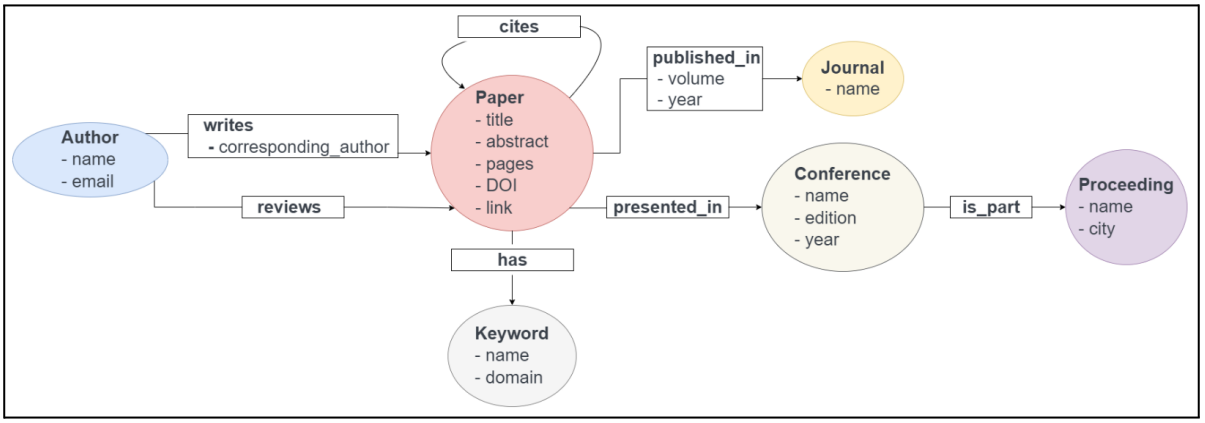
A. Modeling, Loading and Evolving

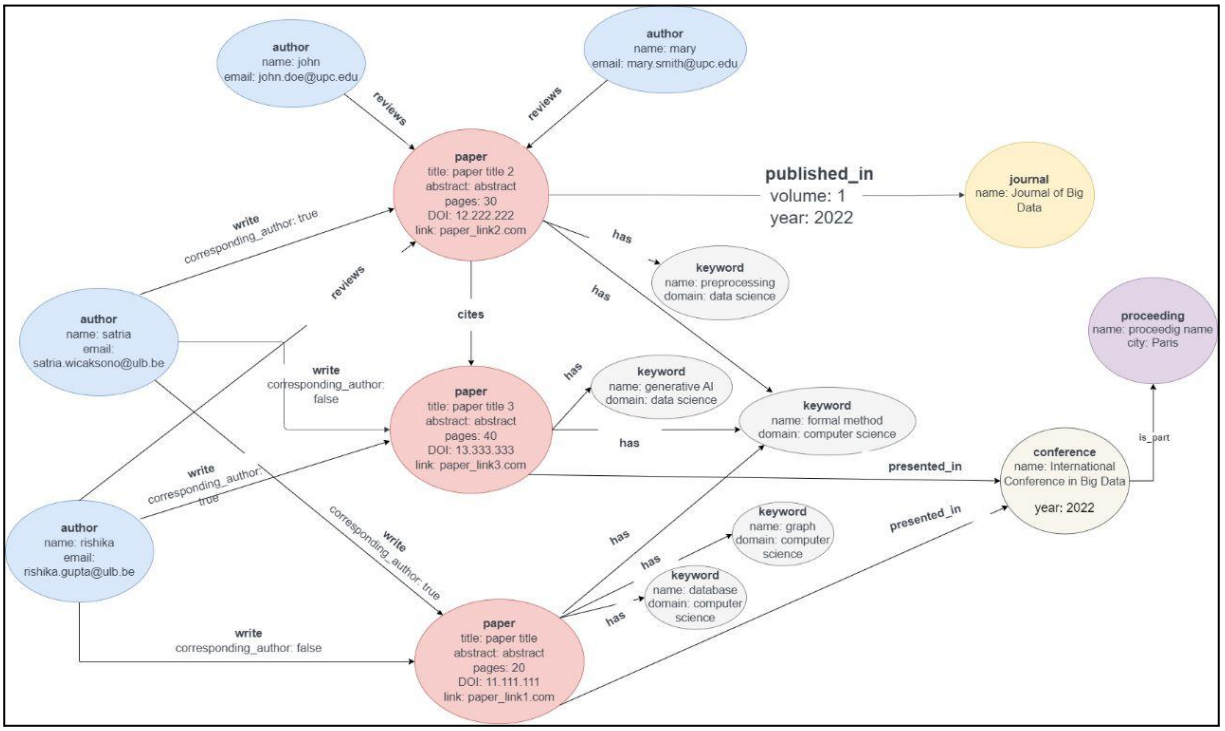
A.1 Modeling

1. Visual representation of metadata-based schema & instance-based schema

Metadata Schema-based:



Instance-based:



2. Justifications for design chosen:
   - Corresponding author: Since, the main author of the paper has the same properties as the author.
   - Reviewer - Since, the reviewer is an author, the properties are that as of any author.
   - Journal: Each journal is modeled as a node since the relation between paper and journal can be represented as edge, with volume and year as the label.

