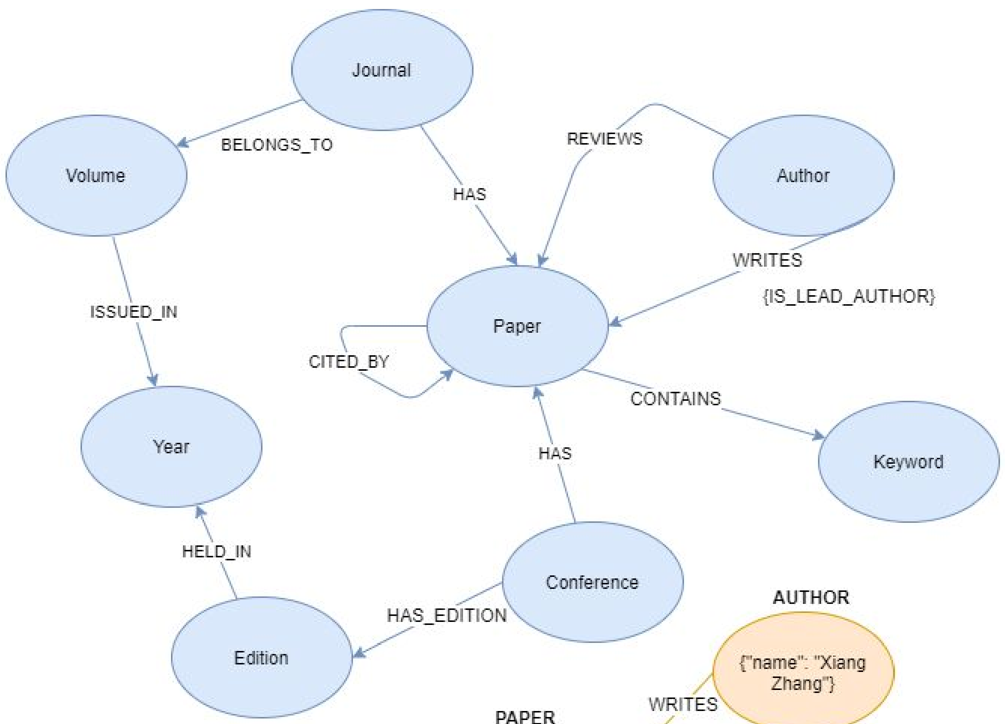
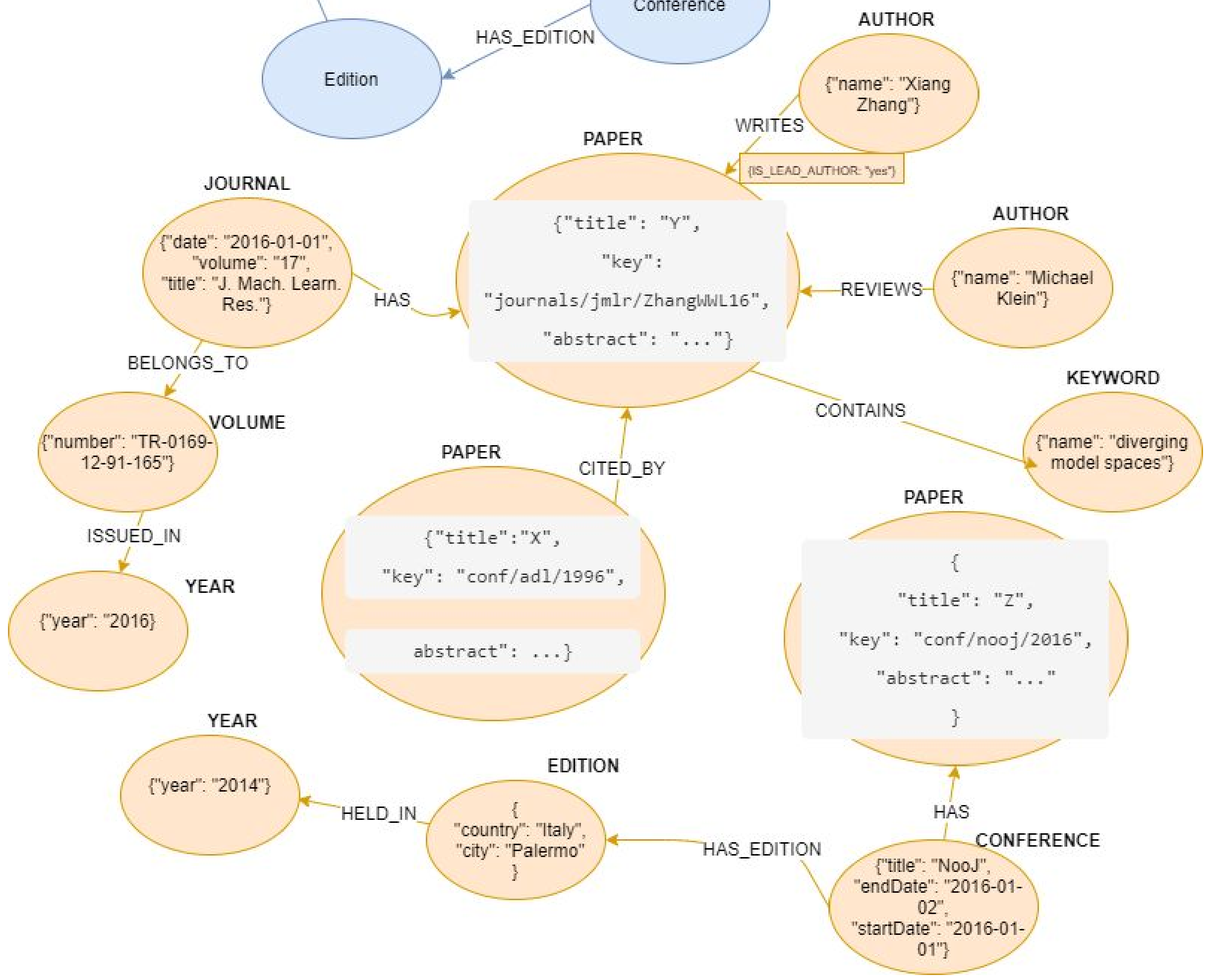
Ariston Harianto Lim
Haroon Rashid

