Report on Graph Databases

Prepared by

Mert Erdemir

Middle East Technical University (METU)
Computer Engineering Department

This report is prepared for Summer Practice held within the project 117E566 supported by TUBITAK.

Ankara, 2018

TABLE OF CONTENTS

I will continue with project information then go on with future plans for the project and finish the report with my conclusion part.

2. INFORMATION ABOUT THE PROJECT

Main purpose of the project is analyzing news data (optionally social media data) by the means of the relationships between the words (such as location, person, organization, and time) and trying to determine the events that happened or can possibly be happen in the future according to that news data.

2.1. PRE-PROJECT PHASE

Before we were informed about our main project, Pinar Karagoz, our supervisor in the METU group of the project, explained the general idea relies behind the research and why it is being examined. She shared common knowledge about the mathematical concepts that we can use in further stages. By this starting point, I started to study on network theory and the mathematical side of it. After we all finished reading project documents provided to us, decided to general weekly planning of my internship. To sum up the phase, I splitted it into different subjects and summarized the information I found.

2.1.1. What is a Graph?

A graph database is a kind of database that represents a mathematical graph which is a directed one. As a definition of a graph, it is simply a collection of elements - typically called Nodes (also called Vertices or Points) - that are joined together by Edges. Each Node represents some piece of information in the Graph, whereas each Edge represents some connection between two Nodes.

2.1.2. What is a Graph Database?

Basically, a Graph Database is simply a Database Engine that models both Nodes and Edges in the relational Graph as first-class entities. This allows us to represent real world problems as data in a a closer fit to what we can draw to papers to explain the problems.

In computing, a Graph Database is a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data. A key concept of the system is the graph (or edge or relationship), which directly relates data items in the store. The relationships allow data in the store to be linked together directly, and in many cases retrieved with one operation.

There are two main category for Graph Databases as design-wise:

- RDF (Resource Description Framework - triple stores - W3C Standard - SPARQL)
- LPG (Labeled Property Graph - node, relations, properties - Generally different query languages.)

In graph databases (in common),

- Nodes represent entities such as people, businesses, accounts, etc.
- Edges, also called as relationships, connects nodes to other nodes. In other words,
- they represent the relationship between nodes.
- Properties are relevant information to nodes such as name, age, height for people.

Common databases that can be used for our research purposes are Neo4j, OrientDB, Apache TinkerPop, Titan and ArangoDB.

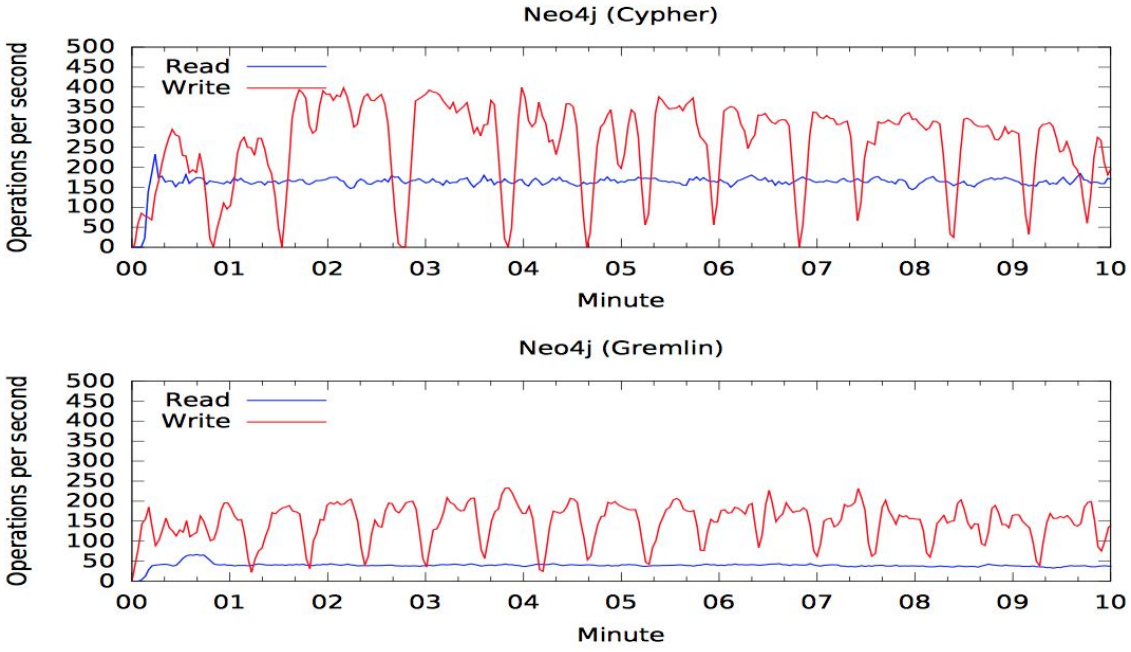
2.1.3. Comparisons Between Graph Databases

