OD: Lab Assignment 1

Antoni Casas
Arnau Abella

Universitat Politècnica de Catalunya

March 22, 2021

A Modeling, Loading, Evolving

A.1 Modeling

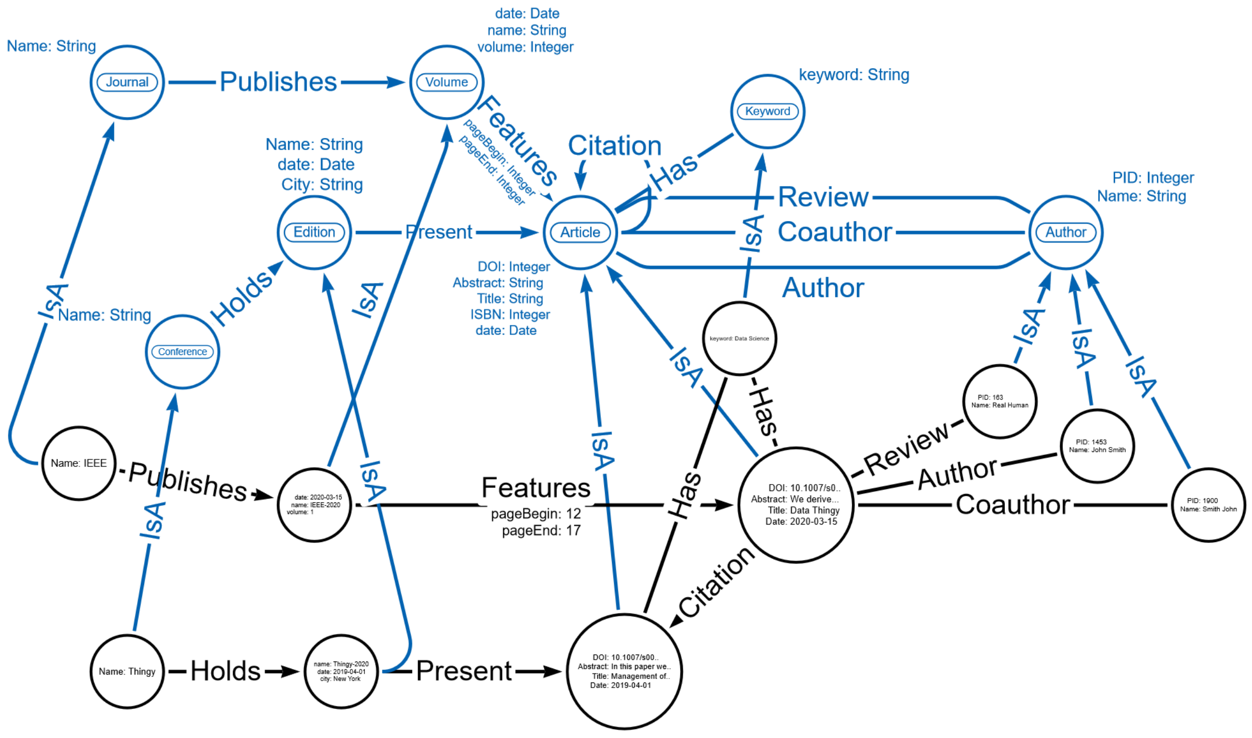

Figure 1: Data Model

Conferences and Journals are kept different due to their semantic differences, even if they could be the same node, data wise.

Conferences can hold events freely, and journals can publish volumes freely, however these must hold articles.

Articles are compromised of their doi, abstract, title and publish date, this date can be used to express differences such as an article being published somewhere before being presented in the edition of the conference, or being published in somewhere like ArxiV before being published in a journal, but also it allows further efficiency when having to obtain the article's date for the impact factor query.

Every author is also a coauthor, author only expresses that such an author is the primary author of that article.

A.2 Instantiating/Loading

In order to instantiate the data for the model of subsection A.1 there is a quite involved process with the following steps:

1. Extract the DOIs of the articles from the DBLP CSV's (see `doiExtraction.py`).
2. Retrieve from *Semantic Scholar* all the information related to the articles (see `retrieveAll.py`).
3. Pre-process the retrieved data and transform it to the corresponding CSVs that are going to be involved in the loading process (see `recordsToCSV.py`).
4. Load the data into *Neo4j* (see `loadNeo4j.py`).