A Modeling, Loading, Evolving

A.1 Modeling





Note:
The metadata is represented by blue circles, and the data is represented by orange circles. For "Paper" nodes, the titles have been changed into a variable to save space in the diagram. Additionally, the edge "IS A" - to connect the metadata to the data - has been omitted for clarity purposes. Instead, we labelled each data node according to its type in **bold.** Some explanations:

- We created a property of "IS_LEAD_AUTHOR" on the "WRITE" edge to indicate the lead authors in case of multiple authors.
- "CONFERENCE" and "JOURNAL" are connected directly to "PAPER". This makes it more efficient for Query 3 and 4.

A.2 Instantiating/Loading

A.2.1: Converting DBLP to CSV

The official website of DBLP database provides data in xml format. This data needs to be converted to CSVs to make use of bulk loading functionality provided by neo4j. For generating CSV files from XML data, the repository from TomHurks was used. The CSV files can be generated by running the following command with the correct xml and output file parameters:

```
XMLToCSV.py xml_filename dtd_filename outputfile
```

The output is a number of csv files which are further processed to extract the relevant information to load into neo4j.

A.2.2: Processing CSV files to extract relevant Data

Processing the XMLs to generate the CSV files resulted in 28 CSVs including the header files. These files contain a lot of redundant data and can be overwhelming to understand it completely. A paper titled DBLP - Some Lessons Learned was very helpful in understanding different pieces of information available in the data. Based on the model designed in the part A1, the relevant attributes present in the data were extracted. A script is written that takes the output from part A.2.1 and produces a new set of CSV files with minimal data.

Following is an overview of how the data is pre-processed to make it ready for neo4j loading:

1. `conferences.csv`: `booktitle` as conference name and year are extracted from `output_inproceedings.csv`. Non-numerical values of year are ignored and NULL values in both of the columns are removed.

2. `journals.csv` : Journals have been extracted from `output_articles.csv` by picking columns `journal`, `volume` and `year`. Non-numerical values of year are ignored and NULL values in both of the columns are removed.

3. `conference_papers.csv`: Conference papers have been picked from `output_proceedings.csv` from columns `key`, `title`, `booktitle` and `year`. Additionally, keywords and abstract are randomly generated and added to the csv as separate columns. abstract is lorem ipsum paragraph, while keywords are extracted as nouns from title to which the database keywords are appended randomly to obtain meaningful results for the Part D recommender.

