existing entity by an activity. The entity is no longer available for use (or further invalidation) after invalidation. Any generation or usage of an entity precedes its invalidation.

4 Methodology and Execution

4.1 Tested Pipelines

We analyzed various preprocessing pipelines, both by executing pipelines already integrated within the DPDS Tool and by creating new pipelines from scratch.

The pipelines in question have been constructed, tested, and analyzed with the aim of identifying potential limitations and areas for improvement in the provenance analysis performed by the DPDS tool.
All pipelines operate on pandas DataFrames.

4.1.1 Existing Pipelines

Five pipelines are integrated within the DPDS Tool and they are presented below:

1. **demo_shell_categorical**: Initially, data frames are **merged** based on common keys, integrating data from different sources. Following this, missing values are **imputed** to maintain the robustness of the dataset, ensuring that no data point is lost due to gaps in the data. **Feature transformation** is then applied to specific columns; for instance, values are manipulated arithmetically to adjust their scale or format, and categorical variables are encoded into binary features through dummy variable creation. This transformation expands the feature space, making the data more suitable for analytical models. Lastly, unnecessary columns are **removed** to streamline the dataset, reducing dimensionality and focusing the analysis on relevant features. Each step is meticulously logged by a provenance tracker, ensuring that every modification is recorded for transparency and reproducibility.

2. **demo_shell_numerical**: The process begins by dynamically appending a **new instance** to the data frame to manage new data entries effectively. Subsequent operations include multiple **merges** of the main Dataframe with other data frames. These merges are based on common keys and are designed to consolidate and align data from multiple sources, enhancing the dataset's integrity. **Imputation** is a key aspect of the preprocessing, where missing values are initially replaced with zeros and subsequently with a default value of ten. **Feature transformations** are then applied to specific columns, `D` and `key2`, where values are doubled, adjusting the feature scale to suit analytical needs. **Space transformations** are another critical operation, converting categorical variables into dummy or indicator variables. This involves creating new columns for each category in the `D` and `E` columns, followed by the removal of the original columns to reduce complexity and focus the analysis on relevant variables. The column `B` is also removed, further streamlining the dataset.

3. **census_pipeline**: Data from the census dataset is loaded into a dataframe with specific column names assigned.
   Preprocessing includes stripping whitespace from several columns and **replacing** specific characters with NaN values to standardize the data.
   Selected categorical features undergo **one-hot encoding** to transform them into a format suitable for modeling, followed by assigning binary values to target categorical columns. Unnecessary columns are **removed** to streamline the dataset.

4. **compas_pipeline**: After data loading, the dataframe may be sampled based on a provided fraction, with each step logged for data provenance. The pipeline's operations focus on refining the dataset by selecting key columns, **eliminating** missing values, and applying strategic **feature transformations** to align with analytical objectives.

Significant transformations include narrowing the dataset to essential demographic and legal variables, converting race features to binary format for simplified modeling, and renaming and inverting a recidivism indicator to serve as a binary label for predictive modeling. Additionally, the script calculates the duration of jail time from date columns to create a predictive numerical feature, removes unnecessary date columns to streamline the dataset, and transforms legal charge degrees into a binary classification of misconduct and felony.

5. **german_pipeline**: Upon loading the dataset, key operations performed include a comprehensive transformation of multiple categorical columns, where original values are **replaced** with more descriptive labels. This transformation facilitates easier interpretation and analysis of the data.
   Further preprocessing includes the **generation** of new columns derived from the 'personal_status' column to separately encode marital status and gender. The original 'personal_status' column is then dropped to streamline the dataset. The pipeline also implements **one-hot encoding** for a series of categorical columns to prepare the data for machine learning models, converting categorical attributes into a binary matrix format.

4.1.2 New Pipelines

Four new preprocessing pipeline were created by us, with new different datasets:

1. **car_pipeline**: The script performs several key data transformations. First it targets specific columns, "car_price" and "car_mileage", and applies a **value transformation** to each value in these columns. If a value is 1000 or greater, it converts the number to a string in k format (e.g., 1500 becomes 1.5k). Then it removes unnecessary columns from the DataFrame, retaining only the relevant data for analysis, it removes the columns "car_transmission", "car_drive", "car_engine_capacity", and "car_engine_hp" from the DataFrame. Next, the first column of the DataFrame is **renamed** to "car_id" for better clarity. Following, whitespaces are **stripped** from text columns, "car_brand", "car_model", and car_city, to ensure data consistency. Finally, the script logs an operation to categorize the age of cars. It **creates** a new column, "car_age_category", and categorizes each car based on its age. Cars that are 3 years old or younger are labeled New, those between 4 and 9 years old are labeled Middle, and those older than 9 years are labeled Old. The updated DataFrame, including this new age category column, is printed to show the results of this transformation. Throughout these transformations, the script logs each step, providing a detailed record of the operations performed on the DataFrame. This logging, combined with provenance tracking, ensures that all changes are transparent and verifiable. When the script is run directly, it parses the command-line arguments and executes the main pipeline function, applying all the described transformations and logging activities.

