Lab report: Property Graphs

Louis Van Langendonck
Pim Schoolkate

March 24, 2022

Instructions on how to use the application

This project is comprised of several `.py` scripts that can each be run from the command line prompt as `python <script-name.py> <inputs>`. Without modification, the folder structure allows the scripts to be ran without much input. For some scripts specific input might be needed, which will be explained by running `python <script-name.py> -h`. The database used in this project needs a specific configuration to be able to run smoothly with the scripts. Firstly, in the configuration file of the database, the line `dbms.directories.import=import` should be commented out, without doing so the scripts wont be able to load in the data. Secondly, the username of the database by default is assumed to be "neo4j", and the password should be "louis". However, different credentials can be used, which requires the user to specify these in the `.py` scripts by adding as options `-u <username>` and `-p <password>`. Lastly, the scripts assume by default that the connection to the database is `"neo4j://localhost:7687"`, however, if this is not the case of the user, it can be specified differently when calling the `.py` scripts by adding `-d <database_connection>` to the call of the script. The Cypher queries are stored as `.txt` files in the subdirectory */source/cypher_queries*, and are retrieved by the python scripts to execute them. This was done to easily modify the Cypher scripts and to avoid hard-coding the queries in python. For inspecting the queries, please refer to these text files, but be mindful not to change anything as a single addition of a `enter` might cause the scripts to not function.

A: Modeling, Loading, Evolving

A.1: Modeling

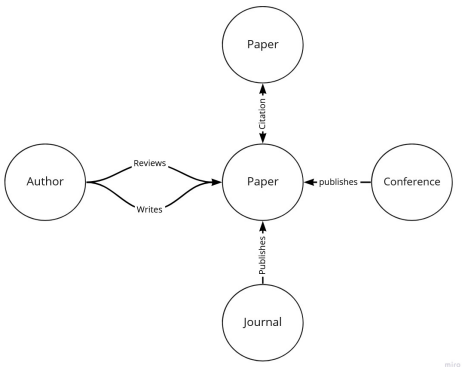

Figure 1: Visualization of graph schema

Based on the description in the project statement and the required queries, a graph was conceived which can be seen in Figure . The reasoning is as follows. Central to this graph database are the papers either published by journal or published by a proceedings in a conference. For simplicity, papers from both sources are contained in one node, having *id*, *DOI*, *ID*, *abstract*, *keywords*, *last_modification*, *pages*, and *title* as properties. The node key for a paper is the unique ID, which automatically creates an index on it, thus improving query performance and data base consistency. Papers can have a relation or edge with another paper if the paper cites the other paper. Citation are thus edge and do not have any properties. As citations in the loading of the database are defined by ID's, matching the right papers makes substantial use of the indexes defined on it. Next, authors play a big role in this graph database as an author can have multiple responsibilities. For one, when an author has contributed to a paper, they have the "Writes" relation with the paper, in which is indicated as property whether the writer is a co-writer or not. Moreover, an author can review a paper they have not contributed to, for which the "Review' relation is used. In section A3, this "Reviews" relation will be reconsidered. Authors have an *ID* and a *name* as their properties and have an index on their name. Furthermore, papers can either be published in a journal or a proceeding of a conference. Because proceedings always have a one to one relation with conferences it is decided to have only one node for conferences. Details on the proceeding are implemented as properties in the 'Publish' relation from a conference to a paper. This improves simplicity and allows for easy querying when looking for relations between nodes and conferences, which will be of major importance later in the project. Conferences have a node for each edition, thus, multiple conferences of the same name can exist in the database. The node key is given by name and date. Conferences have *name*, *series*, *date*, and *location* as their properties. Journals are also represented as nodes for each volume of the journal that is published. The properties of a journal are *name*, *volume*, and *year*. The node key is name, year and volume thus again creating an index accordingly, improving both future query performance and data base consistency

A.2: Instantiating/Loading

To stay as close as possible to working with real data, the data from DBLP is used as a baseline, complemented by generated components not present in the original files. This is obtained in three steps.

