UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAD DE INGENIERÍA INFORMÁTICA DE BARCELONA

Official Máster on Data Science

Property Graph Lab

Semantic Data Management

Daniel Cantabella Vives De La Cortada
Gabriel Zárate Calderón

Barcelona, March 2023

Index

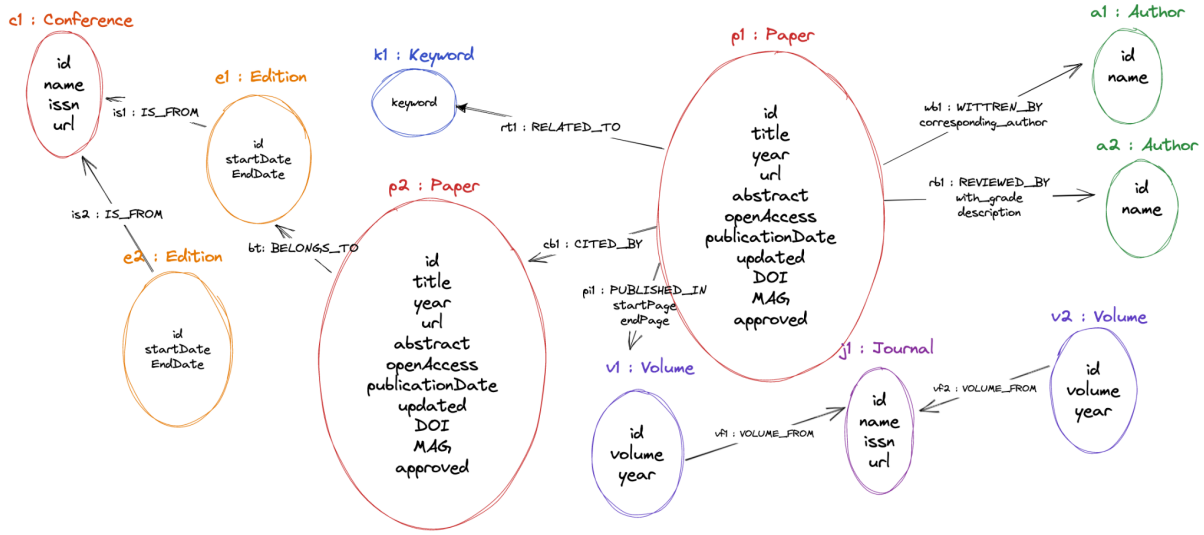
Project Information

The following report will explain the basics of this project. For a more detailed description of the project you can see our git repository program will be located on a git repository in the following link:
https://github.com/DanielCantabella/SDM-Lab1-PropertyGraphs
Before any execution the Neo4j database server needs to be set up as the README.md file indicates.
When the program is executed, it displays a console menu that allows the user to choose which task of the assigned ones wants to execute, so it keeps the modularity asked in the project, that sets that we need to have one program per task, so this menu helps to execute them independently.
If it is the first run you at least need to execute the program for Task A2 before anything to load the data on the database.

A. Modeling, Loading, Evolving

A.1. Modeling

Graph Design:



Node types (in the image, all their attributes are contained inside the nodes):

- **Author**: an author can be either a paper author or a paper reviewer. An author cannot be a reviewer of their own papers.
- **Paper**: scientific paper. A paper can only belong either to one journal or to one conference. A paper can have multiple authors but only one corresponding author. A paper can have multiple reviewers.
- **Keyword**: a paper can contain multiple keywords and a keyword can be contained in many papers.
- **Volume**: a journal volume can contain multiple published papers. Volume refers only to a journal and not to a conference.
- **Journal**: one journal could contain many journal volumes
- **Edition**: a conference edition that can held many papers
- **Conference**: one conference could contain many conference editions

Edges (in the image, all their attributes are contained below the edges name):

- **WRITTEN_BY**: from Paper to Author nodes. it contains a boolean indicating if it is corresponding_author
- **REVIEWED_BY**: from Paper to Author nodes. This author is not actually an author of the paper but its reviewer. An author cannot be the reviewer of its own paper.
- **RELATED_TO**: from Paper to Keyword nodes. A paper can contain many keywords and a keyword can be related with many papers.
- **CITED_BY**: from Paper to Paper node. A paper cannot be cited by itself.

