Property Graphs Lab

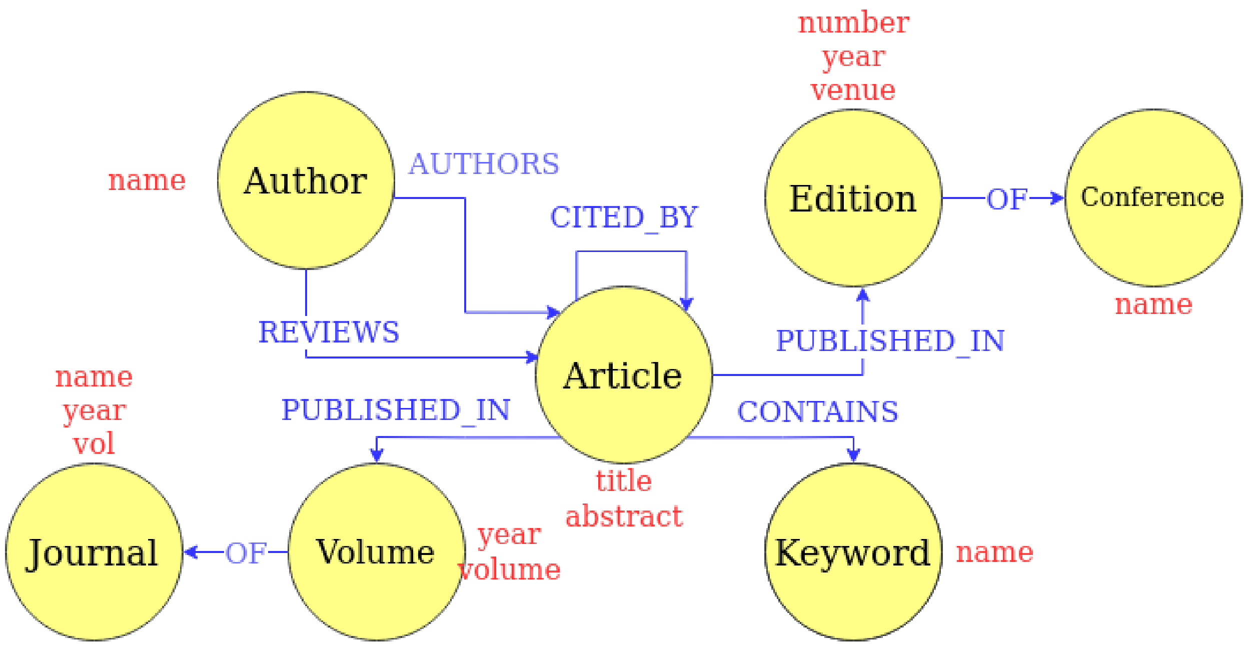
Ferran Noguera Vall
Diego Quintana
Roberto Ruíz

