Semantic Data Management

Rebekka Sihvola   Marek Hradil

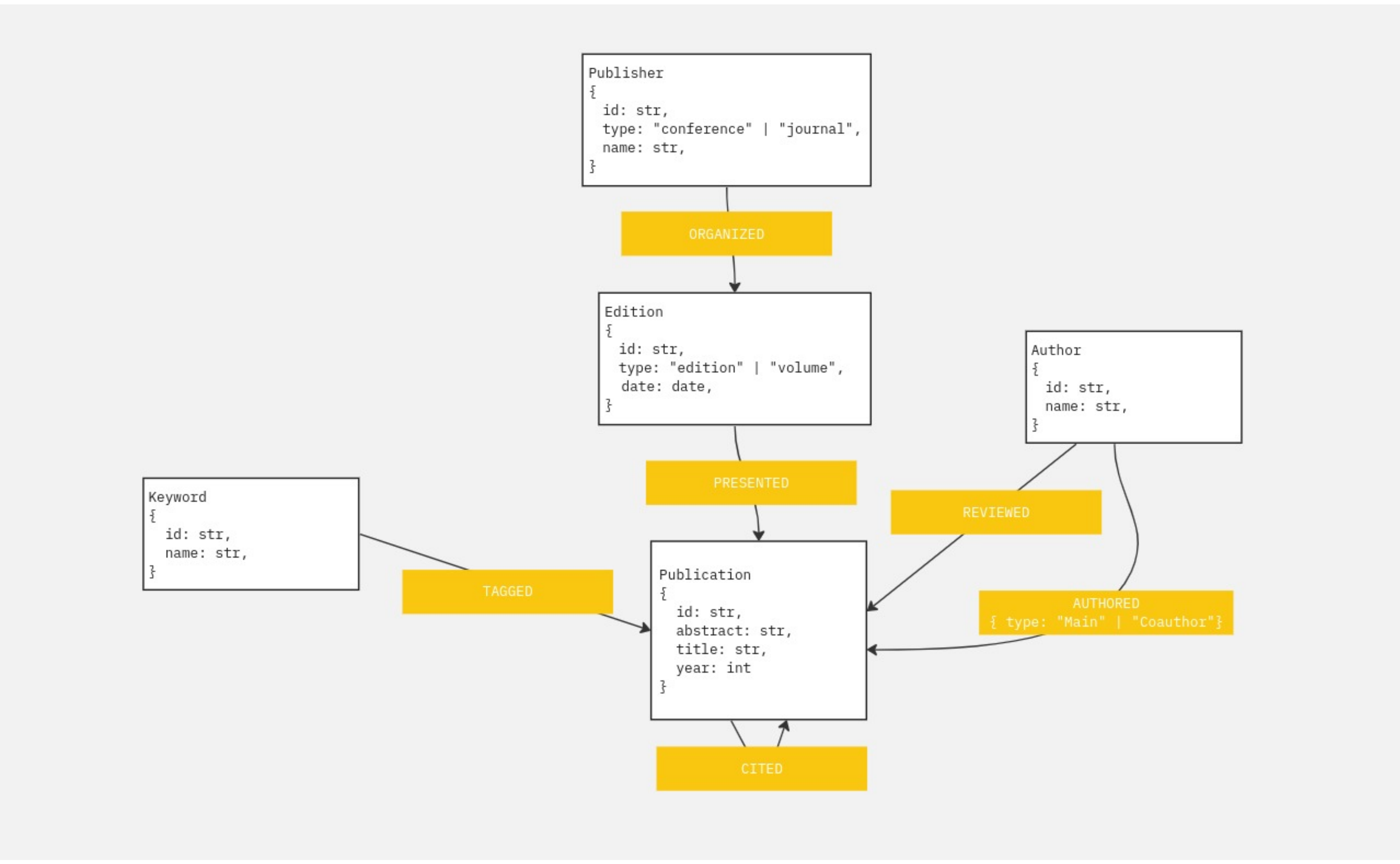
March 2024

Contents

A Modeling, Loading, Evolving

A.1 Modeling

First of all, to make things clear, we present the visual representation of the database schema.



As the schema was modeled as close as possible to the ”business” use case, more than commenting the schema itself, we will focus on what could have been done differently and the possible changes.

