Filip Sotiroski
Md. Jamiur Rahman Rifat

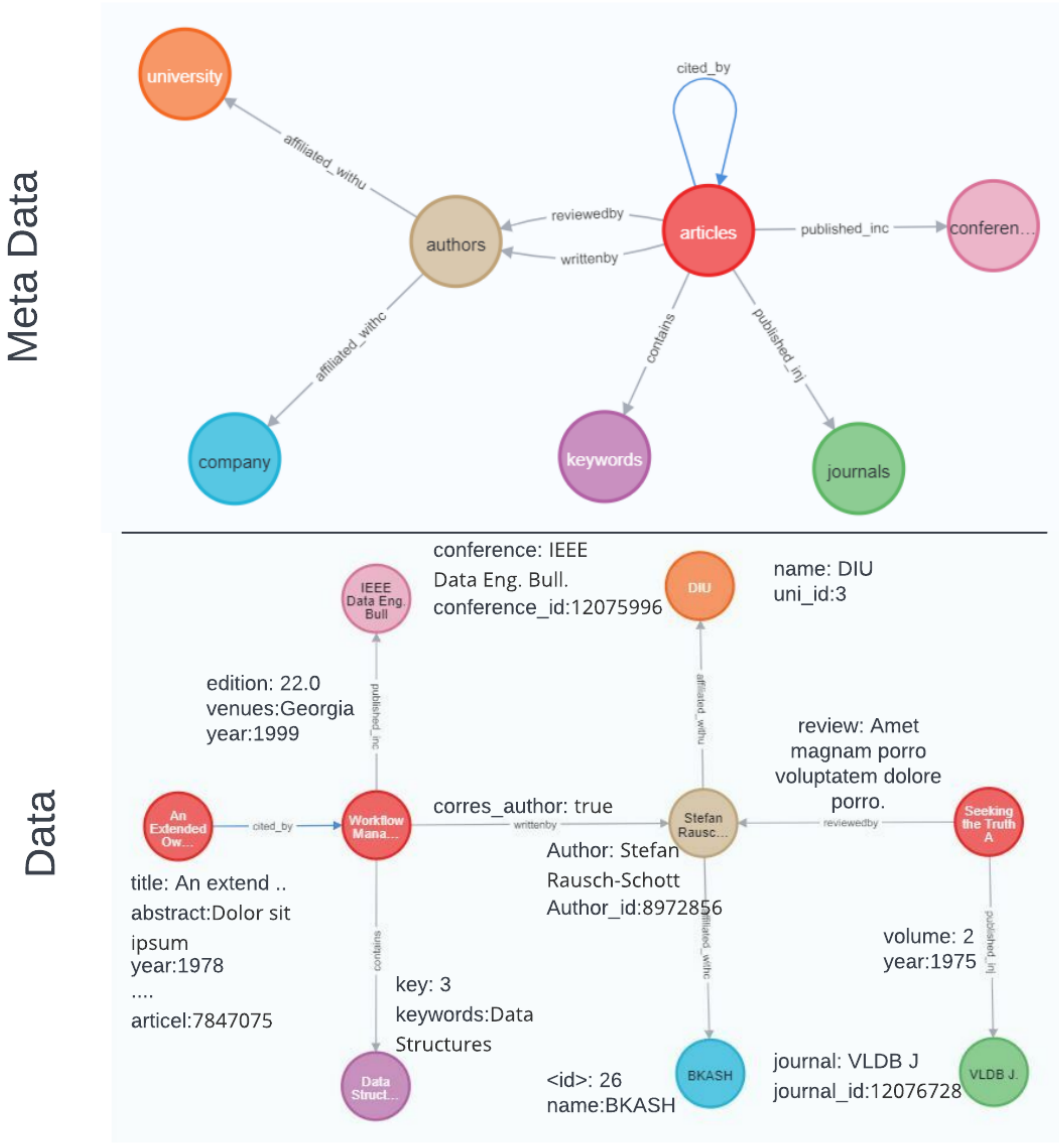
Property Graphs

BDMA - Semantic Data Management - Lab 1

March 2022

Part A - Modeling, Loading, Evolving

A.1 Modeling the graph



Here the articles and journals are the most used nodes as most of the queries are related to these. Keywords are chosen to be separate nodes to use less disk space and faster query time as there will be so many to many relations between keywords. The journals and conferences are also set as separate nodes for this reason. The university and company nodes are added later as a part of the evolution of the graphs. As authors both write and review papers therefore they are linked with two separate relations with the articles. The editions, volume and year are added as a relationship between article and conferences/journals.

