Property Graphs

Research Publications Graph Database Design

Master in Data Science, FIB, UPC
Semantic Data Management

March 16th, 2023

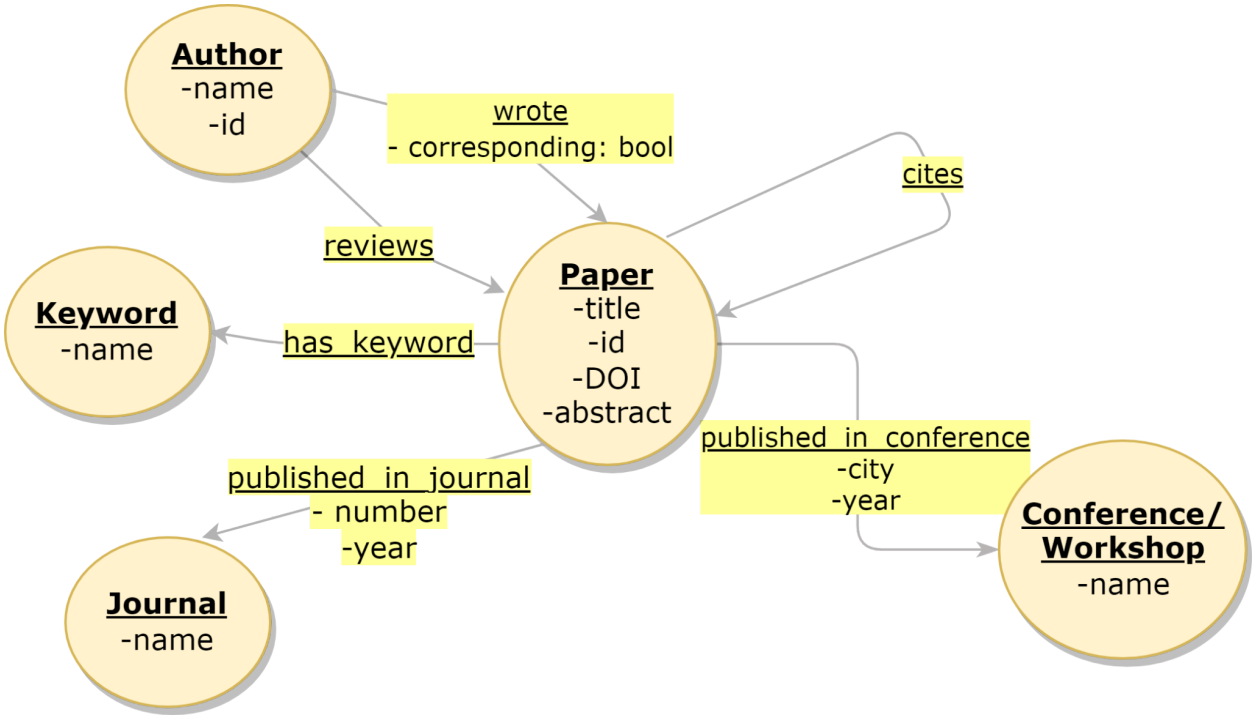
Pau Comas Herrera - Odysseas Kyparissis

Table of Contents

A. Modeling, Loading, Evolving

A1. Modeling

In this chapter of the assignment, the modeling part of the property graph for the research publications is taking place. After reading the design requirements and considering the future queries we were provided by, the following design was implemented:



The main challenges encountered were related to keeping the graph as simple as possible while allowing all possible desired queries. Attributes for nodes and relations were chosen based on the statement but also on the available data from our source. Here are listed some of the decisions we deemed relevant and the justification:

- Implemented *reviews* as a new relation from the node *Author* to *Paper* instead of having a new node *Reviewer*. The specific decision allows adding attributes to the review and saves up space. Space saving occurs due to the fact that a node *Reviewer* would still need to create another relation (e.g. *reviews)* between *Reviewer* and *Paper* nodes. Besides that, while a *Reviewer* is still an *Author* the usage of *Reviewer* as a node would include data redundancy to the solution.
- Implemented relations *published_in_journal* and *published_in_conference* between *Journal* and *Paper*, and *Conference* and *Paper* respectively. In this way, it is possible to contain volume and edition information in the relations instead of having separate nodes

for them. In our queries, it was important to query by the edition of a *Conference* or volume (number and year) of a *Journal*. We believe that the best way to avoid unnecessary relations and nodes was to relate *Paper* with both *Journal* and *Conference* nodes by using a relation. Also, we realized that, for instance, in case we created a node *Volume* it would generate many more different nodes than the expected number of nodes being generated by a *Journal* node.

- Another decision was to include *year* as an attribute in both *published_in_journal* and *published_in_conference* relations instead of using it as an attribute inside node *Paper*. We assumed that a paper can be written in year X and be published in a volume in year Y or, in the other case, be showcased in a conference in a year Z as well. That doesn't allow the simplification of the graph by including the *year* attribute in *Paper*. Nonetheless, when evolving the graph we realized we also need that as well, as we'll see in the next steps.

A2. Instantiating/Loading

In this chapter, the preprocessing of the data and the loading process to Neo4j Graph Database Management System[1] are presented. As it was recommended to use real data for the solution of this assignment, the data for the scientific publications were gathered from DBLP[2]. Due to the complex structure of the data provided by DBLP, we conducted research online and found the representation of the data, in JSON[3] format, from the aminer.org[4] webpage. The latest version of the data offered from *aminer.org* was used (V14, updated in 2023). Due to the fact that the provided archive was very large in size (14 GBs), we randomly selected 70000 papers from it. A script for preprocessing and transforming the data into a suitable format for loading the property graph database was implemented. The steps followed during the preprocessing and loading of the data are described here and more details can be found in *PartA.2 ComasKyparissis.ipynb* file submitted.

To begin with, the selection of the necessary columns took place. Missing data and duplicate observations removal was completed next. Additionally, routines for quality assurance of the data were developed. In more detail, we discarded all the papers published in a Journal, if the information about the year or volume of the journal was missing. Moreover, besides the fact that the information about the references of each paper was available, we needed to make sure that the unique identifiers of the papers, mentioned in the references column, existed in the column of the paper's unique identifier. For the reason that we randomly selected only 70000 out of millions of observations that were included in the raw JSON dataset, we decided to replace the true references with artificial ones from the selected 70000 papers. This procedure was completed by taking into account that a paper can not reference itself, and the sampling was completed by using Poisson distribution[5] with λ equal to 8. In addition, while the conferences did