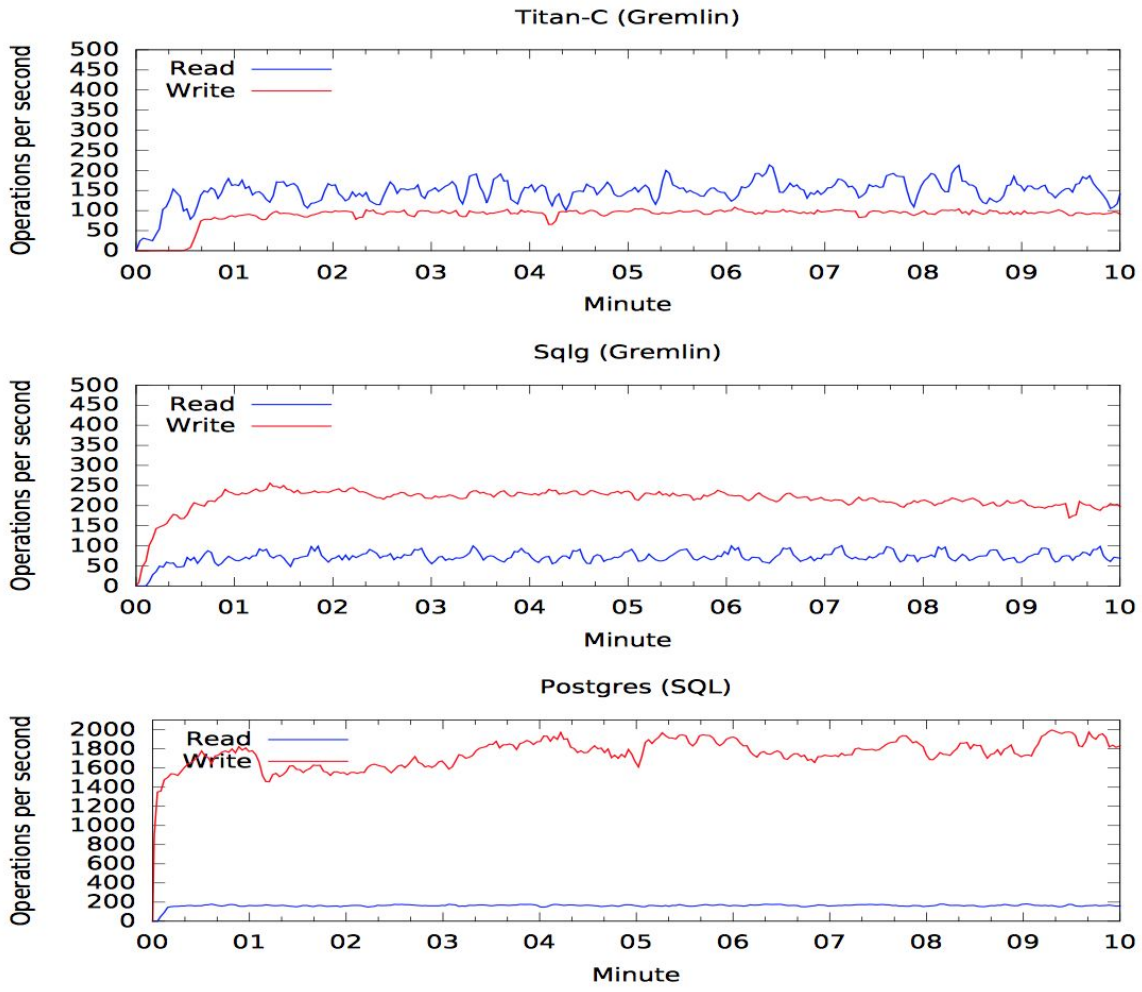
Table 2: Query Latencies in ms — Scale Factor 3