- **PUBLISHED_IN**: from Paper to Volume nodes. Many papers can be published in a journal volume. It contains the starting and ending page where the paper appears.
- **VOLUME_FROM**: from Volume to Journal. A journal can be related with many volumes. A volume can be related to only one journal.
- **BELONGS_TO**: from Paper to Edition. Many papers can belong to the same conference edition.
- **IS_FROM**: from Edition to Conference. A conference can have many editions. An edition can be related to only one conference.

Justifications and assumptions:

- The structure of the graph database was made like this in order to avoid storing the same data more than once and to have an easy understanding.
- As many papers can be related to the same Keyword, representing them as nodes allowed for normalizing the data to avoid repetitions. Representing keywords as nodes provides greater flexibility in the types of queries that can be executed. For example, it becomes possible to query the database for all papers that contain a specific keyword, or to identify papers that share multiple keywords.
- Linking a paper to a conference edition or journal volume was done because it was assumed that one journal has many volumes and a conference has many editions, so one paper can only belong to one of these volumes or editions. This structure allows editions and volumes to be related with different papers, thus avoiding duplicate edition and volume data.
- The reviews structure was made by only linking the reviewers to the paper because of the mentioned statement before (i.e., one paper belongs to only one edition or volume), so it would only have one review associated, so there was no need to have a new node for review and the information per review can be stored on the edge REVIEWED_BY.
- Adding a boolean attribute to the edge was enough to distinguish the corresponding author. Just by filtering by attribute corresponding_author it is possible to find the unique corresponding author.

A.2. Instantiating/Loading

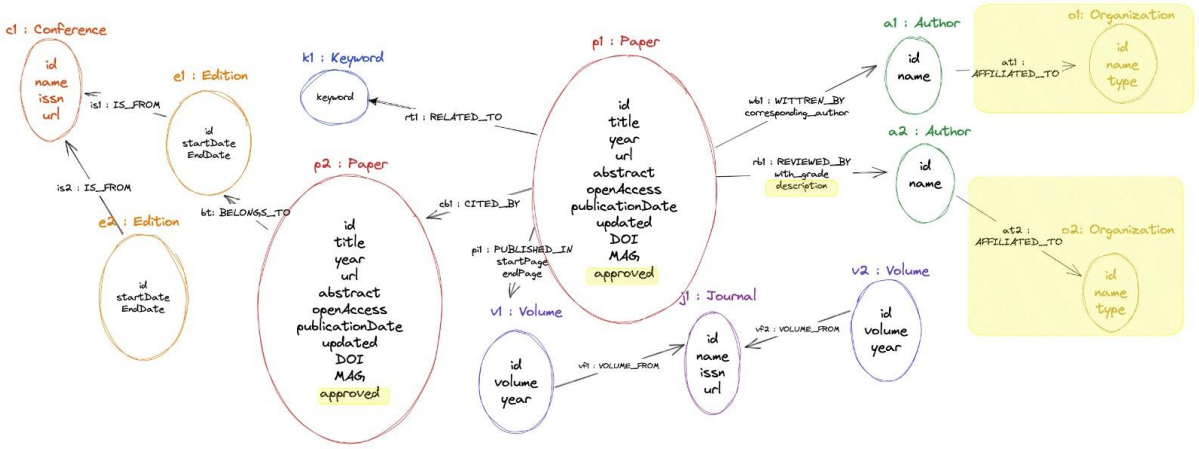
Part of the data we loaded into our graph came from Semantic Scholar. The data consisted of a sample of 100 different examples for each of the contained CSV files. Each CSV file contained information either from scientific papers, from authors or from venues (i.e., conferences and journals). Each of these files contained relationships to other instances, but none of those relationships matched the IDs of the instances in the other files. For this reason, and because our sample was very small (i.e. 100 samples each file), we decided to artificially generate both the relationships and the necessary data not contained in the samples. Here we show the list of the programs we implemented to download and generate the necessary data for the project:

- **a2_getSampleData.py**: this file uses the Semantic Scholar sample API to get the desired datasets as jsonl.gz files.
- **a2_generateCSV.py**: this file reads the jsonl.gz files from the Semantic Scholar API and exports them into CSV files.
- **a2_splitVenues.py**: this file reads the venues file and generates two new files: one with the list of journals and other with the conferences.

