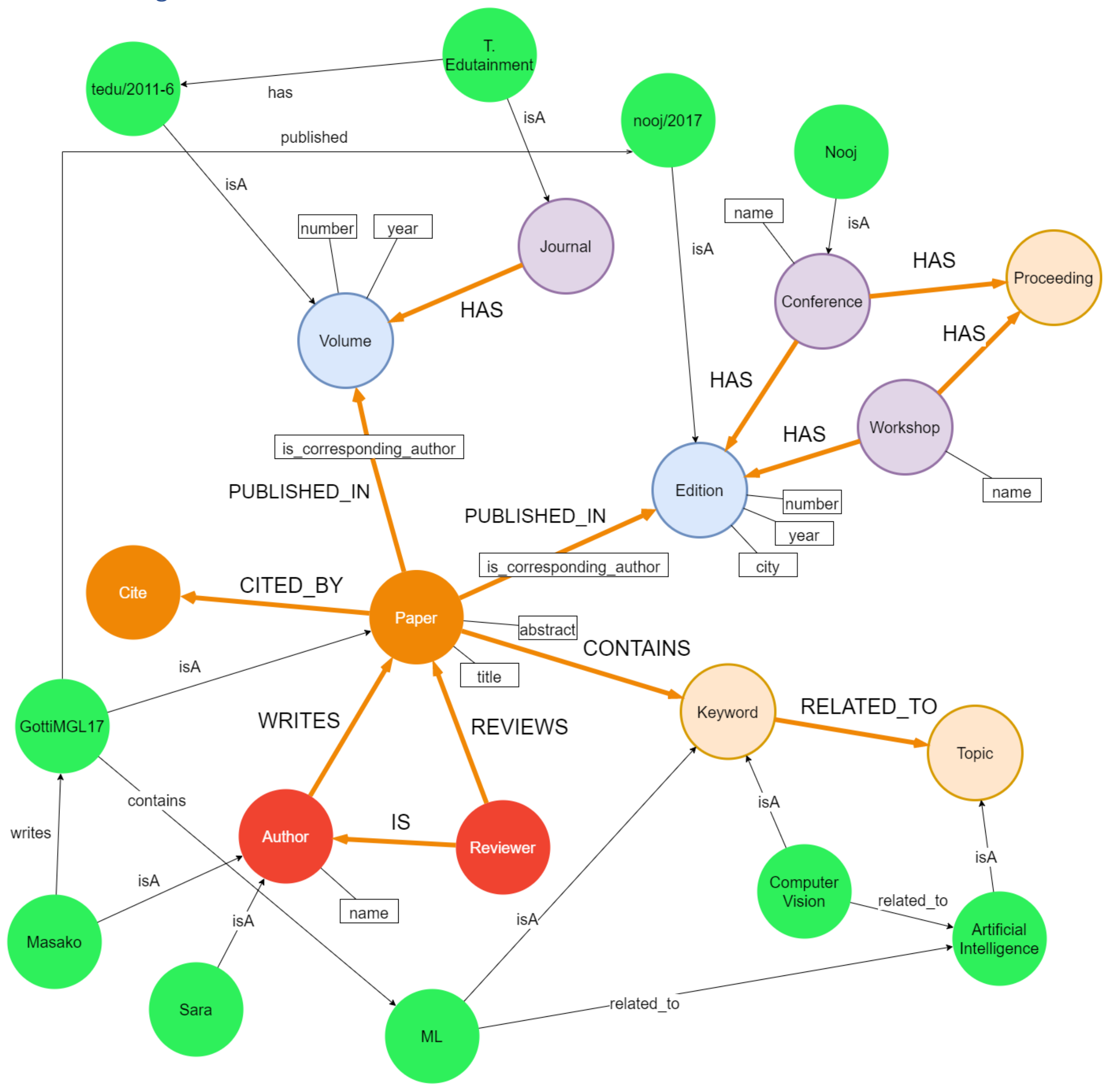
Semantic Data Management

Property Graph Lab

Team members
Ali Arous
Dimitrios Tsesmelis

A. Modeling, Loading, Evolving

A1. Modeling



In the above graph, the green nodes represent the data instances, while all the others ones represent the meta-data. We have visualized all the meta-data -representing our graph model- along with a few data instances so that the whole figure becomes easier to read. For the same sake of simplicity, we have not included all the node attributes that we used while loading the real data. However, such attributes can be found in the loading scripts in *Part B*.

