Open Data: Property Graph

Anass Benali
FIB - UPC Student
Barcelona, Spain
anass.benali@est.fib.upc.edu

Oriol Borrell
FIB - UPC Student
Barcelona, Spain
oriol.borrell.roig@est.fib.upc.edu

March 16, 2020

A Modeling, Loading, Evolving

A.1 Modeling

In Figure 1 is shown the model we created:

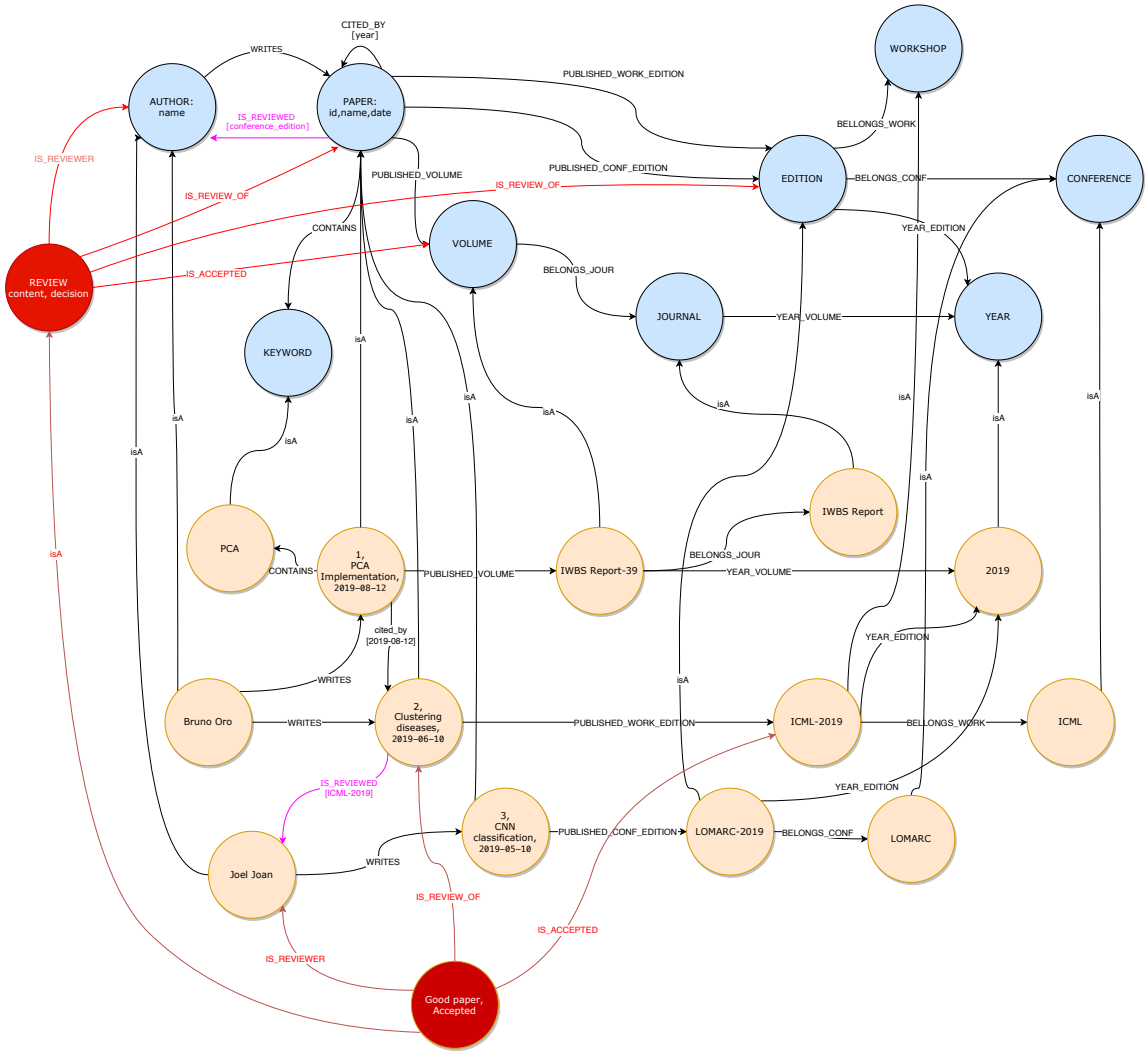

Figure 1: DB Model

We have to take into account that the nodes and edges with a red color are for Section A.3. We represented in a blue color the metadata, and with a orange color a possible representation of the data. The black edges are common for sections A1 and A3. The magenta nodes are specific for A1 and the red specific for A3.

Only the necessary information for the queries of section B are added to the graph, the rest of the information is discarded.