Editions and volumes are for us the point which was given the most consideration. First of all, a decision was made to store volumes and editions as a single type of node called "Edition". This was mostly done for usability and performance reasons (as most of the queries required afterwards do not distinguish between them), so separation would be only costly, even if it better represented what is actually going on. The fact that both editions and volumes are stored under a node called "edition" can be then seen as misleading, however all of the other names we have considered were too abstract.

Editions as relationships were also considered and we still see them as reasonable choice, but representing them as nodes would then make other changes (for example adding name of the editions) easier to implement.

Authoring and coauthoring is denoted just by a property on a given relationship. More beautiful approach would be maybe to model them as two separate relationships, however, in most queries after, this information is not important; separation could then lead again only to a performance hit.

Performance of queries in section in B is relatively efficient. Although, the queries are reasonably simple, they are used with large datasets. Therefore, we have improved the efficiency with indexing on key properties:

`publication_year_index` supports faster retrieval of publication year data, which are used for computing the impact factor of a conference.

`keyword_name_index` supports faster retrieval of keyword names, what is used in the recommender queries, to find quickly the keywords.

As was also mentioned before, we wanted to have the database modelled clearly, still there were some places, where we have duplicated the data to improve performance, mainly having `Edition.type` and `Publisher.type`, where edition type could be determined by publisher type (e.g. edition type is always volume, if the publisher type is journal). This is however advantageous, since in some queries, only journals or conferences are needed, but not the publisher data; one join is thus saved.

Additionally, queries are written in a clear and straightforward way, making it easier for the database to understand and find the information we need. Overall, these optimization strategies lead to timely and responsive data retrieval, making the queries suitable for various analytical tasks.

A.2 Loading

Most of the needed data were taken from SemanticScholar API service. We used mainly `/paper/batch` endpoint to query the data, performing sort of a DFS search through the citation links up to a `DEPTH_LIMIT` depth (that was in this case 2). For the root of the DFS tree, the node with title "Robust and Adaptive Control" was chosen, since it was the so far most cited publication of 2024, according to SemanticScholar. Each `FILE_ENTITIES_LIMIT` (set to 5000) entities found, an API endpoint was invoked, to download the data to prevent unnecessary requests. To traverse the publications in the DFS manner was mainly in a response to a failure of the simpler alternative tried earlier, which was restricting data only to certain years; these were in turn not as connected with the citations as expected, which was making following work harder.

After downloading the JSON data, CSV data was generated. Also, data that were unavailable in the API had to be mocked:

Affiliations For each author, same number of affiliations were generated. Then authors were assigned randomly. Affiliations have also their type - with probability 0.5 they are universities and with probability 0.5 they are companies. The mocked names of the affiliations reflect the type.