2. **mushrooms_pipelines**: The script performs a series of data transformations on a DataFrame related to mushrooms. First, it proceeds to **remove** thirteen specified columns from the DataFrame. These columns include various attributes of mushrooms such as "does-bruise-or-bleed", "gill-attachment", "stem-root", and others that

are deemed unnecessary for the analysis. Next, it **transforms** the 'class' column. It assigns a binary value to the 'class' column where 'e' (edible) is replaced with 1 and 'p' (poisonous) is replaced with 0. This binary assignment simplifies the classification task. The script then **removes** any rows with missing values from the DataFrame, ensuring that the dataset is complete and ready for further analysis without dealing with any NaNs. Lastly, it applies several categorical **value transformations** to the DataFrame. Specific attributes like 'cap-color', 'cap-shape', 'cap-surface', and 'season' are transformed from categorical text values to numerical values. For example, different colors in 'cap-color' are replaced with corresponding numbers, and similarly for 'cap-shape', 'cap-surface', and 'season'. These transformations convert the categorical data into a numerical format that is more suitable for machine learning algorithms.

3. **titanic_pipelines**: The script performs a series of data cleaning and transformation operations on a DataFrame. It starts by **dropping** specific columns, namely 'Name', 'Ticket', and 'Cabin'. These columns are then removed from the DataFrame, reducing it to only the necessary data fields. Next, it **drops** all rows with missing values (NaNs). The DataFrame is updated by removing any row that contains NaN values. Following this, the script selects three columns, 'Pclass', 'Sex', and 'Embarked', which contain categorical data that need to be converted into a numerical format. For each of these columns, it effectively creating **one-hot encoded** versions of the categorical data. These new columns are prefixed with the original column name for clarity. The DataFrame is updated by adding these new columns and then dropping the original categorical column. Through these operations, the script ensures that the DataFrame is cleaned of unnecessary columns, free of incomplete data, and prepared with numerical encodings of categorical variables, making it suitable for further analysis or modeling.

4. **baz_pipeline**:The script first try to **one-hot encode** specific columns. The selected columns for this encoding are 'workclass' and 'native-country'. For each of these columns, the script generates dummy variables, creating one-hot encoded versions of the categorical data. These new columns are prefixed with the original column name to maintain clarity. Next, the script **replace** any '?' characters in the DataFrame with NaN values. This step ensures that placeholders for missing data are consistently represented as NaNs, which can be handled appropriately in subsequent data processing steps. Finally, the script **calculates** a new feature called 'capital-net' by subtracting 'capital-loss' from 'capital-gain'. Another new feature, 'hours-per-week-to-age-ratio', is calculated by dividing 'hours-per-week' by 'age'. These new columns are added to the DataFrame to provide additional insights and potentially useful features for analysis or modeling. Through these steps, the script effectively prepares the DataFrame by encoding categorical data, handling missing values, and creating new informative features.

4.1.3 Smart Sampling Method

Existing pipelines supported simple randomly sampling method to sample dataset in the way to manage the provenance graph (less nodes). The smart sampling method implemented

in the `stratified_sample` function aims to perform stratified sampling on a DataFrame to ensure representative sampling while minimizing information loss. This approach dynamically identifies and uses categorical columns for stratification, falling back to random sampling if suitable stratification is not possible. The method is defined as follows:

```
def stratified_sample(df, frac):
    """
    Perform stratified sampling on a DataFrame.
    """
    if frac > 0.0 and frac < 1.0:
        # Infer categorical columns for stratification
        stratify_columns = df.select_dtypes(include=['object']).columns.tolist()

        # Check if any class in stratify columns has fewer than 2 members
        for col in stratify_columns:
            value_counts = df[col].value_counts()
            if value_counts.min() >= 2:
                # Perform stratified sampling for this column
                stratified_df = df.groupby(col, group_keys=False)
                .apply(lambda x: x.sample(frac=frac))
                return stratified_df.reset_index(drop=True)

        # If no suitable stratification column is found, fall back to random sampling
        sampled_df = df.sample(frac=frac).reset_index(drop=True)
    else:
        sampled_df = df
    return sampled_df
```

The `stratified_sample` function performs stratified sampling on a DataFrame, aiming to maintain the distribution of categorical data in the sampled subset. If stratified sampling is not feasible, it defaults to random sampling.

The function identifies potential stratification columns by selecting columns with the data type `object`, which typically correspond to categorical data.

The function iterates over the identified categorical columns to determine if they are suitable for stratified sampling. A column is considered suitable if each category within the column has at least two members.

If a suitable column is found, the DataFrame is grouped by this column, and stratified sampling is performed on each group. The sampled subsets are then combined into a single DataFrame with the index reset.

If no suitable stratification column is found, the function performs random sampling on the entire DataFrame.

The Benefits of the Smart Sampling Method are:

- **Representation**: Ensures that the sampled data retains the distribution of categorical variables, providing a representative subset.