Regarding our design decisions, we tried to design the model in such a way that the queries in *Part B* are well optimised. More specifically, we made the following decisions:

- The Edition and the Volume are modeled as different nodes from the Conference and the Journal, respectively. We made this choice as most of the queries are related to Conferences (e.g. Find the top 3 most cited papers of each conference.). This means that we do not want to have the Edition as an attribute of the node Conference, as this would affect the efficiency (we would need extra I/Os).
- Similarly, Workshop and Conference are different nodes. If we had both in one node, we would need an extra attribute isConference (Boolean) and in this case, we would have to to look up in this attribute to specify whether the node is a Conference or a Workshop.
- The City of the Edition of the Conference is an attribute of the Edition node, as we assume that no further analysis will be done based on the City. If we had queries that engaged the City, it would be better to have it as a node.
- A Paper may have many Keywords and a Keyword may be related to many Topics and vice versa. That is why Keyword and Topic are 2 separate nodes.
- Regarding the Citations, initially we had a self referencing edge from Paper to Paper. However, while loading the data, we realized that the dataset did not provide the information of which Paper cited which another. The only information available was that a Paper is cited. That is why we have a separated node for Cite representing a citation incident, which is linked to the Paper.

A.2. Instantiating/Loading

Since there was a real dataset available (i.e. DBLP), we decided to use it for our data loading task, in order to keep our data as realistic as possible. As DBLP publishes its data in the form of an XML file with a corresponding DTD, we downloaded the latest one and extracted, transformed and modified the data it contains into CSV files spanning most of our data needs for the proposed model. We used the python script at the following GitHub url for this purpose:
https://github.com/ThomHurks/dblp-to-csv/blob/master/XMLToCSV.py

The output of running the previous script was a Folder containing 28 CSV files, some of them representing content data, while others were used solely for storing headers for other CSV files.

Subsequently, and for each needed file, we wrote a python script that extracts the attributes we are interested in for our model, and generates another intermediate CSV file ready to be bulk loaded into neo4j. The bulk loading scripts were written in conformance to the schema of the intermediate CSV files generated by our Python scripts. This approach helped us keeping the Cypher loading scripts as clean as possible. Following are the remarks on CSV files from DBLP dataset that were used for data extraction:

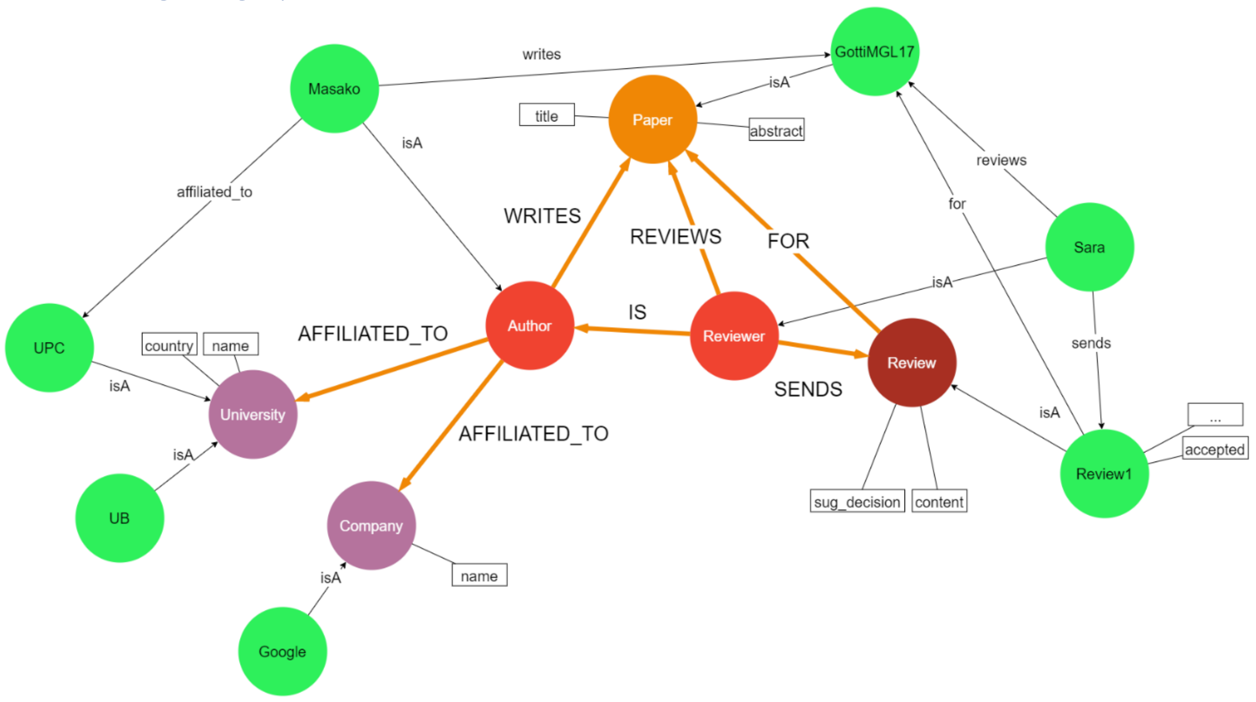
- **output_proceedings.csv:** From this original file we extracted *Conference.csv (2500 records), Edition.csv (10000 records), Workshop.csv* and *Proceedings.csv* files. The relationship between Edition and Conference was created on the fly following each Edition node's creation.
- **output_inproceedings.csv:** From this original file along with its header file, we extracted *Paper(Edition).csv (10000 records)* representing conference papers and their relationships to editions and *Edition_Paper_Author.csv*. We also extracted part of *Authors.csv (25000 records)* from this file, spanning the authors who wrote conference papers.
- **output_article.csv:** From this original file along with its header file, we extracted *Journal.csv (20 records), Volume(Journal).csv (1800 records), Paper(Volume).csv (1800 records)* and part of *Authors.csv.* Taking all this information from this file, helped us store only the journals and its

