UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Lab I: Property Graph

Semantic Data Management

Fathollahi, Mohana
Iborra , Paula

March 31, 2022

A Modeling, Loading, Evolving

A.1 Modeling

In Figure 1 we created a visualization to observe the design of our graph distinguishing data instances (in blue) from metadata (schema in grey). The design decisions that we have made are as follows:

We created **nodes** for: `Article, Author, Journal, Conference, Volume, Edition, City` and `Keyword.` We have decided to separate nodes like Journal-Conference and Volume-Edition since the concept and the information they need to store and relate to the `Article` is different between them. Also we considered `Author` as a single node, and not as separated nodes for writers and reviewers since in that way we would have duplicated information of same people. We have decided to store information of conferences venue in `City` node so conferences in same city and be easily retrieved and in this way we avoid having a property venue with duplicated information among conferences. We merged the concept of workshops and conferences and we have kept them under the same label `Conference` node. We also assumed that a conference is only given once a year, and therefore we just have one edition per year for each conference. The concept Keyword has been considered as a node so in this way we can easily query articles related to similar topics rather than store this information as a property in Article.

For edges we created the following relationships: `PublishedIn, AvailableAt, BelongTo, reviews, Writes, Has, HeldIn` and `Cites`. An `Article` and be either `PublishedIn` a `Volume` or an `Edition`, which are related to `Journal` and `Conference` respectively with different edges. `Authors` are related to Articles in either `writes` or `Reviews` edges (but not both). We decided to include the information of *corresponding* author as a property of the `writes` edge, since this information depends both in the article and the author. It is not a property in `Author` since this can have written many articles and just be the corresponding in some of them, therefore the information need to be stored in the relation and not the `Author` node itself.

A.2 Instantiating/Loading

In this part, we used real data that exist in DBLP and we converted files from XML to CSV format. To retrieve information that we want to build a graph, we used these files and tried to find relation between them.

- `dblp_author_authored_by.csv`: information to relate article_id and author_id. We can find a two columns file with the author with its unique id that wrote article that has a unique id too.
- `dblp_author.csv` relates unique author id to its first name and last name.
- `dblp_ journal_ published_ in.csv` in this file we find relation between journal id and article id, that is, what articles are published in each journal.
- `dblp_ journal.csv` contains information about the name of each journal with respect to its unique id.