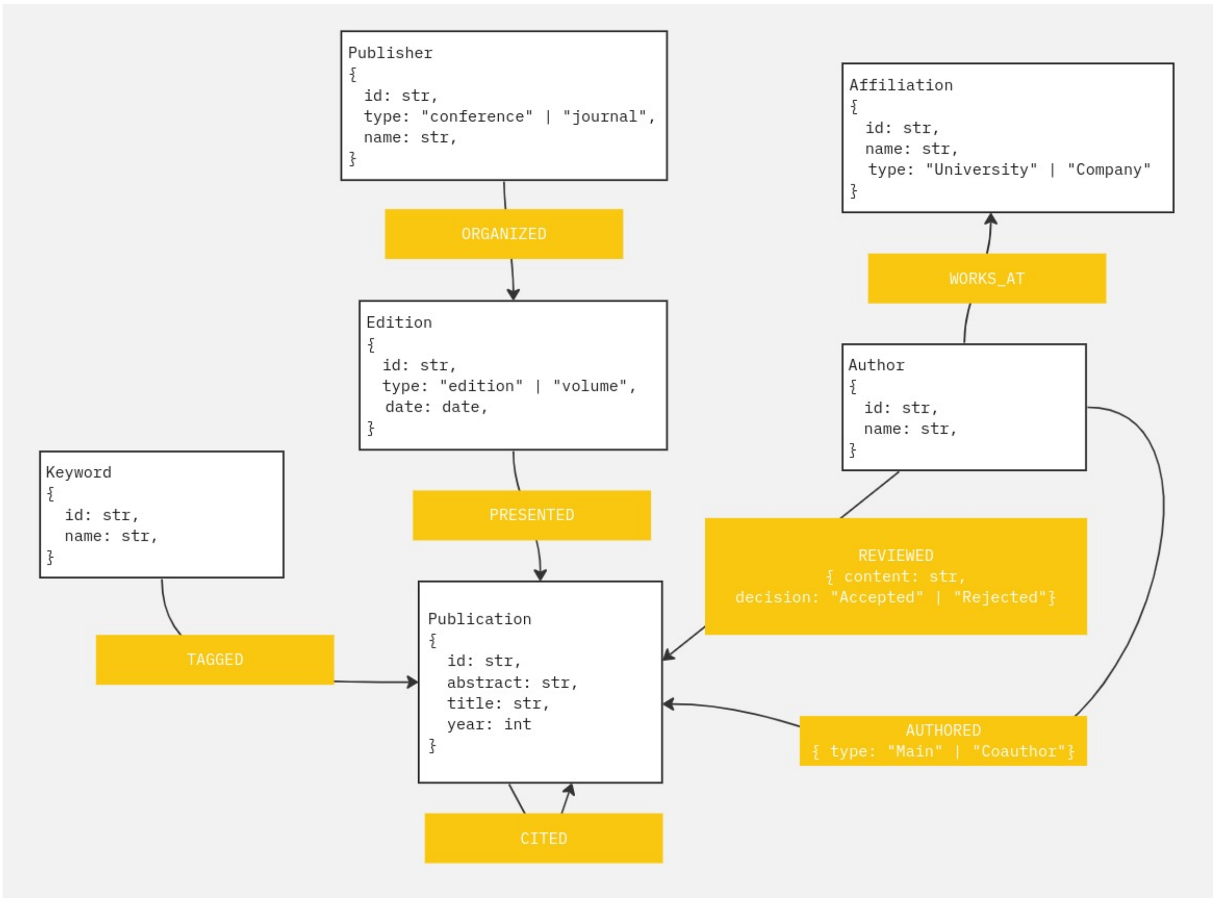
Keywords For each article, 10 keywords were generated from the abstract (using NLP library `yake`). After that all of these keywords were traversed and a syntactic similarity of every pair was measured using library `thefuzz` to merge similar keywords together, to achieve more believable results.

Reviews Lastly, reviews contents and decisions were generated. Using `faker` library, random sentences were generated and with probability 0.1 a negative decision ("Rejected").

When CSV files are done, `docker compose` takes over. Through a custom `Dockerfile`, the data are moved to inside of `neo4j` container to the `/import` folder, waiting for the `import.cypher` query to be run, which then creates the correct nodes and relationships, along with adding some additional constraints to make the database more maintainable.

A.3 Evolving

For evolving, we chose the following schema:


Publisher { id: str, type: "conference" | "journal", name: str, }
ORGANIZED
Edition { id: str, type: "edition" | "volume", date: date, }
PRESENTED
Affiliation { id: str, name: str, type: "University" | "Company" }
WORKS_AT
Author { id: str, name: str, }
Keyword { id: str, name: str, }
TAGGED
REVIEWED { content: str, decision: "Accepted" | "Rejected"}
Publication { id: str, abstract: str, title: str, year: int }
AUTHORED { type: "Main" | "Coauthor"}
CITED

Where a new affiliation node can be seen, along with the `WORKS_AT` relationship and a new contents of `REVIEWED` relationship - `content` and `decision`. Again, both of these decisions seem natural; for example it would not make sense to store affiliation in the author node, since more authors can have the same affiliation and this is an important fact, which then could be queried. The reviews could be stored as a node, however we do not think there is an added value.

B Querying

In this section we will demonstrate four examples of utilizing our database. Next we will present each query and equivalent Cypher query:

1. Find the top 3 most cited papers of each conference

```
MATCH (publisher:Publisher { type: "conference" })-[:ORGANIZED]->(edition:Edition)
-[:PRESENTED]->(publication:Publication)<-[:CITED]-(citingPublication:Publication)
WITH publisher, publication, count(citingPublication) AS citationCount
 ORDER BY citationCount DESC
WITH COLLECT(publication) AS publications, publisher
RETURN publisher AS conference, publications[0..3] AS publications
```

2. For each conference find its community

```
MATCH (a:Author)-[:AUTHORED]->(publication:Publication)<-[:PRESENTED]-(edition: Edition)
<-[:ORGANIZED]-(c:Publisher { type: "conference" })
WITH a, c, COUNT( DISTINCT edition.date) AS editions
WHERE editions >= 4
RETURN c AS publisher, COLLECT(a) AS community
```

