SDM Assignment 1: Property Graphs

Mateo Jácome González and Kathryn Weissman

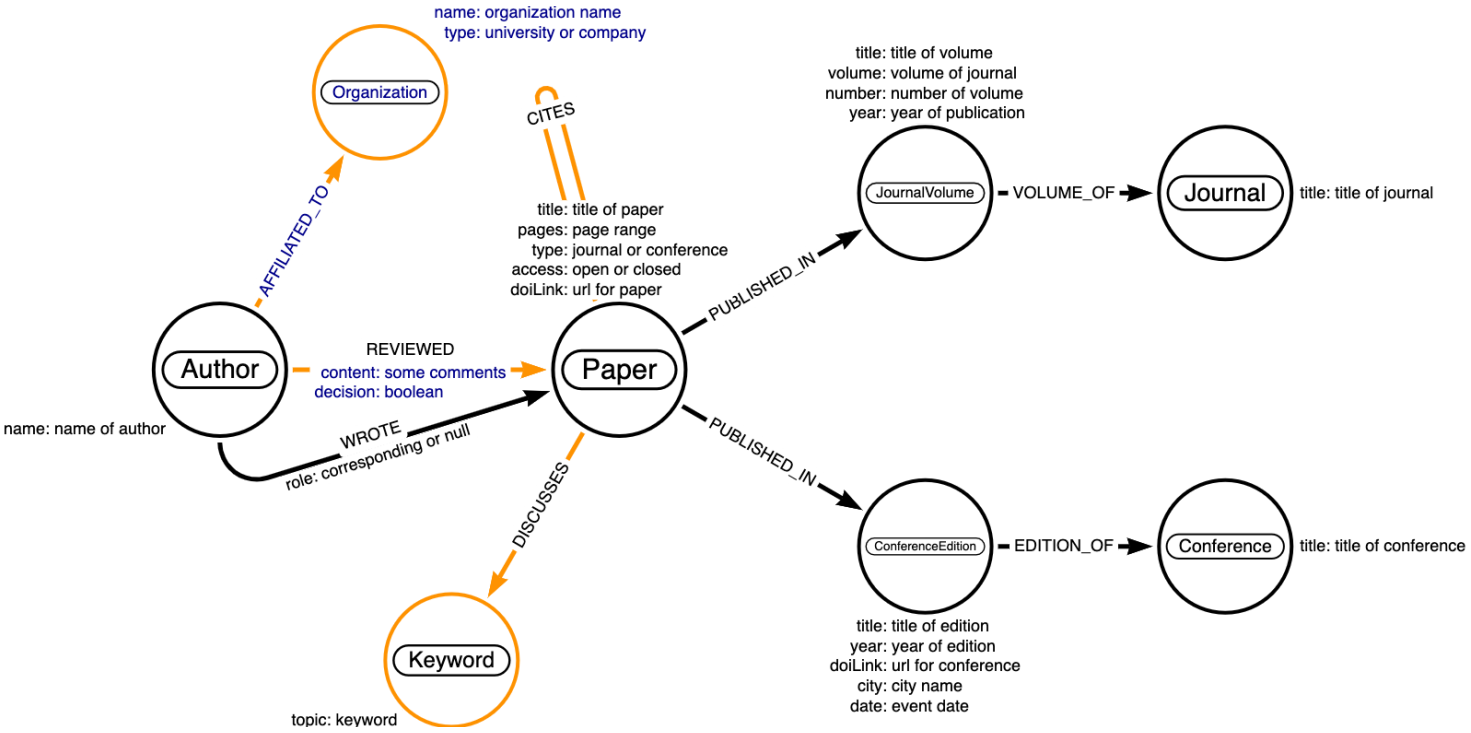
Github Repository, containing all scripts and more: [Github repo]

A. Modeling, Loading, Evolving

A.1. Modeling

Our color-coded schema design is shown in Figure 1. Black nodes and edges represent the entities that we created and loaded from source data. Orange nodes and edges represent synthetically generated data, as explained in section A.2. Nodes, relationships, and properties in blue text were added during graph evolution in section A.3. In general, we use nodes to represent nouns and relationships to represent verbs. 'JournalVolume' and 'ConferenceEdition' nodes represent publications, and 'Journal' and 'Conference' nodes represent their parent publishers or organizations. We chose to represent keywords and organizations as nodes instead of properties for query performance and scalability. 'Paper' and 'Author' have the highest quantity of nodes, so it allows us to filter papers written by authors affiliated to an organization without looking at the properties of every author. Similarly, it allows us to find all papers related to a certain keyword without searching their properties. We represent city as a property of a 'ConferenceEdition' since we don't anticipate using cities for queries.