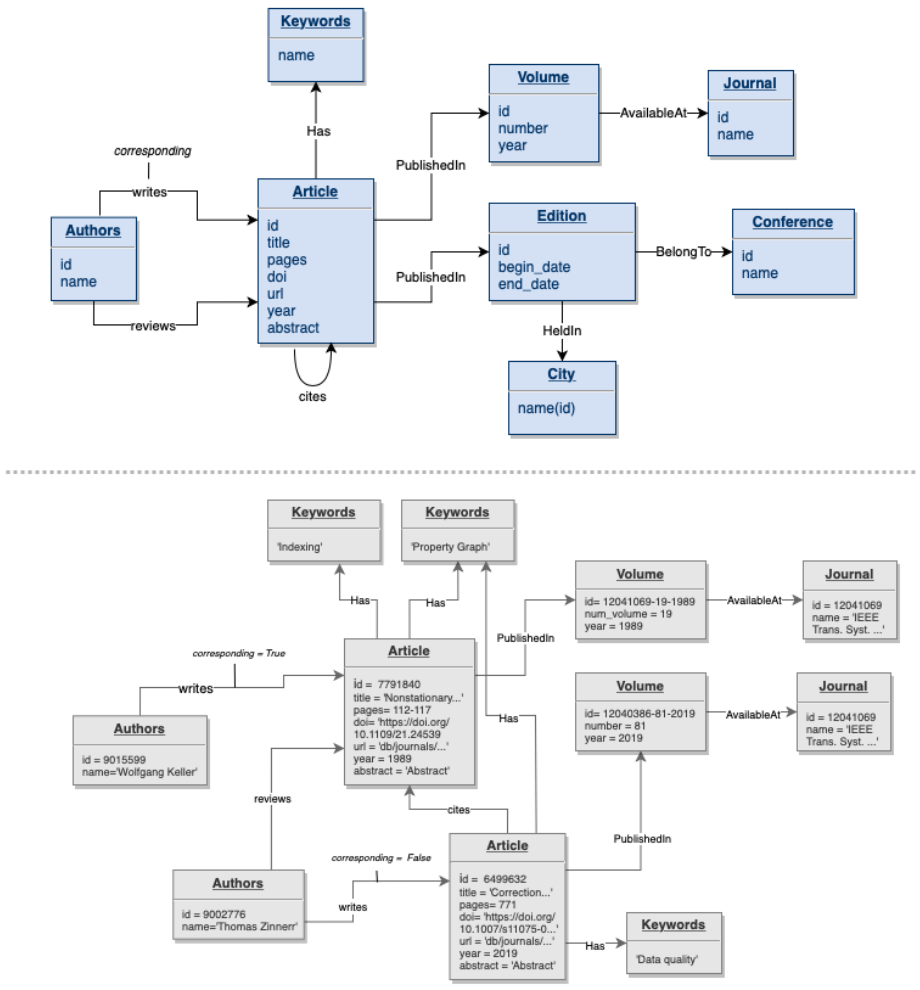

Figure 1: Visual representation of data instances (in blue) and metadata (schema in grey).

- `dblp_ article.csv` has the main information of each article. We have just kept the most relevant information of each article: article_id, title, pages, doi,url,volume and year.

Unfortunately, we could not find other necessary information relating article with conferences. Therefore, for the conference we created artificial information using some of the parameters that comes from journal. For example, we used journal id adding a 0 at the end of journal id to make up the conferences id. In this way we could maintain the relations between articles in conferences (i.e. articles belonging to the same journal id 999 will all have now an id 9990), to maintain a more realistic data-set. Therefore, we consider some journal articles as a conference articles and we could use information from `dblp_artilce` and `dblp_author`. Moreover, for conference venue, we used an external source `worldcities.csv` [5] to add city and country for each conference edition. As previously stated, we consider just one edition per year. Therefore, we created the edition id as a combination of the conference id and the year.

We had near three million articles in our raw dataset `dblp_ article.csv`. Hence, we decided to use just a sample of these data, later dividing samples equally between journal articles and conference articles. Moreover, we applied some preprocessing on samples to not have missing values in features that are important for creating graph, these properties are mentioned in `dblp_ article.csv`.

Finally, with the preprocessed files coming out after executing `preprocess.py` we load the Neo4j graph.

A.3 Evolving the graph

In 2 we can see our evolved graph model. The changed included in this section are the following ones:

- We have created nodes for the affiliations of the authors. We decided to keep affiliations organizations like companies and universities as different nodes since those instances represent very different data. Therefore we included nodes of `University` and `Company` to our graph model and we stored the relation to the `Author` nodes in a new edge `AffiliatedTo`.
- We included additional information of article's reviews as a property of `reviews` edge. We included two properties, `Comment` to store the reviewer's comments or descriptions, and `Acceptance` to store if that reviewers either accepts or rejects the paper. Therefore, this last property is a boolean of either True/False acceptance. This information has been created artificially. Since we consider that all the articles in the graph have been published already, they need to have a majority of True votes in the `Acceptance` property. We ensure that 2 out of the 3 reviewers accepts the paper.

B Querying

We have considered the number of disk accesses (db hits) to check the performance of our queries. In order to evaluate the execution steps of our queries we have used the `PROFILE`.

The following results associated with each query were obtained with 1000 articles and all its related information. The total amount of nodes in the graph was approximately 6000 with 18000 relationships.

To improve the efficiency in our queries, we have specified the labels for nodes/edges that we want to search. In that way *Cypher* just search for nodes with the defined label and avoid searching for all nodes. This increase the performance of our queries and reduce db hits [3]. However, note that for some queries the pattern matching is unique, and therefore, is not really necessary to specify the labels since there is not other path. Additionally, to reduce intermediate result we used WITH after MATCH [4].

Find the top 3 most cited papers of each conference.

```
MATCH (co:Conference)<-[]-()<-[]-(ar:Article)<-[rc:Cites]-()
WITH ar.id as article_id, count(rc) as topcitation,co.id as conference_id
WITH collect(article_id) as all_articles, topcitation,conference_id
RETURN conference_id,all_articles [..3],topcitation
ORDER BY conference_id
```