volumes that have published papers, which was more meaningful to our model for the next tasks.

As for the missing data from DBLP, we generated the following files:

- **Cite.csv:** this file contains 45000 artificial citations generated randomly with a Python script. The citation contains only its own identifier, and a key to the paper it cites. As explained in part A, we care only about how many times a certain paper was cites, with no need to keep the source paper.
- **Keyword.csv:** this file contains keywords for 10000 papers, storing two most important keywords for each paper. The assignment of keywords to papers was done randomly in a Python script.
- **Paper_reviewer.csv:** This file contains the assignment of 1000 reviewers (selected randomly from Authors.csv) to 3000 papers. The assignment rule is that no paper can be assigned more than 3 reviewers, and a reviewer can not be assigned a paper he already authored. This was all achieved in a Python script.
  The used dataset can be found at the following url:
  https://drive.google.com/open?id=1fStYnk5BXQJvGaqZBw-y84i6GgClkIAw

A.3 Evolving the graph



Regarding our modeling decision for the evolved graph, we can make the following notes:

- We decided to separate the Author and Reviewer nodes, as they have different semantics. However, we could also have had one node for both, that would have different edges to Paper (WRITES and REVIEWS).

- The Review is connected both to Reviewer and Paper. A prior initial proposition was to connect the Review only to its Reviewer, but in this way, we would not be able to identify which Review is for which Paper, as a Reviewer may review more than one Paper.

B. Querying

All the following queries (Part B, C and D) can be also found inside our Application.py file.

Query1: Find the h-indexes of the authors in your graph

```
MATCH(a:Author)-[:WRITES]->(p:Paper)-[:CITED_BY]->(c:Citation)
WITH a as authors, p.key as papers, count(c) as number_of_citations
ORDER BY number_of_citations DESC
WITH authors as authors, collect(number_of_citations) as citations_list
WITH authors as authors, citations_list AS citations_list
UNWIND range(0,size(citations_list)-1) as l_index
WITH authors as authors,
CASE
        WHEN citations_list[l_index] >= l_index+1 THEN l_index+1
    ELSE -1
END AS hindex
WHERE hindex <> -1
RETURN authors.name, max(hindex)
```

Query2: Find the top 3 most cited papers of each conference.

```
MATCH(c:Conference)-[:HAS]-(e:Edition)<-[:PUBLISHED_IN]-(p:Paper)-[:CITED_BY]-(t:Citation)
WITH c as c, p as p, count(t) as cites
ORDER BY c.name ASC, cites DESC
RETURN c.name as Conference, collect(p.title)[..3] as Most3CitedPapers
```

Query3: 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)-[:HAS]->(e:Edition)<-[:PUBLISHED_IN]-(p:Paper)<-[w:WRITES]-(a:Author)
WITH c as conference, a as author, count(distinct e) as number_of_editions
WHERE number_of_editions >= 4
RETURN conference.name as Conference, collect(author.name) as Community
```

Query 4: Find the impact factors of the journals in your graph