- Conference: Conference is modeled as a node to accommodate the relation with the proceeding. Each conference belongs to one proceeding.
- Topic of paper relates to keywords (hence edge) and keywords (query) hence nodes
- Domain of keywords: We modeled it as a property of a keyword since there is no requirement yet, but the domain is a superset of a keyword.

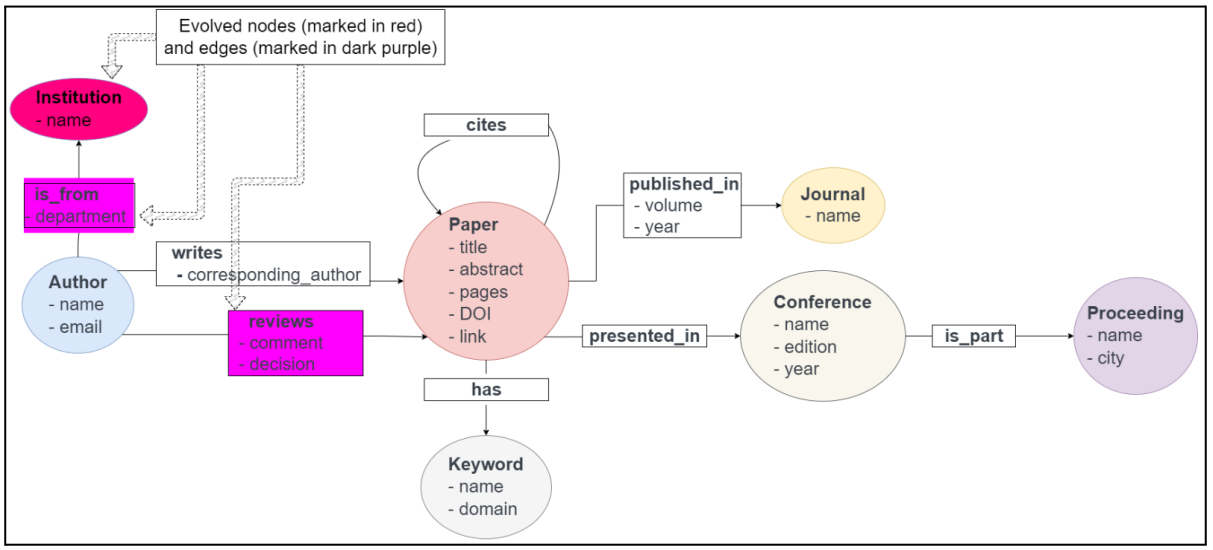
3. Assumption: Corresponding author is false, if the person is a generic author not the main one, and true otherwise.