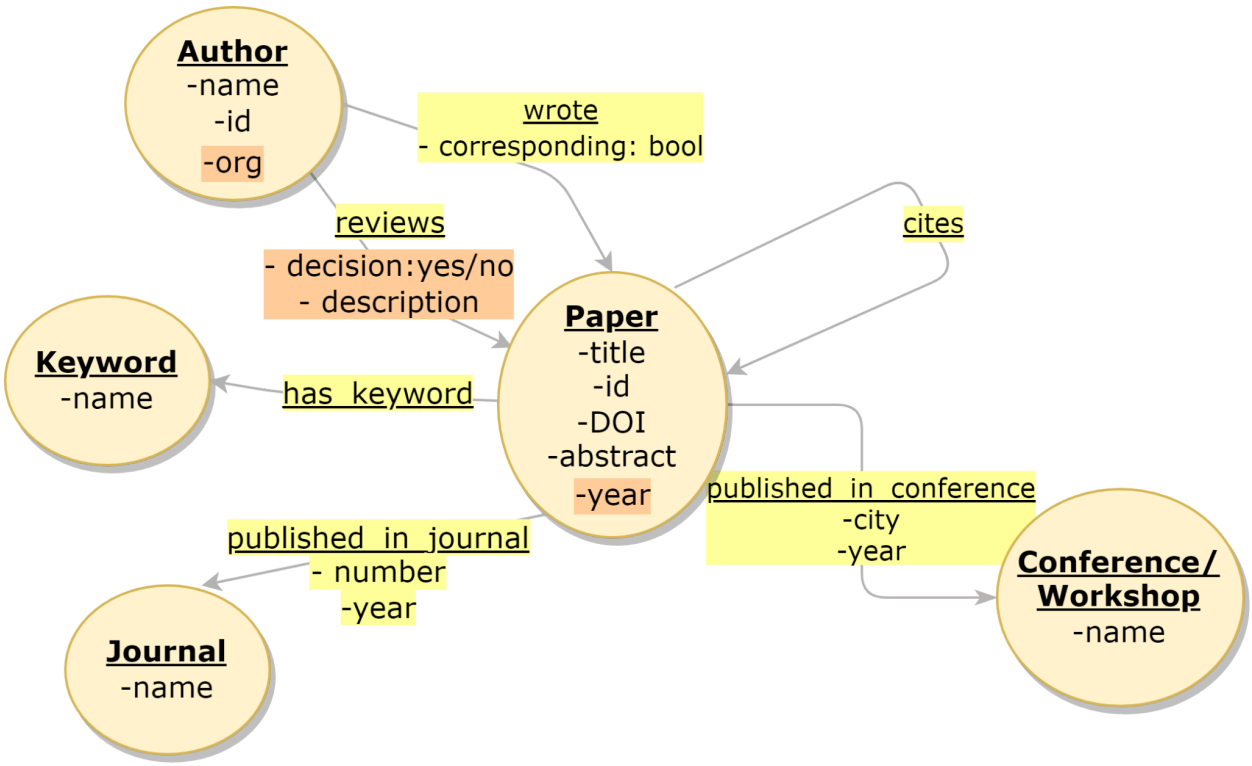
[1] https://neo4j.com/
[2] https://dblp.uni-trier.de/
[3] https://www.json.org/json-en.html
[4] https://www.aminer.org/citation
[5] https://en.wikipedia.org/wiki/Poisson_distribution

not include the city where they took place in, we generated artificial data for this attribute, as well, by selecting random European cities. Furthermore, while data for the reviewers of a paper was missing from the raw dataset, we generated a random sampling mechanism that assigned to each paper 3 random reviewers from the available authors (except the ones who are the authors of the specific paper). Finally, some formatting fixing of specific columns was conducted. To conclude, for performance reasons and after trial and error with the results generated by the queries of parts B and C, we decided to load a total of 10000 papers that follow all the constraints mentioned above.

As the next step of this chapter, based on the graph design decided in part A of the assignment, the creation of the CSVs with the necessary data was completed. For each node and relationship of the graph, the appropriate structure of a CSV was generated and saved into the project's directory. In that way, we could apply bulk loading queries, in order to instantiate the graph and load the preprocessed data. More details about the structure of the CSV files and the loading queries can be found in *PartA.2 ComasKyparissis.ipynb* file as well.

A3. Evolving the graph

One key aspect of graph databases is their flexibility to absorb changes in the data coming into the system. Based on the new specifications and demands presented, we updated our initial graph resulting to the following one:



The new schema allows for monitoring information of a *review* as well as the affiliation (labeled *org)* of every *Author*. The information, about the affiliation and the paper's year of writing, was already provided by our data source, so we just had to update the data instances by merging them. As opposed, synthetic new review decisions and descriptions were generated before updating the graph.

- At that time, we found ourselves at a crossroad of determining if *org* should be an attribute of *Author* or a new node related to it. After we found out that we had hundreds of different organizations, we decided that it was better to store it as an *Author* attribute to avoid generating many new nodes and relations. Although storing it as a new node would ease querying by an organization, it would generate a huge space of objects.