```
Match(j:Journal)
OPTIONAL MATCH (j)-[:HAS]->(v:Volume)
OPTIONAL MATCH (v)<-[:PUBLISHED_IN]-(p:Paper)
OPTIONAL MATCH (p)-[:CITED_BY]->(c:Citation)
WHERE p.year >= '2018' and p.year <= '2019'
RETURN j.name as Journal,  count(DISTINCT p) as Papers18_19, count(c) as Citations2019, count(c)*1.0/count(DISTINCT p) as
IMPACT_FACTOR
ORDER BY IMPACT_FACTOR DESC
```

C. Graph algorithms

Page Rank

```
CALL algo.pageRank.stream(
  'MATCH (n:Paper) RETURN id(n) AS id UNION MATCH (n:Citation) RETURN id(n) AS id',
  'MATCH (n:Paper)-[:CITED_BY]->(m:Citation) RETURN id(m) AS source, id(n) AS target',
  {graph: 'cypher'})
YIELD nodeId, score RETURN algo.asNode(nodeId).key AS page,score ORDER BY score DESC
```

The Page Rank algorithm measures the transitive influence or connectivity of nodes. This means that it measures the importance of a given node, by taking the importance of pointing nodes to it. In our case, we use this algorithm to calculate the importance of the Papers, which is determined by the Citations each of them has. As we explained earlier in the modeling, we modeled the Citation as a separate node and not as a self reference of the Paper node. This affects the result of this algorithm, since Page Rank considers the rank of the Papers that cite other Papers. This means that it works recursively which in our case is impossible as the Citations are not Papers and hence, they are not Cited by other nodes. This also leads to the fact that all the citation nodes contribute with the same importance of 1. Consequently, the calculations of this algorithm in our case is simplified as: The more the Cites a Paper has, the higher its Page Rank value will be.

Betweenness Centrality

```
CALL algo.betweenness.stream(
'MATCH (p) WHERE ANY(lbl IN ["Paper", "Keyword", "Topic"] WHERE lbl IN LABELS(p)) RETURN id(p) as id',
'MATCH (p1)-[r]->(p2) WHERE TYPE(r) IN ["RELATED_TO", "CONTAINS"] RETURN id(p1) as source,id(p2) as
target',  {concurrency:4, graph:'cypher'})
YIELD nodeId, centrality
WITH nodeId AS nodeId, centrality AS centrality
MATCH (k:Keyword) WHERE id(k) = nodeId
RETURN algo.asNode(nodeId).name, centrality ORDER BY centrality desc
```

We use the Betweenness centrality algorithm to identify Keyword nodes that behave as a bridge from one part of the graph to another. More specifically, we are trying to find Topics that are related to Keywords, in such a way that if we remove a very high centrality Keyword from the graph, then we will lose the connection between Papers and some Topics, that were connected before via this Keyword and hence, this Keyword was a bridge between these Papers and these Topics. This information can be useful in our application as we can find Topics that are linked to Papers only via a specific path (Keyword). After that, we can try to find in their content other Keywords that might be related to the same Topic in order to link with these as well. For instance, in our database, we created the Keyword {Machine learning} and 3 Topics that are related only to this Keyword. The betweenness centrality of this Keyword is 3, because if we remove it, we will end up with 3 Topics that are not connected to the rest of the graph.

Another application:

```
CALL algo.betweenness.stream(
'MATCH (p) WHERE ANY(lbl IN ["Paper", "Author", "Reviewer", "Review"] WHERE lbl IN LABELS(p)) RETURN id(p) as id',
'MATCH (p1)-[r]->(p2) WHERE TYPE(r) IN ["WRITES", "REVIEWS", "IS_A", "FOR", "SENDS"] RETURN id(p1) as source,id(p2) as
target', {concurrency:4, graph:'cypher'})
YIELD nodeId, centrality
RETURN algo.asNode(nodeId).name, centrality
ORDER BY centrality desc
```

This is another usecase of the Betweenness centrality algorithm, where we calculate the influential reviewers that sit on the paths among different authors. Here, we conceive the sequence of relationships from a Reviewer to an Author (Throught consecutive relationships: SENDS, FOR, WRITES) as one zoomed out relationship connecting the Reviewer with the Authors of the reviewed paper. Here a Guru reviewer, who could have reviewed many papers of so many authors, locating himself in the center of the cluster of those Authors can be considered as an influential or reputable reviewer, especially if there were not many other Reviewers connecting authors from the same cluster.

In this usecase, we tried to investigate the possibility of looking at a sequence of consecutive relationships in a graph as a one compact relationship in a projected subgraph when applying different algorithms.

D. Recommender