A.2 Instantiating/Loading

1. PartA.2_Gupta_Wicaksono.py file contains the cypher expressions embedded in Python functions used to create, instantiate as well as load data files for the previous section.
   - Dataset used: Semantic Scholar API library for python. The dataset generated post-processing is included under data/, which is attached in the deliverables zip.
   - Pre-processing data: We parse the JSON data from the API to the csv format with the required methods defined in PartA.2_Gupta_Wicaksono.ipynb notebook. We parse the JSON and convert it per node. We also model the relationship between the nodes based on the attributes presented on the JSON data.
   - Other data generation: We used several methods like generating random data using faker library and also randomly generating relationships between the nodes if it is not presented in the data.
2. File - session_helper_neo4j.py instantiates a session with neo4j which is used by python programs for running queries, algorithms, etc. throughout the laboratory assignment.
3. File - drop_create_indexes_neo4j.py includes python calls to create indexes for the nodes present in the designed solution. This is done to utilize the full potential of graph databases by faster data access and retrieval. (Indexing commands in cypher are also available in the file - PartA.2_Gupta_Wicaksono.cypher).

A.3 Evolving the graph

The part of the graph highlighted in Dark Purple is a part of evolution. The edge reviews include the properties of comment (review sent by each reviewer) as well as their suggested decision.



Justifications for design chosen:

- Decision: If reviewers accepted a paper it will be true otherwise false
- Institution: Modeled as node since a lot of authors can belong to the same institution

B. Querying

All the queries are compiled into queries_neo4j.py file, wherein the queries can be run directly using a Python interface.

1. Find the top 3 most cited papers of each conference (*Notes: We are obtaining results only for those conferences wherein all the 3 papers have data (primarily title) and that the title is not empty.)*

```
MATCH (p:Paper) - [r1:cites] -> (citedPaper:Paper) - [r2:presented_in] -> (c:Conference)
WITH c, citedPaper, count(r1) AS numberOfCitations
ORDER BY c, numberOfCitations DESC
WITH c, COLLECT(citedPaper)[0..3] AS top3
WHERE top3[1].title <> 'None' AND top3[2].title <> 'None'
RETURN c.name + " " + c.year as conference,
            top3[0].title AS topCitedPaper1,
            top3[1].title AS topCitedPaper2,
            top3[2].title AS topCitedPaper3;
```

2. For each conference & its community: i.e., those authors that have published papers on that conference in, at least, 4 different editions. (*Notes: We are the community of authors who published papers in the conference in atleast 4 different editions.*)

```
MATCH (a:Author) - [:writes] -> (p:Paper) - [:presented_in] -> (c:Conference)
WITH c.name as conferenceName, a, COUNT(DISTINCT c.edition) AS distinctEditions
WHERE distinctEditions >= 4
WITH conferenceName, a
RETURN conferenceName, COLLECT(a.name) as community;
```

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

```
MATCH(p:Paper) - [r1:cites] -> (citedP:Paper) - [r2:published_in] -> (j:Journal)
WITH j, r2.year as currYear, COUNT(r1) AS totalCitations
MATCH (p2:Paper) - [r3:published_in] -> (j)
WHERE r3.year = currYear - 1 OR r3.year = currYear - 2
WITH j, currYear, totalCitations, COUNT(r3) AS totalPublications
WHERE totalPublications > 0
RETURN j.name AS journalName,
       currYear AS yearOfPublication,
      toFloat(totalCitations)/totalPublications AS impactFactor
ORDER BY impactFactor DESC;
```

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

```
MATCH(a:Author) - [r1:writes] -> (p1:Paper) - [r2:cites] -> (p2:Paper)
WITH a, p2, COLLECT(p1) as papers
WITH a, p2, RANGE(1, SIZE(papers)) AS listOfPapers
UNWIND listOfPapers AS lp
WITH a, lp AS currHIndex, count(p2) AS citedPapers
WHERE currHIndex <= citedPapers
RETURN a.name AS authorName,
        currHIndex as hIndex
ORDER BY currHIndex DESC;
```

C. Recommender

All the queries are compiled into recommender_neo4j.py file, wherein the queries can be run directly using Python.