The script `recordsToCSV.py` does some additional transformation the data from Semantic Scholar since, in some queries, it was not given the expected results:

- Conferences and Journals are synthetically created in order to create a suitable topology for the queries.
- Editions and Volumes too.
- Reviewers are randomly selected among all the authors excluding themselves.
- Keywords are artifically added to make the queries in D possible

All the original and synthetic data can be found at `partA2/data`.

A.3 Evolving the graph

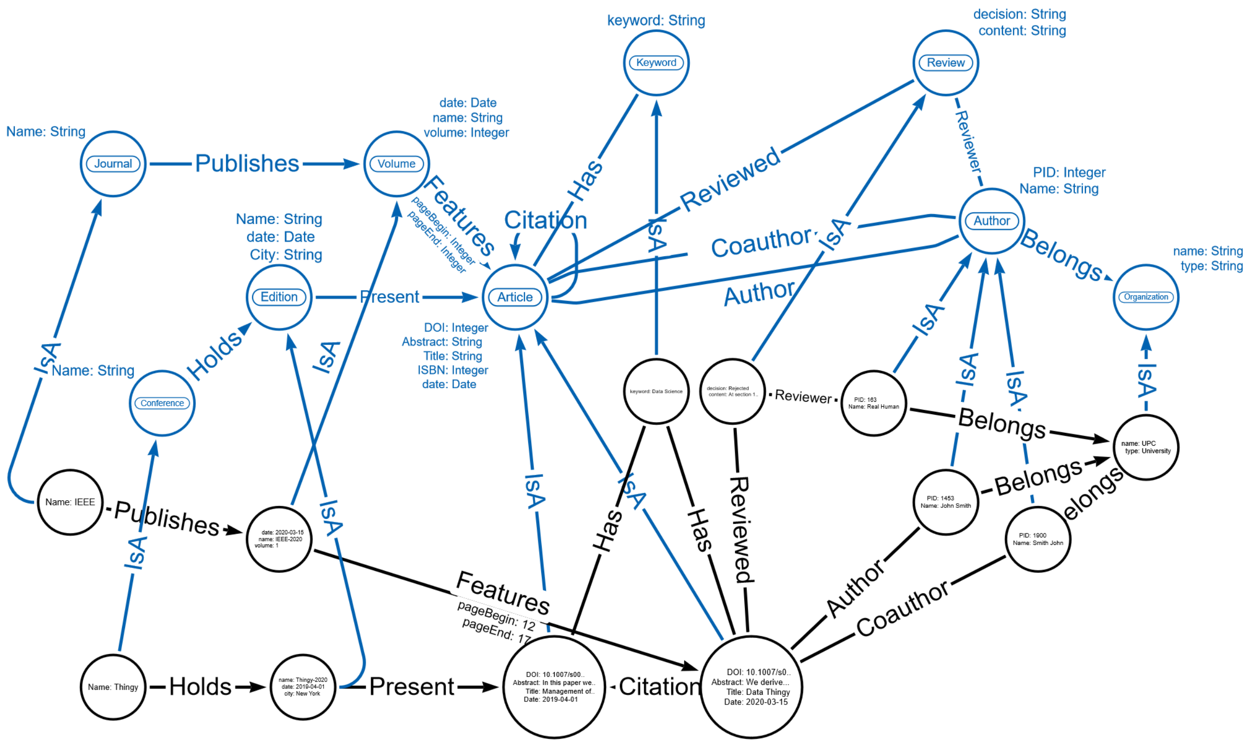

Figure 2: Modified Graph

To adapt the graph to contain the new information, we make the following 2 changes. First, we remove the old relationship of Reviewer we had from author to article, and replace it with a relation to a node Review, which holds the review's contents and its acceptance status, and this node is related to the author who made this review and the article which it reviews.

Secondly, we add a new node, Organization, which stores the organizations, and this node is related to authors, representing via this relationship the organization the author belongs to.