```
// Find the database community
MATCH(k:Keyword)
WHERE k.name in ['data management', 'indexing', 'data modeling', 'big data', 'data processing', 'data storage',  'data
querying']
// Find all the conferences that are related to the database community
WITH collect(k.name) as db_community
MATCH(k:Keyword)<-[:CONTAINS]-(p:Paper)-[:PUBLISHED_IN]->(e)<-[:HAS]-(c)
WHERE k.name in db_community AND ANY(lbl IN ["Edition", "Volume"] WHERE lbl IN LABELS(e)) AND ANY(lbl IN ["Conference",
"Journal"] WHERE lbl IN LABELS(c))
WITH c.name as conference, count(distinct p) as number_of_db_community_papers
MATCH(k:Keyword)<-[:CONTAINS]-(p:Paper)-[:PUBLISHED_IN]->(e)<-[:HAS]-(c)
WHERE ANY(lbl IN ["Edition", "Volume"] WHERE lbl IN LABELS(e)) AND ANY(lbl IN ["Conference", "Journal"] WHERE lbl IN
LABELS(c))
WITH conference as confjour, LABELS(c)[0] as type, number_of_db_community_papers as number_of_db_community_papers ,
c.name as confjour2, count(distinct p) as number_of_papers
WHERE confjour2 = confjour and number_of_db_community_papers * 1.0 / number_of_papers >= 0.9
WITH confjour as confjour, type as type, number_of_db_community_papers as number_of_db_community_papers, number_of_papers
as number_of_papers, number_of_db_community_papers * 1.0 / number_of_papers as percentage
CREATE (cj:dbCommCJ{ name:confjour, type:type, n_dbPapers: number_of_db_community_papers,n_papers:number_of_papers,
percentage: percentage })
WITH cj as cj
MATCH (c:Conference {name: cj.name})-[:HAS]->(e:Edition)<-[:PUBLISHED_IN]-(p:Paper)
CREATE UNIQUE (cj)-[:HAS_PAPER]->(p)
CREATE UNIQUE (c)-[:IS_A]->(cj)
RETURN DISTINCT cj.name AS conf_journal, cj.type AS type, cj.n_dbPapers As number_of_db_community_papers, cj.n_papers AS
number_of_papers, cj.percentage AS percentage
ORDER BY percentage DESC
```

In the first query, we identify the Conferences/Journals that belong to a database community and for each one we create dbCommCJ node, in order to include the inferred knowledge to the graph. In addition, we create edges from the dbCommCJ node to the corresponding Papers (:HAS_PAPER) as well as to the corresponding Conferences/Journals (IS_A).

```
// Find top cited Papers (highest page rank)
MATCH (cj:dbCommCJ)
WITH  collect(cj.name) as dbCommunity
MATCH (cj:dbCommCJ)-[:HAS_PAPER]->(p:Paper)
OPTIONAL MATCH (p)-[:CITED_BY]->(c:Citation)
WHERE c.confjour in dbCommunity
WITH  p as p,cj.name as conference_journal, cj.type as type, count(c) as citations
ORDER BY citations DESC Limit 100
CREATE (tp: TopPaper{n_citations: citations})
CREATE (p)-[:IS_A]->(tp)
RETURN p.title as paper, conference_journal, type, citations
```

Here, we need to mention that we added an attribute to the Citation nodes that represents the Paper by which the Citation is done. Moreover, we create a node for every top Paper and a connection to the corresponding Paper.

```
// Create good and guru authors
CREATE (x:DbReviewer)
CREATE (y:Guru)
WITH x as x, y as y
MATCH (a:Author)-[:WRITES]->(p:Paper)-[:IS_A]->(tp:TopPaper)
WITH a, count(p) as cnt, x as x, y as y
WITH a as a, CASE WHEN cnt>1 THEN 'YES' ELSE 'NO' END as IS_GURU, x as x, y as y
ORDER BY IS_GURU DESC
CREATE (a)-[:IS_A]->(x)
WITH IS_GURU as IS_GURU, a as a, y as y
WHERE IS_GURU = 'YES'
CREATE (a)-[:IS_A]->(y)
// Show good and guru authors
MATCH (a:Author)-[:WRITES]->(p:Paper)-[:IS_A]->(tp:TopPaper)
WITH a, count(p) as cnt
WITH a.name as TopAuthors, CASE WHEN cnt>1 THEN 'YES' ELSE 'NO' END as IS_GURU
RETURN TopAuthors, IS_GURU ORDER BY IS_GURU DESC
```

In the last query, we create 2 new nodes, namely DbReviewer and Guru. Then, we identify all the Authors that can be reviewers of the database community and we link them with the DbReviewer node. We also find the guru Authors and we link them with the Guru node. Finally, we are printing all the top Authors.