- **Flexibility**: Dynamically identifies suitable stratification columns and adapts to the data's characteristics.
- **Fallback Mechanism**: Defaults to random sampling when stratified sampling is not feasible, ensuring that the function always produces a sampled subset.

This smart sampling method enhances the representativeness and reliability of the sampled data, making it particularly useful for data analysis and machine learning tasks where maintaining the distribution of categorical variables is crucial.

4.2 Limitations and Challenges of DPDS

The method by which provenance is tracked by the tool used in our analysis has certain limitations, which can be summarized as follows:

- **Data Type Support**: The tool supports only Pandas dataframes as data types.
- **Excessive Data Generation**: Excessive Neo4J data generation necessitating sampling of the original dataset with a sampling rate between 10% and 20%.
  Considering the census pipeline with a 20% sampling rate (size of the dataset: 3,66 MB), the resulting graph has 274389 Entity and 420313 relationship
- **External Library Tracking Issue**: Inability to track provenance when using external libraries for data cleaning and processing.
  To track operations you have to operate directly to dataframe, you cannot use external libraries, like scikit-learn libraries, there is the lack of a recursive way of tracking provenance.
- **Granularity Limitation**: Lack of options for selecting the granularity of provenance description.
  You cannot choose the granularity level of provenance description, that could be fundamental when we think at different users (that for example do not know Cypher).
- **Limited Data Cleaning Tracking**: Few data cleaning operations are supported and tracked by the tool (look at Table 1).

5 Provenance graph analysis

First, preliminary experiments were conducted to understand the structure and characteristics of the graph. Basic Cypher queries were performed to grasp the information contained within it. By crafting simple Cypher queries, the team was able to count the number of activities performed, identify the number of distinct activities, list the names of activities, and determine which operations utilized specific nodes. These queries enabled the narration of provenance in a more descriptive and high-level manner.
The following examples answer to the queries (applicable to all types of pipeline) refer to the census pipeline.
Here some example of the preliminary interrogations.

```
MATCH (n:Activity) RETURN count(n)
```

count (n)
6

This result indicates the number of operations present in the pipeline.

```
MATCH (n:Activity)
RETURN COUNT(DISTINCT n.function_name) AS activityNumber,
       COLLECT(DISTINCT n.function_name) AS distinctActivities
```

activityNumber	distinctActivities
3	[”Value Transformation”, ”Feature Selection”, ”Feature Augmentation”]

This shows the determination of the number of distinct activities performed and provides a list of the activity names.

```
MATCH (n:Activity)
WITH n.function_name AS activityName, COUNT(n) AS activityCount
RETURN activityName, activityCount
```

activityName	activityCount
”Value Transformation”	3
”Feature Selection”	2
”Feature Augmentation”	1

From this query, we can determine the number of distinct activities performed and obtain a list of the activity names. This information provides insight into the frequency of each distinct activity within the pipeline.

```
MATCH (e : Entity {id : `entity:ed7ff4dd-b198-4c55-88af-
f711e8b04695`})<-[:USED]-(a:Activity) RETURN collect(a)
```

This query matches an entity node with a specified identifier and retrieves all associated activities that have used this node. The results are collected and returned, providing a comprehensive list of activities that have utilized the specified entity.

collect(a.function_name)
["Feature Selection", "Value Transformation"]

5.1 Graph Data Science

After this initial analysis, the graph was subjected to a more in-depth examination leveraging GDS. The Neo4j Graph Data Science (GDS) library is a comprehensive toolkit designed for leveraging graph analytics and machine learning.
It offers a wide range of optimized algorithms for centrality, community detection, similarity, and pathfinding, enabling efficient analysis of large-scale graph data.
The GDS library is essential for extracting valuable insights from graph data.

5.1.1 Path algorithms

Utilizing longest and shortest path algorithms in the Neo4j Graph Data Science (GDS) library allows for the identification of the **most frequently manipulated entries** in a graph. These algorithms help highlight the nodes most impacted by the pipeline.

For example, generating a list of the top-k most manipulated entries reveals the most frequently altered data points. Tracing the path of a node by its ID provides insight into its manipulation history, showing how and how many times it has been altered from the beginning.

Fundamental to this is the use of projection. In graph data science, a projection is an optimized in-memory representation of a graph tailored for analytical operations. It focuses on relevant portions of the graph to enhance computational performance and efficiency. Projections allow for the selective inclusion of nodes, relationships, and properties, reducing computational overhead and memory usage. This process is essential for running advanced graph algorithms effectively, ensuring accurate and practical insights. Projections are a fundamental tool in graph data science, enabling efficient and targeted analysis of complex graph data.

```
//New property to give a weight to edges
MATCH ()-[r:WAS_DERIVED_FROM]-()
SET r.newProperty = 1.0;

//Projection on WAS_DERIVED_FROM
CALL gds.graph.drop(`proj`, false);
CALL gds.graph.project(`proj`, [`Entity`],
{ WAS_DERIVED_FROM:{orientation:`NATURAL`, properties:`newProperty`} } );

//Query finding top-5 longest path
CALL gds.allShortestPaths.stream(
  `proj`,
  {relationshipWeightProperty: `newProperty`}
)
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
RETURN sourceNodeId AS source, targetNodeId
 AS target, distance
ORDER BY distance DESC
LIMIT 5;
```

source	target	distance
178	53	3.0
135	63	3.0
175	31	3.0
179	32	3.0
140	39	3.0

The result of the top-k longest path represents the entity that have been modified the most in the pipeline.