- **a2_generateRelations.py**: this file generates the artificial relationships between the different nodes, and exports them into CSV files.
- **a2_importData2Database.py**: this file contains the connection to the neo4j database. It executes the constraints and the creation of the nodes and edges by loading the data from previously generated CSV files.
- **A2_CantabellaZarate.py:** orchestrates the execution of all the previously explained code

NOTE: Since the relationship between nodes were randomly generated, we didn't expect it to make any sense. The objective of this part was to generate the data to be able to work later with our database.

A.3. Evolving the graph



Changes and justification:

- Reviews:
  - Addition of the **description** attribute in REVIEWED_BY to store the review comments.
  - Addition of the approved boolean in Paper, to store if according to the revisions it is correct
    - This change also included a query that checks in each paper if most of the reviewers grade (with_grade attribute in REVIEWED_BY) is 3 or higher (that was the assumed criteria to consider a paper as approved), and it is executed in the file A3_Evolving.py
  - This was done because of how the graph was designed, it was only needed to store the status of a paper in its node because only it was assumed that one paper can only belong to one Journal Volume or to one Conference Edition, so there is only one evaluation. Description attribute was included in the edge because it was easier to go through the revision edges rather than going inside the node of the paper and look at its revision description. Also because it saves space in the node, it is easier to see who wrote the description of the review in an edge rather than looking at the node itself and matching different descriptions with different reviewers.

```
Unset
MATCH (p:Paper)
OPTIONAL MATCH (p)-[e:REVIEWED_BY]->(:Author)
```

```
WITH p, COUNT(e) as numRev, COUNT(CASE WHEN e.with_grade>2 THEN 1 END) as
numApp
WHERE numRev/2 < numApp
SET p.approved = true
```

- Affiliations:
  - Addition of the node Organization that can only be of type University or Company
  - Addition of the edge AFFILIATED_TO from author to organization
  - Also to a new script was created in python to generate new data to add for the organizations
  - This was done because there was no reason to have different nodes between universities or companies in this case of use. We did not split companies and universities since both would have the same attributes and having an attribute type does not difficult the queries over it. Just filtering by type attribute instead of by nodes allows us to have the model more compacted and interpretable. This way we avoid having two different types of nodes.

B. Querying

1. Find the top 3 most cited papers of each conference.

```
Unset
MATCH (c:Conference)<-[:IS_FROM]-(e:Edition)
MATCH (e)<-[:BELONGS_TO]-(p:Paper)
MATCH (p:Paper)-[cb:CITED_BY]->(:Paper)
WITH c, p, count(cb) as numCited
ORDER BY c, numCited DESC
WITH c, COLLECT({paperId: p.id, numCited: numCited}) AS papers
RETURN c.id, c.name, papers[0..3] AS topPapers
```

2. For each conference find its community: i.e., those authors that have published papers on that conference in, at least, 4 different editions.

```
Unset
MATCH
(author:Author)<-[:WRITTEN_BY]-(paper:Paper)-[:BELONGS_TO]->(edition:Edition
)-[:IS_FROM]->(conf:Conference)
WITH     conf.name     AS     conference,     author.name     AS     authName,
COUNT(DISTINCT(paper)) as numPapers
WHERE numPapers>3
WITH conference, COLLECT(authName) AS community, numPapers
RETURN conference, community, numPapers
```

NOTE: As the data was randomly generated, when generating random numbers across different machines, the output sequence of random numbers can differ between machines. Thus, due to the high

improbability of randomly generating an author publishing exactly four papers in different editions, it's possible that no results may be generated if a search query threshold requires the presence of exactly four publications. To improve the search performance, it's possible to lower the query threshold to see the performance of the query.