March 16th 2020

A Modeling Loading Evolving

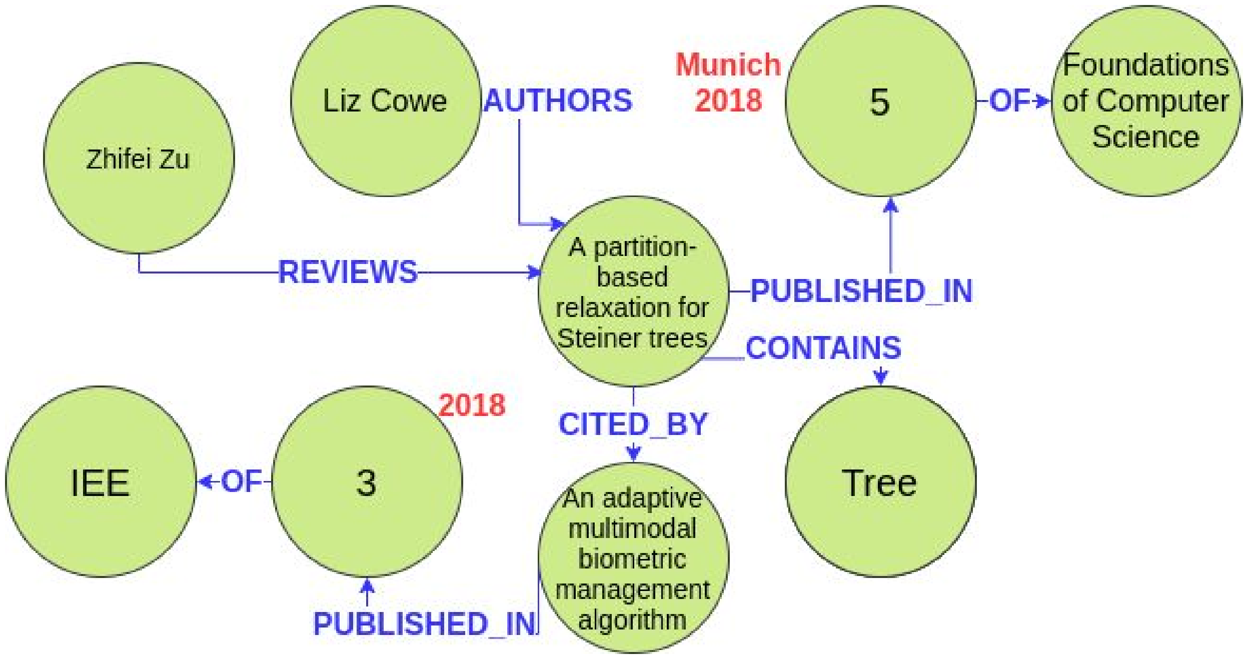
1. Modeling

The metadata can be presented visually as a graph



Here, **Nodes** are written inside the orange circles, whereas RELATIONSHIPS are drawn with blue lines and their respective <u>properties</u> are written in red.

If we instantiate data from the previous graph, it can be seen as



Zhifei Zu
Liz Cowe
AUTHORS
Munich
2018
5
OF
Foundations of Computer Science
REVIEWS
A partition-based relaxation for Steiner trees
PUBLISHED_IN
CONTAINS
2018
CITED_BY
IEE
OF
3
An adaptive multimodal biometric management algorithm
Tree
PUBLISHED_IN

We created the nodes: **Author**, **Conference**, **Article** and **Journal**. Each with its own set of properties. An **Author** either *AUTHOR* or *REVIEW* an **Article**, and an **Author** can not review itself.

After creating these nodes, we had performance issues with the queries from part B. For query B.3, we needed to relate **Edition** to **Conferences**. For query B.4, we related **Volumes** to **Journals.** We also decided that *year* should be an attribute of **Volume** instead of **Journals,** and *venue* an attribute of **Edition** instead of **Conference**. This was done to improve maintenance and reusability since, for instance, if someone wants to change or store the year of a journal, or the venue of a conference, it will have to check every edge, causing a loss in performance and risk having redundant data.

Keywords were decided to be defined as nodes instead of metadata for articles, for an easy implementation of the recommender and also, because even though the number of relations would increase a lot it is expected that most of the keywords will be repeated among the articles causing a lot of redundancy.

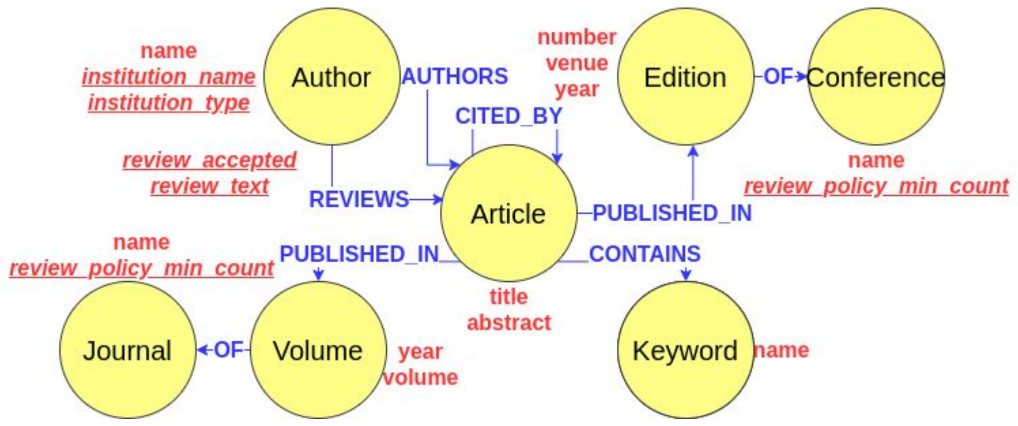
2. Instantiating/Loading

We used the data available from *Semantic Scholar,* downloadable in JSON format[1]. This data provides the information necessary to fill in **Authors, Articles, Keywords, Journals** and **Volumes.** We loaded this data using *apoc*[2] following the instructions from the neo4j website[3].