Figure 1. Graph Database Schema



A.2. Instantiating/Loading

In order to create and populate our database, we first created a database instance manually through the Neo4J Desktop application. We then installed the APOC plugin, which is necessary for the way we implement some of our queries. In order to populate the database, we first processed and imported data coming from the DBLP library. After loading the database with instances of the required entities available in our source data, we created synthetic data for the entities that were still missing, such as citations, keywords, and reviewers.

A.2.1 Preprocessing

Given the massive size of the potential source data, we decided to work with a smaller subset. For this, we obtained XML files from DBLP for individual journals and conferences related to data science, each with multiple volumes and editions. The selected publications can be consulted in table 1. (https: //dblp.uni-trier.de/).

Table 1. Journals and conferences for which DBLP data was downloaded and processed.

Title	Type	DBLP journal link	DBLP XML link
Big Data Mining and Analytics	Journal	link	link
IEEE Transactions on Big Data	Journal	link	link
Data Intelligence	Journal	link	link
IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT)	Conference	link	link
IEEE International Conference on Big Data and Smart Computing (BigComp)	Conference	link	link

Each XML file is the result of a query and limited to the first 1,000 results (papers). Each result tagged as a 'hit' represents a scientific paper of the given journal or conference. In the fragment below, a sample XML structure for one paper in the journal "Big Data Mining and Analytics" can be consulted. The data that we choose to import to the graph database is shown in blue font:

```
<hit score="1" id="4460">
  <info>
    <authors>
      <author pid="53/3308">Hui Bai</author>
      <author pid="37/1091">Yan Yang</author>
      <author pid="29/5259">Jie Wang</author>
    </authors>
    <title>Exploiting more associations between slots for multi-domain dialog state tracking.</title>
    <venue>Big Data Min. Anal.</venue>
    <volume>5</volume>
    <number>1</number>
    <pages>41-52</pages>
```

```
      <year>2022</year>
      <type>Journal Articles</type>
      <access>closed</access>
      <key>journals/bigdatama/BaiYW22</key>
      <ee>https://ieeexplore.ieee.org/document/9663259</ee>
      <url>https://dblp.org/rec/journals/bigdatama/BaiYW22</url>
    </info>
    <url>URL#4460</url>
  </hit>
```

A.2.2 Entity loading from source data

We used the py2neo library to connect Python to our database and run our Cypher queries and other statements. In order to extract and load data of interest from the XML files, we designed three different Cypher scripts: one to process data coming from journals, a second one for processing conference data, and a third one to set a relationship property for corresponding author for each paper (assuming that the corresponding author is the first one in the list of authors). Because the XML files in the DBLP repository are updated periodically, we saved the files we have worked with to a GitHub repository to ensure reproducibility.

In these Cypher scripts, we combine the elements <venue>, <volume>, and <number> as the unique title for the instance of a journal volume. We use the dblp <hit> id to guarantee uniqueness for papers, and import other data which we consider interesting, as indicated with blue font in the sample above.

A.2.3 Synthetic entity generator

Because keywords, citations, and reviewers are not provided in the XML files, we've created three different generators to associate them to the entities in our database using some rules. These generators are built on Python, and are run after the Cypher scripts have finished populating the database.

For keywords, we've created three sets of words associated with three fields in computer science: Databases, Artificial Intelligence, and Supercomputing. Our generator randomly assigns between one and three keywords to each paper, with a very high probability of assigning keywords from one specific keyword set to articles of a specific journal. This ensures that while most of the keywords assigned to a journal or conference belong to the same community, we have some 'noisy keywords' as would happen in reality. This way, we can mimic how conferences and journals that are devoted to a main field still accept papers that discuss some topics outside of it. For this, we created the Databases keyword set using the words suggested in the project statement. This was necessary so that later on we can assign journals and conferences to scientific communities as required in section D yields some results.

For citations, we designed our generator based on two main rules: a paper can't cite itself, and a paper can't cite a paper published in the future. Because the source data has only the year of publication, we sort the papers only by year and abstract any other possible temporal information. Therefore, a paper published in 2021 can cite any other paper published in 2021 or

previous years. Following these rules, we randomly assign between zero and fourteen outgoing citations from each paper (most of the oldest papers in our database have zero or few cites to other papers, yet receive a higher number of citations from newer papers, creating diversity).