B Querying

B.1 Query 1

```
MATCH ((conf:Conference)-[:Holds]-(edi:Edition)-[:Present]-(art:Article)-[cit:Citation]->(art2:Article))
WITH conf,art,count(cit) as citations ORDER BY conf, count(cit) DESC
WITH conf, collect([art,citations]) AS toCut UNWIND toCut[0..3] AS cut RETURN conf,cut[0],cut[1];
```

This query finds the top 3 most cited papers of each conference. To do this, it first obtains the path match obtaining every citation a conference has had, then it aggregates this on unique conferences and articles, obtaining the count of citations of each article. Then, it aggregates these in order to return only the first 3 per conference.

B.2 Query 2

```
MATCH ((conf:Conference)-[:Holds]-(ed:Edition)-[:Present]-(Art:Article)-[:Coauthor]-(aut:Author))
WITH conf,aut,count(DISTINCT ed) as numEds
WHERE numEds>3
RETURN conf,aut;
```

This query finds for each conference find its community: those authors that have published paperson that conference in, at least, 4 different editions. To do this, it first obtains the patch match obtaining for each conference and edition held, all the coauthors, then it aggregates by conference and author, to count the number of different editions, selecting over those aggregations where the number of editions is over 3.

B.3 Query 3

```
MATCH ((jor:Journal)-[pub:Publishes]-(vol:Volume)-[:Features]-(art:Article))
WHERE vol.date.year>($year-3) AND vol.date.year>$year
OPTIONAL MATCH ((art:Article)-[cit:Citation]->(cited:Article))
WHERE cited.date.year=$year
WITH count(cited) AS cited, count(art) as publications, jor
RETURN (cited*1.0) / (publications*1.0) AS IF,jor,$year AS year
```

This query finds the impact factor of the journals for $year. To do this, it first obtains all the articles published the 2 years before $year by each journal, then, it optionally matches those articles with their citations, and selects those citations in $year. Obtaining this way the total count of articles in the previous 2 years, and the citations of the year.

B.4 Query 4

```
MATCH ((aut:Author)-[:Coauthor]-(art:Article))
OPTIONAL MATCH ((art:Article)-[cit:Citation]->(art2:Article))
WITH aut, art, count(cit) as citation
ORDER BY aut, citation DESC
WITH aut,COLLECT(citation) AS citations
RETURN aut,citations
```

This query finds the h-indexes of the authors. To do this, it first obtains the path belonging, for each main author, to their published articles, then optionally matching it with the citations, this to also obtain authors that haven't been cited (h-index of 0), then it aggregates by author and article, obtaining the amount of citations per article, then it collects these in order, to be processed outside of Neo4J in a simple for loop.

C Graph algorithms

C.1 Page Rank

Page Rank is used in our graph to measure the relevance of an article. The traditional approach of estimating the relevance by the number of references can be considered naïve since it does not take into account the relevance of the references. For example, an article with 20 relevant citations should be more relevant that another article with 25 non-relevant citations.

For running page rank on our graph we prepared a *python* script (see `partC/pageRank.py`) which prints the page rank of each node in descending order.

```
CALL gds.pageRank.stream($graphName)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).title AS name, score
ORDER BY score DESC, name ASC
LIMIT $limit
```

Table 1 includes the most relevant articles with respect to its page rank in our database.

Title	Page Rank
Quad trees a data structure for retrieval [...]	155.4
A fast algorithm for Steiner trees	80.4
The log-structured merge-tree (LSM-tree)	62.8
Concurrency of operations on B-trees	58.3
The temporal logic of branching time	58.2
Amounts of nondeterminism in finite automata	46.1
Subtyping for session types in the pi calculus	31.1
The SB-tree an index-sequential structure [...]	30.4
Optimal scheduling for two-processor systems	26.2
Branching processes of Petri nets	23.9

Table 1: Top 10 most relevant papers w.r.t. Page Rank

C.2 Louvain Method

The Louvain method can be applied to our graph, in particular to the articles nodes, to find related articles. We prepared a script (see `partC/louvain.py`) to run the Louvain method and find the hidden communities. The following script is part of the code to obtain the communities from our graph.

```
CALL gds.louvain.stream($graphName)
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).title AS title, communityId
ORDER BY communityId ASC
```

In table 2 we display the largests communities in our graph which contains 449 communities in total. For each community we will display its size and three randomly selected articles that belong to the community. Although our database does not contain the topic of the articles, it is clear that the communities are related by topic of publication. For example, community 2 is about tree data structures, community 5 is about program verification, community 6 is about programming language theory, and so on.