B. Querying

In this section, the solutions for the requested queries are presented. In the following blocks, the Cypher implementation can be found:

- Find the top 3 most cited papers of each conference:

```
Unset
MATCH (c:Conference)<-[:published_in_conference]-(p:Paper)
OPTIONAL MATCH (p)<-[:cites]-(cp:Paper)
WITH c, p, COUNT(DISTINCT cp) AS citation_count
ORDER BY c.name, citation_count DESC
WITH c, COLLECT(p)[..3] AS top_papers
RETURN c.name AS conference_name, [paper IN top_papers | paper.title] AS top_paper_titles
```

- 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 (a:Author)-[:wrote]->(p:Paper)-[pub:published_in_conference]
->(c:Conference)
WITH c, a, collect(DISTINCT pub.city + pub.year) as edition
WHERE size(edition) >= 4
RETURN c.name AS conference, collect(DISTINCT a.name) AS community, edition
```

- Find the impact factors[6] of the journals in your graph:

[6] https://en.wikipedia.org/wiki/Impact_factor

```
Unset
MATCH (j:Journal)
OPTIONAL MATCH (p:Paper)-[pub:published_in_journal]->(j:Journal)
WHERE pub.year = $year_2 OR pub.year = $year_1
OPTIONAL MATCH (p)<-[:cites]-(cp1:Paper{year:$year})
WITH j, p, COUNT(DISTINCT cp1) AS citation_count
WITH j, COUNT(DISTINCT p) AS paper_count, SUM(citation_count) AS citation_sum
RETURN j.name AS journal_name,
CASE WHEN paper_count = 0 THEN 0 ELSE citation_sum/paper_count END AS impact_factor
ORDER BY impact_factor desc
```

In this case, we obtain the impact factor of a specific year by passing *$year,* and the previou two years (*$year1* and *$year2)* as parameters to the function that executes the specific query (more details in *PartB ComasKyparissis.ipynb*.

- Find the h-indexes[7] of the authors in your graph:

```
Unset
MATCH (a:Author)-[:wrote]->(p:Paper)<-[:cites]-(c:Paper)
WITH a, p, COUNT(DISTINCT c) AS citations
WITH a, COLLECT({paper: p, citations: citations}) AS papers
WITH a, REDUCE(s = {index: 0, h: 0}, x IN papers |
CASE WHEN x.citations >= s.index THEN {index: s.index + 1, h: s.index + 1}
ELSE s END) AS hIndex
RETURN a.name AS author_name, hIndex.h AS h_index ORDER BY h_index DESC
```

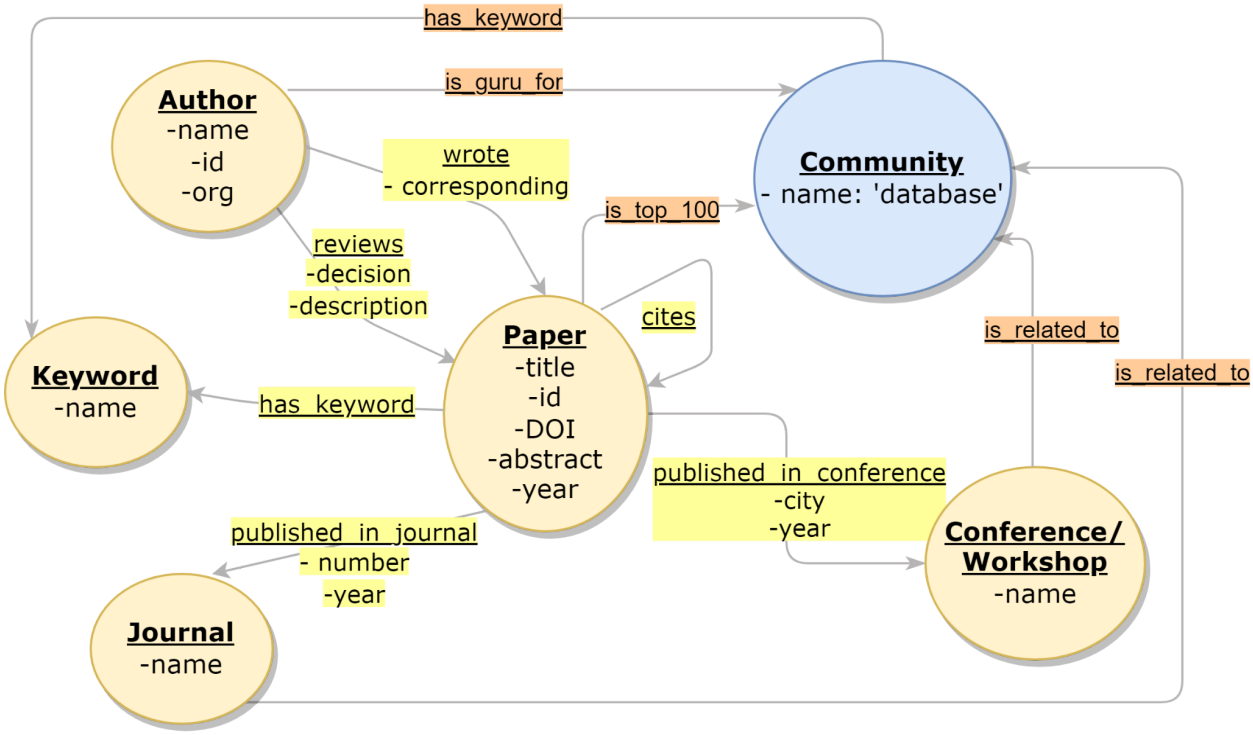
C. Recommender

The goal of this section is to create a reviewer *recommender* for editors and chairs. The idea is to be able to identify potential reviewers for the database community. For that, we need to further update the graph with step-wise inferred information to embed the recommendation information for future use. The updated graph can be found in the following Figure.

Firstly, we had to identify the *database* community which is defined by a set of keywords. Thus, we created the node *Community* and applied the relation *has_keyword* to the *Keyword* nodes of the graph that include as a *name* attribute one of the strings of the following list: *data management, indexing, data modeling, big data, data processing, data storage,* and *data querying*. Right after, for every *Conference* and *Journal* node with at least 90% of papers with keywords from the community ‘database’, we created a *is_related_to* relation to that community. Next step of the recommender solution includes fetching the top 100 papers of conferences and journals, from the ‘database’ community, by calculating the page rank based on citations to

[7] https://en.wikipedia.org/ wiki/H-index

them. For that, we had to project our whole graph into the subset of 'database' papers and citations, and then apply `gds.pageRank.stream` to obtain the scores. The inferred new information about which are the top 100 'database' papers was appended to the graph by means of a *is_top_100* relation between *Paper* and *Community* nodes.



Finally, only after this long process, we were able to search for very good potential reviewers for the database papers, as well as, for *gurus,* meaning very reputed authors that would be able to review for top conferences. For identifying very good potential reviewers we had to query all the authors who wrote at least one of the top-100 papers of the *database* community. For identifying gurus as those authors that have written at least two papers among the top-100 identified. Consequently, we developed the query which in the future can easily be applied and provide possible reviewers for *database* papers. Furthermore, we related the respective *Author* nodes (gurus) to the *Community* node, with a *is_guru_for* relation.

All the process and query creation is described in great detail in *PartCComasKyparissis.ipynb*. In the notebook, it can be seen that, in our use case, the only guru for the *database* community would be *Michael Stonebraker*, with *node-id 31091*.

D. Graph algorithms

Beyond regular queries, graphs allow the exploitation of graph theory and the use of well-known graph algorithms. For experimenting with this feature, we present here an application of Dijkstra's shortest path finding algorithm and Node similarity algorithm.

D1. Dijkstra's shortest path

This algorithm is a graph traversal algorithm that finds the shortest path between a starting node and all other nodes in a weighted graph. It works by maintaining a set of visited nodes and a priority queue of unvisited nodes, selecting the node with the smallest tentative distance, and relaxing the adjacent nodes until the destination is reached.

In our graph, we apply it to find the lowest cost in terms of *citations* hops between any two *Paper* nodes. For that, we need to first build a projection of our graph:

```
Unset
CALL gds.graph.project.cypher('paper_cites',
MATCH (p:Paper) RETURN id(p) AS id',
MATCH (p:Paper)-[c:cites]->(p2:Paper) RETURN id(p) AS source, id(p2) AS target, type(c) as
type')
```

In *paper_cites* we store the desired projection, where we can directly execute the `gds.allShortestPaths.dijkstra.stream` call, allowing us to obtain the shortest path cost from a source node to all the nodes of the projection. With that in mind, and by filtering the results as needed, we can obtain the cost from two papers in terms of citation hops (each one with cost weight = 1):

```
Unset
MATCH (p:Paper {title: $titol1}) WITH id(p) as id_P, p
MATCH (p2:Paper {title: $titol2}) WITH id(p) as id_P, p, id(p2) as id_P2, p2
CALL gds.allShortestPaths.dijkstra.stream('paper_cites', {sourceNode: p})
YIELD sourceNode, targetNode, totalCost
WHERE targetNode = id_P2
RETURN gds.util.asNode(sourceNode).title as sourceNode, gds.util.asNode(targetNode).title
as targetNode, totalCost
```

For demonstration purposes, if the following parameters are set as follows: *$titol1 = 'Undirected graphs of entanglement 2'* and *$titol2 = 'Fingerprint recognition based on minutes groups using directing attention algorithms'*, the shortest path in terms of citations is of length **5**. More details can be found in *PartDComasKyparissis.ipynb.*

	SourceNode	TargetNode	Cost
0	Undirected graphs of entanglement 2	Fingerprint recognition based on minutes group...	5.0

D2. Node similarity

The Node Similarity algorithm in Neo4j is a similarity measure algorithm that allows to compare nodes based on their relationships with other nodes in the graph. It can be used to identify nodes that are similar to each other in terms of their connectivity patterns. The algorithm works by first creating a matrix that represents the similarity between each pair of nodes in the graph. The matrix is initialized with zeros and then updated with values that reflect the strength of the relationships between nodes. By iterating over the matrix, it compares each pair of nodes and updates the similarity score for each pair based on the similarity scores of their neighbors. The algorithm uses a weighted measure of the similarity between each node's neighbors to compute the overall similarity score between the two nodes. The Node Similarity algorithm is based on the concept of the Jaccard similarity coefficient, which is a measure of the similarity between two sets. In the context of the Node Similarity algorithm, the sets are the sets of neighboring nodes for each pair of nodes being compared.

In the specific solution, the algorithm was performed in a projection of the overall graph that includes the *Paper* and *Keyword* nodes. The main goal of applying this algorithm, to the specific projection, was to identify papers that are very similar, based on the keywords they are using. The generation of the projection is completed by using `gds.graph.project.cypher` function from the *gds* library and it is presented in the following block:

```
Unset
CALL gds.graph.project.cypher(
  'papers_and_keywords',
  'MATCH (a) WHERE a:Paper OR a:Keyword RETURN id(a) AS id, labels(a) AS labels',
  'MATCH (a:Paper)-[r:has_keyword]->(k: Keyword) RETURN id(a) AS source, id(k) AS target,
type(r) as type')
  YIELD
  graphName AS graph, nodeQuery, nodeCount AS nodes, relationshipCount AS rels
```

Once the projection of the graph has been created the application of the Node similarity algorithm was possible. In order to do so, the following Cypher code was used:

```
Unset
CALL gds.nodeSimilarity.stream('papers_and_keywords',
{degreeCutoff: 5, similarityCutoff: 0.5})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).title AS Paper1, gds.util.asNode(node2).title AS Paper2,
similarity
ORDER BY similarity DESCENDING, Paper1, Paper2
```

Finally, the output of the algorithm is a matrix that represents the similarity between each pair of *Paper* nodes in the graph, which can be used for further analysis or visualization. For space-saving reasons, the results produced by applying the Node similarity algorithm are presented in *PartDComasKyparissis.ipynb.*