Finally, we assigned three existing authors as reviewers for each paper. To make the database more life-like, we first select all the potential reviewers by querying for the set of authors that wrote papers about any of the topics that a paper discusses. Then, we randomly assign three of these authors to be the reviewers.

A.3. Evolving the graph

To evolve the graph, we first updated the recently created ':REVIEWED' relationships to include a 'content' property and a 'decision' property. The 'content' property simply stores a string containing the text 'some comments', where the reviewer remarks would be stored. The decision property stores a boolean value regarding whether the reviewer recommended acceptance of the paper. We generated those properties ensuring that all articles have at most one negative review.

In order to create the entities necessary for representing author's affiliations to companies and universities, we first web-scraped the list of the world's top 50 universities according to the QS ranking, and the list of the world's largest 50 companies by revenue. We created nodes with the label 'Organization' and two properties: 'name', and 'type', the latter referring to whether the organization is a company or a university. We then created an ':AFFILIATED_TO' edge between each author and a randomly assigned university or company, making sure that universities are oversampled with respect to companies, since they're commonly the organizations to which most scientific authors belong to. The updated schema is shown in blue in Figure 1 on page 1.

B. Querying

B.1. Top 3 Papers per Conference by Citations

```
MATCH(c:Conference)
CALL {
        WITH c
        MATCH(c)<-[:EDITION_OF]-(ce:conferenceEdition)<-[:PUBLISHED_IN]-(p:Paper
        )<-[cites:CITES]-(:Paper)
        RETURN p, count(cites) as numCitations
        ORDER BY numCitations DESC
        LIMIT 3 }
RETURN c.title as Conference, collect(p.title) as Papers, collect(numCitations)
as Citations
```

B.2. Community Members of Conference

```
MATCH(conference:Conference)<-[:EDITION_OF]-(ce:conferenceEdition)<-[:PUBLISHED
_IN]-(:Paper)<-[:WROTE]-(author:Author)
WITH count(distinct(ce.title)) as EditionsParticipated, author.name as Author,
conference.title as Conference
WHERE EditionsParticipated>=4
RETURN Conference, collect (Author) as Community
```

B.3. Impact Factors of Journals

```
MATCH
(j:Journal)<-[vol:VOLUME_OF]-(jv:JournalVolume)<-[pub:PUBLISHED_IN]-(p:Paper)
WHERE jv.year IN ['2019','2020']
OPTIONAL MATCH
(p)<-[cites:CITES]-(p2:Paper)-[pub2:PUBLISHED_IN]->(jv2:JournalVolume)
WHERE jv2.year = '2021'
RETURN j as Journal, (toFloat(COUNT(cites) / toFloat(COUNT(pub)))) as
ImpactFactor_2021
```

B.4. H-Indexes of Authors

```
MATCH(a:Author)-[wrote:WROTE]->(p:Paper)
OPTIONAL MATCH (p)<-[cites:CITES]-(p2:Paper)
WITH count(cites) as numCites,a,p
ORDER BY numCites DESC
WITH a.name as Author, (collect(numCites)) as CitationList
WITH [i IN RANGE(0, SIZE(CitationList)-1) WHERE CitationList[i] >= i] as
H_index, Author
UNWIND H_index as h
RETURN Author, max(h) as H_index
ORDER BY H_index DESC
```

C. Graph algorithms

After studying the different proposed algorithms, we decided to implement the node similarity algorithm and the Louvain community detection algorithm. We did this because the similarity analysis enables the use of the community detection algorithm, nicely linking the use of the two algorithms. In the fragment below, the most important code parts of our implementation are shown:

1. Create in-memory sub graph for Author Similarity by Paper

```
CALL gds.graph.create("Author_Similarity_Graph_Papers",
["Author","Paper"],"WROTE");
```

2. Mutate the in-memory graph adding a similarity score between pairs of authors.

```
CALL gds.nodeSimilarity.mutate("Author_Similarity_Graph_Papers",
{mutateRelationshipType:'Similar',mutateProperty:'score',
topK:10, similarityCutoff:0.1});
```

3. Mutate the graph by assigning authors to a community with the Louvain algorithm.

```
CALL gds.louvain.mutate("Author_Similarity_Graph_Papers",
{nodeLabels:['Author'], relationshipTypes:['Similar'],
relationshipWeightProperty:'score',
mutateProperty:'louvainCommunity'});
```