A.2. Loading the graph

The data for this graph is provided from the DBLP website. We exported the XML to CSV using the following tool: DBLP-to-CSV. We followed the instructions, sources and tools provided by the lab guide.
Although the source was not perfect, it provided us the majority of the data that we have in the graph. We were missing:

- Keywords, which were manually generated
- Contains relationship, which were randomly assigned to the articles
- Cited relationship, which was randomly assigned to the articles.
- The corresponding author of an article. Authors were a list in articles.
- Conferences, only the Journals were provided. We decided to use half of the journals for conferences and the others for the journals.
- Company and University were also missing, they are added in the A.3. part. Same for the relationships between them and the authors.
- The authors were extracted from the Articles as there was a list of Authors as a column.
- The review relationship was also created randomly.
- The abstract for the articles was generated with Lorem Ipsum paragraphs.

After creating the remaining CSV files we used the **neo4j-admin imp#ort** command to fully load the graph in a new database. For the nodes we were following the convention to use `:ID` for the ID's of the nodes. For the relationships we had to precise the `START_ID` and `END_ID` to refer to the start and end nodes. The full one-liner can be found in import.sh.
You can see all the preprocessing in detail in the PartA1_SotiroskiRifat.ipynb file.

A.3. Evolving the graph

The code for this part can be found in the PartA3_SotiroskiRifat.py file.
The Company and University were generated randomly as new CSV files. Using the LOAD CSV Cypher command. We keep university and company as separate nodes because there could be many to many relations between author and university and, author and company.
affiliations_withc relations were created between author and company and *affiliations_withu* between authors and university nodes.

In order to do the suggested decision we are setting it as a property to the *reviewedby* relationship. We use the rand() function and set it as true for two thirds of the reviews.
Later, we set the reviewed property for the article if all the suggestions for the reviews are set to true. We do this by collecting the reviews for each article and checking if all of them are true. Then we set the reviewed variable to true.

These queries are done directly with Cypher while the graph is in the database.

Part B - Querying

The code for this part can be found in the PartB_SotiroskiRifat.py file.

Most citations

We are interested in finding the top citations for each conference. At first we matched all the articles published in the conferences then counted the number of articles that had cited this paper. At last, aggregated based on the conference name.
The query for this task is in the citation_query function.
:

"con"	"arti"
"ACM SIGMOD Digit. Rev."	[["Review - Predicate Migration: Optimizing Queries with Expensive Predicates.",3],["Review - Relational Databases for Querying XML Documents: Limitations and Opportunities.",3],["Review - Schema Versioning for Multitemporal Relational Databases.",2]]
"Appl. Math. Comput."	[["Retraction notice to "Synchronization analysis of linearly coupled singular systems" [AMC 248 (2014) 536-549].",2]]
"Commun. ACM"	[["An Approach for Reverse Engineering of Relational Databases.",2],["Comparing Data Modeling Formalisms.",2],["Managerial Considerations.",2]]

Community query

The objective of the query is to find for each conference the authors who published in more than 4 editions. Therefore, we match all the articles that have been published in conferences and written by authors. The aggregation is then done on the conferences and authors with a count on how many edition that article has been published in. The query then filters the last number wiith greater than 4. We then return the result as a list of authors of that community.
The query for this task is in the communities_query function.

	conferences	community
1	"SIGMOD Rec."	["Richard T. Snodgrass", "Jim Melton", "Andrew Eisenberg"]
2	"IEEE Data Eng. Bull."	["Yannis E. Ioannidis", "Surajit Chaudhuri", "Jennifer Widom", "Hector Garcia-Molina", '

Impact Factor

The equation of finding impact factor is:

$$\frac{\text{Citations}_y}{\text{Publications}_{y-1} + \text{Publications}_{y-2}}$$

The number of citations were calculated first as like the first query then using the name of the journal and the year of publication we again calculated the publications number for the previous

two year and calculated the impact factor using the above formula. At last we aggregated the result on each journal. As the amount of data is less and the year is widely distributed therefore some journals yielded impact factor equal to 0 which have been excluded here.

The query for this task is in the impact_factor_query function

"name"	"impact_factor"
"ACM Comput. Surv."	0.79
"ACM Trans. Database Syst."	0.69
"IEEE Trans. Knowl. Data Eng."	0.35
"VLDB J."	0.21

H-index

To find the h-index we were following to following formula:

$$h\text{-index}\ (f) = \max\{i \in \mathbb{N} : f(i) \geq i\}$$

After gathering all the citations of each paper from an author in one list *citations* we use the CALL subquery where we apply the logic from the formula. In essence, we go through each element using the *enumerate* list and get the number of papers that match the h-index condition. We then return from the subquery the h-index for that author.
The query for this task is in the hindex_query function
We see from result the top h-indexes of the authors.