Obtain data

The `.xml` data is transformed to `.csv` using the https://github.com/ThomHurks/dblp-to-csv tool

Preprocess data

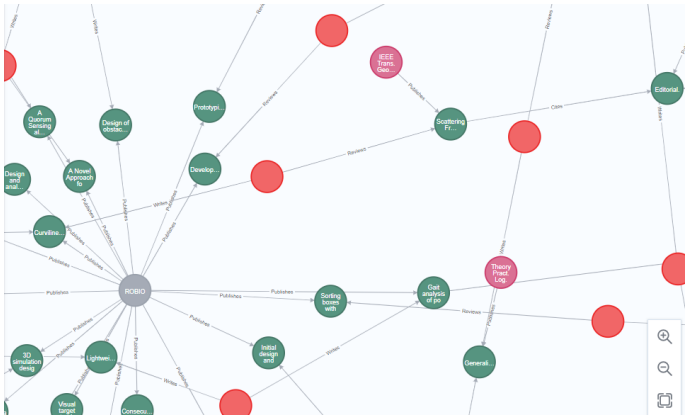

Figure 2: Sample result of loading the data in NEO4J.

These files are now processed through a script called 'preprocess_csv.py', which can be ran via the command line. The preprocessing can be roughly divided into two operations.

On the one hand, relevant files and columns are selected joined together. First the files are sampled to reduce the amount of data to process. To facilitate loading the data the graph database, all relevant data is eventually gathered into two files. 'article.csv' contains papers from journals and their corresponding attributed defined in . Secondly, 'proceeding_papers.csv' contains papers from proceedings/conferences and all relevant info. However the generated files do not provide all necessary data.

Data, relevant and not provided by the `.xml`-files is generated. For each paper, random keywords and an abstract are generated using the `lorem`-package in Python. However, for illustrating some concepts later on in the project, database specific keywords are added to most papers belonging to a handful of conferences and journals. Next, for every paper, from zero to thirty-five ID's of papers citing them, are randomly picked from all other ID's and put into a list, also making sure it can't cite itself. Completely analogously, one to four author names are randomly picked and chosen as reviewers. Lastly, some locations from a hand-made list are randomly assigned to conferences.

Load into NEO4J

The resulting files are now read into NEO4J using two cypher `Load CSV`-queries, accompanied by some constraints, indicating what defines each node. The details of each of these cypher queries can be found in 'article.txt' and 'proceeding.txt' respectively. To illustrate the result, an example of a subgraph is presented in Figure .

A.3: Evolving

In order to evolve the graph, an assumption is made that the data that has to be inserted is from two different sources. The first containing per row information about one author and their affiliation, and the second containing information about one author with the review they wrote, and their final decision. Because this data was generated, in this case, for each author, whether they reviewed a paper or not, an affiliation, a review, and a decision was generated for every author in the data set. This allows for using the same data set for the three queries that are needed to update the graph. The data was generated with the following heuristics. First, 10 affiliation where made up and uniformly distributed over the authors. Secondly, using Lorem Ipsum, a random text was generated for each author. Lastly, 80% of authors approved the paper and 20% disapproved the paper, whether they wrote a review or not. In order to satisfy the requirements, the graph needs to be updated first.

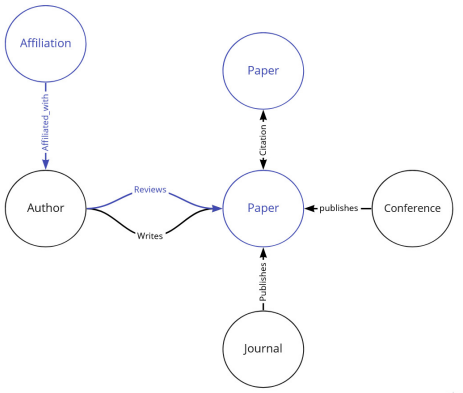

Figure 3: Visualization of graph schema after update

Figure shows the changes in blue with regards to the graph in figure . First and most notably, a new node "affiliation" has been added which connects to all authors that are affiliated with this specific node. As a property, it stores the name of the affiliation. The review edge was also updated and now stores as properties the review the author (or reviewer) wrote and the decision that was made. Lastly, each paper node now holds the property "decision" which can be either be "Accepted", "NotAccepted", and in case of no review "NotReviewed". It was assumed that there could be papers in the data set which would not be accepted. This was mostly done in order to show how such a query could be done, and prove the adequacy of the authors of this report.

In order to update the graph, and load in the data at the same time for all the nodes that are affected (we do not want to add empty nodes or properties for those that are not affected), three different queries were composed, which are all executed by the A3UpdateGraph.py file in the sources folder. It first updates the affiliation with the following query, which creates new node for the affiliation and connects it to the corresponding author for each line in the csv file. The query can be found in */source/cypher_queries/update_affiliation.txt*.