4. Export the graph as a new database for exploration and delete the in-memory graph.

```
CALL gds.graph.export('Author_Similarity_Graph_Papers', { dbName:
'AuthorCommunity', additionalNodeProperties: ['name','title']});
CALL gds.graph.drop("Author_Similarity_Graph_Papers");
```

C.1. Insights into our Node Similarity implementation

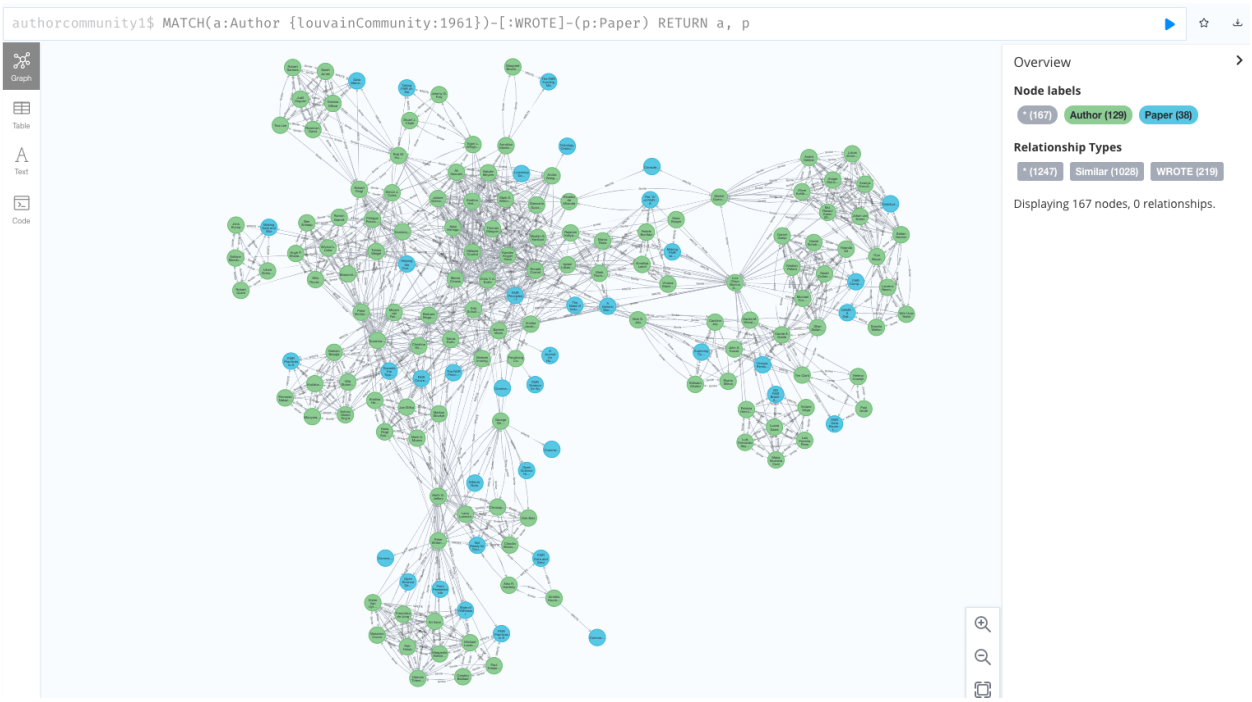
The neo4j Node Similarity algorithm uses the Jaccard metric to compute pairwise similarities. The input is a bipartite connected graph, and the algorithm ignores disconnected nodes. We use the node sets of `:Author` and `:Paper`, using the relationship `:WROTE` to examine the similarity between authors. This allowed us to view analysis of real data instead of synthetically generated data. In this domain, structurally equivalent nodes represent authors who are co-authors on all of the papers they publish, and have a similarity score of 1 between them. The lowest score of 0 represents authors who have never co-authored a paper, however the similarity cutoff threshold for the algorithm is slightly higher than 0, so those results do not appear. Scores closer to 1 indicate a higher number of collaborations between two authors and scores closer to 0 indicate a lower amount, considering the ratio of number of collaborations to the total number of papers both authors have published. After running the algorithm, we used the results to update the graph with similarity relationships between co-authors and added the Jaccard score as a property of the relationship.