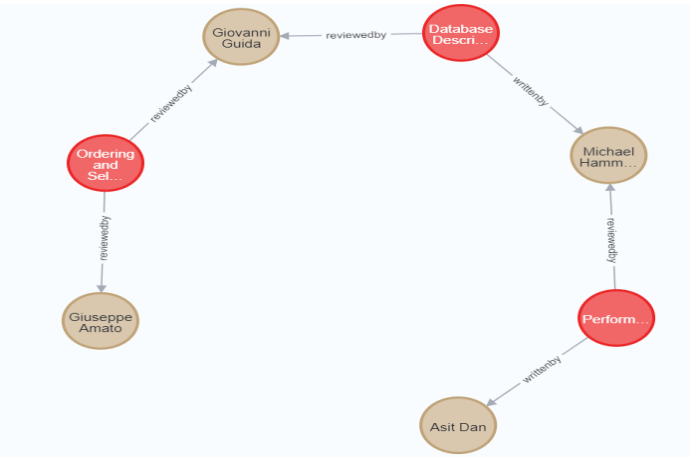
"authors"	"citations"	"hindex"
{"author":"Elisa Bertino","isGuru":true,"author_id":9058019}	[9,5,3]	3
{"author":"Alfons Kemper","author_id":9681662}	[10,9,7]	3
{"author":"Philip S. Yu","author_id":9054679}	[9,6,4,2]	3

Part C - Graph algorithms

The code for this part can be found in the PartC_SotiroskiRifat.py file.

Dijkstra - Shortest path algorithm

In this problem we tried to identify the shortest distance between the author “Asit Dan” and “Giuseppe Amato”. The total cost of the shortest path found to be 6 as expected. As our graph is directed therefore to get a path from one node to another we make that undirected. On the figure on the RHS you can see the result.
The shortest path can be very useful in order to find the path through which (i.e.) two authors are connected to each other.
In this case, as we don’t have weights, the Dijkstra algorithm uses 1 as weight.
The query for this task is in the shortest_query function



Louvain - Community detection algorithm

We chose to do a Louvain community detection as we are interested in finding the ‘clusters’ of nodes that are gravitating around another node. There are articles who contain one or more keywords. It is interesting to see towards which keywords these articles are more related to.
We create the graph using the following syntax:

```
gds.graph.create('myGraph', ['articles', 'keywords'], 'contains')
```

Notice that from the initial graph in the database we create a view that will be used by the library. We select the *articles* and the *keywords* and the *contains* relationship. The algorithm by itself is homogeneous, so both articles and keywords are treated as same nodes.
We then applied the stream methods in order to run the Louvain algorithm:

```
gds.louvain.stream('myGraph')
```

We get that we have 8 communities, the same amount of keywords in our database.
After writing the communities on the article and keywords nodes (i.e. community 0), we do a simple match on the contains relationship and the authors that wrote the articles to see what changes were made. We notice that now we can see to which community the authors are related to.

The query for this task is in the louvain function.

Part D - Recommender

The code for this part can be found in the PartD_SotiroskiRifat.py file.

The first thing we need to do is to define the communities by the keyword. In our case, we assumed that the keywords are based on the keywords nodes. As mentioned in part A, we have synthetically generated the keywords and randomly assigned to each article with the contained relationship. For this case we reused the Louvain community detection algorithm. We use the same graph view that we created. The only difference is that we use the write method to apply the property to the keywords and article nodes.

```
CALL gds.louvain.write('myGraph', {writeProperty: 'community'})
```

The query for this step is in the step1 function

Next, we need to assign the community id to a conference/journal. We do this by assigning the community id depending on the majority of the articles that are in those conferences/events. What our Cypher code does is that it finds the number of papers of each community and each conference. The difficult part is how to divide that number with the total number of papers on that conference in order to get the percentage. That's why we used a list in which we 'store the indexes' and then unwind it in order to be able to calculate the percentage.
Finally we order the results by the percentage and collect and get the first row as it is the highest percentage for that conference.
Note: in our database we don't have communities that represent 90% of the papers for that conference, therefore the threshold can be changed, but either way we get the community with the maximum number of articles.
The query for this step is in the step2 function

This is the results we obtain after ordering by the percentage and getting filtering the results:

"com"	"conf"	"percentage"
7	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.1919191919191919
5	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.1919191919191919
6	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.18181818181818182
3	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.1111111111111111
4	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.1111111111111111
0	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.0707070707070707
2	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.0707070707070707
1	{"journal_id":12075516,"conference":"SIGMOD Rec.","community":7}	0.0707070707070707

The third task is to run the PageRank algorithm. For each community we run the algorithm and assign the score to those articles. Here, we need to use the Cypher notation when creating the graph. For the second parameter we return the ID's of the nodes (articles), and the third is the relationship, cited_by.
We use the Cypher notation as we need to do this subgraph for all the communities. In this example, for simplicity, we chose community 6.

The query for this step is in the step3 function
After creating we use the write method on the 'pageRank' graph and we write the score as a property.

"article"	"pageRank"
7816474	0.385875
7791764	0.34125000000000005
8919322	0.34125000000000005
7846912	0.3213015625

At the end, we simply match the conference with those articles in that conference and the corresponding community and we set the property top100 as true. This property will be used in the next step.

The query for this step is in the step4 function.