Additionally, tracing the path of a given node by its ID helps to understand the entire manipulation history of that entry. By examining the shortest path, we can see the direct sequence of operations applied to the node, while the longest path can reveal more complex, indirect manipulations. This approach allows us to quantify how many times an entry has been manipulated from the beginning of the pipeline to its current state. These capabilities are essential for tasks such as impact analysis, where understanding the propagation of changes through the pipeline is crucial. They also assist in identifying potential bottlenecks or heavily trafficked nodes, which can be critical for optimizing pipeline performance and ensuring data integrity.

```
//finding path of a given node by its ID
CALL gds.allShortestPaths.stream(
  `proj`,
  {relationshipWeightProperty: `newProperty`}
)
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE sourceNodeId = 382
RETURN collect(targetNodeId) as pathNodes
```

pathNodes
[264, 314, 328, 382]

In this example is highlighted the path of a particular node from him to his origin.

5.1.2 Community detection

Another significant application of the Neo4j Graph Data Science (GDS) library is community detection. Different community detection methods are applied, one of those is the Louvain method, refer to Appendix A for more details. Two projections proved to be the most useful for the analyses and are as follows:

- Projection WAS_DERIVED_FROM: In this context, each community represents a node that has been modified over time, along with its modifications. This allows for the observation of the evolution of specific nodes and their associated changes, providing a detailed view of node transformations and their histories.

  ```
  CALL gds.graph.project(`proj`, [`Entity`],
  { WAS_DERIVED_FROM:{orientation:`NATURAL`} } );
  ```

- Projection All Relationships between entities: In this scenario, each community represents an activity. This projection aids in understanding how different activities are grouped based on the relationships between nodes, offering insights into the functional clusters within the graph.

```
CALL gds.graph.project(`proj`, [`Entity`],
{ WAS_DERIVED_FROM:{orientation:`NATURAL`},
USED:{orientation:`NATURAL`}, WAS_GENERATED_BY:
{orientation:`NATURAL`}} );
```

Additionally, other community detection algorithms, such as **Weakly Connected Components** and Core of nodes, were utilized to further analyze the structure and dynamics of the graph. In graph theory, different algorithms offer unique insights into graph structures. **Core decomposition** identifies k-cores, which are subgraphs where each node has at least k neighbors. This technique is particularly useful for detecting influential nodes and dense subgraphs, especially in social network analysis, refer to Appendix B for more details. Weakly Connected Components (WCC) identify maximal subgraphs where any two nodes are connected, regardless of edge direction, refer to Appendix C for more details. Each algorithm serves a distinct purpose, making them valuable tools for analyzing different aspects of graph data.
Eigenvector Centrality is an algorithm that measures the transitive influence of nodes. Relationships originating from high-scoring nodes contribute more to the score of a node than connections from low-scoring nodes. A high eigenvector score means that a node is connected to many nodes who themselves have high scores, refer to Appendix D for more details.
Betweenness Centrality is a measure of the influence a node has over the flow of information in a graph, refer to Appendix E for more details . It identifies nodes that act as bridges between different parts of the graph. This metric is particularly useful for detecting bottlenecks in the data flow. For example, using betweenness centrality, we can identify key processes, such as one-hot encoding, that may serve as critical junction points in the data pipeline.

The application of these community detection algorithms enables the effective identification of clusters of nodes, thereby elucidating significant activities and modifications within the graph. This approach provides profound insights into data provenance and the impact of various processes on the dataset, contributing to a comprehensive understanding of the underlying data dynamics.
Applications follows:

- Here the application of **Louvain** to the provenance graph (first with the projection on WAS_DERIVED_FROM and secondly with the projection on all relationships):

  ```
  CALL gds.louvain.mutate(`proj`, {mutateProperty:`communityId`});
  CALL gds.graph.nodeProperty.stream(`proj`,`communityId`, [`Entity`])
  YIELD nodeId, propertyValue
  WITH gds.util.asNode(nodeId) AS n, propertyValue AS communityId
  WHERE n:Entity
  RETURN n, communityId
  ORDER BY communityId ASC
  ```

  The communities correspond to the derived entity aggregated.

