NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Adding Provenance to an Existing Python Project and Showcasing it through Graph Models

Data Modelling Course

Professor Carlos Damásio
FCT NOVA - 2024/2025
Francisco Barros - Nº69333
Francisco Silva - Nº67548

Abstract 2
Introduction 2
Provenance 3
Python Provenance Library 4
Neo4J - Graph Database 4
Code 5
Provenance Package 5
Modifications 5
Applied Fixes 6
FastAPI Social Mock Server 7
Challenges with FastAPI Integration 8
Semi-Load Test For Data Generation 9
Cypher Queries 9
Results 10
Provenance Package Output 10
Neo4J Graph 11
Lineage of a Post 15
Interactions of a User 16
All Edited Posts 16
Pair of Users with most Interactions with Each Other 17
Users Who Never Interacted with Each Other Directly 17
Possible Solutions and other Implementations in the Future 18
Changing Serialization in Provenance Package 18
Adding Graph Database as Type of Data Repo 18
Conclusion 19
Bibliography 20
Appendix 21

Abstract

This project aims to investigate the usage of data provenance open-source libraries to implement into pre-existing projects without provenance systems previously built in, and in doing so exporting the artifact data into graph databases and see the advantages and disadvantages of using such a system to analyze provenance.

Introduction

Data provenance (Mucci, 2024), the metadata that records the details of the data origins by capturing its transformations and its various processes, is extremely useful in data systems by being able to track information and being able to check its authenticity. Therefore, being able to easily add provenance to existing projects by simply adding a package [https://pypi.org/project/provenance/] and then annotating relevant functions would be extremely useful to any project, and this is what this project aims to test by using the provenance library made in python and exporting the resulting artifacts into a graph database, such as Neo4J, where provenance can be more easily seen visually.

This theme has been chosen as the project of the Data Modelling course due to its connection in both tracking and modelling of data using provenance and the usage of graph models taught in the first half of the course.

The code for this project can be seen at the following repository - https://github.com/FrancisMB2001/project-md-provenance

Provenance

Provenance, as written about in the introduction, is the complete history of the data, including its origin (where it came from, when, how was it uploaded, etc…), to the transformations it suffered doing in its lifetime.

The primary purpose of using provenance is to provide transparency of the data in the system, accountability for the data created and the authenticity of said data. This is usually important in all systems, but especially machine learning systems where decisions depend on the quality of the given data, and by keeping records of how reliable this data is, better decisions can be made.

Therefore using provenance in any system can be thought of as a positive addition to it, as it subtracts nothing from the original system but instead adds another layer of detail to it. However, adding provenance to a system can be tricky, since keeping track of data is tricky, due to the amount of information that passes through a system, especially in its transformations through functions in the code.

Provenance can also be divided into various levels of granularity, which consists in the level of detail captured about the origin and transformations of the data. It can go from a high-level where it only tracks the sequences of operation (data input -> transforming the data -> data output), to a much lower level where the entire data flow is monitored, from its exact transformations and operations in every single step.

A more detailed analysis of provenance can be read in Pierre Senellart's paper (Senellart, 2019) about the provenance in databases with the usage of mathematical concepts combined with evaluation queries to check provenance in a database.

Python Provenance Library

The “Provenance” package [https://pypi.org/project/provenance/] (bmabey & scott_nielsen, 2020) is a library made for python projects with the objective of adding provenance to them by simply annotating functions with “`@provenance`” and letting the package take care of everything else.

It works by caching the computed results of the annotated functions across various tiered stores (disk, S3 and SFTP), with provenance then being tracked and stored in an artifact repository, with this repository able to be configured and the location of where it’s stored changed.

In theory, the package would then be able to create artifacts that would be able to track how any object is created, the transformations it suffered and the result it currently stores.

The package was specifically designed to be used in machine learning pipelines and projects, due to how important provenance is to these projects, as previously explained in the “Provenance” section. We however, used a different type of project to see how flexible the package is in adapting to other types of projects, and due to our limited knowledge in the machine learning area.

Neo4J - Graph Database

Neo4J is a graph database management system designed to manage highly connected data, in other words, with a lot of relations. With that design in mind, it’s the perfect choice to store and query provenance data, due to the lineage of data being the primary point of provenance.

It works by storing data in a graph format, where nodes represent entities and arcs represent relations between the entities, in our project the nodes being data values and functions, and the arcs being the calls and returns of the functions with said data nodes as inputs/outputs.

Code

Like previously stated in the Introduction, all the code discussed in this report can be seen in the following github repository - https://github.com/FrancisMB2001/project-md-provenance

The project's code is basically separated into three main categories, the social mock app annotated with provenance package (`main.py`), the load test to generate provenance data (`test_provenance.py`) and finally the cypher queries that are used to transform and import data into Neo4J and then query the data in graph form.

Provenance Package

Modifications

To integrate provenance into the application, the online documentation was followed. From the very start challenges were found due to the package itself having outdated libraries and functions that were exclusively for Linux machines, while the project development was on Windows. After solving these issues, provenance was configured with success to store its results in memory. However, to export the results into Neo4J, it was necessary to store them persistently, to which PostgreSQL was used.

To facilitate in the usage of the project we tried implementing a local version of the provenance package (custom_provenance folder) that the main code would use instead of the pip version, however, after multiple attempts we didn't manage to properly implement due to issues related with the local repositories that the provenance package uses, therefore the installation of the provenance package remains the same, but the user will need to manually go into the package installation directory and do some fixes in order to be able to run the provenance package.

