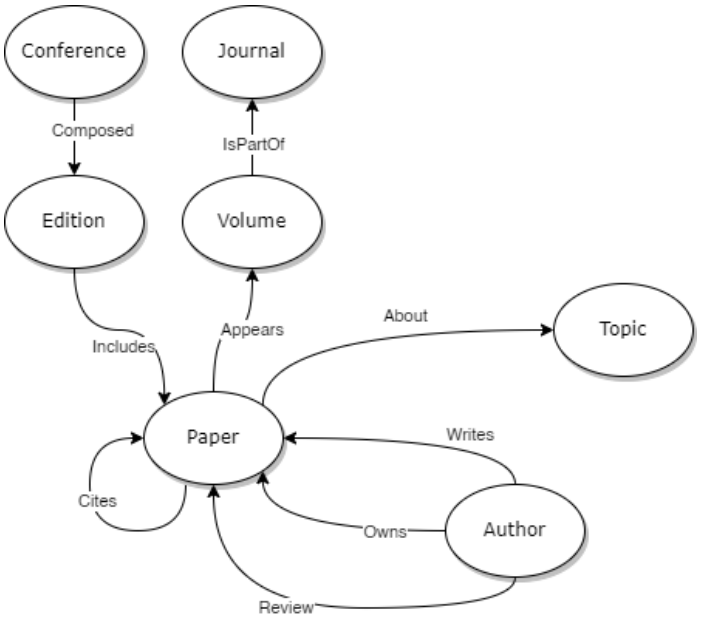
SEMANTIC DATA MANAGEMENT - LAB1 (Property graph)

Niccolò Morabito
Andrés Espinal

A.1 Modeling



Nodes

- author: *id, name, email*
- topic: *id, topic*
- paper: *id, title, year, language, isbn, abstract*
- conference: *id, name*
- volume: *id, title*
- edition: *id, name, number, city*
- journal: *id, name*

Edges

- paper_about_topic
- paper_cites_paper
- author_writes_paper
- author_owns_paper
- author_reviews_paper
- conference_composed_edition
- volume_ispart_journal
- edition_includes_paper
- paper_appears_volume

Justifications:

- *year* was encoded as a property in nodes *edition* and *volume* instead of as a new node because it didn't seem relevant for the queries, doesn't appear to require constant updates and also simplified the model.
- *abstract* was also encoded as a property of node *paper* because each paper has one abstract and that abstract can be only related to one paper
- Mapping *proceeding* as a node didn't provide any value for the queries, so was discarded from the model
- We are assuming constraints regarding *numreviewers = 3* and *author can't be a reviewer of his paper* are enforced in the application side, hence they are not checked in the graph

A.2 Instantiating/Loading

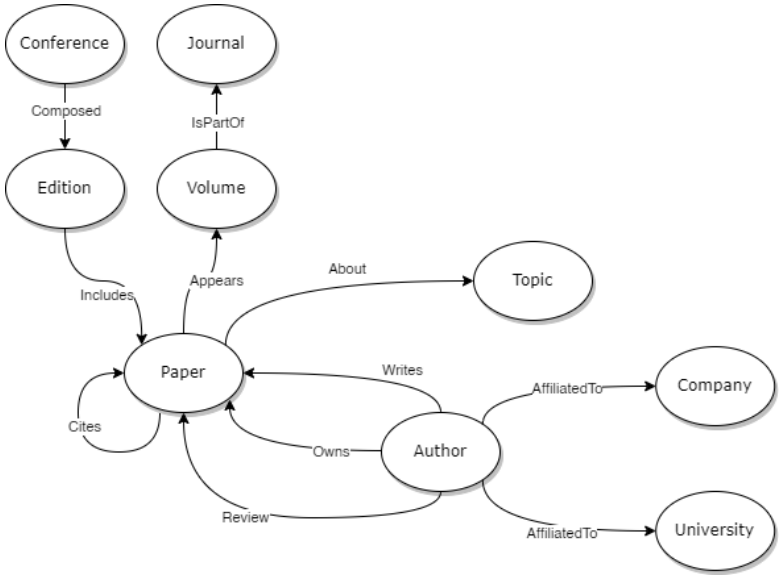
The dataset used was extracted from Citation Network Dataset. The source CSV files were wrangled and generated using a Python script (see DataWranglingResearchDB.py file). Some entities *(Volume, Journal, Conference, Edition, University, Company* for instance*)* didn't exist in the original dataset and alternative mechanisms to generate random data were used for this purpose. A combination of Mockaroo and random sampling of pandas series was used to generate any missing nodes, edges and properties required for the lab. Additionally, the distribution of values for some of this data was intentionally skewed to illustrate the scenarios specified by the queries, which means not all the data of the graph is entirely realistic. After these CSV datasets were generated, *LOAD CSV* functionality in Neo4j was used to load all of the data into the database (see PartA.2_EspinalMorabito.py file).

