Universitat Politècnica de Catalunya

Semantic Data Management

LAB Assignment 1

Property graphs

Enric Reverter
enric.reverter@estudiantat.upc.edu

Míriam Méndez
miriam.mendez.serrano@estudiantat.upc.edu

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Curs 2022/2023 Q1

Contents

A Modeling, Loading, Evolving

A.1 Modeling

The design chosen for modeling the graph in accordance with the described objective is depicted in Figure 1. Thus, the nodes are: `:Article`[1], `:Author`, `:Keyword`, `:Volume`, `:Journal`, `:Edition`, and `:Conference`. Their properties can be observed in the diagram as well. Note how `Article` is the node with more properties. Regarding edges, the following are created: `:Author` and `:Article` can be related through: *:HAS_WRITTEN*, *:IS_CORR_OF* and *:HAS_REVIEWED*; `:Article`'s nodes are connected through *:CITES_TO*; `:Article` and `:Keyword` relate through *:IS_ABOUT*; `:Article` is *:PRESENTED_AT* at `:Edition` (`:Volume`), which at the same time is *:HOLD_AT* `:Conference` (`Journal`).

Regarding nodes, that articles and authors need to be such type is evident. Then, one could argue that editions (volumes) and conferences (journals) could be merged within a single node, but this would result in storing redundant data, for which they have been differentiated. This means queries that require information from both articles and conferences (journals) will require an additional hop through edition (volume). Another option could be to add an additional edge that skips the above-mentioned hop, but this would increase the complexity of the graph at it scales, so it has not been considered. Finally, keywords are also stored as nodes rather than article's properties, so the communities can be efficiently queried.

Concerning edges, the ones between keywords, articles, editions (volumes), and conferences (journals) are again, self-evident. This includes the citations among articles. Then, authors and articles share three types of edges: *:HAS_WRITTEN*, *:IS_CORR_OF*, and *:HAS_REVIEWED*. The first two could be merged into a single one, but it is reasoned that if a query related to corresponding authors was needed, it would be problematic to scan all the edges for the property (i.e., each article has been written by many authors). This allows to efficiently filter the `cypher` queries in such a case and only takes one additional edge per article. The latter describes a different action than these.

Regarding properties, most of them are self-evident, but city and year within editions (volumes) are worth discussing. It is assumed that an edition of a conference is defined by its city and year, so another option would be to link each conference with a city-year pair of nodes and remove the edition one. However, this would complicate the queries that need to account for different editions. Instead, if the city-year nodes are created in addition to edition ones, redundant data (i.e., repeated year and city labels) could be solved, but the trade-off would be less efficient queries, as more hops would be required. The network complexity would incidentally increase as well. For that reason, they are kept as properties, though for computing measures such as the impact factor, it would probably be more efficient to have years as nodes. The properties of each entity can be depicted in Figure 1.

[1] Article is interchangeable to scientific paper in this context.

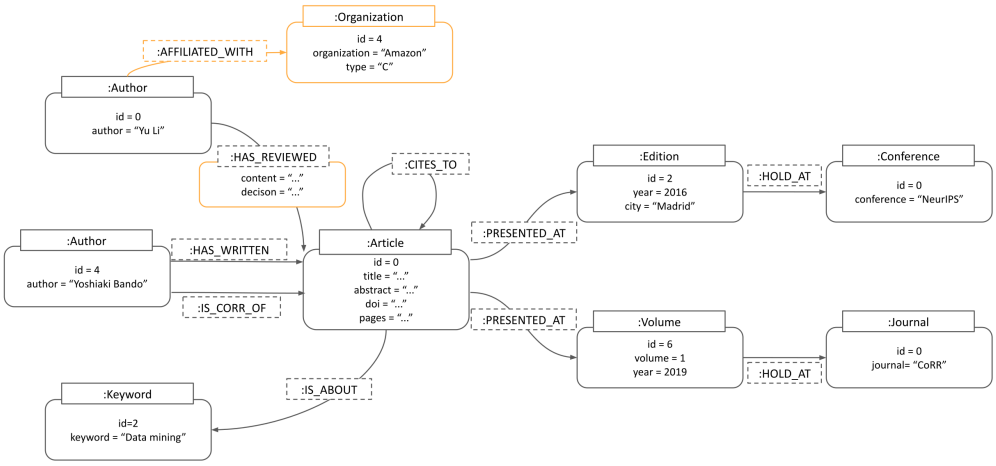

Figure 1: Proposed architecture design. Node entities are represented by rectangular boxes with straight lines. Edge entities are represented the same, but with dashed lines. Properties are depicted within rounded boxes. In orange, the entities added during model evolution.