Applied Fixes

To develop this project Windows machines were used. Provenance was installed within anaconda, so the following directories might be slightly different based on where *pip* is installed. With this note in mind the modifications are the following:

If installed globally (not using a virtual env), modify the global packages, in the example below using Anaconda 3, in:

`C:\Users\{Username}\anaconda3\Lib\site-packages\provenance\utils.py`

If installed using a virtual environment instead, navigate to the project folder and change the packages installed in the virtual environment, in:

`{Project-Location}\{Virtual-Env-Name}\Lib\{Python-Version}\site-packages\provenance\utils.py`

The import

```
from collections import OrderedDict, Sequence
```

is outdated and should be changed to

```
from collections import OrderedDict
from collections.abc import Sequence
```

If installed globally, in:

`C:\Users\{Username}\anaconda3\Lib\site-packages\provenance\repos.py`

If installed using a virtual environment, in:

`{Project-Location}\{Virtual-Env-Name}\Lib\{Python-Version}\site-packages\provenance\repos.py`

On function `def _process_info():` we can't use `p.num_fds()` on Windows machines, so the whole function was switched by the code that can be seen on the appendix section as (1).

The configuration of the yaml file was done through the provenance guide, accessible on https://provenance.readthedocs.io/en/latest/intro-guide.html.

To properly connect provenance with the postgres database the

```
db: postgresql://localhost/provenance-intro
```

line was slightly modified to

```
db:
postgresql://<myPostgresUsername>:<myPostgresPassword>@localhost/provenance-dbapi
```
.

Where <**myPostgresUsername**> and <**myPostgresPassword**> are the personal credentials of postgres. Note, assuming no postgres username was set, the default is “postgres”.

FastAPI Social Mock Server

A small application was made in python (with the help of ChatGPT) to simulate a social network application with various endpoints to mock the normal operations a user does in a social network such as:

- Register a user
- Login
- Creating or Editing a Post
- Commenting a Post, or editing said comment
- Liking posts or comments

The application is minimal and was done using "FastAPI" as the web framework to build the HTTP endpoints and "uvicorn" to host the server, only having the code to the aforementioned operations and using a very simple dictionary to serve as in-memory database.

The endpoints functions were then annotated with "`@p.provenance`" to then utilize the provenance package, thus in theory being able to add provenance to the application.

This mock server was not our first choice as a test bed for the provenance library, however, after multiple days spent trying to find a good open-source python project that would be a good fit, we couldn't find one. This was due to the project being outdated, having major issues (such as extremely slow in running), the tests being incomplete or the code being so complex it would be impossible to properly annotate and get useful date out of it without having to annotate the entire project and understand what each function was doing. This was also surely due to our inexperience in python projects, specifically in django.

To run the application just navigate to the project folder, and do the following terminal commands:

1. `python3 -m venv prov-testing`
   a. Generates a new virtual environment for the project, optional but recommended due to needing to change the source code of the provenance, and the location of it on the virtual environment is much easier to access than on the global packages.
2. `source prov-testing/bin/activate`
   a. Activates the virtual environment previously created. Skip the command if not using a virtual environment.
3. `pip install fastapi uvicorn provenance pyyaml`
   a. Installs the required packages to run the server.
4. `uvicorn main:app`
   a. Starts the server.

Challenges with FastAPI Integration

When transitioning from the example in the documentation to the application (which uses FastAPI), significant challenges were encountered. Functions that didn't use the FastAPI annotations worked as expected, but for the ones that did use, instead of storing the data of the objects (e.g., username, password from user) in provenance (stored in PostgreSQL), only the object type was stored. This seems specifically tied to incompatibility between the types of objects FastAPI uses (*pydantic, [https://docs.pydantic.dev/latest/]*) and the conversion library Provenance uses (*joblib,* [https://joblib.readthedocs.io/en/stable/]) since *joblib* is particularly focused on pipeline jobs.

On a first attempt to make a workaround for this issue, classes with functions inside were defined with each function having the provenance annotation “`@p.provenance`”. However, this approach was unsuccessful. Although the application initially ran, any method calls (e.g., POST /register) caused the tests to fail because provenance did not return the expected values, despite them being initially sent.

On a second workaround, an attempt was made to modify the FastAPI framework itself to integrate provenance. Unfortunately, this approach was also unsuccessful, as the framework could not be made to run correctly with the provenance modifications.

As a third and final attempt, we started using auxiliary functions that receive primitive data as arguments, and having these functions being annotated instead of the endpoint functions. While this is more of a “hack” than a proper solution it works due to the provenance package monitoring every value that enters the function, therefore we’re “sideloading” the information that enters and leaves the endpoint with the attributes of the object instead of the object itself (for example, a post is created and saved, and then the auxiliary function is called with the values of the post). This works due to us using primary keys (unique usernames, post ids, comment ids, etc…) in our object classes and then using them to connect them in the graph later on.

Semi-Load Test For Data Generation

To populate the database, a small python script was made (`test_provenance.py`) to generate provenance data by calling the endpoints multiple times of the mock social app with HTTP requests.

What is done :

1. A static user is registered and logged on, he creates two posts, edits one of them, creates two comments and likes his first post and comment. This is done to always have a static value of provenance we can check the correctness of.
2. Multiple users are then created, registered and logged on. They create a multitude of posts and comments, some editing their created posts and some liking certain posts and comments. The number of resources created is random.

After this, the PostgreSQL connected through the config of the provenance package will have multiple rows added to its "artifacts" table.

To run this test, simply navigate to the project folder and run the following command in the terminal : "`$ python test_provenance.py`". Attention, that the server needs to be running first, otherwise there will be no response to the HTTP requests.

Cypher Queries

Cypher queries were created to generate the graphs based on the CSV exported from PostgreSQL. When running the Cypher queries, it is only necessary to double-check the *`inputs_json`* parameter, as any structural differences in the data format will only occur within this specific parameter.

This code can be found in the project directory with the name "*prov_social_data_gen.cypher*".

The queries generate the nodes, relations and properties of functions *`registT`*, *`loginT`*, *`createPostT`*, *`createCommentT`*, *`editPostT`*, *`likePostT`*, and *`likeCommentT`*. For each function:

- Relevant nodes, namely **User**, **Post**, **Comment**, and **Action**, are created, with their respective properties, and/or matched in the graph.
- Relationships like **REGISTERED**, **LOGGED_IN**, **CALLS**, **CREATED**, **EDITED**, **ON**, and **LIKED** are generated accordingly to represent the interactions and dependencies between users, posts, comments, and actions.