A.3 Evolving the graph

Justifications:

- In order to store the reviews sent by each reviewer, some properties were added to the relationship *review* edge between nodes *author* and *paper*. The properties added were the *review* (textual description), *timestamp* and *decision* (boolean).
- Moreover, since now every conference or journal may have a different policy on the number of reviewers per paper, property *numreviewers* was added to the *Journal* and *Conference* nodes
- Finally, since an author can be now affiliated with a company or a university, a node was added for both the entities and a relationship *affiliatedto* was created to connect them

The final version of the diagram is therefore:



B Querying

The requested queries are:

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

```
MATCH (paper: paper) <-[includes]- (edition) <-[Composed]- (conf: conference)
MATCH (paper) <-[cites]- (paper2 : paper)
WITH conf, paper, COUNT(distinct paper2) as NumCitations
ORDER BY conf, NumCitations DESC
RETURN conf.name AS conference, COLLECT(paper.title)[..3] AS topThree
```

2. For each conference find its community

```
MATCH (conf: conference)-[composed]-> (e: edition) -[includes]-> (paper) <-[writes]-
(aut: author)
WITH conf, aut, COUNT(DISTINCT e) AS numEditions
WHERE numEditions > 4
RETURN conf.name AS conference, COLLECT(DISTINCT aut.name) AS authors
```

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

```
MATCH (j: journal) <-[is_part_of]- (volume) <-[appears]- (paper) <-[cites]- (citingPaper:
paper)
WITH j, toInteger(citingPaper.year) AS citationYear, COUNT(distinct citingPaper) as
citingPapers
ORDER BY j.name, citationYear
MATCH (j) <-[is_part_of]- (volume) <-[appears]- (citablePaper: paper)
WITH j, citationYear, citingPapers,
toInteger(citablePaper.year) AS publicationYear, citablePaper
WHERE publicationYear = citationYear-1
OR publicationYear = citationYear-2
RETURN j.name AS journal, citationYear, citingPapers AS numCitations,
COUNT(distinct citablePaper) AS numCitablePapers,
toFloat(citingPapers) / COUNT(distinct citablePaper) AS impactFactor
ORDER BY j.name, citationYear
```

4. Find the h-indexes of the authors in your graph (see https://en.wikipedia.org/wiki/H-index, for a definition of the h-index metric): The *h*-index is the largest number *h* such that h articles have at least *h* citations each.

```
MATCH (aut: author) -[Writes]-> (p: paper)
OPTIONAL MATCH (p: paper) -[cites]-> (p2: paper)
WITH aut, p, COUNT(DISTINCT p2) as numCitations
ORDER BY aut, numCitations DESC
WITH aut, COLLECT(numCitations) AS numCitationsArray,
SIZE(COLLECT(numCitations)) AS numPapers
RETURN aut.name AS author, SIZE([i IN RANGE(1, numPapers) WHERE i <=
numCitationsArray[i-1]]) AS h_index
ORDER BY h_index DESC
```

C Graph algorithms

Algorithm 1: **PageRank**

Use Case Scenario: Every year the National Research Institute (NRI) gathers statistics related to paper citations done in the research field. The authors of the most cited paper are awarded with an NRI recognition and a €10000 prize. NRI has hired you as a data analyst, to identify which was the most cited paper of the year.

Implementation in Neo4J

1. The first step is to create a named graph representing a paper citing another paper

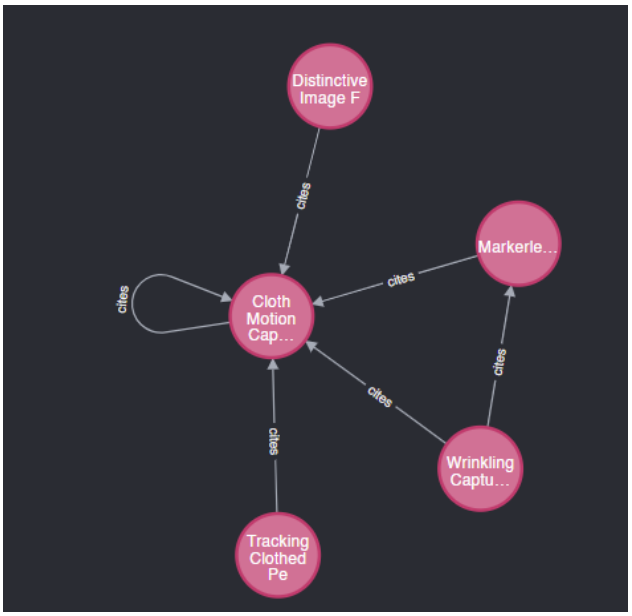
```
CALL gds.graph.create('PaperCitesPaper', 'paper', 'cites')
```