n	communityId
(:Entity {instance: [0],feature_name: "age", index: 14507, id: "6d55714f-e208-448f-931c-632a49871d66", type: "int", value: 23})	0
(:Entity {instance: [0],feature_name: "workclass", index: 14507, id: "114dda05-e6f0-43ae-98c7-26464c963c36", type: "str", value: "Private"})	1
(:Entity {instance: [1],feature_name: "workclass", index: 14507, id: "c0d6fd6d-e768-4bba-9f37-9a6975d70ebe", type: "str", value: "Private"})	1
(:Entity {instance: [0],feature_name: "fnlwgt", index: 14507, id: "91359319-3e5f-4990-82ad-6c40e7a2ae6f", type: "int", value: 256211})	2
(:Entity {instance: [0],feature_name: "education", index: 14507, id: "d4b8fe07-3a71-43cc-8778-9d8c84206f22", type: "str", value: "Some-college"})	3
...	
(:Entity {instance: [0],feature_name: "education", index: 16695, id: "f1916711-3c81-4e53-a080-3d66e2c2e832", type: "str", value: "Some-college"})	298
(:Entity {instance: [1],feature_name: "education", index: 16695, id: "01682cf9-ae25-4f4e-969d-e81dafd38d28", type: "str", value: "Some-college"})	298

Now the number of communities corresponds to the number of the activities/operations (6).

n	communityId
Instance [0], Age, Value: 23	0
Instance [0], Education-Num, Value: 10	0
Instance [0], Workclass, Value: Private	0
Instance [0], Education, Value: Some-college	0
Instance [0], Marital-Status, Value: Never-married	0
...	
Instance [0], Workclass, Value: Private	6
Instance [0], Education, Value: Some-college	6

Depending on the type of projection used, the interpretation of community detection results can vary significantly. For instance, when employing the "WAS_DERIVED_FROM" projection, the identified communities provide information on entities that represent the same element, revealing the evolution and modifications of specific nodes over time. Instead, when utilizing the "All Relationships between entities" projection, the communities identified represent entities involved in a single operation, offering insights into how various activities are interconnected based on the relationships between nodes. This distinction is crucial as it enables the analysis to be tailored to

specific aspects of the graph, thereby providing nuanced and context-specific insights into the data.

- Here the application of **Core decomposition** to the provenance graph (applied on the projection on WAS_DERIVED_FROM):

```
CALL gds.kcore.stream(`proj`)
YIELD nodeId, coreValue
RETURN gds.util.asNode(nodeId).id AS id, coreValue ORDER BY coreValue DESC;
```

id	coreValue
"entity:114dda05-e6f0-43ae-98c7-24664c963c36"	1
"entity:d4b8fe07-3a71-437c-8778-9d8c84206f22"	1
"entity:cb531e8f-1c1b-4062-9768-7e1c0fb17341"	1
"entity:d26ef0a2-b1a6-4962-bbb8-8062d80e3c77"	1
"entity:9a1fb027-f17d-463c-a4e1-e4224c2f7306"	1
...	
"entity:d74be72c-b5df-42e3-8f2b-c1e3eb6cd2cf"	1
"entity:119d28d4-23aa-4c02-91e5-35998b8c7989"	1
"entity:b0c56962-ffce-4073-887b-e8922c315de1"	1

We can see that each entity has core 1, so the same entity was manipulated at most 2 times consequently.

- Here the application of **the Weakly Connected Component method** to the provenance graph (applied on the projection on WAS_DERIVED_FROM):

```
CALL gds.wcc.stream(`proj`)
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).id AS name, componentId
ORDER BY componentId, name;
```

There have been no groundbreaking discoveries compared to the Louvain method.

- Here the application of the **Eigenvector Centrality** method to the provenance graph (applied on the projection with all relationships and including also Activities):

```
CALL gds.eigenvector.stream(`proj`)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS name, score
ORDER BY score DESC, name ASC
```

The interpretation of this query results are useful, because they express the most involved entities and activities (highest score means highest involvement)

- Here the application of the **Betweeness Centrality** method to the prove- nance graph (applied on the projection with all relationships and icluding also Activities):

```
CALL gds.betweenness.stream(`proj`)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS name, score
ORDER BY score DESC
```

name	score
"activity:7b395fc7-3dda-47c1-be68-001e8a25c95f"	4161843857.0
"activity:f220ba94-9eaa-406f-a10e-02a011c5a8fa"	254417328.0
"activity:c1edd059-97e6-4b94-8a4a-02556a3514fb"	1034595.0
"entity:6d79a2e1-429a-43b3-aead-731805487e0"	58679.52827918171
"entity:ebb8980f-37c2-4090-b273-1d5b09d7e0f"	58679.52827918171
...	
"entity:cb8c5549-9100-4942-b963-fc9d60689201"	58679.52827918171
"entity:17918da6-9471-4a9f-9eff-8b3419c53938"	58679.52827918171

From the results listed above, it is evident that the nodes with the highest flow scores are activities. This observation is consistent with the definition of operations and their role as potential bottlenecks within the graph. The high betweenness centrality scores of these activity nodes indicate their critical position in the flow of information, reinforcing their significance in the network's structure and functioning.

6 Exploratory Study on Pseudo-RAG for Graph Interaction

After producing a series of Cypher queries, our intuition was to leverage this work to avoid repetition. This led us to consider using a Large Language Model (LLM) with a pseudo-Retrieval-Augmented Generation (pseudo-RAG) approach.