System	Neo4j		Titan-C	Titan-B	Sqlg	Postgres	Virtuoso	
Query Language	Cypher	Gremlin	Gremlin	Gremlin	Gremlin	SQL	SQL	SPARQL
Point lookup	9.08	122	39	65	16.1	0.25	0.35	3
1-hop	12.82	101	240	223	34	1.4	2.15	1.23
2-hop	368	275	439	1271	2526	29	11.55	16.62
Shortest Path	21	4813	10732	13948	10243	2242	4.81	26

Table 3: Query Latencies in ms — Scale Factor 10

System	Neo4j		Titan-C	Titan-B	Sqlg	Postgres	Virtuoso	
Query Language	Cypher	Gremlin	Gremlin	Gremlin	Gremlin	SQL	SQL	SPARQL
Point lookup	11.16	177	42	236	16.9	0.32	0.41	3
1-hop	14.1	377	129	2117	43	1.62	2.22	1.71
2-hop	579	683	1570	12978	4408	46	15.92	52
Shortest Path	16	4053	17379	-	7003	3648	7.09	32





Images are taken from: https://event.cwi.nl/grades/2017/12-Apaci.pdf

From the benchmark results, we can clearly say that Neo4j is better than the other Graph Databases in several ways. It is also an important point that Neo4j is having frequent sudden drops in performance. Using Gremlin is common for most of the Graph Databases. However, it reduces performance and increases execution time significantly.

Table 2: MIW and QW results (sec)

Graph	Workload	Titan	OrientDB	Neo4j
EN	MIW	9.36	62.77	**6.77**
AM	MIW	34.00	97.00	**10.61**
YT	MIW	104.27	252.15	**24.69**
LJ	MIW	663.03	9416.74	**349.55**
EN	QW-FN	1.87	**0.56**	0.95
AM	QW-FN	6.47	3.50	**1.85**
YT	QW-FN	20.71	9.34	**4.51**
LJ	QW-FN	213.41	303.09	**47.07**
EN	QW-FA	3.78	0.71	**0.16**
AM	QW-FA	13.77	2.30	**0.36**
YT	QW-FA	42.82	6.15	**1.46**
LJ	QW-FA	460.25	518.12	**47.07**
EN	QW-FS	1.63	3.09	**0.16**
AM	QW-FS	0.12	83.29	**0.302**
YT	QW-FS	24.87	23.47	**0.08**
LJ	QW-FS	123.50	86.87	**18.13**

Table 3: CW results (sec)

Graph-Cache	Titan	OrientDB	Neo4j
Graph1k-5%	2.39	**0.92**	2.46
Graph1k-10%	1.45	**0.59**	2.07
Graph1k-15%	1.30	**0.58**	1.88
Graph1k-20%	1.25	**0.55**	1.72
Graph1k-25%	1.19	**0.49**	1.67
Graph1k-30%	1.15	**0.48**	1.55
Graph5k-5%	16.01	**5.88**	12.8
Graph5k-10%	15.10	**5.67**	12.13
Graph5k-15%	14.63	**4.81**	11.91
Graph5k-20%	14.16	**4.62**	11.68
Graph5k-25%	13.76	**4.51**	11.31
Graph5k-30%	13.38	**4.45**	10.94
Graph10k-5%	46.06	**18.20**	34.05
Graph10k-10%	44.59	**17.92**	32.88
Graph10k-15%	43.68	**17.31**	31.91
Graph10k-20%	42.48	**16.88**	31.01
Graph10k-25%	41.32	**16.58**	30.74
Graph10k-30%	39.98	**16.34**	30.13
Graph20k-5%	140.46	**54.01**	87.04
Graph20k-10%	138.10	**52.51**	85.49
Graph20k-15%	137.25	**52.12**	82.88
Graph20k-20%	133.11	**51.68**	82.16
Graph20k-25%	122.48	**50.79**	79.87
Graph20k-30%	120.94	**50.49**	78.81
Graph30k-5%	310.25	**69.38**	154.60
Graph30k-10%	301.80	**94.98**	151.81
Graph30k-15%	299.27	**94.85**	151.12
Graph30k-20%	296.43	**94.67**	146.25
Graph30k-25%	294.33	**92.62**	144.08
Graph30k-30%	288.50	**90.13**	142.33
Graph40k-5%	533.29	**201.19**	250.79
Graph40k-10%	505.91	**199.18**	244.79
Graph40k-15%	490.39	**194.34**	242.55
Graph40k-20%	487.31	**183.14**	241.47
Graph40k-25%	467.18	**177.55**	237.29
Graph40k-30%	418.07	**174.65**	229.65
Graph50k-5%	642.42	**240.58**	348.33
Graph50k-10%	624.36	**238.35**	344.06
Graph50k-15%	611.70	**237.65**	340.20
Graph50k-20%	610.40	**230.76**	337.36
Graph50k-25%	596.29	**230.03**	332.01
Graph50k-30%	580.44	**226.31**	325.88