3. Find the impact factors of the journals in your graph

```
MATCH (a:Publication)<-[:PRESENTED]-(e:Edition)<-[:ORGANIZED]-(b:Publisher)
WHERE a.year >= 2022 AND a.year <= 2023
WITH b, a
OPTIONAL MATCH (cited:Publication)<-[:CITED]-(a)
WHERE cited.year = 2024
WITH b, COUNT(DISTINCT CASE WHEN a.year = 2022 THEN a END) AS publications2022,
            COUNT(DISTINCT CASE WHEN a.year = 2023 THEN a END) AS publications2023,
            COUNT(cited) AS citations2024
RETURN b AS publisher,
       (citations2024 * 1.0) / (publications2022 + publications2023) AS impactFactor
ORDER BY impactFactor DESC
```

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

```
MATCH (a:Author)-[:AUTHORED]->(p:Publication)
OPTIONAL MATCH (p)<-[:CITED]-(otherPublication:Publication)
WITH a, p, COUNT(otherPublication) AS citationsCount
 ORDER BY citationsCount DESC
WITH a, COLLECT(citationsCount) AS citationCounts
WITH a, REDUCE(s = 0, count IN RANGE(1, SIZE(citationCounts)) |

CASE WHEN citationCounts[count - 1] >= count THEN count ELSE s END) AS hIndex
RETURN a.name AS authorName, hIndex
ORDER BY hIndex DESC
```

C Recommender

In this section, we'll develop a basic recommender system. Firstly, we'll identify potential reviewers for the database community using our database. Then, we'll outline each step along with its corresponding Cypher query.

1. Defining database community We assume that the database community is defined through the following keywords: data management, indexing, data modeling, big data, data processing, data storage and data querying.

```
MATCH (keyword:Keyword)
WHERE k.name IN ["data management", "indexing", "data modeling", "big data",
"data processing", "data storage", "data querying"]
RETURN keyword
```

2. Finding the publishers of database community Next, our task is to locate conferences and journals pertinent to the database community, specifically those tailored to the field of databases. Publishers are deemed relevant to the community if at least 90pct. of the papers they publish contain keywords associated with the database community.

```
MATCH (k:Keyword)
WHERE k.name IN ["data management", "indexing", "data modeling",
"big data",
"data processing", "data storage", "data querying"]
MATCH (k)-[:TAGGED]->(pub:Publication)

MATCH (p:Publisher)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)

WITH p AS publisher, COUNT(pub) AS totalPublications, COLLECT(pub) AS allPublications

MATCH (p)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)<-[:TAGGED]-(k:Keyword)
WHERE k.name IN ["data management", "indexing", "data modeling", "big data",
"data processing", "data storage", "data querying"]
WITH publisher, totalPublications, COUNT(pub) AS taggedPublications, allPublications

WITH publisher, totalPublications, taggedPublications, allPublications,
taggedPublications * 1.0 / totalPublications AS taggedRatio
WHERE taggedRatio >= 0.9

RETURN publisher
```

3. Top papers of database community We determine the top papers from these publishers by identifying those with the highest number of citations within the same community (i.e., papers within the database community publishers). This process yields the top 100 papers from the database community.

```
MATCH (p:Publisher)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)

WITH p AS publisher, COUNT(pub) AS totalPublications, COLLECT(pub) AS allPublications

MATCH (p)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)<-[:TAGGED]-(k:Keyword)
WHERE k.name IN ["data management", "indexing", "data modeling", "big data",
"data processing", "data storage", "data querying"]
WITH publisher, totalPublications, COUNT(pub) AS taggedPublications, allPublications

WITH publisher, totalPublications, taggedPublications, allPublications,
taggedPublications * 1.0 / totalPublications AS taggedRatio
WHERE taggedRatio >= 0.9

UNWIND allPublications AS paper
WITH DISTINCT paper
MATCH (paper)<-[c:CITED]-()
WITH paper, COUNT(c) AS numCitations
 ORDER BY numCitations DESC
LIMIT 100

MATCH (author:Author)-[:AUTHORED]->(paper)
WITH author, COLLECT( DISTINCT paper) AS authoredPapers
WHERE SIZE(authoredPapers) >= 2

RETURN author AS guru, authoredPapers AS topPapers
```

4. Finding database gurus Finally, we want identify gurus, i.e., reputable authors that would be able to review for top conferences. Gurus are those authors that are authors of, at least, two papers among the top-100 identified.