The idea is to enable natural language interaction with the graph, supporting a lower granularity abstraction of provenance. This approach aims to allow non-expert users, who may not be familiar with Cypher, to effectively interact with the graph

This preliminary exploration demonstrates the potential of using LLMs to facilitate natural language interaction with graph databases, aiming to make data provenance analysis accessible to non-expert users. Future work will focus on refining this approach and expanding its capabilities.

6.1 Architecture and Realization

The realization involves the following steps, for all the details refer to (6) :

1. **Neo4j Graph Initialization:** The Neo4j graph connection is established using the `Neo4jGraph` class from the `langchain_community.graphs` module, with the appropriate connection details.

2. **Cypher Generation Template:** A prompt template (`CYPHER_GENERATION_TEMPLATE`) is defined to guide the LLM in translating natural language questions into Cypher queries. The template includes instructions and examples to ensure accurate translation based on the graph schema.

3. **Prompt Template Creation:** The `PromptTemplate` class is used to create the `cypher_generation_prompt` with the defined template and input variables (`schema` and `question`).

4. **Cypher QA Chain Construction:** The `GraphCypherQAChain` is created using the `from_llm` method, which integrates the LLM (`chat`), the graph connection, and the cypher generation prompt.

5. **Schema Printing:** The schema of the graph is printed to provide an overview of the available nodes and relationships.

6. **Interactive Q&A Loop:** An interactive loop allows users to input questions in natural language. These questions are translated into Cypher queries by the LLM, and the resulting queries are executed against the graph. The results are then displayed to the user.

The essential tools used are Groq (1) that allowed us to use an LLM (in particular "llama3-70b-8192") and langchain (2) a widespread framework for developing applications powered by large language models.

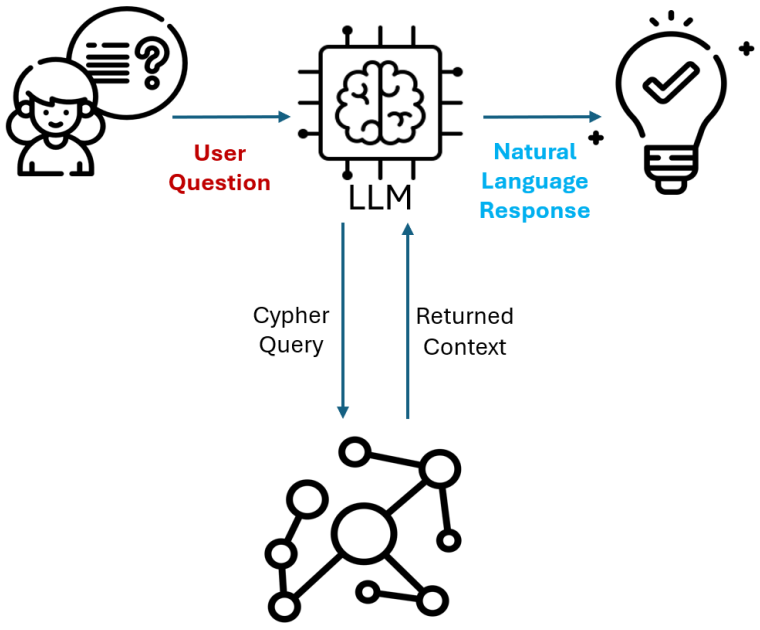

Figure 2: LLM-approach architecture

Following, high-level description of the process with a graphic support (Figure 2):

1. The user make a question in natural language, that is given to the Large Language Model.

2. The LLM elaborates the translation in Cypher and quries the Neo4J graph.

3. LLM receives the answer by the graph and translates it back in natural language.

The application works extremely well for simple question, moreover furnishing few examples to the template it can answer even to difficult queries also involving complex graph algorithms.

6.2 Application Demonstrations

To demonstrate the functionality of the developed application, we will present several illustrative examples.
As previously explained, the application performs exceptionally well for "simple" queries.

Below, we present examples of both simple and more complex queries. At the end of the chains, the user's natural language query is shown first, followed by the graph's response as interpreted by the LLM.

6.2.1 Counting Activities

In this example, the user queries the graph to count all activities. The LLM interprets the natural language query, generates the corresponding Cypher query, and executes it against the graph. The response indicates that there are six activities in the graph.

```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (n:Activity) RETURN count(n)
Full Context:
[{'count(n)': 6}]

> Finished chain.
count all the activities
There are 6 activities.
```

Figure 3: Counting Activities in the Graph

6.2.2 Identifying the Most Frequently Used Entity

In this example, the user queries the graph to identify the most frequently used entity. The LLM interprets the natural language query, generates the corresponding Cypher query, and executes it against the graph. The response indicates that the entity with the ID 'entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1' is the most frequently used.

```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (a:Activity)-[:USED]->(e:Entity)
WITH e, COUNT(a) as count
ORDER BY count DESC
RETURN e.id LIMIT 1
Full Context:
[{'e.id': 'entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1'}]

> Finished chain.
return the id of the entity used the most
The id of the entity used the most is entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1.
```