4. `conf_venues.csv:` Conference venues are generated from the title of the conference by extracting the city and the country using python's geotext library.

5. `conference_authors.csv`: Authors have been extracted from `output_inproceedings.csv` files. Authors are further divided into lead authors and coauthors for each paper. Journal authors have been extracted using the same strategy from `output_article.csv`.

6. `conference_reviewers.csv` : To generate reviewers for the paper, three authors are randomly selected and assigned to the paper. Furthermore, a random text description of the review is also added that will be used to evolve the graph in the next section.

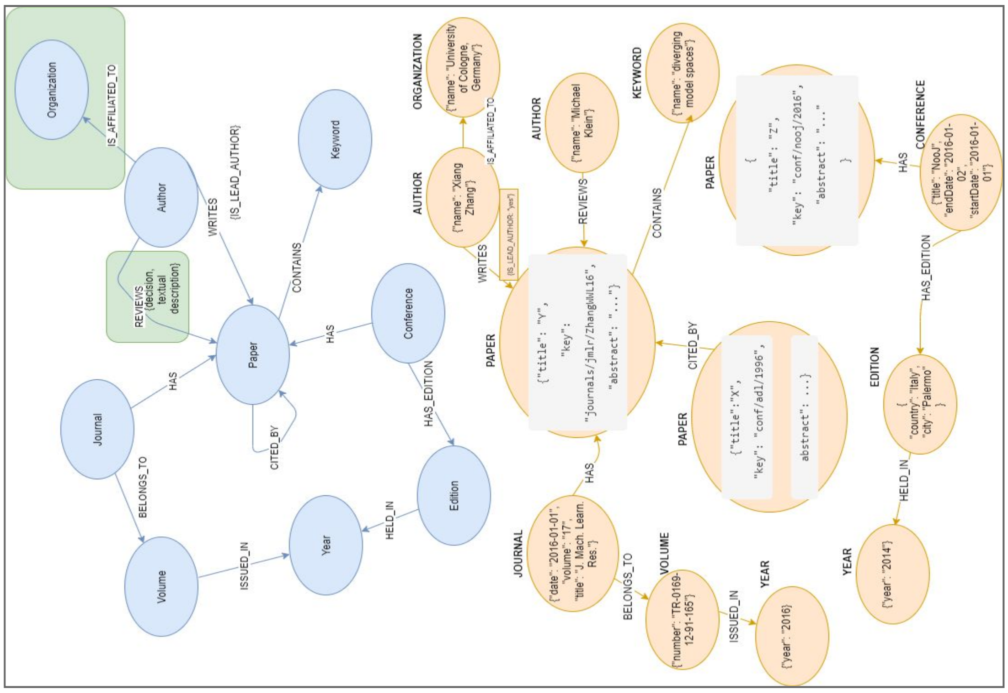
7. `authors_affiliations.csv` : The author's affiliation is extracted from `output_schools.csv`. Each author is assigned a random school as his/her affiliation.

A ready to go script is provided with the project submission.

A.2.3: Loading Data into the Neo4J graph

Data is loaded into Neo4j using the Neo4j Python driver provided by Neo4j. LOADCSV is used to load the data and the MERGE, MATCH AND UNWIND are used where needed to load data into nodes and edges of the graph. The project submission provides the ready-to-go script to load data into Neo4j.

A.3 Evolving the Graph


Organization
IS_AFFILIATED_TO
Author
WRITES
{IS_LEAD_AUTHOR}
Keyword
REVIEWS {decision, textual description}
CONTAINS
Paper
HAS
Conference
Journal
HAS
CITED_BY
HAS_EDITION
Edition
BELONGS_TO
HELD_IN
Year
Volume
ISSUED_IN
ORGANIZATION
{"name": "University of Cologne, Germany"}
AUTHOR
{"name": "Xiang Zhang"}
AUTHOR
{"name": "Michael Klein"}
KEYWORD
{"name": "diverging model spaces"}
PAPER
CONFERENCE
{"title": "NooJ", "endDate": "2016-01-02", "startDate": "2016-01-01"}
PAPER
PAPER
EDITION
{"country": "Italy", "city": "Palermo"}
JOURNAL
{"date": "2016-01-01", "volume": "17", "title": "J. Mach. Learn. Res."}
VOLUME
{"number": "TR-0169-12-91-165"}
YEAR
{"year": "2016"}
YEAR
{"year": "2014"}