C.2. Insights into our Louvain implementation

The similarity score was then used as the relationship weight property between authors on the follow-up Louvain community detection algorithm, which uses hierarchical clustering to group nodes in communities. By using Louvain to assign authors to communities after computing Jaccard similarity, we can see where the strongest links are among authors and their co-authors at a more global level. After running both algorithms, the largest Louvain community has 129 authors who wrote 38 papers, and many of the papers are about FAIR principles and practices. Without prior knowledge about the topic of the papers, computing similarity followed by

community detection for authors was useful to find papers on a similar topic due to the collaboration of authors who have common research interests. In the figure below, we show a diagram of the largest Louvain Community, with author nodes in green and paper nodes in blue. Results of queries regarding the authors and papers in the largest community are saved as exports in .csv format on github.

Figure 2. Graph of largest Louvain Community of authors (green) and their papers (blue)



D. Recommender

We decide to implement our recommender system in a manner that, given a community and a set of keywords that the community discusses and studies, we could get as output a set of potential reviewers and gurus, regardless of whether the community we want to query for is the Database community, as proposed in the exercice, or one of the other two that we defined sets of keywords for earlier: Supercomputing or Artificial Intelligence.

Our implementation is based on Python code, and thus the queries shown below contain some pythonic string formatting that is modified when the code is run in python. These modifications incorporate some variables such as the community we're querying for, or its set of keywords. We suggest reading these queries in our Part D python script to better understand the context and the string formatting that we perform to modify these Cypher scripts. These modifications would allow a conference chair or journal editor to easily find authors and gurus for any community in our database (and not only for the Database community) by simply calling a function with the desired community as a parameter.

D.1 Define Research Community

```
MATCH (c:Community), (k:Keyword)
WHERE c.name = "{comm}" AND k.topic = "{kw}"
MERGE (c)-[:STUDIES]->(k)
```

D.2 Identify Relevant Conferences & Journals

```
MATCH (co:Community)-[:STUDIES]-(keyword)
WITH apoc.map.fromLists([co.name], [collect(keyword.topic)]) AS
community_kws
MATCH (kw:Keyword)-[]-(p:Paper)-[]-(edition)-[]-(jc)
WHERE (jc:Conference or jc:Journal) AND (edition:JournalVolume OR
edition:conferenceEdition)
UNWIND keys(community_kws) AS comm
WITH comm, jc, ID(p) AS paper, kw.topic AS keyword,
CASE
    WHERE kw.topic in community_kws[comm] THEN true ELSE null
    END as related
WITH jc, comm, paper,
CASE
    WHEN any(x in collect(related) WHERE x IS NOT null) THEN 1 ELSE 0
    END AS rel_papers
WITH jc, comm, avg(rel_papers) AS rel_score
MATCH (jc), (community:Community {name: comm})
WHERE rel_score >= 0.9
MERGE (jc)-[rel:IS_PART_OF]->(community)
SET rel.rel_score = rel_score
```

D.3 Identify Top Papers

1. Instantiate graph for running the PageRank algorithm for each community's papers' subgraph.

```
CALL gds.graph.create.cypher(
     '{comm}_Papers',
     'MATCH
          (comm:Community)<-[:IS_PART_OF]-()<-[:EDITION_OF|VOLUME_OF
          ]-()<-[:PUBLISHED_IN]-(p:Paper)
     WHERE comm.name = "{comm}" RETURN id(p) AS id, labels(p) AS
     labels',
     'MATCH (p1)-[r:CITES]->(p2) RETURN id(p1) AS source, id(p2) AS
     target, type(r) AS type',
     {{validateRelationships: false}})
```

```
      YIELD graphName AS graph, nodeQuery, nodeCount AS nodes,
      relationshipCount AS rels
```

2. Run PageRank over the graphs previously created

```
CALL gds.pageRank.stream('{comm}_Papers')
      YIELD nodeId, score
      WITH nodeId, score
      ORDER BY score DESC, nodeId ASC LIMIT 100
      MATCH (p:Paper) WHERE ID(p) = nodeId
      SET p.{comm}_top100 = true
      RETURN p.title, p.top100
```

D.4 Identify Reputable Authors

```
MATCH (a:Author)-[:WROTE]->(p:Paper {{{comm}_top100: true}})
RETURN a.name AS author,
CASE WHEN count(*) >= 2 THEN true ELSE false END AS guru
ORDER BY guru DESC
```

In table 2 we show the partial results for the recommender. All authors in the list would be recommended reviewers for any journal or conference belonging to the Database community. Additionally, the ‘guru’ column indicates with a boolean whether they can be considered a guru (an author with at least two top-100 articles for the community).

Table 2. Recommender results (partial).

Author	Guru
Harksoo Kim	true
Chong-kwon Kim	true
…	…
Jonghwan Chung	false