The second query updates the review edges to contain the decision and the review and is done as follows. The "Reviews" edge for each author to a paper is found with a simple match statement, after which a new edge named "Review" is made including the properties of the text of the review and the decision. The distinction between "Reviews" and "Review" is important as the former refers to an action, whereas the latter refers to an actual thing, and thus, in the last line of code, the former is deleted as it is no longer needed. The query can be found in */source/cypher_queries/update_review.txt*.

Lastly, in order to set the property whether a paper was accepted or not requires a bit more code. In order to decide on whether the paper is accepted, a major voting is used, meaning that we need the total amount of reviews for each paper, and the reviews for which the decision was "yes" (meaning the paper was approved). With these queries, and several `WITH` statements, a `CASE` statement is used to differentiate between a majority of approvals and a minority of approvals. When a tie occurs, the paper is accepted anyways. The result of this `CASE` statement is then `SET` as the new properties of that specific paper. The query can be found in */source/cypher_queries/update_decision.txt*.

B: Querying

Each query is provided with its corresponding cypher statement. The results can be obtained by running the queries, all available from the command line. A brief linguistic description of the results are provided however.

Top three most cited papers per conference

This query is a straightforward count of incoming cite edges per paper, per conference. Query:

```
WITH pa, co, count(*) as cites
ORDER BY co, cites DESC
WITH co, collect([pa,cites]) as papers
RETURN co.name as conference, co.date as date, coalesce(papers[0][0].title,"") as first_paper,
coalesce(papers[0][1],"") as times_cited_1, coalesce(papers[1][0].title,"") as second_paper,
coalesce(papers[1][1],"") as times_cited_2, coalesce(papers[2][0].title, "") as third_paper,
coalesce(papers[2][1], "") as times_cited_3
```

Result: The first conference "MFCS" of "2019-05-14" has as its most cited paper 72 citations, its second too and its last 70 citations.

Authors that published papers in 4 different editions

To optimize this query, first, all conferences in the data base that do not have four different editions are disregarded, which corresponds to pushing down selection in `SQL`. From this intermediate result, the requested authors are now found and collected into a list. Query:

```
MATCH (co:Conference)-[:Publishes]->(:Paper)
WITH co.name as conference, count(distinct co.date) as editions
WHERE editions > 3
CALL {
    MATCH (co)-[:Publishes]->(:Paper)<-[:Writes]-(au:Author)
    WITH co.name as conf, au.name as author, count(distinct co.date) as contrib
    WHERE contrib > 3
    RETURN conf, collect(author) as community
}
RETURN conf, community
```

Result: It only returns a result if at least 10000 rows are included in the load as there are very little conferences with four editions in the sample. The result is a conference called 'Medical Imaging: Image Processing' and its corresponding community.

Journal impact factor

The journal impact factor is calculated using [2]

$$\mathrm{IF}_y = \frac{\mathrm{Citations}_y}{\mathrm{Publications}_{y-1} + \mathrm{Publications}_{y-2}}. \tag{1}$$

This is executed by first calculating the citations for a certain journal and year and then subquerying the sum of publications of the two years before using the `call`-command. If no publications are made the prior years, the impact factor is set to zero. Query:

```
MATCH (jo:Journal)-[]-(pa:Paper)<-[ci:Cites]-()
WITH jo.name AS name ,jo.year AS year, ToFloat(COUNT(ci)) AS citations
CALL {
    WITH name, year
    MATCH (jo2:Journal {name: name})-[pu:Publishes]-()
    WHERE jo2.year IN [year-2, year-1]
    RETURN ToFloat(COUNT(pu)) AS prior_publications
}
RETURN name, year, CASE prior_publications WHEN 0 THEN 0 ELSE (citations/prior_publications) END AS
    impact_factor
ORDER BY impact_factor DESC
```

Result: The journal with the top impact factor is called "IEEE Trans. Geosci. Remote. Sens." in 2019 with impact-factor 119.

Author h-index

The author h-index is defined as the maximum value of h such that the given author has published at least h papers that have each been cited at least h times [1]. To calculate this, both the amount of published papers and received citations are calculated and the minimum taken. Query:

```
MATCH (au:Author)-[wr:Writes]->(pa1:Paper)<-[ci:Cites]-(pa2:Paper)
WITH au, wr, pa1, COUNT(ci) as times_cited
WITH times_cited, au.name as author, COUNT(wr) as amount_published
RETURN author, (CASE WHEN amount_published <= times_cited THEN amount_published ELSE times_cited END
    ) as h_index
ORDER BY h_index DESC
```

Result: The highest h-value of 3 is for author "Robert Schober".