The evolved graph has been shown in Figure A.3 - with the evolved nodes and relationships shown in green boxes. The graph has been evolved to include review as textual description as well as the decision of the review. Furthermore, the author's affiliation has been established by introducing a new node Organization and adding an "IS_AFFILIATED_TO" edge to establish a connection. Another change is introducing some properties to the edge "Reviews". We added the variable "decision" to indicate the type of reviews received by each paper.

B Querying

Query 1

```
MATCH (a:Author)-[w:WRITES]->(p:Paper)<-[cb:CITED_BY]-(c:Paper)
WITH a.name AS authorName, p.title AS paperName, count(c.title) AS nCitation
ORDER BY authorName, nCitation DESC
WITH authorName, collect(nCitation) AS citation
UNWIND range(0, size(citation)-1) AS position WITH authorName,
CASE WHEN citation[position] <= (position+1)
    THEN citation[position]
    ELSE (position+1)
END AS index
RETURN authorName, max(index) AS hIndex
```

Query 2

```
MATCH (c:Conference)--(p:Paper)<--(p2:Paper)
WITH c.name AS confName, p.title AS paperName, count(p2.title) AS nCitation
ORDER BY confName, nCitation DESC
WITH confName, collect(paperName) AS citation
RETURN confName, citation[..3] AS Top3CitedPapers
```

Query 3

```
MATCH (y:Year)--(e:Edition)--(c:Conference)-->(p:Paper)<--(a:Author)
WITH c.name as confName, a.name AS authorName,
     count(distinct y.year) AS nYear
WHERE nYear >= 4
RETURN confName, collect(authorName) AS authorNames
```

Query 4

```
MATCH
(y:Year)--(v:Volume)--(j:Journal)--(a:Author)--(p:Paper)<-[:CITED_BY]-(p2:Paper)
WITH j.title as journalName, a.name as authorName, y.year as jYear,
     count(p2.title) as citation
WHERE jYear in [(date().year-1), (date().year-2)]
RETURN journalName,  sum(citation)/count(authorName) as ImpactFactor
ORDER BY ImpactFactor DESC
```

C Graph Algorithms

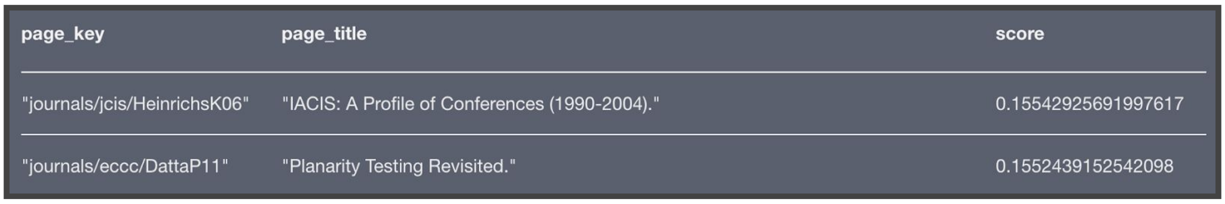
Algorithm 1: ArticleRank (Centrality Algorithm)

The first algorithm chosen for this part is ArticleRank, which is a variant of the famous PageRank Algorithm. Where ArticleRank differs from PageRank is that PageRank assumes that relationships from nodes that have a low out-degree are more important than relationships from nodes with a higher out-degree. ArticleRank weakens this assumption.

```
AR(A) = (1-d) + d (AR(T1)/(C(T1) + C(AVG)) + ... + AR(Tn)/(C(Tn) + C(AVG))
```

where **C(A)**: num. of links going out of Page A and **C(AVG)**: Avg. numLinks going out for all pages.

```
CALL algo.articleRank.stream('Paper', 'CITED_BY', {iterations:20,
dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.asNode(nodeId).key AS page_key, algo.asNode(nodeId).title AS
page_title,score
ORDER BY score DESC LIMIT 10;
```