Images are taken from:

https://pdfs.semanticscholar.org/73dd/7060a97f8ae5728ac2533926aee492400261.pdf

The article states that when user uses small graphs (small amount of data) the results are comparable and even it can be said that they are similar for all of the databases. In some cases, like need for successive local queries (OrientDB) or single insertion operations (Titan), databases are beginning to differ from each other. In addition to these differences, Neo4j is the winner by far when user uses big and complex graphs. Neo4j is more efficient for storing and querying graph data. Since we are going to use a lot of nodes/edges due to the nature of the news/event analysis (we need to consider all of the words and their relations in the given texts), Neo4j seems the best option.

NoSQL Performance Bechmark 2018
Absolute & normalized results for ArangoDB, MongoDB, Neo4j and OrientDB

	single read (s)	single write (s)	single write sync (s)	aggregation (s)	shortest (s)	neighbors 2nd (s)	neighbors 2nd data (s)	memory (GB)
ArangoDB 3.3.3 (rocksdb)	100%	100%	100%	100%	100%	100%	100%	100%
	23.25	28.07	28.27	01.08	0.42	1.43	5.15	15.36
ArangoDB 3.3.3 (mmfiles)	102.16%	102.55%	103.89%	102.40%	816.06%	122.07%	99.32%	92.87%
	23.76	28.79	29.37	1.10	3.40	1.75	5.12	14.27
MongoDB 3.6.1 (Wired Tiger)	422.38%	1123.36%	1652.09%	136.65%		518.83%	192.88%	50.64%
	98.24	315.33	466.99	1.47		7.42	9.94	7.70
Neo4j 3.3.1	153.65%		149.37%	203.45%	199.94%	208.96%	214.22%	240.68%
	35.73		43.22	2.18	0.83	2.99	11.04	37.00
PostGres 10.1 (tabular)	231.17%	129.03%	127.70%	29.62%		307.96%	76.87%	26.68%
	53.77	36.22	36.10	0.32		4.41	3.96	4.10
PostGres 10.1 (jsonb)	135.96%	104.34%	101.55%	204.55%		292.57%	126.14%	35.36%
	31.62	29.29	28.70	2.20		4.19	6.50	5.43
OrientDB 2.2.29	198.84%	110.37%		2526.29%	12323.67%	636.45%	400.97%	107.04%
	46.25	30.98		27.19	51.34	9.11	20.67	16.45

Image is taken from:
https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb

This article tries to compare some graph databases, document databases and relational databases performances according to ArangoDB database itself. Benchmarking is done by ArangoDB creator himself. I think that it is not a good data for comparison since it is done with the way that how ArangoDB works, or expected to work with the

best performance. However, I wanted to include this benchmark in my report to leave the ArangoDB-side open for further discussions.

According to articles that I mentioned, the best choice for our needs seems to be Neo4j. It has being kept updated, improved. It is the most known and being used Graph Database in the market. It has more built-in functionalities for algorithms. It also supports Gremlin (a common query languages across different Graph Databases) which still doesn't have a better performance than the original query language, Cypher, for Neo4j. On the other hand, besides better performance for large amount data, licensing or being charged for more nodes can be another handicap. For this problem, we have to determine the domains of our project by the means of nodes, edges, properties, etc.

