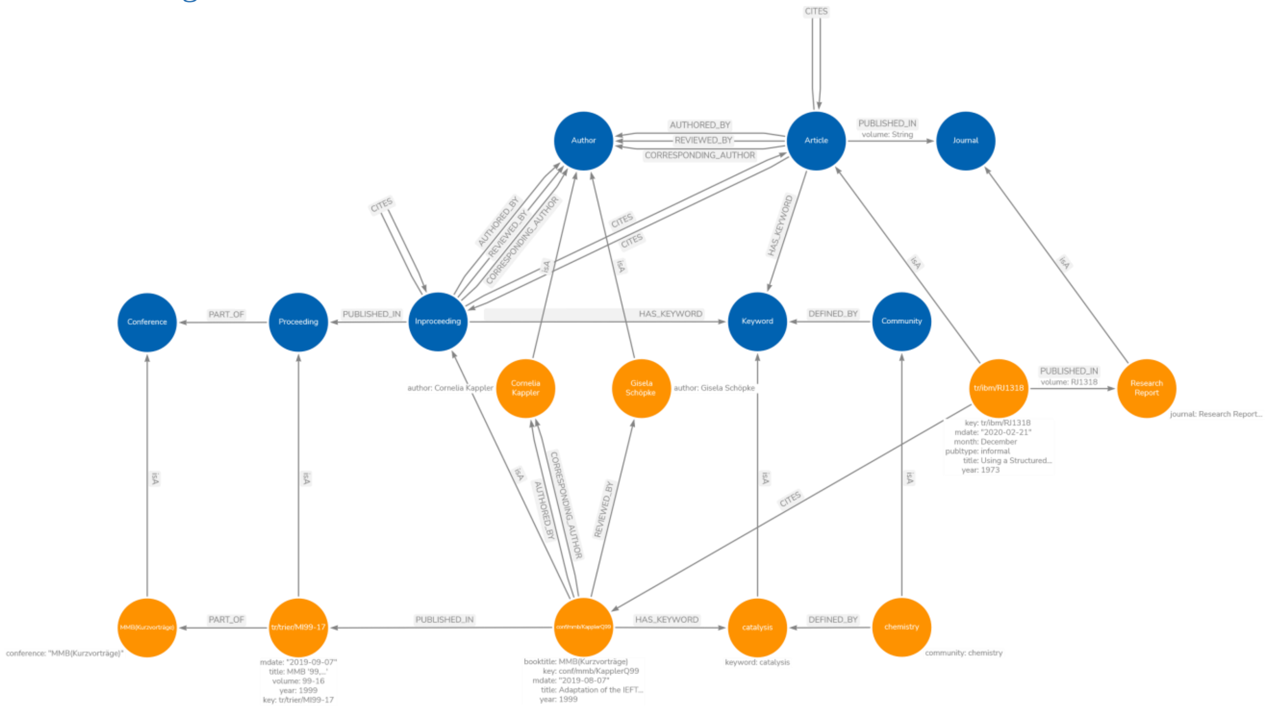
LAB Assignment: Property Graphs

Team members: Bogdana Živković, Nikola Ivanović

A Modeling, Loading, Evolving

A.1 Modeling



Firstly, it should be noted that the presented model heavily corresponds to the dblp dataset, which was used to instantiate the graph, while trying to fulfill the specified requirements. Regarding this, there are several design choices that we think should be justified. The first of these is the choice to include both nodes of type *Article* and *Inproceeding* to model research papers, in order to preserve all the semantics from the dataset (we realize that all scientific papers could have been modeled with a single type). Furthermore, *Articles* relate to *Journals* with a *PUBLISHED_IN* edge, with the journal *volume* as an edge property. This decision was made due to many records of dblp dataset not containing any information about the volume. While we realize that volumes could have been separate nodes (with nonexistent volumes in the dataset being represented as volume 1), we decided to model them as properties, because they do not hold any semantic value without their corresponding journals. On the other hand, *Proceedings* (Conference editions), are modeled as nodes, because they hold more information in the dblp dataset. Thus, *Inproceedings* relate to *Proceedings*, which relate to *Conferences*. Both *Articles* and *Inproceedings* relate to their *Authors* with *AUTHERED_BY* edges, with one additional *CORRESPONDING_AUTHOR* edge per node. Although this approach introduces redundancy, we estimate it to be minor. *Citations* and *reviews* are simply modeled as edges in the graph. Finally, *Keywords* are present as separate nodes and relate to their appropriate communities.