Community	Size	Articles
1	86	*M-nets: An algebra of high-level Petri nets [...]* *On Gurevich's theorem on sequential algorithms* *Branching processes of Petri nets*
2	73	*A fast algorithm for Steiner trees* *Applications of efficient mergeable heaps for optimization problems on trees* *On efficient implementation of an approximation algorithm for the Steiner tree problem*
3	71	*Quad trees a data structure for retrieval on composite keys* *Picture deformation* *Dynamic multi-dimensional data structures based on quad and k-d trees*
4	70	*An algorithmic and complexity analysis of interpolation search* *Optimal merging of 2 elements with n elements* *A characterization of database graphs admitting a simple locking protocol*
5	67	*The 'Hoare logic' of concurrent programs* *General predicate transformer and the semantics [...]* *Specification and verification of database dynamics*
6	51	*The clean termination of iterative programs* *Duality in specification languages: a lattice-theoretical approach* *The clean termination of Pascal programs*

Table 2: Communities found by Louvain method.

D Recommender

D.1 Query 1

```
MERGE (comm: Community{name: $name})
WITH comm
MATCH (kw: Keyword)
WHERE kw.keyword in $keywords
MERGE (comm)-[:Related]-(kw)
```

This query creates a community (if it does not exist) and the relationships between the community and the keywords that define it.

D.2 Query 2

```
MATCH (n)-[:Holds|Publishes]-()-[:Present|Features]-(art:Article)
OPTIONAL MATCH (art:Article)-[:Has]-(kw: Keyword)
WITH n, count(distinct art) as total, sum(CASE WHEN kw.keyword in $keywords THEN 1 ELSE 0 END) as subtotal
WHERE subtotal/total >= 0.9
MATCH (comm: Community{name: $name})
MERGE (n)-[:Belongs]-(comm)
RETURN n.name as name
```

This query finds conferences and journals related to each community and creates the relationship between them.

D.3 Query 3

```
%Obtain Cypher projection

CALL gds.graph.drop('commCalc')

CALL gds.graph.create.cypher('commCalc',
MATCH (c:Community {name: '\$community'})-[:Belongs]-()-[]-()-[]-(art:Article) RETURN id(art) as id,
MATCH ((c:Community {name: '"+communityName+"'})-[:Belongs]-()-[]-()-[]-(art:Article))
MATCH (art)-[:Citation]->(art2:Article)-[]-()-[]-()-[:Belongs]-(c)
RETURN id(art) as source, id(art2) as target)
YIELD graphName, nodeCount, relationshipCount, createMillis

%Obtain pagerank from projection

CALL gds.pageRank.stream('commCalc')
YIELD nodeId, score RETURN gds.util.asNode(nodeId).doi AS doi,score ORDER BY score DESC LIMIT 100;

%Generate top100 relation

MATCH (art:Article) WHERE art.doi IN $doiList
MATCH (comm:Community {name: $commName})
MERGE (art)-[:Top100]-(comm)
```

The first part of query 3 obtains, for a single $community its projection made of only articles that belong to it, with the relation of their citations, it is done this way due to RAM limitations when performing page rank due to its high demand for RAM, performing page rank with the projection of only one community at a time.

The second part of the query performs the page rank, and finally, the third part, uses the list of doi obtained to generate a relation from each of the articles belonging to it to its community, the Top100 relation. All this is repeated for each community

D.4 Query 4

```
MATCH (aut:Author)-[:Coauthor]-(art:Article)-[:Top100]-(c:Community)
WITH aut,c,count(art) AS articles
MERGE (aut)-[:ReviewCandidate]-(c)
WITH aut,c,articles
WHERE articles>1
MERGE (aut)-[:ReviewGuru]-(c)
```

Query 4 uses the relation obtained in query 3, the Top100, to obtain all authors belonging to it, then creates the ReviewCandidate relation with their corresponding community, then, if the author has more than 1 article, then it also creates the ReviewGuru relation.