The rest of data needed in our database in order to fulfill the following exercises was not found on this json, so it had to be faked using the python Faker library[4], where, **Conferences, Editions** nodes**,** CITED_BY, REVIEWS relationships, among others, were simulated.

The full process of munging data and faking the rest is detailed and available as a jupyter notebook in Github[5] and also attached together with this delivery.

3. Evolving the Graph



1 http://s2-public-api-prod.us-west-2.elasticbeanstalk.com/corpus/
2 https://github.com/neo4j-contrib/neo4j-apoc-procedures
3 https://neo4j.com/docs/labs/apoc/current/import/load-json/
4 https://faker.readthedocs.io/en/master/
5 https://github.com/diegoquintanav/od-lab-1/blob/master/notebooks/1-part-a.ipynb

The graph considered in exercise A.1 is flexible enough, allowing us to adapt to the new constraints by just adding a few properties to several nodes.

First we created two new properties for each **Author**. These properties will let us know, if any, the name of the affiliated institution and whether the institution is a university or company (type). Secondly, a new property was created in the nodes **Conference** and **Journal**, setting the minimum reviews policy for each of them. We also added two new properties to the REVIEWS relationships to know if a review is accepted or not and also the review text on the decision.

B. Cypher queries

All three queries are located and ready to be executed in the file called, available as a jupyter notebook in Github[6].

1. H-Index:

The maximum value of *h* such that the given author/journal has published *h* papers that have each been cited at least *h* times.

```
MATCH (author:Author)-[:AUTHORS]->(article:Article)-[:CITED_BY]->(article2:Article)
WHERE article <> article2
WITH author, count(article) as citations
ORDER BY citations DESC
WITH author, collect(citations) as c_citations
WITH author, [i in range(0,size(c_citations)-1)
    WHERE i <= c_citations[i] | i][-1] + 1 as h_index
RETURN author.name as author, h_index
```

2. Most Cited Papers:

Finds the 3 most cited papers in each conference.

```
MATCH(a1:Article)-[:CITED_BY]->(a2:Article)-[:PUBLISHED_IN]->(ed:Edition)<-[:IN]-(cf:Conference)
WITH a1, cf, count(a1) as citations
ORDER BY citations DESC
WITH cf, collect(a1 {.title,citations}) as c_citations
RETURN cf.name AS conference_name, c_citations[..3] as three_most_cited
```

3. Community for a conference

Authors that have published in at least 4 editions of the conference.

```
MATCH (au:Author)-[:AUTHORS]->(ar:Article)-[e1:PUBLISHED_IN]->
(ed:Edition)<-[:IN]-(co:Conference)
WITH au, collect(ed) as editions, co
WHERE size(editions) >= 4
RETURN au as author, co as conference, size(editions) as n_publications
```

[6] https://github.com/diegoquintanav/od-lab-1/blob/master/notebooks/2-part-b.ipynb

4. Impact factors

This query takes a parameter `$year` that will compute the impact factor of each journal on that specific year. The impact factor is the yearly average number of citations received by articles published in the last two years in the journal.

```
MATCH (article:Article {year: $year})
WITH collect(article) AS current_papers
MATCH (article:Article)-[:PUBLISHED_IN]->(:Volume)-[:OF]->(journal:Journal)
WHERE article.year = ($year - 1) OR article.year = ($year - 2)
MATCH (article)-[:CITED_BY]->(citer:Article)
WHERE citer IN current_papers
WITH count(citer) AS n_cites, article, journal
RETURN journal.name as journal, toFloat(SUM(n_cites)/COUNT(article)) AS impact_factor;
```

C. Graph Algorithms

1. PageRank

PageRank is an algorithm that measures the importance of a given node in a graph considering its connections, which can either be or not be weighted. The purpose of this query is to find the most prestigious paper, understanding that a paper is more prestigious the more citations it receives.

```
CALL algo.pageRank.stream(
    'MATCH (a:Article)
     WHERE exists ((a)-[:CITED_BY]->())
     RETURN ID(a) AS id',
    'MATCH (a1:Article)-[:CITED_BY]->(a2:Article)
     RETURN id (a1) AS source, ID(a2) AS target',
    {graph : 'cypher', iterations:20, dampingFactor:0.85})
YIELD nodeId, score
MATCH (a:Article) WHERE ID(a) = nodeId
RETURN a.title AS title, score
ORDER BY score DESC;
```