Figure 4: Identifying the Most Frequently Used Entity

6.2.3 Determining the Heaviest Activity

In this example, the user queries the graph to determine the heaviest activity. The LLM interprets the natural language query, generates the corresponding Cypher query, and executes it against the graph. The response indicates that the activity with the ID ‘activity:a28bd41d-7ec7-4264-a38d-5de67aa9f645‘ has the most features.
It turns out that for the LLM heaviest is equal to the actvity with most entity attached.

```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (a:Activity)-[:USED]->(e:Entity)
WITH a, size(collect(e)) as weight
ORDER BY weight DESC
RETURN a.id LIMIT 1
Full Context:
[{'a.id': 'activity:a28bd41d-7ec7-4264-a38d-5de67aa9f645'}]

> Finished chain.
return the id of the heaviest activity
The id of the heaviest activity is a28bd41d-7ec7-4264-a38d-5de67aa9f645.
```

Figure 5: Determining the Activity with the Most Features

6.2.4 Identifying the Most Utilized Entity

In this example, the user queries the graph to identify the most utilized entity. The LLM interprets the natural language query, generates the corresponding Cypher query, and executes it against the graph. The response indicates that the entity with the ID ‘entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1‘ is the most frequently used.

```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (a:Activity)-[:USED]->(e:Entity)
WITH e, COUNT(a) as count
ORDER BY count DESC
RETURN e.id LIMIT 1
Full Context:
[{'e.id': 'entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1'}]

> Finished chain.
return the id of the entity used the most
The id of the entity used the most is entity:92ab3b4e-88ba-4213-b01e-865dd4672fd1.
```

Figure 6: Identifying the Most Utilized Entity

6.2.5 Counting Communities

In this example, the user queries the graph to count all communities using the Louvain method. The LLM interprets the natural language query, generates the corresponding Cypher query, and executes it against the graph. The response indicates that there are six communities in the graph, identified using the specified relationships and including both entity and activity nodes in the projection.

```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
CALL gds.graph.drop('proj', false)
YIELD graphName AS droppedGraph
WITH droppedGraph
CALL gds.graph.project('proj', ['Activity', 'Entity'], {
  WAS_DERIVED_FROM:{orientation:'NATURAL'},
  WAS_GENERATED_BY:{orientation:'NATURAL'},
  WAS_INVALIDATED_BY:{orientation:'NATURAL'},
  USED:{orientation:'NATURAL'}
})
YIELD graphName AS projectedGraph
WITH projectedGraph
CALL gds.louvain.mutate('proj', {mutateProperty:'communityId'})
YIELD communityCount
RETURN communityCount AS result
Full Context:
[{'result': 6}]

> Finished chain.
Count all communities using Louvain on WAS_DERIVED_FROM, WAS_GENERATED_BY, WAS_INVALIDATED_BY, USED; including both entity and activity in the projection
There are 6 communities using Louvain on WAS_DERIVED_FROM, WAS_GENERATED_BY, WAS_INVALIDATED_BY, USED; including both entity and activity in the projection.
```

Figure 7: Counting Communities Using Louvain

7 Conclusion

In this project, we successfully achieved our initial objective of analyzing graph provenance to extract specific information and insights. To accomplish this, we utilized various tools such as Graph Data Science (GDS) and implemented a Large Language Model (LLM)-based query generator. The results of our analysis were consistent with our expectations, demonstrating the efficacy of our methods.

The fine-grained provenance data obtained from our preprocessing pipelines posed challenges in terms of query time efficiency and accessibility for users unfamiliar with Cypher, the query language for graph databases.

The use of algorithms applied to graphs (via GDS) has demonstrated that there is room for the synthesis of certain information. It is likely that the high level of detail in provenance documentation, as integrated into the DPDS tool, is redundant and significantly burdens the graph.
The integration of an LLM-based approach significantly mitigated these challenges by simplifying the query process, making it more user-friendly and accessible to a broader range of users.

This project demonstrated that the combination of GDS tools and LLM-based query generation is a powerful approach for graph provenance analysis. It provides detailed, coherent insights while addressing the challenges of query complexity and user accessibility. Future work could explore the synthesize of the provenance to get a simpler and faster research.

References

[1] Groq. URL: `https://groq.com/`.

[2] Langchain. URL: `https://python.langchain.com/v0.2/docs/introduction/`.

[3] Adriane Chapman, Luca Lauro, Paolo Missier, and Riccardo Torlone. Supporting better insights of data science pipelines with fine-grained provenance. *ACM Transactions on Database Systems*, 49(2):1–42, April 2024. URL: `http://dx.doi.org/10.1145/3644385`, `doi:10.1145/3644385`.

[4] Moreau Luc, Cheney James, and Missier Paolo. Constraints of the prov data model. 2013. URL: `https://www.w3.org/TR/prov-constraints/`.