3. Find the impact factors of the journals in your graph

```
Unset
MATCH (j:Journal)<-[e1:VOLUME_FROM]-(v:Volume)<-[e2:PUBLISHED_IN]-(p:Paper)
with j, toInteger(v.year) as year, count(e2) as numCit
OPTIONAL MATCH
(j)<-[:VOLUME_FROM]-(v2:Volume{year:year-1})<-[e3:PUBLISHED_IN]-(:Paper)
WITH j, year, numCit, count(e3) as numPub1
OPTIONAL MATCH
(j)<-[:VOLUME_FROM]-(v3:Volume{year:year-2})<-[e4:PUBLISHED_IN]-(:Paper)
WITH j, year, numCit, numPub1, count(e4) as numPub2
return j.id, year, numCit, numPub1, numPub2, CASE numPub1+numPub2 WHEN 0
THEN 0.0 ELSE toFloat(numCit)/(numPub1+numPub2) END AS IF order by j.id,
year
```

4. Find the h-indexes of the authors in your graph

```
Unset
MATCH (a:Author)<-[WRITTEN_BY]-(p:Paper)-[c:CITED_BY]-> (:Paper)
WITH a.name as authorName, p.title AS title, COUNT(c) AS numCites
ORDER BY numCites DESC
WITH authorName, COLLECT(numCites) AS numCitesList
WITH authorName, [x IN range(1,size(numCitesList))
WHERE x<=numCitesList[x-1]| [numCitesList[x-1],x] ] AS hIndexList
RETURN authorName,hIndexList[-1][1] AS h_index
```

C. Recommender

To develop the recommender C_Recommender.py was created, where step by step the cypher commands of each stage of the recommender were executed:

- **Stage 1**: Defining the database community, by creating the node community and an instance with name database and linking it with the respective defined keywords to the database community by creating the edge DEFINED_BY:

```
Unset
CREATE CONSTRAINT communityNameConstraint FOR (community:Community) REQUIRE
community.name IS UNIQUE;
```

```
CREATE (community:Community{name: 'database'}) WITH community
MATCH (keyword: Keyword)
WHERE keyword.keyword IN ['data management', 'indexing', 'data modeling',
'big data', 'data processing', 'data storage', 'data querying']
CREATE (community)-[:DEFINED_BY]->(keyword);
```

- **Stage 2**: Selecting the associated conferences and journals to the database community by checking if at least 90% of the papers that belong to them contained the keyword linked to the community, and linking them to the community node of database by creating the edge IN_COMMUNITY

```
Unset
MATCH
(p:Paper)-[:BELONGS_TO|PUBLISHED_IN]->(e)-[:VOLUME_FROM|IS_FROM]->(ven)
WITH ven.id AS venue, COUNT(p) AS numPapers, ven
MATCH
(com:Community)-[:DEFINED_BY]->(k:Keyword)<-[:RELATED_TO]-(p:Paper)-[:PUBLIS
HED_IN|BELONGS_TO]->(e)-[:VOLUME_FROM|IS_FROM]->(ven2)
WHERE ven2.id=venue
WITH ven2, ven2.id AS venue2, COUNT(distinct(p)) AS numPapersWithKeywords,
numPapers, com
WITH ven2, venue2, numPapersWithKeywords, numPapers, com,
(toFloat(numPapersWithKeywords)/numPapers) AS percentage
WHERE percentage>=0.9
MERGE (ven2)-[:IN_COMMUNITY]->(com);
```

- **Stage 3**: Selecting the top 100 papers according to its highest page rank of the database community, this by matching the papers of the conferences and journals that belong to the database community, linked by the edge IN_COMMUNITY and then adding an attribute called is_database_com_top to this papers and setting it in true

```
Unset
MATCH (n)-[:IN_COMMUNITY]->({name: "database"})
MATCH (n)<-[:VOLUME_FROM|IS_FROM]-(n2)
MATCH (n2)<-[:PUBLISHED_IN|BELONGS_TO]-(p:Paper)
MATCH (p:Paper)-[cb:CITED_BY]->(:Paper)
WITH p, count(cb) as numCited
ORDER BY numCited DESC LIMIT 100
SET p.is_database_com_top = true;
```