A.2 Instantiating / Loading

To ensure a realistic scenario, data from DBLP is used, as prescribed by the problem statement. However, some fields described in the assignment are missing, for which a fraction of data has been generated. It is worth to mention that neither indices nor materializations have been considered.

First, data is downloaded as a *xml* file and then converted to *csv* through the dblp-to-csv tool. In total, data occupies around 4 GB of memory.

Second, data is preprocessed with *Python*. Realistic data being used encompasses that of: articles, authors, editions, volumes, conferences, journals, and its edges. Then, only a fraction of the articles is selected for the analysis (500 instances of "articles" and 500 instances of "inproceedings"). All this scientific papers have been randomly selected among 4 different conferences and 4 different journals to increase the chance of the authors being related to more articles. All years are taken into account. Generated data affects keywords, citations, and some properties. A total of 50 keywords related to data science have been generated, which are then randomly assigned to articles (approximately 5 per article). Moreover, due to the characteristics of Section C, an additional injection of keywords is done to those papers of a specific conference. Regarding citations, they have been generated in such a way that the newer an article, the less citations it receives. There is also a constraint to avoid citations from more recent papers to older ones. Finally, additional properties such review content and abstracts have been generated through GPT-2.

Finally, data is loaded to a *neo4j* database by means of the `LOAD CSV` command. Assuming data is initially stored in a relational-style, there is a *csv* file for each "dimension" (e.g., journals) and one for the "facts" (articles). As an example, the authors are loaded as follows:

```
LOAD CSV WITH HEADERS FROM file_path AS row
MERGE (a:Author {id: toInteger(row.id), name: row.author})
```

Finally, the articles are created and linked through:

```
LOAD CSV WITH HEADERS FROM file_path AS row
MERGE (p:Article {id: toInteger(row.id), title: row.title, ...})
WITH p, row
UNWIND split(row.author_ids, '|') as author
MATCH (a:Author {id: toInteger(author)})
MERGE (a)-[:HAS_WRITTEN]-(p)
...
```

Note how by using `MATCH` and `MERGE`, if the files increase on size, nodes and edges will not be duplicated.

A.3 Evolving the graph

It is now asked to consider new data. More precisely, it concerns the need to store the review content and its corresponding decision (approved or rejected) by each author. Also, affiliations of the authors, which can be either universities or companies, have to be added. It is assumed that the papers found in the DBLP data were all accepted, for which the majority of the incoming reviews for each article will be considered accepted.

To address this, the two properties, content and decision, are added within the *:HAS_REVIEWED* relationship. The former contains the review text, which has been generated like the abstracts, while the latter stores the decision made, which can be either "Accepted" or "Rejected". To support the second functionality, a new node entity, `:Organization`, is created. Entities have been randomly chosen from a list of infamous companies and universities. Each organization has two properties, name and type, where the latter specifies whether it is a University or Company. The, the relationships between authors and organizations are set as seen in Section A.2. To evolve the properties of the reviews, `SET` is used as in the example depicted below.

```
LOAD CSV WITH HEADERS FROM file_path AS row
MATCH (:Author {id: toInteger(row.author_id)})-[r:HAS_REVIEWED]-(:Article {
    id: toInteger(row.article_id)})
SET r.review=row.content, r.decision=row.decision
```

B Querying

Once data has been loaded into the database, the queries described in the assignment are defined as depicted below.

- **Find the top 3 most cited papers of each conference.**

  ```
  MATCH (c:Conference)<-[:HOLD_AT]-(e:Edition)<-[:PRESENTED_AT]-(p:Article
      )<-[r:CITES_TO]-(q:Article)
  WITH c, p, COUNT(r) AS num_citations
  ORDER BY c.name, num_citations DESC
  WITH c, COLLECT({title:p.title, citations:num_citations}) AS papers
  RETURN c.name AS Conference, papers[..3] AS top_3_papers
  ```

  Straightforward query, the citations of each paper are counted, then the results are sorted and collected to just return the first 3 instances per conference. The article with more citations is

called "A causal optimal filter of the second degree.", from the conference named *EUSIPCO*. Not much can be said since the citations are synthetic.