To achieve this, we first read the CSV file and use the APOC plugin to facilitate data conversion from the file, making it easier to handle in Cypher. The CSV is read in rows, and APOC is used to split and process the data.

The graph is generated by firstly creating the nodes with their properties, and then we establish the majority of the relationships between them.

The Cypher queries created for analytical purposes will be discussed later in the subsections appropriately identified under Neo4J Graph on Results section, and can be seen in code in the file "*prov_analytics.cypher*".

Results

Provenance Package Output

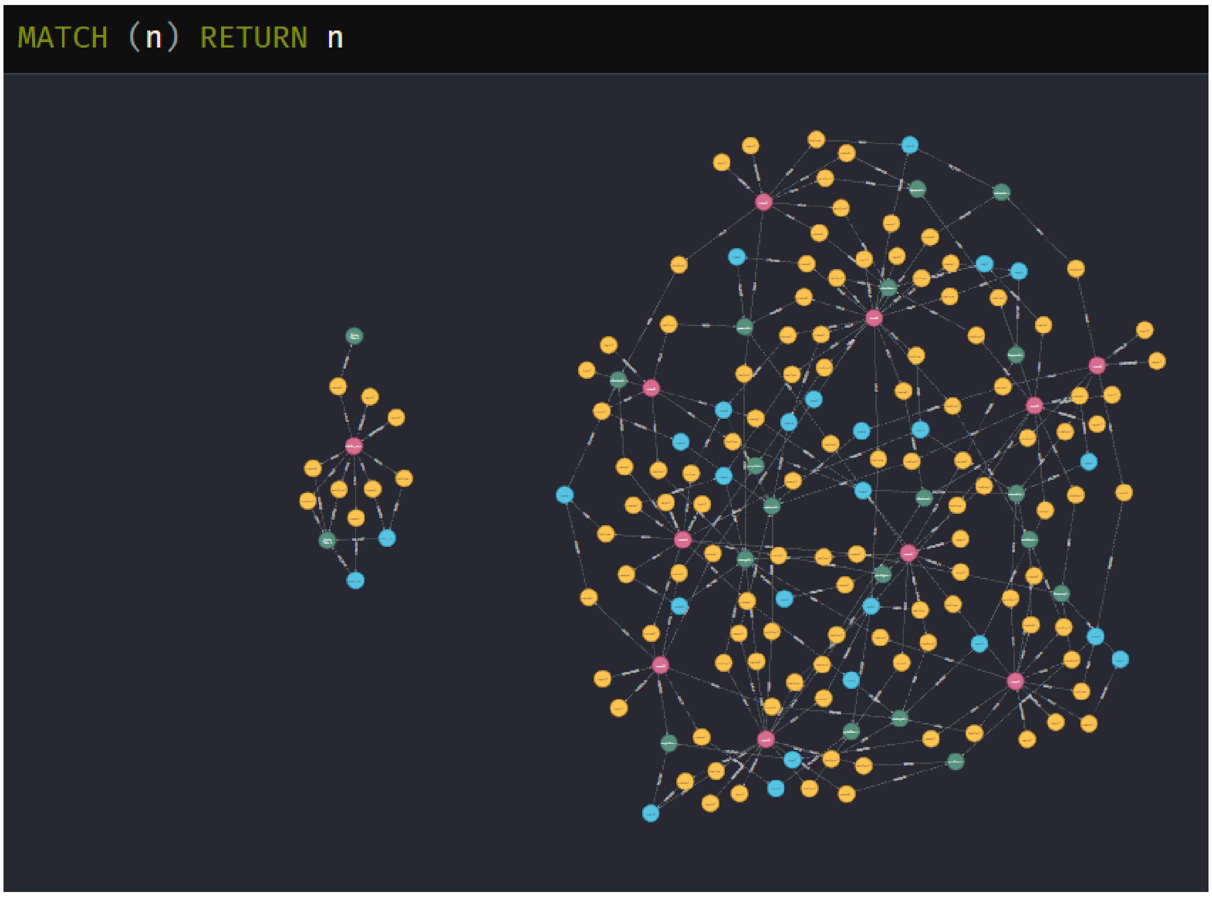
The provenance package generates a CSV file that contains multiple variables, such as *provenance_id*, *value_id*, *run_id*, *name*, *fn_name*, *computed_at*, *added_at*, and *inputs_json*, among others. However, for our specific case, only a small subset of this data is mandatory.

Specifically, we utilize the content of *inputs_json* and *fn_name* as mandatory data. The *inputs_json* column contains the data that is being transferred, while *fn_name* allows us to identify the function that was executed. We optionally include the *computed_at* field to add some metadata.

The remaining columns in the resulting CSV file are completely discarded. This raises potential concerns about the suitability of the use of the provenance package on our project, since, while not exclusively, the provenance package is designed to track the execution of multiple functions, with emphasis on nested function calls, allowing a comprehensive overview of the execution path. Contrary to our endpoints which compute only a single primary function (excluding auxiliary functions).