1. Create and define research communities

```
MATCH (n: ResearchCommunity) DETACH DELETE n;
MERGE (rc:ResearchCommunity {name:"Databases"});
MATCH (k:Keyword)
WHERE k.name IN ['Data Management', 'Indexing', 'Data Modeling', 'Big Data', 'Data Processing', 'Data Storage','Data Querying']
WITH k
MATCH (rc:ResearchCommunity {name:"Databases"})
MERGE (k) - [r1:belongs_to] -> (rc)
RETURN k AS keyword,
      r1 AS belongs_to,
      rc AS researchCommunity;
```

2. Find conferences/journals related to the research community (*Note: We are considering papers which have data (specifically title) with 90% or more than papers include atleast one keyword from the Databases community (has*1..))*

```
MATCH (rc:ResearchCommunity {name:"Databases"}) <- [:belongs_to] - (k:Keyword) <- [:has*1..] - (cp:Paper) - [r1] -> (x)
WHERE x:Journal OR x:Conference AND x.title <> 'None' AND r1 IN ['published_in','presented_in']
WITH cp, COUNT(DISTINCT cp) AS numberOfCommunityPapers
MATCH (p:Paper) - [r2] -> (x)
WHERE x:Journal OR x:Conference AND r2 IN ['published_in','presented_in']
WITH p, cp, x, numberOfCommunityPapers, COUNT(DISTINCT p) AS numberOfPapers
WHERE (toFloat(numberOfCommunityPapers) / (numberOfPapers)) >= 0.9
```

```
RETURN p.title AS paperName,
   labels(x)[0] AS confJour,
   x.name AS nameOfConfJour;
```

3. Find top papers of these conferences/journals using the highest page rank provided the number of citations from the papers of the same community.

```
CALL gds.graph.drop('papers_belongingTo_databases_community',false);
CALL gds.graph.project.cypher('papers_belongingTo_databases_community',
'MATCH (rc:ResearchCommunity {name:"Databases"}) <- [:belongs_to] - (k:Keyword) <-
[:has*1..] - (cp:Paper) - [r1:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH cp, COUNT(DISTINCT cp) AS numberOfCommunityPapers
MATCH (p:Paper) - [r2:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH p, cp, x, numberOfCommunityPapers, COUNT(DISTINCT p) AS numberOfPapers
WHERE (toFloat(numberOfCommunityPapers) / (numberOfPapers)) >= 0.9
RETURN id(p) AS id;',
'MATCH (p1:Paper) - [:cites] -> (p2:Paper)
RETURN id(p1) AS source,
          id(p2) AS target',
       {validateRelationships:FALSE});

CALL gds.pageRank.write('papers_belongingTo_databases_community', {
   maxIterations: 20,
   dampingFactor: 0.85,
   writeProperty: 'pagerank'
   });

MATCH (rc:ResearchCommunity {name:"Databases"}) <- [:belongs_to] - (k:Keyword) <-
[:has*1..] - (cp:Paper) - [r1:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH rc, cp, COUNT(DISTINCT cp) AS numOfCommunityPapers
MATCH (p:Paper) - [r2:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH rc, p, cp, x, numOfCommunityPapers, COUNT(DISTINCT p) AS numberOfPapers
WHERE (toFloat(numOfCommunityPapers) / (numberOfPapers)) >= 0.9
WITH rc, p
ORDER BY p.pagerank DESC
WITH rc, COLLECT(DISTINCT p) AS p
RETURN rc.name, p[0..100] AS top100Papers;
```

4. Find potential reviewers (guru authors) who can review top conferences.

```
MATCH (rc:ResearchCommunity {name:"Databases"}) <- [:belongs_to] - (k:Keyword) <-
[:has*1..] - (cp:Paper) - [r1:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH rc, cp, COUNT(DISTINCT cp) AS numOfCommunityPapers
MATCH (p:Paper) - [r2:published_in|presented_in] -> (x)
WHERE x:Journal OR x:Conference
WITH rc, p, cp, x, numOfCommunityPapers, COUNT(DISTINCT p) AS numberOfPapers
WHERE (toFloat(numOfCommunityPapers) / (numberOfPapers)) >= 0.9
WITH rc, p
ORDER BY p.pagerank DESC
WITH rc, COLLECT(DISTINCT p) AS p
WITH rc.name as communityName , p[0..100] AS top100Papers
UNWIND top100Papers as topPapers
WITH COLLECT(DISTINCT topPapers) as topPapersList
MATCH (a:Author) - [r3:writes] -> (p:Paper)
WHERE p IN topPapersList
WITH a, COUNT(DISTINCT r3) AS numberOfWrittenPapers
WHERE numberOfWrittenPapers >= 2
RETURN a.name AS potentialReviewerName,
       a.email AS potentialReviewerEmail,
       numberOfWrittenPapers AS guru;
```