[5] Neo4j. Graph algorithms. URL: `https://neo4j.com/docs/graph-data-science/2.6/algorithms/`.

[6] Neo4j. Repo github. URL: `https://github.com/pasqualeleonardolazzaro/DPDS_analysis.git`.

A Louvain method for community detection

The Louvain method for community detection is a method to extract non-overlapping communities from large networks created by Blondel et al. from the University of Louvain (the source of this method's name). The method is a greedy optimization method that appears to run in time $O(n \cdot \log n)$ where n is the number of nodes in the network.

The inspiration for this method of community detection is the optimization of modularity as the algorithm progresses. Modularity is a scale value between -0.5 (non-modular clustering) and 1 (fully modular clustering) that measures the relative density of edges inside communities with respect to edges outside communities. Optimizing this value theoretically results in the best possible grouping of the nodes of a given network. But because going through all possible iterations of the nodes into groups is impractical, heuristic algorithms are used.

In the Louvain method of community detection, first small communities are found by optimizing modularity locally on all nodes, then each small community is grouped into one node and the first step is repeated. The method is similar to the earlier method by Clauset, Newman, and Moore that connects communities whose amalgamation produces the largest increase in modularity. The Louvain algorithm was shown to correctly identify the community structure when it exists, in particular in the stochastic block model.

B K-core decomposition

The K-core decomposition constitutes a process that separates the nodes in a graph into groups based on the degree sequence and topology of the graph.
The term *i-core* refers to a maximal subgraph of the original graph such that each node in this subgraph has a degree of at least i. The maximality ensures that it is not possible to find another subgraph with more nodes where this degree property holds.

The nodes in the subgraph denoted by *i-core* also belong to the subgraph denoted by *j-core* for any $j < i$. The converse, however, is not true. Each node u is associated with a core value which denotes the largest value i such that u belongs to the *i-core*. The largest core value is called the degeneracy of the graph.
Standard algorithms for K-core decomposition iteratively remove the node of lowest degree until the graph becomes empty. When a node is removed from the graph, all of its relationships are removed, and the degree of its neighbors is reduced by one. With this approach, the different core groups are discovered one-by-one.
K-core decomposition can have applications in several fields ranging from social network analysis to bioinformatics. Some of the possible use-cases are presented here.

C The Weakly Connected Components Algorithm

The Weakly Connected Components (WCC) algorithm finds sets of connected nodes in directed and undirected graphs. Two nodes are connected if there exists a path between them. The set of all nodes that are connected with each other form a component. In contrast to Strongly Connected Components (SCC), the direction of relationships on the path between two nodes is not considered. For example, in a directed graph $(a) \rightarrow (b)$, a and b will be in the same component, even if there is no directed relationship $(b) \rightarrow (a)$.

WCC is often used early in an analysis to understand the structure of a graph. Using WCC to understand the graph structure enables running other algorithms independently on an identified cluster .

D Eigenvector Centrality

In graph theory, **eigenvector centrality** (also called eigencentrality or prestige score) is a measure of the influence of a node in a connected network. Relative scores are assigned to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.

Google's PageRank and the Katz centrality are variants of the eigenvector centrality.

Using the adjacency matrix to find eigenvector centrality

For a given graph $G := (V, E)$ with $|V|$ vertices let $A = (a_{v,t})$ be the adjacency matrix, i.e. $a_{v,t} = 1$ if vertex v is linked to vertex t, and $a_{v,t} = 0$ otherwise. The relative centrality score, x_v, of vertex v can be defined as:

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in V} a_{v,t} x_t$$

where $M(v)$ is the set of neighbors of v and λ is a constant. With a small rearrangement this can be rewritten in vector notation as the eigenvector equation:

$$A\mathbf{x} = \lambda\mathbf{x}$$

In general, there will be many different eigenvalues λ for which a non-zero eigenvector solution exists. However, the connectedness assumption and the additional requirement that all the entries in the eigenvector be non-negative imply that only the greatest eigenvalue results in the desired centrality measure. The v-th component of the related eigenvector then gives the relative centrality score of the vertex v in the network. The eigenvector is only defined up to a common factor, so only the ratios of the centralities of the vertices are well defined. To define an absolute score, one must normalise the eigenvector e.g. such that the sum over all vertices is 1 or the total number of vertices n. Power iteration is one of many eigenvalue algorithms that may be used to find this dominant eigenvector. Furthermore, this can be generalized so that the entries in A can be real numbers representing connection strengths, as in a stochastic matrix.

E Betweeness Centrality

In graph theory, betweenness centrality is a measure of centrality in a graph based on shortest paths. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimized. The betweenness centrality for each vertex is the number of these

shortest paths that pass through the vertex.
Betweenness centrality was devised as a general measure of centrality; it applies to a wide range of problems in network theory, including problems related to social networks, biology, transport, and scientific cooperation.

Betweenness centrality finds wide application in network theory; it represents the degree to which nodes stand between each other.