- **Stage 4**: Selecting the potential reviewers for the database community by selecting the authors of those 100 top papers were selected and an attribute called potential_database_com_rev was added to the authors and set in true. Then to define the gurus it was checked if this authors had 2

or more papers in this community and if so the attribute database_com_guru was added and set in true

```
Unset
MATCH (p:Paper{is_database_com_top:true})-[:WRITTEN_BY]->(a:Author)
SET a.potential_database_com_rev = true;

MATCH (p:Paper{is_database_com_top:true})-[:WRITTEN_BY]->(a:Author)
WITH a, count(*) as num_papers
WHERE num_papers >1
SET a.database_com_guru = true;
```

D. Graph algorithms

1. Node Similarity:

This algorithm would be applied to the nodes of type Author, focusing on the relation WRITTEN_BY with the Paper nodes, and would compare the similarity between these nodes. The result would be ordered by the similarity in descendent order. The idea of this algorithm application is to get the Authors that tend to write papers together. This would be useful, for example, to search authors to collaborate with, so I have an author in which I trust, so by using the similarity calculated by this algorithm, I can choose other authors that have high similarity with the author I trust, because if they have a high similarity, it means that they have a lot of papers together.

Execution:

Create the graph projection where the algorithm would be applied:

```
Unset
CALL gds.graph.project(
    'similarity_graph',
    ['Author', 'Paper'],
    { WRITTEN_BY: {orientation: "REVERSE" }});
```

Algorithm execution over the graph projection:

```
Unset
CALL gds.nodeSimilarity.stream('similarity_graph')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).id AS Author1, gds.util.asNode(node2).id AS
Author2, similarity
ORDER BY similarity DESCENDING, Author1, Author2
```

Results:
In the output it can be seen a list of two authors id and the calculated similarity between them, ordered in a descendant way by this similarity value, this would be helpful as it was said before to have a list of authors that work together a lot. But this is not 100% truthful because this similarity value would depend on the number of papers of the authors, so for instance it would not be the same a similarity of 0.5 between two authors that have 2 papers each and between two authors with 20 papers, so it is a relative value.

2. Louvain algorithm:

In this case, we utilized the Louvain algorithm. The algorithm takes into account shared edges between nodes and looks for the highest modularity gain (i.e., a measure that assesses the best partitions of nodes between communities). This algorithm was useful for identifying communities of strongly connected papers, so when we projected only part of our graph (i.e., Paper nodes and CITED_BY edges), we were able to find different communities of papers based on the citations between them. In our case, the use of this algorithm could be applied in various contexts. For example, it could be interesting to find communities of papers that related to each other to identify joint research groups (i.e., papers that worked in similar areas). It could be also useful to find papers from a community that an author may have overlooked or to help a student find papers related to their Master's thesis.

Execution:

We first created the graph projection where the algorithm would work on. Notice that we only projected Papers and CITED_BY edges.

```
Unset
CALL gds.graph.project('LouvainGraph', 'Paper', 'CITED_BY')
```

Algorithm execution over the graph projection:

```
Unset
CALL gds.louvain.stream('LouvainGraph')
YIELD nodeId, communityId, intermediateCommunityIds
RETURN COLLECT(gds.util.asNode(nodeId).title) AS title, communityId
ORDER BY communityId ASC
```

Results:
The results of running the Louvain algorithm showed the existence of 8 distinct communities, ranging from 2 to 27 papers each. We observed that the communities grouped different numbers of papers that were strongly related to each other due to the large number of citations. These findings suggest that there are specific areas of research within our dataset that are highly interconnected and may indicate potential research collaborations or interdisciplinary topics. Further analysis of the communities may provide insights into the relationships between the papers and help to identify research gaps and opportunities for future work.