- **For each conference find its community.**

```
MATCH (a:Author)-[:HAS_WRITTEN]->(p:Article)-[:PRESENTED_AT]->(e:Edition
    )-[:HOLD_AT]->(c:Conference)
WITH c, a, COUNT(DISTINCT e) AS neditions
WHERE neditions >= 4
RETURN c, COLLECT(DISTINCT a.name) AS authors
```

Again, straightforward query, authors that written in at least 4 editions within a conference are matched, then returned for each conference. For example, in *EUSIPCO*, only one author has written in more than 4 editions, Rosario Avino.

- **Find the impact factors of the journals in your graph.** The impact factor is computed through the following formula:

$$\mathrm{IF}_y = \frac{\mathrm{Citations}_y}{\mathrm{Publications}_{y-1} + \mathrm{Publications}_{y-2}}$$

where the citations and publications are calculated within each journal community.

```
MATCH (j:Journal)<-[:HOLD_AT]-(v:Volume)<-[:PRESENTED_AT]-(p:Article)<-[
    c:CITES_TO]-(:Article)
WITH j.name AS journal_name, v.year AS year, toFloat(COUNT(c)) AS
    ncitations
CALL {
    WITH journal_name, year
    MATCH (:Journal {name: journal_name})-[:HOLD_AT]-(v:Volume)-
    [r:PRESENTED_AT]-(:Article)
    WHERE v.year IN [year-1, year-2]
    RETURN toFloat(COUNT(r)) AS preceding_count
}
RETURN journal_name, year, CASE preceding_count WHEN 0 THEN -1
ELSE ncitations/preceding_count END AS if
ORDER BY if DESC
```

This query can be split in two parts. First, the total number of citations within the papers in a journal are counted by matching the pattern. Then, by means of a sub-query, the publications of the two years prior to every volume presented are added up. Finally, the impact factor is computed through the division of this two measurements. The result is a list of journal-year pairs with its impact factor. The highest impact factor is achieved by *CoRR* in 2019, with a value of 37.5.

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

```
MATCH (a:Author)-[:HAS_WRITTEN]->(p:Article)<-[:CITES_TO]-(:Article)
WITH a.name AS Author, p.title AS Title, count(*) AS ncites
ORDER BY ncites DESC
WITH Author, collect(ncites) as Cites
```

```
WITH Author, [x IN range(1, size(Cites)) WHERE x <= Cites[x-1] |  [Cites
    [x-1],x]] AS Hindexes
RETURN Author, Hindexes[-1][1] as Hindex
ORDER BY Hindex desc
```

The h-Index of an author is the maximum number h such that h publications have at least h citations. To compute this, the number of citations by article and author was ordered in descending order, then in the range equal to the size of the citations when the number of citations in a given position is greater than or equal is satisfied. The size of the last in the list is considered as the H-index for a given author.

The results show the author with the highest h-index is Shiguo Xu, with a value of 9. It is followed by Rosario Avino and others, with an h-index equal to 6, which is in line with the results of the second query.

C Recommender

A simple recommender system is implemented. It works as in a retriever-reader architecture, where the retriever fetches the outstanding scientific papers according to the PageRank (PR) algorithm, and the reader scans for the authors. More precisely, the articles being retrieved are those of the communities related to databases.

To do so, the conferences and journals where at least 90% of its papers contain a specific set of keywords related to databases are filtered. First, a community node is created and linked with the keywords conforming it. This is done to ensure scalability, as there could be many communities sharing some keywords. The following query creates the database community:

```
CREATE (s:Community {id: 0, community: 'database'})
WITH s
MATCH (k:Keyword)
WHERE ANY(keyword IN ['Data', 'data'] WHERE k.name CONTAINS keyword)
MERGE (s)-[:DEFINES]-(k)
```

Due to how data has been generated, it is known that if the keywords contain the word "data" they are already a match for the described community. It would yield the same with an exact match on the keywords mentioned in the statement. Once the community is created, the conference and journals are filtered in the following way:

```
MATCH (n)
WHERE (n:Conference OR n:Journal)
MATCH (n)-[:HOLD_AT]-(:Edition)-[:PRESENTED_AT]-(a1:Article),
(n)-[:HOLD_AT]-(:Edition)-[:PRESENTED_AT]-(a2:Article)-[:IS_ABOUT]-(k:
    Keyword)
     -[:DEFINES]-(s:Community {community: 'database'})
WITH n, COUNT(DISTINCT a1) AS total_articles, COUNT(DISTINCT a2) as
    matching_articles
WHERE toFloat(matching_articles)/total_articles >= 0.9
WITH n, toFloat(matching_articles)/total_articles AS matching_score
SET n.community = 'database'
```