A.2 Instantiating/Loading

We used the real data found on DBLP to generate our graph. We obtained the provided huge XML file by DBLP and converted it to CSV using the python script provided by Thom Hurks. We only kept the information we needed to instantiate the graph and fulfill the queries of section B. We also preprocessed the data in order to have it more clean. We trusted the journal and proceeding "url" respectively to classify the papers. We assumed that the workshops are all the proceedings that contain 'workshop' in the title and that the others are conferences. Moreover, for the journals we use the readily available "journal" and "volume" names, however, for the proceedings this information was not available so we inferred it from the "url".

However, we did not find the paper citations at DBLP so we generated them at random using a Chi-squared distribution. Also the keywords of a paper are missing from DBLP, so we derived them from the title of the paper. We cleaned the word from any non alphabetical character. We took as a keyword each cleaned word that has more than 3 characters.

We used indexes to improve the efficiency of the queries and be able to use a large amount of data. We used 300.000 papers (100.000 of each type). At the end of all sections, this amount of papers generated us a total number of 408.595 nodes and 6.650.293 edges. Even with this big amount of data the queries of the following sections execute in a performant way and therefore in reasonable time.

A.3 Evolving the graph

When evolving the graph, we mainly took two options into account. The first one was to represent the reviews with an edge from *Paper* to *Author*, and store as a property the $conference$ it belongs, the *content*, and the *decision*. The second one was to create a node *Review* with the *content*, and the *decision* as properties, and create edges to the authors (the reviewers), the paper witch is reviewed, and the conference it belongs. We decided this last option, represented in red in Figure 1.

The reason we took this last option is because it lets you obtain the reviews directly, and with distance 1 you can navigate to *Paper*, *Author* (if there is more than 1 you have to do a "table scan", but we don't expect to have a lot of authors), or $Conference$. With the other option the conference of each review was more difficult to reach.

When generating the graph, we only created the nodes that will be needed to fulfill the queries of Section B. As the reviews are not present in future sections, we will not generate them in Neo4j. We created the queries for task 2, but this queries are using a *reviews.csv* file that is not generated. This is because we didn't have that information in the database, and because is not needed for queries of

sections B, C or D. We will also not generate the metadata nodes and edges. We understand that this information is implicitly stored with the labels and already represented in the diagram of A.1.

B Querying

When creating the queries the general idea that we applied is first thing the pattern matching that resolves our query, and after that apply some selection, aggregation function or whatever is needed. Whenever the type of an edge or node is not necessary for the result of the query, we not included in the pattern matching in order to reduce its complexity.

B.1 Finding h-indexes

```
MATCH (author:Author)-[:WRITES]->(paper1:Paper)-[paper2:CITED_BY]->()
WITH author, paper1, COUNT(paper2) AS number
ORDER BY author.author_name, number DESC
WITH author, collect(number) AS list
WITH author, list, range(1, size(list), 1) AS indexes
WITH author, reduce(hindex=0, index IN indexes |
    CASE WHEN list[index-1] >= index THEN hindex+1 ELSE hindex END
) as hindex
RETURN author, hindex
```

B.2 Most 3 cited paper of each conference

```
MATCH (conference:Conference)<-[:BELONGS_CONF]-(edition:Edition)
<-[:PUBLISHED_CONF_EDITION]-(paper1:Paper)<-[cited:CITED_BY]-()
WITH conference, paper1, COUNT(cited) AS number
ORDER BY conference.conference_name, number DESC, paper1.paper_id
WITH conference, collect(paper1) AS paper_list
UNWIND paper_list[0..3] AS top3
RETURN conference, collect(top3) AS paper
```

B.3 Finding conference community

```
MATCH (author:Author)-[:WRITES]->(:Paper)-[:PUBLISHED_CONF_EDITION]->
(edition:Edition)-[:BELONGS_CONF]->(conference:Conference)
WITH author, conference, collect(DISTINCT edition) AS editions,
size(collect(DISTINCT edition)) AS numberEditions
WHERE numberEditions>=4
RETURN conference, collect(author)
```

B.4 Finding the impact factor of journals

```
MATCH (vol:Volume)-[:YEAR_VOLUME]->(y:Year)
WHERE y.year IN ["2018", "2019"]
WITH vol
MATCH (journal:Journal)<-[:BELONGS_JOUR]-(vol:Volume)
<-[pub:PUBLISHED_VOLUME]-(p:Paper)<-[cited:CITED_BY]-()
RETURN journal, COUNT(cited)/COUNT(DISTINCT pub) AS impactFactor
```

C Graph algorithms

C.1 Page Rank

```
CALL algo.pageRank.stream("Paper", "CITED_BY",
{iterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.asNode(nodeId).paper_id AS page,score
ORDER BY score DESC
```

For each paper we obtain it's PageRank score. The papers with high score are the ones that are more relevant. The number and quality of the citations are estimated to determine the importance of a paper. The relevancy is understood such that a paper is more relevant if is cited by many other relevant papers. That relevancy of a paper is distributed to the paper's citations. With this, we obtain rank the papers of all the graph. From that we could get the top papers of the graph. This algorithm makes sense because the papers cite other papers which is symmetric to the original purpose it was developed for, where websites link to other websites.