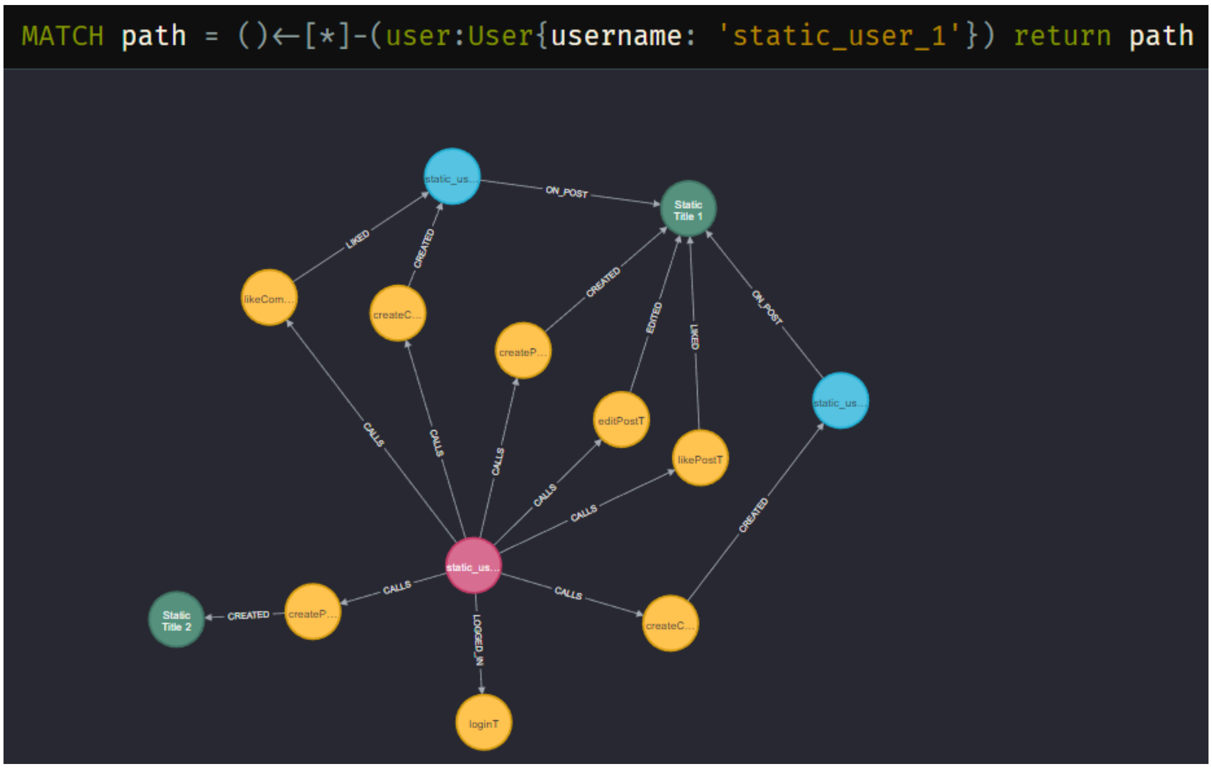
Neo4J Graph

After generating the data through the FastAPI app and our test file, exporting the data from PostgreSQL as a CSV file and finally transforming it and importing it into Neo4J, the result can then be seen visually in the database:



```
MATCH (n) RETURN n
```

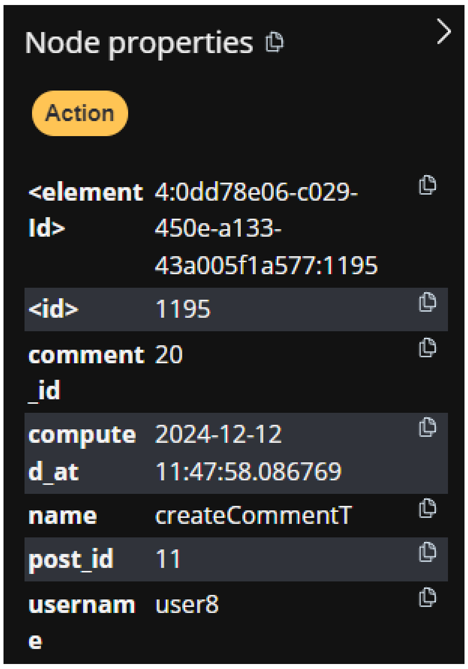
As can be seen, there's basically two graphs in the model, the first one (smaller and on the left) consists of the first lines of the load test (talked about in the "Semi-Load Test For Data Generation" section) where the static user is created and then does it's actions (creating posts, editing, liking and commenting). This user doesn't interact with any other user or their content in the system, therefore being isolated, and serves as a guiding stone since the rest of the test is randomized while the static user is not. The graph can be seen more in detail in the following image:



```
MATCH path = ()←[*]-(user:User{username: 'static_user_1'}) return path
```

The model schema has the following properties:

- Yellow Nodes - Actions (or functions), that are called by a user.
    - **computed_at** - Timestamp of when the function was computed.
    - **name** - Name of the function called.
    - **username** - Username of the user that called the function.
    - Other attributes related to the specific function (respectively **post_id** or **comment_id**, that depends on the **name** attribute of the action node, for example when name is *createPostT* or *createCommentT*).

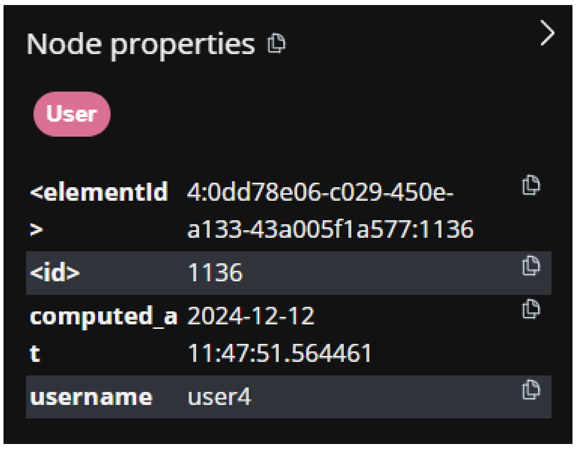


Node properties

Action

<element Id>	4:0dd78e06-c029-450e-a133-43a005f1a577:1195
<id>	1195
comment_id	20
computed_at	2024-12-12 11:47:58.086769
name	createCommentT
post_id	11
username	user8

- Pinkish Red Nodes - Users.
    - **computed_at** - Timestamp of when the user node was created.
    - **username** - Username of each user, it is used as a unique identifier.