A.2 Instantiating/Loading

Firstly, unique key constraints were created for every node type. In Neo4j, creating a unique constraint also implicitly creates an index.

```
CREATE CONSTRAINT articleKeyConstraint IF NOT EXISTS
FOR (article:Article)
REQUIRE article.key IS UNIQUE
```

As previously stated, we used the dblp dataset acquired from https://dblp.uni-trier.de/ to instantiate most of the elements in the graph. However, the information that is not present in the dataset (e.g. citations, reviews, keywords…)was generated synthetically or randomly. Conversion of dblp data from XML to CSV format was done using the following tool https://github.com/ThomHurks/dblp-to-csv. Considering that this project was intended for exercise, we loaded only a subset of the whole dataset. When testing our application, we loaded 1000 articles, their appropriate authors and journals, and 500 proceedings with the corresponding inproceedings, authors and conferences. Taking into account that the information about corresponding authors was not in the dataset, we assigned this role to the first author in the list.

```
LOAD CSV WITH HEADERS FROM 'file:///output_article.csv' AS line
FIELDTERMINATOR ';'
WITH line LIMIT %d
CALL {
    WITH line
    MERGE (j:Journal {journal: line.journal})
    CREATE (a:Article {mdate: date(line.mdate), key: line.key,
publtype: line.publtype,
        title: line.title, month: line.month, year:
toInteger(line.year)})
    CREATE (a)-[:PUBLISHED_IN {volume: line.volume}]->(j)
    WITH line, a
    UNWIND split(line.author, '|') AS authors
    MERGE (author:Author {author: authors})
    CREATE (a)-[:AUTHORED_BY]->(author)
    WITH author, a, line
    WHERE author.author = split(line.author, '|')[0]
    CREATE (a)-[:CORRESPONDING_AUTHOR]->(author)
}
IN TRANSACTIONS
```

Reviews and citations were generated randomly in a very similar manner, using the **apoc** library. We decided to randomly generate exactly three reviews for every *Article*/*Inproceeding* as a *REVIEWED_BY* edge. On the other hand, the number of citations per article was randomly generated.