2. The next step is to estimate the memory being used for the algorithm execution.

```
CALL gds.pageRank.write.estimate('PaperCitesPaper', {
 writeProperty: 'pageRank',
 maxIterations: 20,
 dampingFactor: 0.85
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

This returns the following table, as seen even though the amount of nodes and relationships is considerable, the amount of memory required to run the algorithm is negligible in a modern system:

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
100000	27870	2413288	2413288	"2356 KiB"

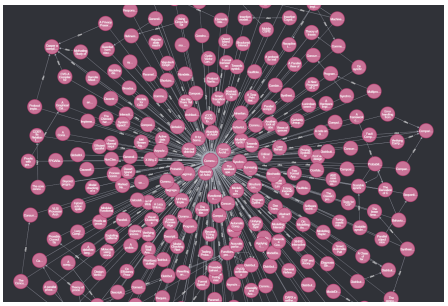


3. The algorithm requires two parameters *maxIterations* which is 20 by default, and *dampingFactor* which is 85 by default. The first parameter indicates the number of times the algorithm will run while the second indicates a dampening factor that defines the probability of picking a related node randomly. After visualizing what is going on with the paper called *Cloth Motion Capture*, we find something interesting, this paper is citing itself. This is a common problem in the PageRank algorithm called **Rank Sink**. To fix this we calibrate the *dampingFactor* to a value that removes this node as the first occurrence, the cutoff that worked for this is *58*. After

running the algorithm with this value, the paper with the loop is no longer selected as the most important.

```
CALL gds.pageRank.stream('PaperCitesPaper', {maxIterations: 20,dampingFactor: 0.58})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS paper_id, gds.util.asNode(nodeId).title AS
paper_title, score AS pagerank_score
ORDER BY pagerank_score DESC
```

As shown on the top 10 results, paper *Communicating Sequential Processes* is now on top and values appear more evenly distributed than before.



paper_id	paper_title	pagerank_score
53e99808b7602d970201a17e	Communicating sequential processes	75.69138261
53e99808b7602d970201b87e	Cloth Motion Capture	74.10306012
53e9986eb7602d97020ab93b	Distinctive Image Features from ScaleInvariant Keypoints	52.201689
53e9983db7602d9702065035	Latent dirichlet allocation	38.72673156
53e99842b7602d970206b4d5	Modeling annotated data	25.97264826
53e99822b7602d970204503f	SupportVector Networks	24.41625483
53e99884b7602d97020bca3f	A simple transmit diversity technique for wireless communic	24.38646074
53e9983db7602d97020698e6	Light field rendering	17.51884288
53e99803b7602d9702015987	Foundations of Databases	17.26400275

Let's now inspect how the graph looks for this node. As shown below the top paper has a vast amount of citations. This is a massive difference from the top node selected before.

Conclusion: Upon further examination and explanation by the data analyst, NRI decides to award the authors of the paper called '*Communicating Sequential Processes'*.

Algorithm 2: **Similarity**

Use Case Scenario: Amanda is a researcher working on her master thesis at university. Looking for reference material to support some of her ideas she found a really nice paper, but this paper is lacking several pages with tables providing important evidence to support her hypothesis. This is a well-studied topic and Amanda knows another researcher might have included these tables on other papers like the one she is reviewing. Amanda is against the clock and must deliver her master thesis tomorrow, As such, she needs to find 10 related papers that might contain this supporting evidence.
Implementation in Neo4J

1. The first step is to create a graph in memory to represent *paper is about topic*, to do that we run:

```
CALL gds.graph.create('PaperAboutTopic', ['paper', 'topic'], 'about')
```

2. After we have our graph in memory we estimate the cost of executing the algorithm in memory. For this we execute:

```
CALL gds.nodeSimilarity.stream.estimate('PaperAboutTopic', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
```

Which results in the following table:

nodeCount	relationshipCount	bytesMin	bytesMax	requiredMemory
145229	654419	48815296	56948120	"[46 MiB ... 54 MiB]"

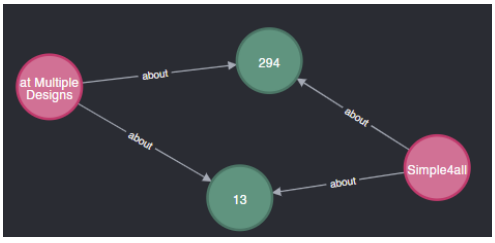
As seen above, the amount of memory required is negligible for a modern system, even if complexity is quadratic on the amount of nodes.

3. Given we have the conditions to run the algorithm we run it through:

```
CALL gds.nodeSimilarity.stream('PaperAboutTopic')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).id AS Paper1Id, gds.util.asNode(node2).id AS Paper2Id,
similarity AS SimilarityScore
ORDER BY SimilarityScore DESCENDING, Paper1Id, Paper2Id
```

As seen on the top 10 similar papers the Similarity Score equals to 1. This means that this papers talk exactly about the same topics.

Paper1Id	Paper2Id	SimilarityScore
53e99792b7602d9701f54b94	53e997ecb7602d9701fe99ee	1
53e99792b7602d9701f54b94	53e997ecb7602d9701fea0ca	1
53e99792b7602d9701f54b94	53e997f5b7602d9701ffa63d	1
53e99792b7602d9701f54b94	53e99800b7602d970200c103	1
53e99792b7602d9701f54b94	53e99808b7602d9702019fcb	1
53e99792b7602d9701f54b94	53e99838b7602d970206056c	1
53e99792b7602d9701f54b94	53e99842b7602d97020718c6	1
53e99792b7602d9701f54b94	53e9984fb7602d9702086777	1
53e99792b7602d9701f54b94	53e998a2b7602d97020d85d4	1



As an example of this, we check the 1st tuple of nodes. Both papers talk about *Computer Science* and *Engineering Drawing*. In the same way, we expect the papers with low similarity scores to be very different from each other. After reordering the results we find that the 10 papers with the smallest Similarity Scores are the following. **Conclusion:** After analyzing the data you were able to provide a list of 10 highly similar papers, both papers talk about *Computer Science* and *Engineering Drawing*.

D Recommender

1.

```
MATCH (p: paper) -[about]-> (t: topic)
WHERE toLower(t.name) IN ['data management', 'indexing', 'data modeling', 'big data',
'data processing', 'data storage', 'data querying']
```

```
SET p.community = "database"
RETURN p
```

2. For conference:

```
MATCH (c: conference) -[composed]-> (edition) -[includes]-> (p: paper)
WHERE p.community = "database"
WITH c, COUNT(DISTINCT p) AS totalDatabaseCommunity
MATCH (c) -[composed]-> (edition) -[includes]-> (p_: paper)
WITH c, totalDatabaseCommunity, COUNT(DISTINCT p_) AS total
WHERE toFloat(totalDatabaseCommunity) / total > 0.9
SET c.community = "database"
RETURN c
```

For journal:

```
MATCH (j: journal) <-[ispartof]- (volume) <-[appears]- (p: paper)
WHERE p.community = "database"
WITH j, COUNT(DISTINCT p) AS totalDatabaseCommunity
MATCH (j) <-[ispartof]- (volume) <-[appears]- (p_: paper)
WITH j, totalDatabaseCommunity, COUNT(DISTINCT p_) AS total
WHERE toFloat(totalDatabaseCommunity) / total > 0.9
SET j.community = "database"
RETURN j
```

3.

```
MATCH (c: conference) -[composed]-> (edition) -[includes]-> (p: paper)
WHERE c.community = "database"
OPTIONAL MATCH (p) <-[cites]- (cp: paper) <-[includes]- (edition) <-[composed]- (cc:
conference)
WHERE cc.community = "database"
WITH p, COUNT(DISTINCT cp) as numCitations
ORDER BY numCitations DESC LIMIT 100
SET p.top = True
RETURN p, numCitations
```

Similarly to the previous point, we need another query for finding the top papers for the journals in the database community.

4.

```
MATCH (a: author) -[writes]-> (p: paper)
WHERE p.community = "database" AND p.top = True
SET a.can_review_database = True
RETURN a
```

And for gurus:

```
MATCH (a: author) -[writes]-> (p: paper)
WHERE p.community = "database" AND p.top = True
WITH a, COUNT(DISTINCT p) AS numTopPapers
WHERE numTopPapers > 1
SET a.database_guru = True
RETURN a, numTopPapers
```