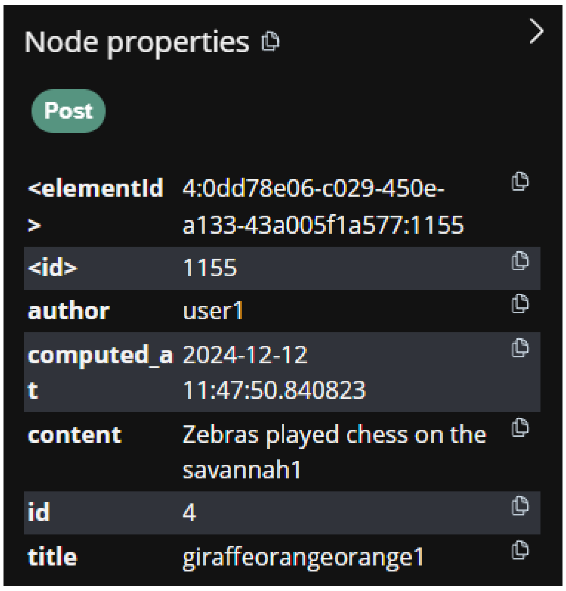


Node properties

User

<elementId>	4:0dd78e06-c029-450e-a133-43a005f1a577:1136
<id>	1136
computed_at	2024-12-12 11:47:51.564461
username	user4

- Green Nodes - Posts, created by users.
    - **computed_at** - Timestamp of when the post node was created.
    - **content** - The text content of the post written by the user.
    - **id** - A unique identifier for the post.
    - **title** - The headline or brief summary of the post, created by the user to describe its content.

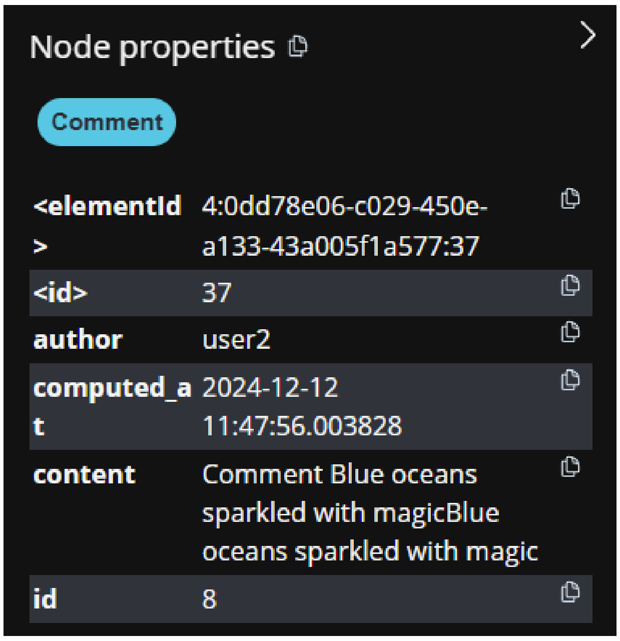


Node properties

Post

<elementId>	4:0dd78e06-c029-450e-a133-43a005f1a577:1155
<id>	1155
author	user1
computed_at	2024-12-12 11:47:50.840823
content	Zebras played chess on the savannah1
id	4
title	giraffeorangeorange1

- Blue Nodes - <u>Comments</u>
    - **author** - The user who created the comment, the username is stored as value.
    - **computed_at** - Timestamp of when the comment node was created.
    - **content** - The text content of the comment written by the user.
    - **id** - A unique identifier for the comment.



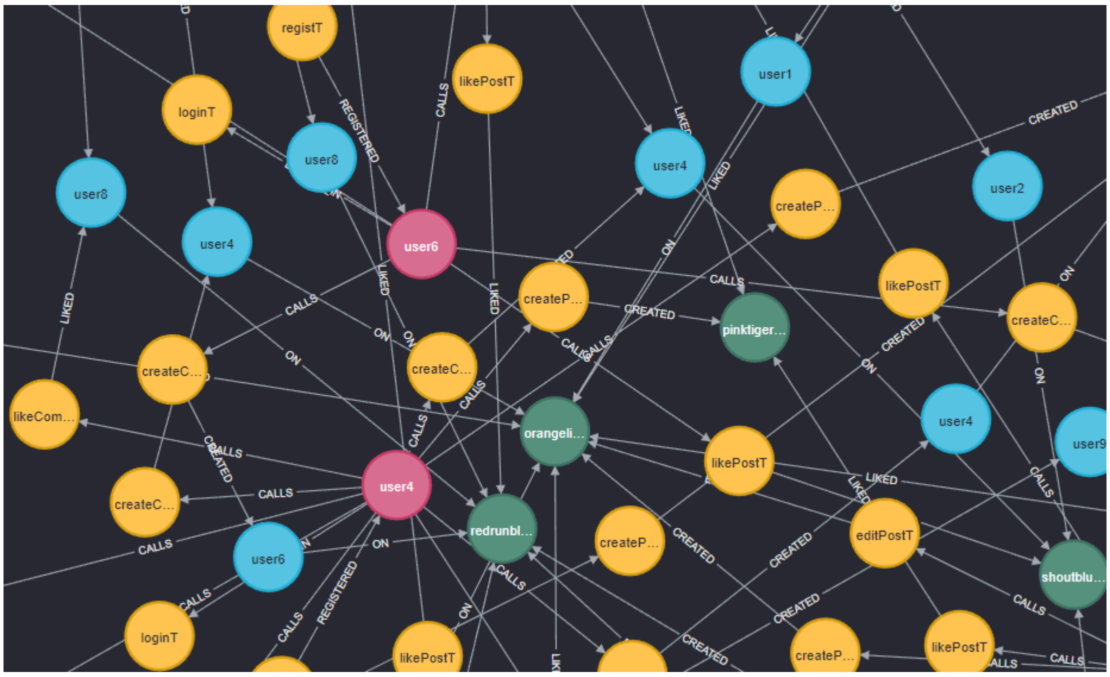
Node properties

Comment

<elementId>	4:0dd78e06-c029-450e-a133-43a005f1a577:37
<id>	37
author	user2
computed_at	2024-12-12 11:47:56.003828
content	Comment Blue oceans sparkled with magicBlue oceans sparkled with magic
id	8