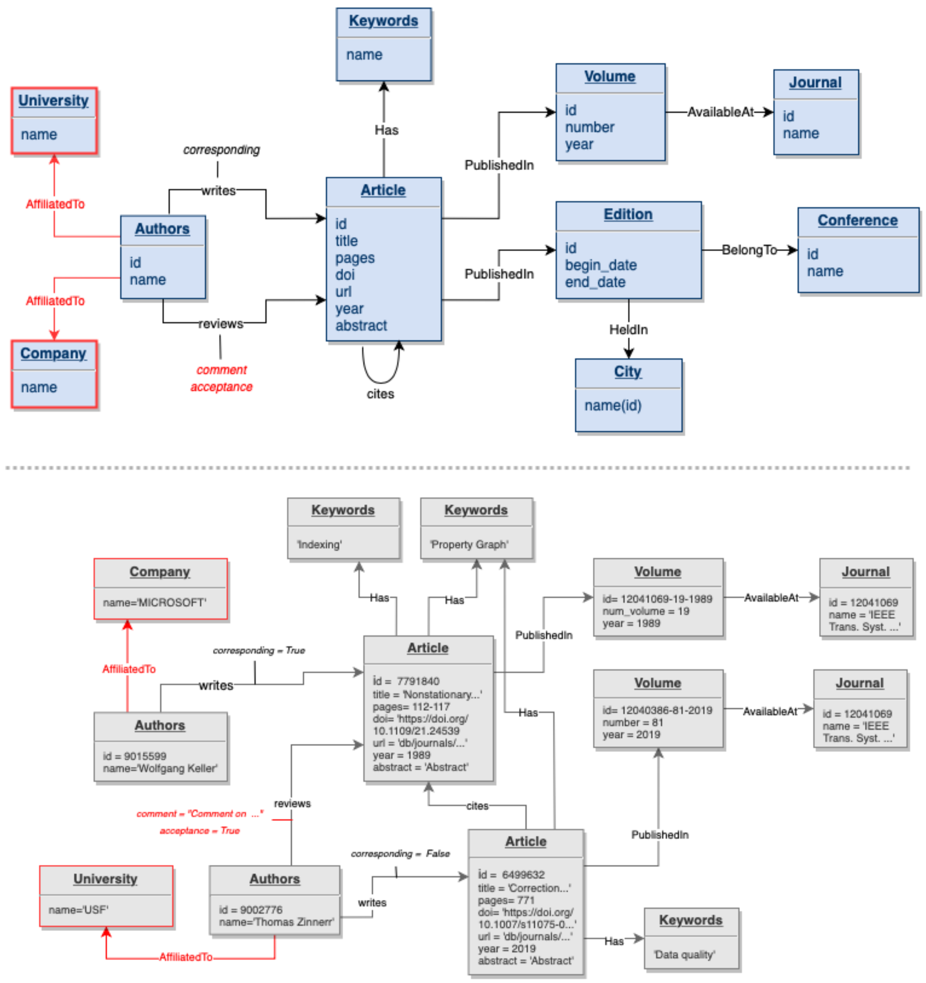

Figure 2: Visual representation of data instances (in blue) and metadata (schema in grey) with the changes introduced highlighted in red.

For each conference find its community.

```
MATCH (co:Conference)<-[]-(ed:Edition)<-[]-(:Article)<-[:writes]-(at:Author)
WITH co.id AS conference_id , at.id as author_id,COUNT(ed) AS editions
WHERE editions >= 4
RETURN conference_id , collect(author_id),editions
order by conference_id
```

Find the impact factors of the journals in your graph.

```
MATCH (j:Journal)<-[:AvailableAt]-(:Volume)<-[:PublishedIn]-
(:Article)<-[c:Cites]-(ar:Article)
WHERE ar.year = 2021
WITH j, count(c) AS citations
MATCH (j:Journal)<-[:AvailableAt]-(v:Volume)<-[:PublishedIn]-(a:Article)
WHERE v.year in [2019,2020]
WITH j,citations,count(a) as last_articles
RETURN j.id,1*(citations)/last_articles AS impact
ORDER BY j.id
```

Find the h-indexes of the authors in your graph.

```
MATCH (at:Author)-[:writes]->()<-[ci:Cites]-()
WITH at.id AS author_id, COUNT(ci) AS citation
ORDER BY author_id, citation DESC
WITH author_id , COLLECT(citation) AS all_citation
RETURN author_id ,SIZE([x IN RANGE(1,SIZE(all_citation))
WHERE x <= all_citation[x-1]]) AS h_index
ORDER BY h_index DESC , author_id
```

C Graph algorithms

The first algorithm we considered is *PageRank* to fins the most important or relevant articles in our graph. The *PageRank* algorithm calculated the importance of each node within the graph, based on the number incoming relationships and the importance of the corresponding source nodes. The underlying assumption is that a page is only as important as the pages that link to it [2].