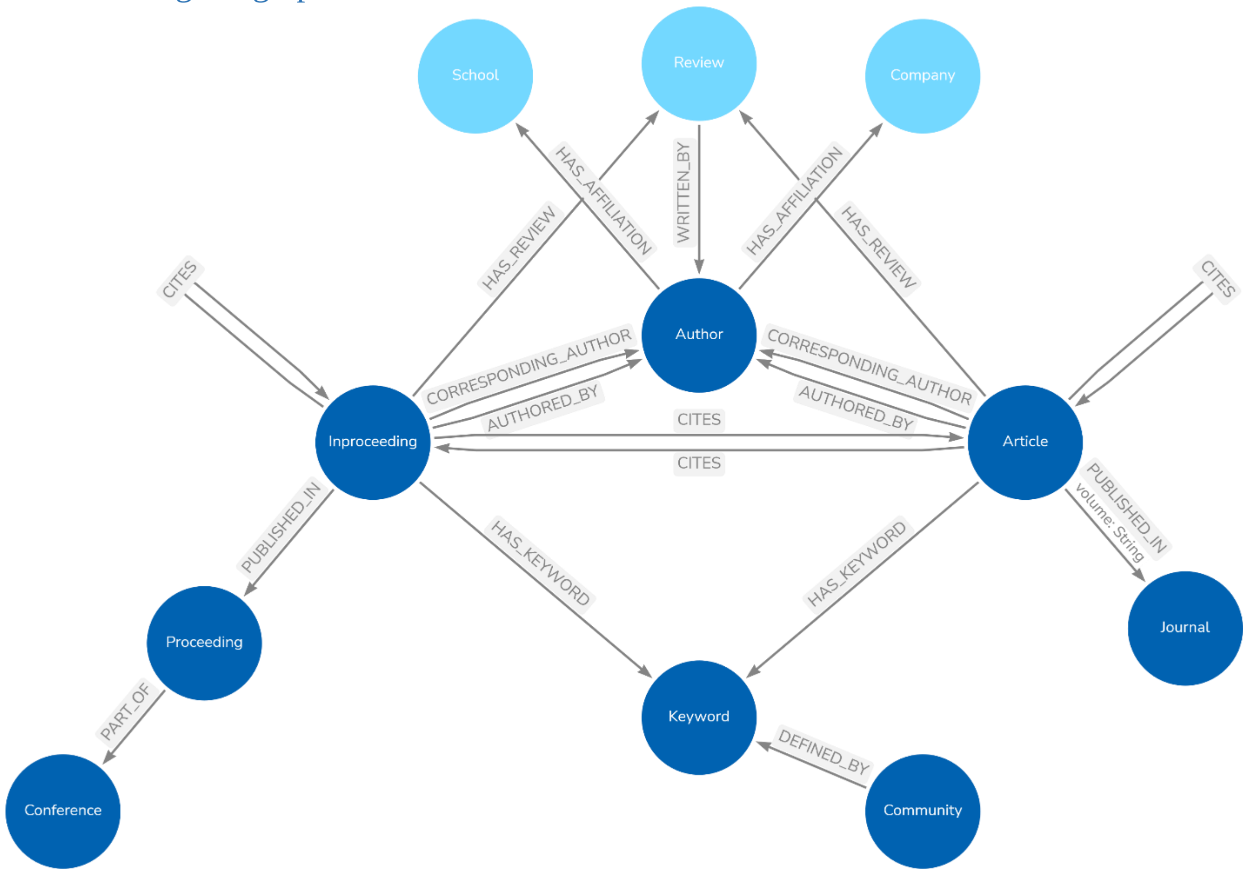
```
MATCH (p1:Article|Inproceeding)
CALL {
    WITH p1
    MATCH (p2:Article|Inproceeding) WHERE p1.key <> p2.key
    WITH p1, apoc.coll.randomItems(COLLECT(p2),
apoc.coll.randomItem(range(0, 10))) as papers
    FOREACH (paper IN papers | CREATE (p1)-[:CITES]->(paper))
}
IN TRANSACTIONS
```

Finally, concerning the keywords, we have compiled an example csv file where each row has a keyword and a corresponding community.

```
database,data management
database,indexing
database,data modeling
```

From this document, both *Keywords* and *Communities* were loaded into the graph. Following that, a random number of keywords was randomly related to every *Article*/*Inproceeding* in a similar way as citations.

A.3 Evolving the graph



First modification to the graph that we suggest is adding a node representing an individual *Review*. This node should have its appropriate properties: *content* and *accepted*. This modification was very simple. The *accepted* property describes a suggested decision and is randomly assigned value of 0 or 1.

```
MATCH (p)-[r:REVIEWED_BY]->(a)
DELETE r
CREATE (p)-[:HAS_REVIEW]->(:Review {content: 'Review content',
accepted: apoc.coll.randomItem(range(0, 1))})-[:WRITTEN_BY]-
>(a)
```

Next, we added *accepted* property to all *Articles* and *Inproceedings*, where we stored the final decision for the research paper. Although this added information is already present in the graph, we considered it a better option than counting the votes every time we need information.

```
MATCH (p:Article|Inproceeding)
WITH p, CASE
    WHEN COUNT {(p)-[:HAS_REVIEW]->(:Review {accepted: 1})} > 1
THEN 1
    ELSE 0
END AS accepted
SET p.accepted = accepted
```

In the end, in order to create authors' affiliations, we first had to load universities and companies. For this purpose, we loaded schools and publishers from dblp dataset and assigned them randomly to each author similarly to how we assigned citations.

In the graph the edge *REVIEWED_BY* has been replaced with *Review* nodes and *HAS_REVIEW* and *WRITTEN_BY* edges as can be seen on the updated version of graph metadata. Two new nodes, *School* and *Company*, have also been added to the graph metadata, along with their corresponding HAS_AFFILIATION edges.

B Querying

In this section, we will give a brief descriptions of the queries and present the cypher code to express them.