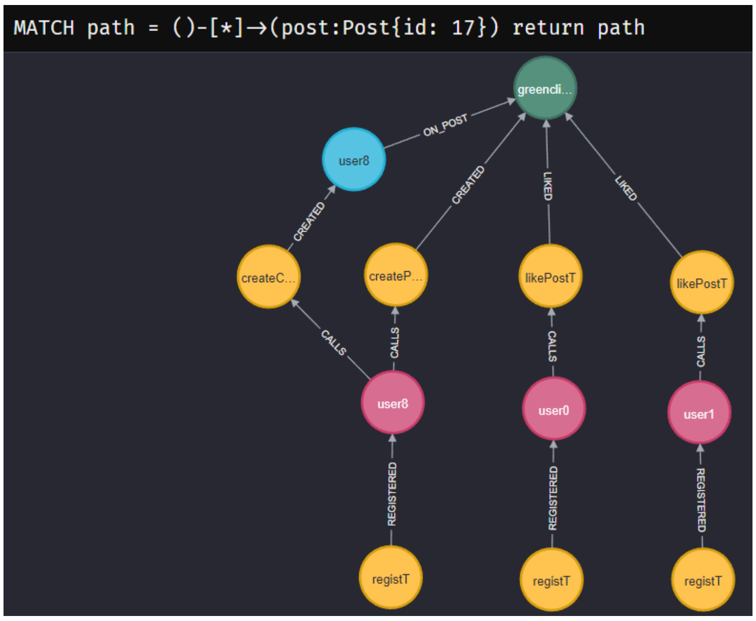
Having explained the schema of the model, with the smaller static user graph as an easier to understand example, the graph on the right comes from the provenance data of 10 users interacting with the system, that in any way or another relate to one or another through their actions (such as liking another users post or commenting on their posts), therefore making a graph where every node is connected (though there is a very slim possibility while running the test, that one user creates no posts, no comments and doesn't like any other users posts or comments).

The graph, as can be seen in the below image, can be quite confusing due to the amount of nodes and relations in the model. To better illustrate provenance using the graph, we have created multiple cypher queries to better analyze the graph. The queries written as code can each be seen in the file "*prov_analytics.cypher*".



Lineage of a Post

As one of the provenance's main points, get the lineage of a post where we can see the creation of a post (by who, when and what) and the transformations it suffered (post edits, comments or likes).

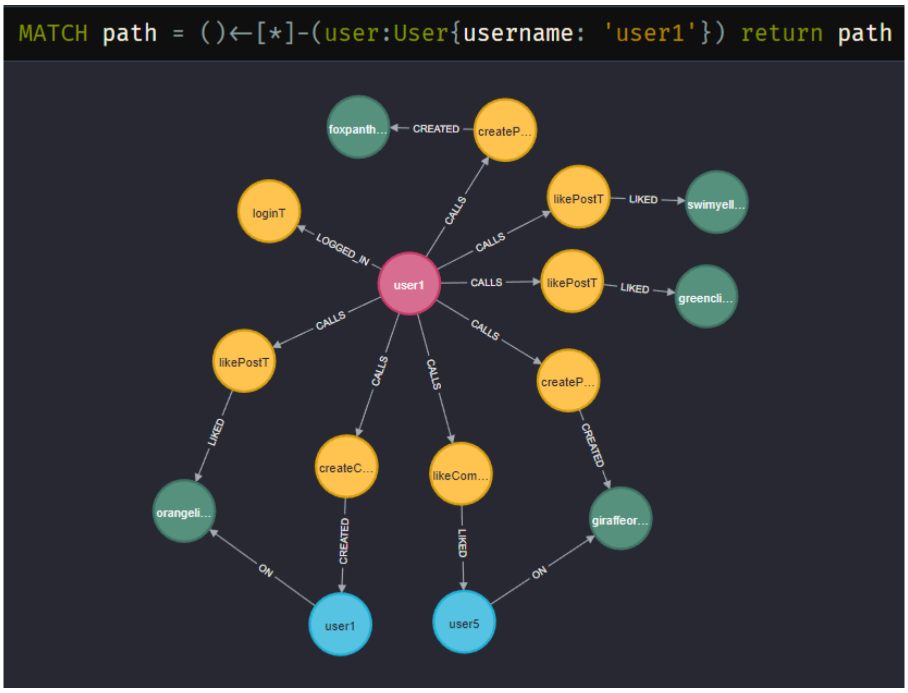


```
MATCH path = ()-[*]→(post:Post{id: 17}) return path
```

The specific query filters a specific post by its id, in this case post with the id 17 with the title "greenclimbdolphin8". From the result we can see that the user8 created the post and commented on it. Besides this, user0 and user1 liked the post.