2. Triangle Counting

Triangle Counting algorithm counts the number of triangles on a given relation graph.
The intention of this query is to find triangles between citations of the articles to detect those authors citing themselves in small groups, which can be due to non detected communities or questionable citing practices.

```
CALL algo.triangleCount.stream('Article', 'CITED_BY')
YIELD nodeId, triangles, coefficient
RETURN algo.asNode(nodeId).title AS title, algo.asNode(nodeId).id as id, triangles, coefficient
ORDER BY coefficient DESC
```

This part is provided as a Jupyter Notebook and it is also available in Github[7]

[7] https://github.com/diegoquintanav/od-lab-1/blob/master/notebooks/3-part-c.ipynb

D. Recommender

Research Community definition.

The database community is defined using the keywords: `data management`, `indexing`, `data modeling`, `big data`, `data processing`, `data storage`, `data querying`. Since these keywords are not in the database, we will create them, and we will assign them randomly to existing articles.

Finding Conferences and Journals

Based on the existing pattern in the graph we just created, e.g.

```
MATCH (:ResearchCommunity {name: 'databases'})-[:CONTAINS]->(:Keyword)<-[:CONTAINS]-
(article:Article)-[:PUBLISHED_IN]->(instance)-[:OF]->(publication)
```

We can start finding those conferences and journals that share the same keywords with the research community. Counting the number of publications in this path, over the number of publications in a more general path i.e. without considering the keywords, we can obtain the ratio between them. If this ratio is bigger than 0.9, we will assign these conferences as RELEVANT_TO the **ResearchCommunity**.

```
MATCH (article:Article)-[:PUBLISHED_IN]->(instance)-[:OF]->(publication)
WHERE (publication:Journal OR publication:Conference) AND (instance:Volume OR instance:Edition)

WITH instance, count(article) AS total_articles
MATCH (:ResearchCommunity{name:'databases'})-[:CONTAINS]->(:Keyword)<-
      [:CONTAINS]-(article:Article)-[:PUBLISHED_IN]->(instance)-[:OF]->(publication)
WHERE (publication:Journal OR publication:Conference) AND (instance:Volume or instance:Edition)

WITH publication, total_articles, COUNT(DISTINCT article) as related_articles
WHERE toFloat(related_articles)/toFloat(total_articles) >= 0.9

MATCH (rc:ResearchCommunity {name: 'databases'})
MERGE p=(publication)-[:RELEVANT_TO]->(rc)
```

Top Papers of Selected Conferences and Journals

Using the same path as above, we can obtain the PageRank score of the articles in the path. If we sort them and keep the first one hundred, then we can recommend these articles to the research community.

```
CALL algo.pageRank.stream(
       'MATCH (c:ResearchCommunity {name:
"databases"})<-[:RELEVANT_TO]-(publication)<-[:OF]-(instance)<-[:PUBLISHED_IN]-(article:Article)
WHERE (instance:Volume OR instance:Edition) AND (publication:Journal OR publication:Conference)
RETURN article',
       'CITED_BY', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score
MATCH (a:Article) WHERE ID(a) = nodeId
WITH a, score
ORDER by score DESC
WITH a, score LIMIT 100
```

```
MATCH (rc:ResearchCommunity {name: "databases"})
MERGE p=(a)-[:RECOMMENDED_TO]->(rc)
RETURN p, score;
```

Finding community gurus

Finally, for discovering the gurus, we need to remember that all the papers RECOMMENDED_TO the research community are those in the top 100 of the PageRank algorithm. The authors of these articles then lie in the path

```
MATCH (a:Author)-[:AUTHORS]->(b:Article)-[:RECOMMENDED_TO]->(rc:ResearchCommunity {name:
'databases'})
```

If we COUNT the number of articles they published in this path, we can then define as gurus those authors that have 2 or more publications.

```
MATCH (a:Author)-[:AUTHORS]->(b:Article)-[:RECOMMENDED_TO]->(rc:ResearchCommunity {name:
'databases'})
WITH a, count(b) as n_articles, rc
WHERE n_articles >= 2
MERGE (a)-[:GURU_OF]->(rc)
```

This part is available as a Jupyter Notebook and it is also available in Github[8]

[8] https://github.com/diegoquintanav/od-lab-1/blob/master/notebooks/4-part-d.ipynb