C: Algorithms

0.0.1 Similarity of Conferences

One question of interest with this data, is to see how similar conferences are in terms of the authors attending them. In order to get insight in this interest, the `Node Similarity` algorithm from the Graph Data Science package in Neo4j can be used, which computes the similarity of two nodes according to the following equation for the Jaccard Similarity Score:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

In this case, $|A|$ denotes all the outgoing edges of node a and $|B|$ denotes all the outgoing edges of node b. Thus $|A \cup B|$ denotes all the shared edges between a and b which are the edge that go between a and b

An author is considered to attend two conferences if they contributed to papers from published by both conferences. Thus, to make it easy to execute the node similarity algorithm, a new helper node called "CommonAuthor" is made which connects two conferences. To do this, the following query is executed.

```
MATCH (c:Conference) - [:Publishes] -> (p:Paper) <- [:Writes] - (a:Author) - [:Writes] -> (p2:Paper)
    <- [:Publishes] - (c2:Conference) WHERE c.name <> c2.name AND c <> c2
WITH c, c2, a
MERGE (c) - [:connects] -> (ca:CommonAuthor {name: a.name}) <- [:connects] - (c2)
```

Next, a `gds.graph` is build with only the nodes of interest: conference and CommonAuthor, which is done as follows:

```
CALL gds.graph.create(
    'myGraph',
    ['Conference', 'CommonAuthor'],
    {connects: {
            type: 'connects'}});
```

Then, to compute the similarity the following query is performed:

```
CALL gds.nodeSimilarity.stream('myGraph')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Conference1, gds.util.asNode(node2).name AS Conference2,
    similarity
ORDER BY similarity DESCENDING, Conference1, Conference2
```

Lastly, the helper nodes are deleted again, as they are no longer needed in the graph.

```
MATCH (c:CommonAuthor) DETACH DELETE c
```

The results of this algorithm are dependent on which nodes are loaded in the graph database and are thus only discussed in terms of what they represent. Simply, the node similarity returns the Jaccard Similarity Score of two nodes, in the case of this query, in descending order. There higher the score, the more similar nodes are.

Community detection

A community can be defined as a group of vertices such that nodes inside the group are connected with many more nodes within the group than outside the group. One way of detecting these is using the Louvain algorithm, which bases its method on optimizing the so called modularity Q, a measure for the relative edge density within compared to outside a group [3], thus leading to interesting density-based results. The modularity score goes from $Q \in [-\frac{1}{2}, 1)$, when a single community exist $Q = 0$, meaning that the whole graph is just one community. When Q is negative, this means that the graph is random, and no single community can be determined. The best selection of communities would give us the highest Q equal to 1.

In order to apply the Louvain algorithm to the current data base, the following use case is proposed. Given all authors nodes and their relations to other nodes, find communities of authors. This could be of use for example if a given author is looking for co-writers in the same field, that work with the same people, cite the same papers, etc. The first step is to find all authors that have a node in common by running the following query.

```
match (a:Author) - [] -() - [] - (a2:Author)
where a <> a2
with a, a2
merge (a) - [:knows] -(a2)
```

Now the graph projection is created.

```
CALL gds.graph.create(
    'myGraph',
    'Author',{ knows: {
            orientation: 'UNDIRECTED'}})
```

Next, the Louvain algorithm is called.

```
CALL gds.louvain.stream('myGraph')
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC
```

The first result is a list of all author nodes and to which community they belong. Next, the following command is run.

```
CALL gds.louvain.stats('myGraph')
YIELD communityCount
```

This returns how many total communities are present in the graph.

D: Recommender

A simple reviewer recommender for editors and chairs is now created in the context of the database community. First the concept of a community is introduced and a corresponding node created. It is a node connected to all papers containing a keyword related to databases with the 'Part_of' relation. The corresponding query is provided below.

```
CREATE CONSTRAINT ON (c:Community) ASSERT (c.name) IS NODE KEY;
CREATE (:Community {name: 'database'});
MATCH (pa:Paper)
MATCH (co:Community {name: 'database'})
WHERE ANY(x IN pa.keywords WHERE x IN ['data management', 'indexing', 'data modeling', 'big data', '
    data processing', 'data storage', 'data querying'])
MERGE (pa)-[:Part_of]->(co);
```

Because the keywords are syntactically generated, all interpretations considering this community can not be considered realistic. However, to illustrate how such a community might look, graph 4 plots the community with its corresponding papers and their respective authors and journals or conferences.