Interactions of a User

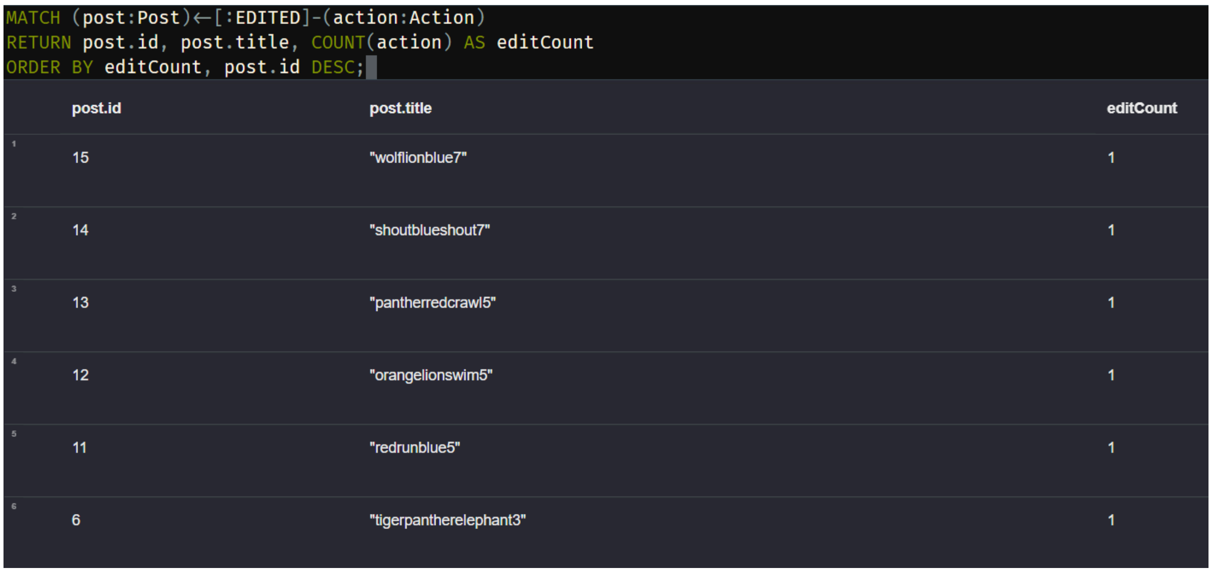
Get the interactions a user has done in the system, these include logging in, creating posts and comments, and liking posts and comments.



```
MATCH path = ()←[*]-(user:User{username: 'user1'}) return path
```

All Edited Posts

This query returns all posts that have suffered edits, unfortunately our provenance testing only decides if a user edits his post once or never, therefore the max amount of edits a post can have is 1. Still, the query returns the essential information of showcasing the posts that were edited.

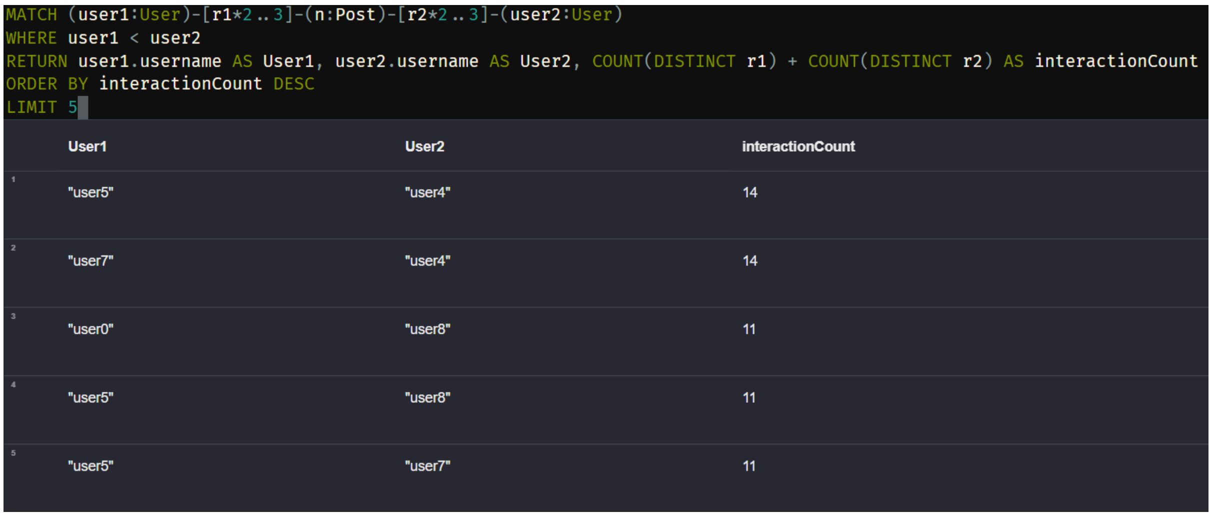


```
MATCH (post:Post)←[:EDITED]-(action:Action)
RETURN post.id, post.title, COUNT(action) AS editCount
ORDER BY editCount, post.id DESC;
```

	post.id	post.title	editCount
1	15	"wolflionblue7"	1
2	14	"shoutblueshout7"	1
3	13	"pantherredcrawl5"	1
4	12	"orangelionswim5"	1
5	11	"redrunblue5"	1
6	6	"tigerpantherelephant3"	1

Pair of Users with most Interactions with Each Other

Show the top 5 pairs of users with the most interactions with each other, this means users who liked each other's post or comments, or commented on each other's posts.