The type of node, either conference or journal, is filtered previously to query the whole pattern. Then, it is checked whether the majority of its articles contain any keywords with the specified fields in them. The way total articles and matching articles are computed by counting the number of times the articles match the above patterns. Some values need to be cast into floats to avoid getting 0 as a result. The only entity that suffices this constraint is the *NeurIPS* conference, with 98% of its articles containing such keywords. Finally, the community label for that conference is set to database. This could be done in a list, since a journal/conference could be part of more than one.

Then, in order to identify the top papers of such community, *Neo4j Graph Data Science* (GDS) is used. First, the papers related to the community (i.e., *NeurIPS* are projected into a subgraph, where the PR can be applied. To do the projection, the following query is run:

```
CALL gds.graph.project.cypher(
    'db-community',
    'MATCH (n) WHERE (n:Conference OR n:Journal) MATCH (n {community: "
       database"})
        -[:HOLD_AT]-()-[:PRESENTED_AT]-(p:Article) RETURN id(p) as id',
    'MATCH (p1)-[:CITES_TO]-(p2) RETURN id(p1) AS source, id(p2) AS target',
    {validateRelationships: False})
YIELD
    graphName,
    nodeQuery,
    nodeCount,
    relationshipQuery,
    relationshipCount,
    projectMillis
```

Note how journal/conference is filtered by its community. A parameter could be used instead of the static name. With the community properly cast in the subgraph, the tools from GDS can be implemented. For that, PR is called through the following query:

```
CALL gds.pageRank.stream('db-community', {
  maxIterations: 20,
  dampingFactor: 0.85
})
YIELD nodeId, score
WITH gds.util.asNode(nodeId).title AS name, score
ORDER BY score DESC, name ASC
LIMIT 100
MATCH (p:Article {title: name}), (s:Community {community: 'database'})
MERGE (p)-[:TOP_OF]-(s)
```

The PR is called with a damping factor of 0.85, which is a common value amongst literature. Before explaining the query, PR is briefly described. The general equation is the following:

$$PR_{k+1}(u) = d * \left(\sum_{v \in BL(u)} \frac{PR_k(v)}{out(v)} + \sum_{w \text{is a sink}} \frac{PR_k(w)}{N} \right) + \frac{1-d}{N} \tag{1}$$

where k refers to the ongoing iteration, u is the vertex being updated, $v \in BL(u)$ are the vertices

pointing to u, w refers to those vertices that act as sinks, N is the total number of vertices, and d is the damping factor.

Put simply, the PR algorithm not only takes into account the number of citations an article receives, but also the quality of the origin doing the citation. That is, important articles should receive citations from other important articles. See the original paper for more details [1].

Hence, the results obtained from the PR algorithm are sorted and only the top 100 articles are kept. These are related to the community by means of the relation *:TOP_OF*. As such, the top papers of a community can be easily queried. Then, the authors that have written those articles are tracked and counted. Those that have written in some articles are considered potential reviewers. Those in 2 or more articles can be considered gurus. The gurus are obtained through:

```
MATCH (a:Author)-[:HAS_WRITTEN]-(p:Article)-[:TOP_OF]-(s:Community {
    community: 'database'})
WITH a, count(p) AS articleCount
WHERE articleCount >= 2
RETURN a.name AS authorName, collect(DISTINCT articleCount) AS articleCounts
```

The gurus are: Mihai Datcu, Byonghyo Shim, Kok Yong Lim, L. F. Abbott, Li-Wei Ko, and Jiantao Jiao. The first has written in 3 articles, the rest in 2.

The results of each stage are in accordance to the data fed into the system. The only nuisance worth mentioning, is the fact that a great number of articles received a PR score equal to the damping factor, which means the citations within the community are not that high. Again, this makes sense due to the citations being uniformly distributed among different communities. Recall, only the year was taken into account to skew its distribution.