Now, all conferences and journals related to the database are found. These are those conferences/journals which published papers are for 90 % part of the database community. To find these, this proportion is calculated and filtered on the 90 % threshold using the following two queries.

```
MATCH (jo:Journal)-[:Publishes]-(pa:Paper)
WITH jo.name as journal, ToFloat(count(distinct pa)) as total_papers
CALL {WITH journal
    MATCH (jo {name:journal})-[:Publishes]-(pa)-[:Part_of]-(co:Community {name:          'database'})
    RETURN jo.name, ToFloat(count(distinct pa)) as papers_in_community}
WITH papers_in_community/total_papers as prop_in_community, journal
WHERE prop_in_community > 0.9
RETURN journal, prop_in_community;
MATCH (con:Conference)-[:Publishes]-(pa:Paper)
WITH con.name as conference, ToFloat(count(distinct pa)) as total_papers
CALL {WITH conference
    MATCH (con {name:conference})-[:Publishes]-(pa)-[:Part_of]-(co:Community {name:          '
    database'})
    RETURN con.name, ToFloat(count(distinct pa)) as papers_in_community}
WITH papers_in_community/total_papers as prop_in_community, conference
WHERE prop_in_community > 0.9
RETURN conference, prop_in_community;
```

Then, the pagerank algorithm is used to find the top-100 papers of the community, which is based on the within-community amount of citations per paper. The pagerank algorithm is applied in NEO4J using the `Graph Data Science` plug-in. First a projection to a subgraph on which to apply the algorithm is created using the following query.

```
CALL gds.graph.create.cypher('database_papers',
  'MATCH (co:Community {name: "database"})-[]-(p:Paper) RETURN id(p) AS id',
  'MATCH (p)-[c:Cites]->(m) RETURN id(p) as source, id(m) as target',
  {validateRelationships: False})
YIELD
  graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipCount AS rels;
```

Next, the pagerank for within citation is calculated as follows and each of the corresponding papers connected to the community by a new "Top_100_papers" relation.

```
CALL gds.pageRank.write.estimate('database_papers', {
  writeProperty: 'pageRank',
  maxIterations: 20,
  dampingFactor: 0.5})
CALL gds.pageRank.stream('database_papers')
YIELD nodeId, score
WITH gds.util.asNode(nodeId).title  AS title, score
ORDER BY score DESC, title ASC
LIMIT 100
MATCH (pa:Paper {title:title})-[]-(co:Community {name:'database'})
MERGE (pa)-[:Top_100_paper {score:score}]->(co);
```

Finally, to decide which authors of the community are best for reviews, all authors contributed to at least two papers of these top 100 are selected using following query.

```
MATCH (a:Author)-[w:Writes]-(pa:Paper)-[:Top_100_paper]-(co:Community {name:'database'})
WITH a.name as author, count(w) as amount_of_top_papers_written
WHERE amount_of_top_papers_written >= 2
RETURN author, amount_of_top_papers_written
ORDER BY amount_of_top_papers_written DESC;
```

This results in only one sole name: "Yi Yang 0001".

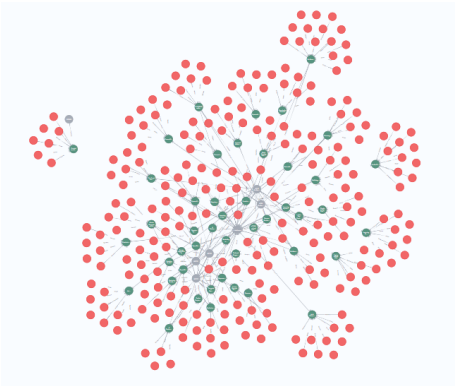

Figure 4: Visualization of community node.

References

[1] Wikipedia. H-index — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=H-index&oldid=1078308819, 2022. [Online; accessed 24-March-2022].

[2] Wikipedia. Impact factor — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Impact%20factor&oldid=1078580576, 2022. [Online; accessed 24-March-2022].

[3] Wikipedia. Louvain method — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Louvain%20method&oldid=1073360472, 2022. [Online; accessed 24-March-2022].