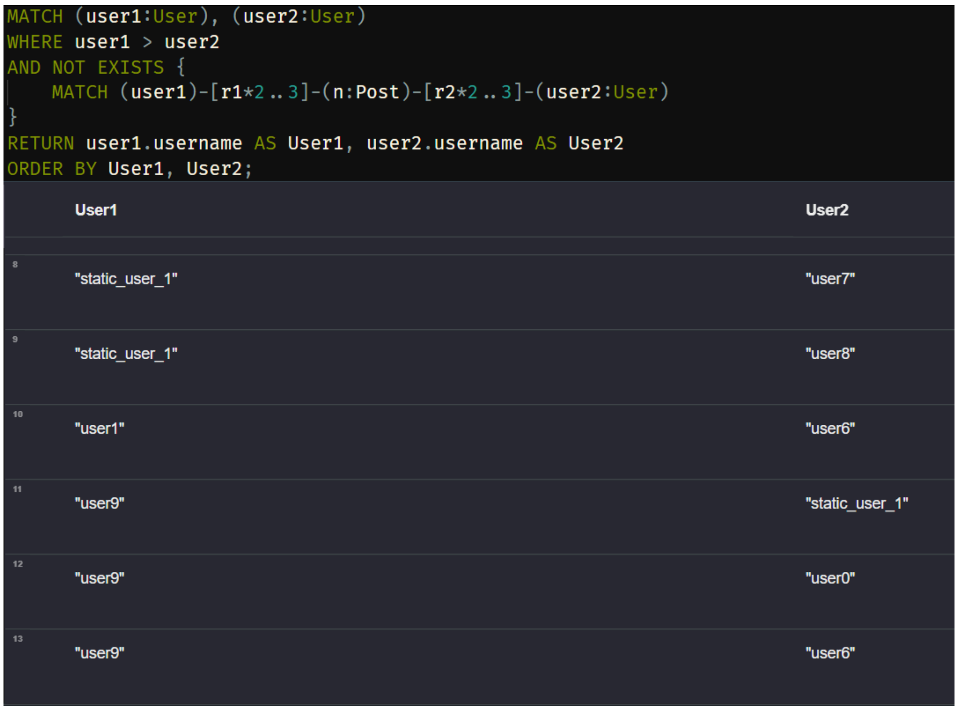


```
MATCH (user1:User)-[r1*2..3]-(n:Post)-[r2*2..3]-(user2:User)
WHERE user1 < user2
RETURN user1.username AS User1, user2.username AS User2, COUNT(DISTINCT r1) + COUNT(DISTINCT r2) AS interactionCount
ORDER BY interactionCount DESC
LIMIT 5
```

User1	User2	interactionCount
"user5"	"user4"	14
"user7"	"user4"	14
"user0"	"user8"	11
"user5"	"user8"	11
"user5"	"user7"	11

As a result, we see user4 with the most interactions with both user5 and user7. This sort of query could be very useful in actual social network systems to understand the actual users that drive the most traffic in the network, thus showcasing usefulness of provenance in a data analysis matter.

Users Who Never Interacted with Each Other Directly



```
MATCH (user1:User), (user2:User)
WHERE user1 > user2
AND NOT EXISTS {
    MATCH (user1)-[r1*2..3]-(n:Post)-[r2*2..3]-(user2:User)
}
RETURN user1.username AS User1, user2.username AS User2
ORDER BY User1, User2;
```

User1	User2
"static_user_1"	"user7"
"static_user_1"	"user8"
"user1"	"user6"
"user9"	"static_user_1"
"user9"	"user0"
"user9"	"user6"

Show the users who have never personally interacted with each other on the system, this means that they have never liked or commented on each other's posts or comments. Similar to the previous in showcasing user behaviour in the system.

Possible Solutions and other Implementations in the Future

Changing Serialization in Provenance Package

One way to fix our issues would be to change the actual serialization code in the provenance package to allow the reading of complex objects and save their attributes in the artifact information. However, there would still be the issue of how the package would know that the object would be the same object codewise due to changing pointer values during runtime, therefore needing an unique attribute (such as an identifier or primary key) to be sure that we are modifying the artifact the object belongs to.

Adding Graph Database as Type of Data Repo

Not as a solution to any specific problem, but a good addition to the library would be the addition of a repo type of graph database besides the existing ones (such as postgresQL or in-memory), therefore automatically making the transformation and conversion of data to a graph database in run-time instead of saving to another type of database, exporting it outside, running transformations queries and then finally being able to utilize it in a graph database.

Conclusion

While in the end we did manage to get some workable and usable data from the provenance package working correctly in Neo4J, and actually making a connected graph which was something we didn't expect to actually manage, there were multiple points we wished worked better (from the testbed project selection to the actual provenance package flexibility).

Having written that, I do think some interesting information did come from this project, specifically what one can already do with said provenance package (even if in a limited capability), but more interestingly, the possible expansion of this package or something that derives from it to implement provenance in a more accessible manner, since the foundation is there.

This new package would then be able to properly add provenance to projects that aren't necessarily within the machine learning area, such as web applications like the one we simulated in the project.

In conclusion, while the project didn't achieve every goal we set out to achieve, I think a lot of interesting knowledge was learnt from this project, from a data modelling point of view with the understanding of using graphs to get information we wouldn't otherwise get (or as easily) in a normal Entity-Relation model database to learning the open-source environment (in python in this specific case) and how we can use community tools and libraries, and help them by fixing or expanding said tools.

Bibliography

References

bmabey & scott_nielsen. (2020, 12 2). *Provenance 0.14.1*. PyPi. Retrieved 11 25, 2024, from https://pypi.org/project/provenance/

Cypher Cheat Sheet. (n.d.). Cypher Cheat Sheet. Retrieved 12 10, 2024, from https://neo4j.com/docs/cypher-cheat-sheet/5/all/

FastAPI. (n.d.). FastAPI. Retrieved December 8, 2024, from https://fastapi.tiangolo.com/

Mucci, T. (2024, 07 23). *What is data provenance?* IBM. Retrieved 11 23, 2024, from https://www.ibm.com/think/topics/data-provenance

Neo4j, Inc. (n.d.). *What is a graph database - Getting Started*. Neo4j. Retrieved December 2, 2024, from https://neo4j.com/docs/getting-started/graph-database/

Senellart, P. (2019). Provenance in Databases: Principles and Applications. *Provenance in Databases: Principles and Applications*, *1*(1), 109. https://pierre.senellart.com/publications/senellart2019provenance.pdf

Appendix

(1)

```
def _process_info():
    pid = os.getpid()
    p = psutil.Process(pid)

    # Check if num_fds exists before calling it (Windows compatibility)
    num_fds = None
    if platform.system() != "Windows":
        try:
                num_fds = p.num_fds()  # Only on Unix-like systems
        except Exception as e:
                num_fds = None  # In case of errors
        else:
            # Use net_connections instead of connections for Windows
            num_fds = len(p.net_connections())
    return {
        'cmdline': p.cmdline(),
        'cwd': p.cwd(),
        'exe': p.exe(),
        'name': p.name(),
        'num_fds': num_fds,
        'num_threads': p.num_threads(),
    }
```