Query 1

Find the top 3 most cited papers of each conference.

```
MATCH (p:Inproceeding|Article)-[c:CITES]->(i:Inproceeding)-
[:PUBLISHED_IN]->(pr:Proceeding)-[:PART_OF]->(conf:Conference)
WITH i, conf, COUNT(c) AS cnt
ORDER BY COUNT(c) DESC
WITH conf.conference AS conferences, collect(i.title) AS
inproceeding, collect(cnt) AS counts
RETURN conferences, inproceeding[0..3] AS inproceedings,
counts[0..3] as number
ORDER BY conferences
```

Query 2

For each conference find its community: i.e., those authors that have published papers on that conference in, at least, 4 different editions.

```
MATCH (c:Conference)<-[:PART_OF]-(p:Proceeding)<-
[:PUBLISHED_IN]-(:Inproceeding)-[:AUTHORED_BY]-(a:Author)
WITH c.conference AS conference, a.author AS author,
COUNT(DISTINCT p) AS cnt
WHERE cnt > 3
MATCH (c:Conference)
OPTIONAL MATCH (c{conference:conference})<-[:PART_OF]-
(p:Proceeding)<-[:PUBLISHED_IN]-(:Inproceeding)-[:AUTHORED_BY]-
(a:Author{author:author})
RETURN c.conference AS conference, COLLECT (DISTINCT a.author)
AS community
ORDER BY SIZE(community) DESC
```

Query 3

Find the impact factors of the journals in your graph.

```
MATCH (p:Inproceeding|Article)-[c:CITES]->(i:Inproceeding)-
[:PUBLISHED_IN]->(pr:Proceeding)-[:PART_OF]->(conf:Conference)
WITH i, conf, COUNT(c) AS cnt
ORDER BY COUNT(c) DESC
WITH conf.conference AS conferences, collect(i.title) AS
inproceeding, collect(cnt) AS counts
RETURN conferences, inproceeding[0..3] AS inproceedings,
counts[0..3] as number
ORDER BY conferences
```

Query 4

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

```
MATCH ()<-[c:CITES]-(p)-[:AUTHORED_BY]->(a:Author)
WITH a.author AS author, p.title AS paper, COUNT(c) AS
citations
ORDER BY author, citations DESC
WITH author, COLLECT(citations) AS cite_list
WITH author, cite_list, [x IN range (0, SIZE(cite_list)-1)
WHERE x < cite_list[x]] AS indexes
RETURN author, SIZE(indexes) AS h_index
```

C Recommender

Considering that keyword data was not present in the dblp dataset, we had to syntethically generate it. Thus, research communities and the keywords that define them were already loaded, as described in the section A.2. Next, conferences and journals were related to communities if 90% of their papers contained one of the keywords that defined those communities. This was performed for all communities (not only the database community) as we require this information for further analysis that will be presented in part D.

```
MATCH (j:Journal)<-[:PUBLISHED_IN]-(a:Article)-[:HAS_KEYWORD]-
>(k:Keyword)<-[:DEFINED_BY]-(c:Community)
WITH j, c, COUNT(DISTINCT a) AS cnt
MATCH (j)<-[:PUBLISHED_IN]-(a:Article)
WITH j, c, cnt, COUNT(a) AS total
WHERE cnt*1.0/total >= 0.9
CREATE (j)-[:BELONGS_TO]->(c)

MATCH (conf:Conference)<-[:PART_OF]-(p:Proceeding)<-
[:PUBLISHED_IN]-(i:Inproceeding)-[:HAS_KEYWORD]->(k:Keyword)<-
[:DEFINED_BY]-(c:Community)
WITH conf, c, COUNT(DISTINCT i) AS cnt
MATCH (conf)<-[:PART_OF]-(p:Proceeding)<-[:PUBLISHED_IN]-
(i:Inproceeding)
WITH conf, c, cnt, COUNT(i) AS total
WHERE cnt*1.0/total >= 0.9
CREATE (conf)-[:BELONGS_TO]->(c)
```

In the following steps, we first generated a subgraph that contained all papers published in journals/conferences that belong to the database community and the citations that exist between them.

```
MATCH p=(n1)-[:CITES]->(n2)
WHERE (EXISTS {
    MATCH (:Community{community:"database"})<-[:BELONGS_TO]-
(:Journal)<-[:PUBLISHED_IN]-(n1)
} OR EXISTS {
    MATCH (:Community{community:"database"})<-[:BELONGS_TO]-
(:Conference)<-[:PART_OF]-(:Proceeding)<-[:PUBLISHED_IN]-(n1)
})
AND
(EXISTS {
    MATCH (:Community{community:"database"})<-[:BELONGS_TO]-
(:Journal)<-[:PUBLISHED_IN]-(n2)
} OR EXISTS {
    MATCH (:Community{community:"database"})<-[:BELONGS_TO]-
(:Conference)<-[:PART_OF]-(:Proceeding)<-[:PUBLISHED_IN]-(n2)
})
RETURN gds.alpha.graph.project('papers', n1, n2)
```