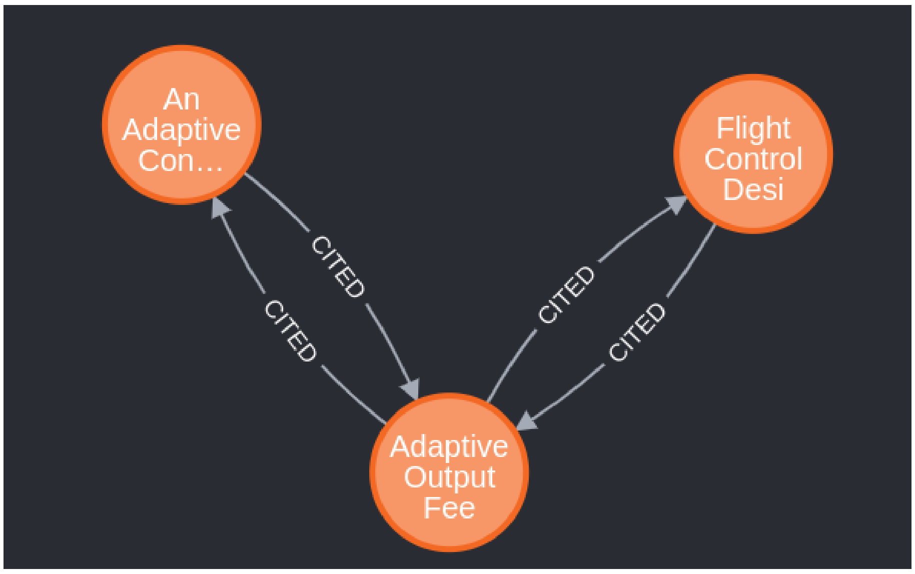
```
MATCH (p:Publisher)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)
WITH p AS Publisher, COUNT(pub) AS TotalPublications, COLLECT(pub) AS AllPublications
MATCH (p)-[:ORGANIZED]->(edition:Edition)-[:PRESENTED]->(pub)<-[:TAGGED]-(k:Keyword)
WHERE k.name IN ["data management", "indexing", "data modeling", "big data",
"data processing", "data storage","data querying"]
WITH Publisher, TotalPublications, COUNT(pub) AS TaggedPublications, AllPublications
WITH Publisher, TotalPublications, TaggedPublications, AllPublications,
     TaggedPublications * 1.0 / TotalPublications AS TaggedRatio
WHERE TaggedRatio >= 0.9
UNWIND  AllPublications AS paper
WITH DISTINCT paper
MATCH (paper)<-[c:CITED]-()
WITH paper, COUNT(c) AS numCitations
ORDER BY numCitations DESC
LIMIT 100
MATCH (author:Author)-[:AUTHORED]->(paper)
WITH author, COLLECT(DISTINCT paper) AS authoredPapers
WHERE SIZE(authoredPapers) >= 2
RETURN author AS Guru, authoredPapers AS TopPapers
```

D Graph algorithms

Using GDS native projection we have constructed a graph of publications, where citations serve as edges. Then following two algorithms were chosen:

SCC Generally, when considering graph of citations as an oriented graph it should be acyclic. If a cycle is found, this must represent some invalidity of the data or communities that cite each other. Nevertheless both scenarios should not happen. We were thus interested, if the data obtained does not obtain such defects.

Every strongly connected component with a size bigger than one contains a cycle, which is the defect we were looking for. We therefore utilized SCC's to find strongly connected components and to our surprise we were able to find a community of three publications with unclear citations. These were:



Louvain Our other interest was detecting "communities" of publications. For this we used Louvain algorithm, again invoked on the same publication graph as SCC. After listing out the communities, the query follows with creating `Community` nodes and `BELONGS_TO` relationships, so it is easier to visualize the communities. The top 3 largest communities were then:

- Adaptive Control Systems - size 70
- Adaptive Flight Control Systems - size 49
- Adaptive Control for Aerospace Systems - size 46

The names were created using ChatGPT which was prompted to summarize all of the titles. Why so many control systems communities? For us this has to do with the root of the DFS tree, which was in a similar area - it would be then logical that citations and citations of citations are also publications in same area.