We decided to use this algorithm since we considered that it keeps the subjective idea of article's importance better than other centrality algorithms that just consider the incoming/outgoing edges. *PageRank* extends this idea to articles by not counting citations from all articles equally, but normalizing them by the number of citations on the citing article [1].

The final query that we used is the following one:

```
CALL gds.graph.create('ArticleCitations','Article','Cites');
CALL gds.pageRank.stream('ArticleCitations',
{maxIterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).title as title, score
ORDER BY score DESC;
```

The second algorithm that we have considered is the *Louvain* algorithm. This hierarchical clustering algorithm aims to find clusters by modularity, which consists in uncovering communities by partitioning a graph into smallest modules and then measuring the strength of the groupings, whose quality is measured by the modularity [2]. We used *Louvain* to get author communities. Authors that work more together belong to the same community and may have similar lines of research.

```
CALL gds.graph.create('AuthorCommunity', ['Author','Article'], 'writes');
CALL gds.louvain.stream('AuthorCommunity', {maxLevels: 20, maxIterations:20})
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
ORDER BY name ASC;
```

D Recommender

D1. Define the research communities.

```
CREATE (co:Community {name: 'community_db'})
WITH co
MATCH (k:Keyword)
```

```
WHERE k.name IN ["Data management", "Indexing", "Data modeling", "Big data",
"Data processing", "Data storage", "Data querying"]
MERGE (k)-[:IS_in]->(co)
RETURN co,k
```

D2. Find the conferences and journals related to the database community.

```
MATCH (j)<-[:AvailableAt]-()<-[:PublishedIn]-(a)
OPTIONAL MATCH (a)-[:Has]->(k)-[:IS_in]->(co)
WITH j, a.id AS article,
COUNT(k) > 0 AS keywords
WITH j, COLLECT(keywords) AS all_article,article
WHERE 1*(SIZE([x IN all_article WHERE x=true]))/SIZE(all_article) >= 0.9
WITH j
MATCH (co:Community)
WHERE co.name ='community_db'
MERGE (j)-[:Connect]->(co)

MATCH (con)<-[:BelongTo]-()<-[:PublishedIn]-(ar)
OPTIONAL MATCH (ar)-[:Has]->(k)-[:IS_in]->(co) WITH con, ar.id AS article,
COUNT(k) > 0 AS keywords
WITH con, COLLECT(keywords) AS all_article,article
WHERE 1*(SIZE([x IN all_article WHERE x=true]))/SIZE(all_article) >= 0.9
WITH con
MATCH (co:Community)
WHERE co.name ='community_db'
MERGE (con)-[:Connect]->(co)
```

D3. Identify the top 100 papers of these conferences/journals.

```
CALL gds.graph.create.cypher('community_graph',
    'MATCH (co:Community {name: "community_db"})<-
    [:Connect]-()<-[]-()<-[:PublishedIn]-(ar:Article)
     RETURN id(ar) AS id',
    'MATCH (ar:Article)-[:Cites]->(top_ar:Article)
     RETURN id(ar) AS source, id(top_ar) AS target',
    {validateRelationships: FALSE});
CALL gds.pageRank.stream('community_graph')
YIELD nodeId, score
WITH gds.util.asNode(nodeId) as ar, score
ORDER BY score DESC
LIMIT 100
MATCH (c:Community {name: 'community_db'})
CREATE (ar)-[:isTopRated]->(c)
```

D4. Author of any of these top-100 papers considered a potential good match to review database papers, and gurus.

```
MATCH (co:Community {name: "community_db"})<-[:isTopRated]-(ar:Article)
<-[:writes]-(at:Author)
WITH co,at , SIZE(COLLECT(ar.id)) AS articles
MERGE (at)-[:isPotentialReviewer]->(co) WITH co, articles
WHERE articles > 1
MERGE (at)-[:isGuru]->(co)
```

References

[1] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine". In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[2] *Neo4j Graph Data Platform. 2022. The Neo4j Graph Data Science Library Manual v2.0 - Neo4j Graph Data Science.* URL: https://neo4j.com/docs/graph-data-science/current/ (visited on 03/27/2022).

[3] *Query optimization.* URL: https://neo4j.com/docs/cypher-manual/current/query-tuning/basic-example/.

[4] *Query$_o$ptimization.* URL: https://medium.com/neo4j/cypher-query-optimisations-fe0539ce2e5c.

[5] *simplemaps - World Cities Database.* URL: https://simplemaps.com/data/world-cities (visited on 03/30/2022).