The algorithm used uses a damping factor and a random walking. The damping factor is weighting of the relevance of the citations. Then, 1 minus the damping factor is the base relevance of a node. Therefore, the probability of following citations is the damping factor and to jump to a random node is 1 minus the damping factor.

C.2 Weakly Connected Components

```
CALL algo.unionFind("Paper", "CITED_BY", {
  write: true,
  writeProperty: "componentId"
})
YIELD nodes AS Nodes, setCount AS NbrOfComponents, writeProperty AS PropertyName

CALL algo.unionFind.stream("Paper", "CTIED_BY")
YIELD nodeId, setId
RETURN algo.asNode(nodeId) AS Name, setId AS ComponentId
ORDER BY ComponentId, Name
```

Connceted Components is a community algorithm which tell us the number of connected components for the given input nodes and edges. For the papers and the citations, we identify the number of different disjoint groups of papers that are not related to each other by any citation. If we call the stream function we can get each paper and it's component number. This helps us analyze the connectivity of the graph and identify groups/clusters of related papers.

D Recommender

In order to store the result from each query, we created a node of label *Community*. We will create edges that join this *community* node to the results of each query. In the Section D.3 we also store the score obtained with the page rank in a property of the edge. We defined different types of edges, one for each query. Each section queries over this node in order to obtain the result of the previous steps.

D.1 Defining research community

```
CREATE (community:Community {name: "database"})
WITH community
MATCH (keyword:Keyword)
WHERE keyword.keyword IN ["data", "management", "indexing",
"modeling", "processing", "storage", "querying"]
MERGE (community)<-[:IS_COMMUNITY]-(keyword)
```

D.2 Conferences/Journals of the community

```
MATCH (community:Community)<-[:IS_COMMUNITY]-(keyword:Keyword)
<-[:CONTAINS]-(paper1:Paper)-[]->()-[]->(n2)
WHERE n2:Journal OR n2:Conference
WITH community, n2, COUNT(paper1) AS communityPapers
MATCH (paper2:Paper)-[edge]->()-[]->(n2)
WITH community, n2, communityPapers, COUNT(paper2) AS totalPapers,
(1.0*communityPapers/COUNT(paper2)) AS percentage
WHERE percentage >=0.5
MERGE (community)<-[:RELATED]-(n2)
RETURN n2, totalPapers, communityPapers, percentage
```

We used a percentage threshold of 0.5 because otherwise we obtain too few database communities. We adopt this change because we are using the real data and it does not change the structure of the query.

D.3 Top papers of the community

```
CALL algo.pageRank.stream(
  'MATCH (community:Community {name: "database"})
```

```
  <-[:RELATED]-()<-[]-()<-[]-(p:Paper)
  RETURN id(p) AS id',
  'MATCH (community:Community {name: "database"})
  <-[:RELATED]-()<-[]-()<-[]-(p:Paper)
  <-[:CITED_BY]-(p2:Paper)-[]->()-[]->()-[:RELATED]->(community)
  RETURN id(p2) AS source, id(p) AS target',
  {graph:'cypher'}
) YIELD nodeId, score
WITH algo.asNode(nodeId) as node, score order by score desc LIMIT 100
MATCH (community:Community {name: "database"})
MERGE (community)<-[:IS_TOP_PAPER {score:score}]-(node)
RETURN node.paper_title, score
```

We do a graph projection to compute the PageRank only taking into account the papers and citations of the database community.

D.4 Gurus of the community

```
MATCH (community:Community {name:"database"})<-[IS_TOP_PAPER]-(paper1:Paper)
<-[:WRITES]-(author1:Author)
WITH community, paper1, author1
MATCH (community)<-[IS_TOP_PAPER]-(paper2:Paper)<-[:WRITES]-(author2:Author)
WHERE paper1.paper_id <> paper2.paper_id and author1 = author2
RETURN DISTINCT author1
```

E Final Thoughts

We are aware that this section is not asked, but we wanted to express our opinion/problems we found during the performance of the lab. This lab was pretended to be done in 26 hours of work. As we found a lot of problems, we needed approximately 60 hours to have what we think is a decent work, but not as good as we wanted.

The first problem was that the dataset that were given had some errors (for example some open quotes that were not closed) where we lost some time. After that, we decided to use the real dataset, the one from *dblp*. But in this one we found that was some information that was not available: This dataset did not contain neither citations (in the dblp website was explained that in a future they will have that information, but not now) nor keywords. Finding "easy" solutions for this luck of information made us loose some more time, and obtain a data that was not completely real.

Moreover, the main purpose of the lab should be focused on practising and learning Cypher and Neo4j as stated on the preamble of the lab "We will practice how to create and manage graphs, as well as querying/processing them". Instead the way it is, almost all the time in devoted to preprocessing which it is not part of the content of this course.

We think that if a cleaner dataset is given, that contains all the information needed to fulfill the queries, we could spend more time working with cypher, neo4j and the other content of this course.