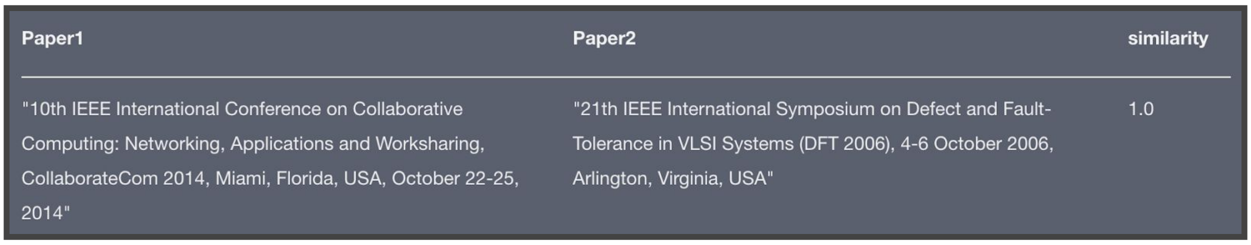
page_key	page_title	score
"journals/jcis/HeinrichsK06"	"IACIS: A Profile of Conferences (1990-2004)."	0.15542925691997617
"journals/eccc/DattaP11"	"Planarity Testing Revisited."	0.1552439152542098

ArticleRank is used to find the relevance of the page based on the number of citations. We applied the ArticleRank on the Paper Node and the relevance is determined by the "CITED_BY" relationship. The results in the above figure shows the top two most relevant papers based on the citations as determined by the ArticleRank algorithm.

Algorithm 2: NodeSimilarity (Graph Similarity Algorithm)

The second algorithm chosen is call NodeSimilarity which belongs to the class of Graph Similarity Algorithms. The algorithm compares the set of nodes based on their connections to other nodes. Two nodes are considered similar if they share many of the same neighbors. NodeSimilarity computes the pairwise similarities based on the Jaccard metric, which is basically intersection over union of two sets.

```
CALL algo.nodeSimilarity.stream('Paper | Keyword', 'MENTIONS', { direction:
'OUTGOING'})
YIELD node1, node2, similarity
RETURN algo.asNode(node1).title AS Paper1, algo.asNode(node2).title AS Paper2,
similarity
ORDER BY similarity DESCENDING, Paper1, Paper2
```



Paper1	Paper2	similarity
"10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2014, Miami, Florida, USA, October 22-25, 2014"	"21th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2006), 4-6 October 2006, Arlington, Virginia, USA"	1.0

For the purpose of this lab, we have decided to find the similar papers based on the keywords using NodeSimilarity Algorithm. The query to perform the algorithm as well as the top result is shown above. The results above show the similarity score of 1.0, which means that the keywords are exactly similar for the papers returned. The Jaccard metric score is 1.0, because intersection of keywords for these two papers is maximum and their union is minimum for all the pairs of Paper nodes in the graph. The algorithm is optimized by ignoring the nodes that are disconnected which results in the faster query response time.

D Recommender

Part 1: Get the Database paper communities

```
MATCH (c)-[:HAS]->(p:Paper)-[:MENTIONS]->(k:Keyword)
WHERE k.name IN {search_keywords}
   WITH c, COLLECT(p.title) as papersList, toFloat(COUNT(p)) AS totalDBPapers
   MATCH (c)-[:HAS]->(p1:Paper)
    WITH c, totalDBPapers, papersList,  toFloat(COUNT(p1)) AS total, (totalDBPapers
/ toFloat(COUNT(p1))) AS ratio
   WHERE ratio > 0.9
   RETURN COLLECT(c.title) as confName, papersList
```

Part 2: Determine the relevance of the paper in the community based on the number of citations, return top 100 most relevant papers

```
CALL algo.pageRank.stream('Paper', 'CITED_BY', {iterations:1, dampingFactor:0.85})
YIELD nodeId, score
WITH algo.asNode(nodeId).title AS page, score
WHERE page IN """ + str(papers) + """
RETURN page, score
ORDER BY score DESC LIMIT 100;
```

Part 3: Return Gurus as the authors of the most relevant papers from Part2

```
Match (a:Author)-[w:WRITES]->(p:Paper)
WHERE p.title IN """ + str(top_papers) + """
WITH a.name as authorNames, count(w) as count
WHERE count >= 2
RETURN collect(authorNames) as Gurus
```

Each part uses the results from the previous query, and the final query returns the top authors in the database community based on the keywords specified.