D. Graph Algorithms

D.1 Node Similarity Algorithm

Justification: Two nodes are considered similar if they are shared common neighbors together, it is calculated by computing Jaccard Index or Overlap Index between two nodes [1]. For this assignment, this algorithm will be useful to identify which paper is most similar to each other based on their keywords, as nodes (in this case papers) which share similar keywords will most likely be similar to each other.

```
CALL gds.graph.drop('myGraph1',false);
CALL gds.graph.project('myGraph1', ['Paper','Keyword'], 'has');
CALL gds.nodeSimilarity.write.estimate('myGraph1', {
   writeRelationshipType: 'SIMILAR',
   writeProperty: 'score'
});
CALL gds.nodeSimilarity.stream('myGraph1')
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).title AS Paper1,
   gds.util.asNode(node2).title AS Paper2,
   similarity
ORDER BY similarity DESC;
```

"Paper1"	"Paper2"	"similarity"
"Presentation and validation of the Radboud Faces Database"	"Data Science"	1.0

D.2 Betweenness Centrality Algorithm

Justification: Betweenness algorithm measures the influence of nodes in the terms of flow of information, it will calculate the shortest path between nodes and measure which node was visited the most [2]. Betweenness centrality is useful to give an idea how useful a node is in a network [3]. Given the context of the assignment, it is useful in terms of paper citation to know which one is considered as an influential paper, as it will measure how many times the paper is involved in the chain of citation. Undirected graph is used instead of directed graph since we only want to know how many times a node is involved in a citation chain.

```
CALL gds.graph.drop('myGraph2',false);
CALL gds.graph.project('myGraph2', 'Paper',
        {cites: {orientation: 'UNDIRECTED'}
});
CALL gds.betweenness.write.estimate('myGraph2',
        { writeProperty: 'betweenness' }
);
CALL gds.betweenness.stream('myGraph2')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).title AS title,
        score
ORDER BY score DESC;
```

"title"	"score"
"Descriptive Complexity, Canonisation, and Definable Graph Structure Theory"	604.2730376736839
"Open Data Science"	572.7653095131653
"The COG database: an updated version includes eukaryotes"	569.5128689054188

Other Notes:

- The entire code for the laboratory can be accessed via - https://github.com/risg99/SDM_Lab_1
- Queries on GitHub are restricted with 'LIMIT 5' in most cases, as they were used for testing purposes.
- Each task (from loading (Part A.2) till implementing algorithms (Part D), it's done using Python (that integrates and invokes Cypher commands). Within the deliverables file, python programs corresponding to each task can be located in deliverables/python_programs/task.py

(can have any of .py/.ipynb extensions). Similarly, cypher queries for executing each of the tasks are stored with .cypher extension under deliverables/cypher_codes/task.cypher.

- Indexes are also included in the assignment for enabling quicker query execution times.
- Requirements.txt includes the installation commands for the python libraries used in the assignment.

References:

[1] "Node similarity - neo4j graph data science," *Neo4j Graph Data Platform*. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/algorithms/node-similarity/. [Accessed: 15-Mar-2023].

[2] "Betweenness centrality - neo4j graph data science," *Neo4j Graph Data Platform*. [Online]. Available: https://neo4j.com/docs/graph-data-science/current/algorithms/betweenness-centrality/. [Accessed: 15-Mar-2023].

[3] S. Gago. "The betweenness centrality of a graph". 2007. Available: https://upcommons.upc.edu/bitstream/handle/2117/972/BetGa07.pdf;sequence=7