Next, we called gds.pageRank on the subgraph and created a *TOP_PAPER* egde connecting the papers with top 100 highest page rank scores.

```
CALL gds.pageRank.stream(
    'papers'
)
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS paper, score
ORDER BY score DESC, paper ASC
LIMIT 100
MATCH (c:Community{community:'database'})
CREATE (paper)-[:TOP_PAPER]->(c)
```

From there on, the process of finding the authors of the top papers was straightforward, as well as finding gurus and connecting them to the database community node.

```
MATCH (a:Author)<-[:AUTHORED_BY]-(p)-[r:TOP_PAPER]-
>(c:Community{community:'database'})
MERGE (a)-[:GOOD_MATCH]->(c)
```

```
MATCH (a:Author)<-[:AUTHORED_BY]-(p)-[r:TOP_PAPER]-
>(c:Community{community:'database'})
WITH a, c, COUNT(p) AS cnt
WHERE cnt > 1
CREATE (a)-[:GURU]->(c)
```

D Graph algorithms

In this section, we will illustrate how graph algorithms can be applied to this particular graph. The first example shows how the betweenness centrality algorithm can be used to determine the interdisciplinarity of journals in the dataset. As we know, betweenness centrality indicates which nodes have a high influence on the flow of information in the graph. In this case, interdisciplinarity refers to the number of communities a certain journal can be related to. The first step of this procedure is connecting journals to their appropriate communities, as was done in section C. Next, we create a subgraph that contains all articles in the dataset and only citations between separate journals that belong to different communities.

```
MATCH p=(j1:Journal)<-[:PUBLISHED_IN]-(a1:Article)-[:CITES]-
>(a2:Article)-[:PUBLISHED_IN]->(j2:Journal)
WHERE j1 <> j2 AND NOT EXISTS {
    MATCH (j1)-[:BELONGS_TO]->(:Community)<-[:BELONGS_TO]-(j2)
}
RETURN gds.alpha.graph.project('subgraph', a1, a2)
```

Lastly, we call the gds.betweenness algorithm to assess the scores of articles in the subgraph. This score represents the measure of the article participating in citation chains between different communities. Once these scores are obtained, we derive the corresponding journal's betweenness score as the average of the scores of all of its articles.

```
CALL gds.betweenness.stream(
    'subgraph'
)
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS article, score
MATCH (article)-[:PUBLISHED_IN]->(j:Journal)
RETURN j.journal AS journal, AVG(score) AS average
ORDER BY average DESC, journal ASC
```

The second algorithm that we've decided to implement is the node similarity algorithm, for determining similar authors in order to recommend future collaborations between them. We assumed that two authors were more similar if they have published research papers for the same journals/conferences. To achieve this, we first created new relations between authors and journals/conferences that their research papers were published in.

```
MATCH p=(a:Author)<-[:AUTHORED_BY]-(:Article)-[:PUBLISHED_IN]-
>(j:Journal)
MERGE (a)-[:WRITES_FOR]->(j)

MATCH p=(a:Author)<-[:AUTHORED_BY]-(:Inproceeding)-
[:PUBLISHED_IN]->(:Proceeding)-[:PART_OF]->(c:Conference)
MERGE (a)-[:WRITES_FOR]->(c)
```

Then, we created a subgraph that contains authors, journals/conferences and newly created *WRITES_FOR* relations.

```
MATCH (a:Author)-[:WRITES_FOR]->(p:Journal|Conference)
RETURN gds.alpha.graph.project('writesFor', a, p)
```

Finally, we invoke gds.nodeSimilarity algorithm that calculates a similarity score between two every pair of authors from the dataset. This result is addionally be sorted to indicate the best matches.

```
CALL gds.nodeSimilarity.stream('writesFor')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).author AS Author1,
gds.util.asNode(node2).author AS Author2, similarity
ORDER BY similarity DESCENDING, Author1, Author2
```