D Graph algorithms

The main goal of this section is to enhance the skills on using graph algorithms for querying graph data. To accomplish this task, we have selected the Louvain algorithm for community detection and the Node Similarity algorithm for measuring similarity.

D.1 Node similarity

The Node Similarity algorithm compares nodes based on their connections in a bipartite graph (Figure 2). Two nodes are considered similar if they share many of the same neighbors. Given two sets of nodes, the Node Similarity computes the *Jaccard* and *Overlap* similarity metrics.

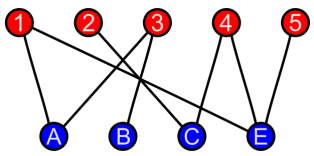

Figure 2: Example of bipartite graph

In the project context, the sets consist of two node types: `Article` and `Keyword`. The comparison between `Article` nodes will be based on their connections with the `Keyword` nodes. The similarity score of articles will be determined by the number of common keywords they share. Articles with a higher number of common keywords will have a higher similarity score.

```
CALL gds.graph.project('bigraph',['Article','Keyword'],'IS_ABOUT');
```

```
CALL gds.nodeSimilarity.stream('bigraph')
YIELD node1, node2, similarity
WHERE gds.util.asNode(node1).title < gds.util.asNode(node2).title
RETURN distinct gds.util.asNode(node1).title AS Article1, gds.util.asNode(
    node2).title AS Article2, similarity
ORDER BY similarity DESCENDING, Article1, Article2
```

This algorithm has time complexity $O(n^3)$ and space complexity $O(n^2)$, and a *topK* parameter can be set to limit memory usage, but it wasn't necessary.

The Table 1 illustrates the most similar articles that were obtained computing the code above.

Article1	**Article2**	**similarity**
A Model Selection Approach for Corruption Robust Reinforcement Learning.	A semantic framework for data retrieval in large remote sensing databases.	0.6667
A Novel Content Caching and Delivery Scheme for Millimeter Wave Device-to-Device Communications.	Combination Synchronization of Three Identical or Different Nonlinear Complex Hyperchaotic Systems.	0.6667
A Survey on Neural Recommendation: From Collaborative Filtering to Content and Context Enriched Recommendation.	Nonparametric Bayesian matrix factorization for assortative networks.	0.6667

Table 1: Top 3 pairs of articles with highest similarity score.

D.2 Louvain

The Louvain algorithm maximizes a modularity score for each community to detect communities in large networks. The score measures the density of connections within a community compared to how connected they would be in a random network.

In the context of the project, the Louvain algorithm is used to identify the communities of `Article` nodes based on citations, hence we are using only the *CITES_TO* edge. Thus, if citations are used correctly, i.e., to mention ideas that are truly used in the article, articles from the same community should be more or less on the same topic.

Running the algorithm with the provided code produces 62 clusters (community IDs) and a list of articles with their respective community IDs. The Table 2 shows an example of the query output with three articles belonging to the same community (communityID = 758).

Articles	**communityId**
GAN-Based SAR-to-Optical Image Translation with Region Information.	758
Gaussian Process Classification Bandits.	758
Guaranteed Contraction Control in the Presence of Imperfectly Learned Dynamics.	758

Table 2: Sample of related articles belonging to the communityId 758

The algorithm is recursively merging communities into a single node and also executes modularity clustering on the condensed graphs. This modularity ranges from -1 to 1, where a higher value indicates a stronger community structure.

The result of the modularity with this algorithm was of 0.3065 which suggests that the clustering has a moderate level of quality, meaning that the article communities identified by the algorithm are

more densely connected within themselves than with the rest of the network, based on the citations that means that there is a high percentage in self-citation.

```
CALL gds.graph.project('graph','Article',{CITES_TO: {orientation: '
    UNDIRECTED'}});
CALL gds.louvain.stream('graph')
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).title AS Article, communityId
ORDER BY communityId ASC, Article ASC;
CALL gds.louvain.stats('graph')
YIELD communityCount
```

The worst-case runtime complexity per iteration of the algorithm is $O(max\{\frac{M+n\cdot\lambda}{p}, \lambda_{\max}\})$, where p denotes the number of processing cores, λ is the average degree of a vertex and $\lambda_{\max}$ is the maximum degree of a vertex. The space complexity is $O(m+n)$.

References

[1] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999. URL: http://ilpubs.stanford.edu:8090/422/.