Also, for supporting more Graph Databases, we can work in two different paths for starting while still using Neo4j. We can test our news data with both Gremlin extension and Cypher language, and see which brings out better performance for our usage.

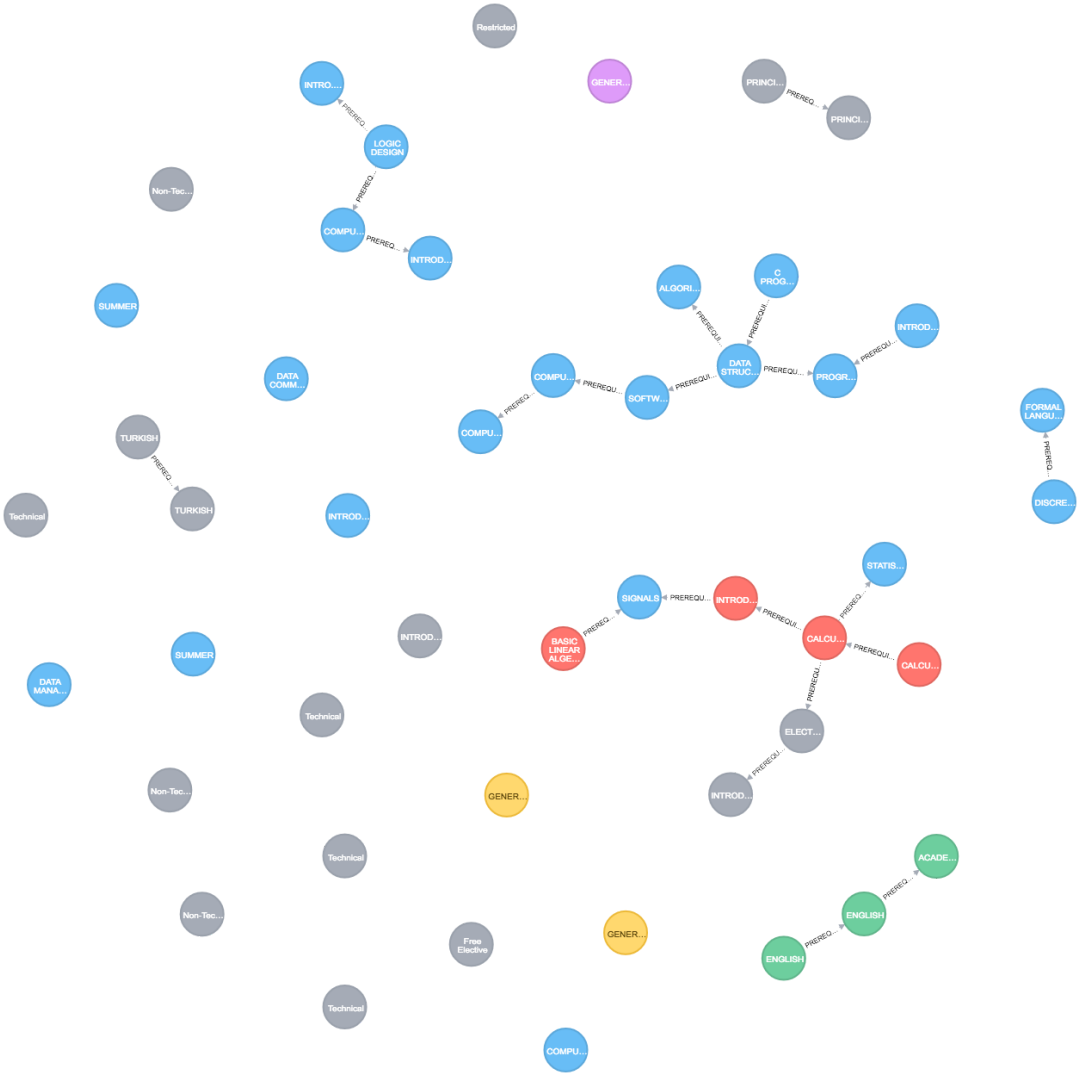
2.2. UNDERSTANDING GRAPH DATABASE PHASE

After selecting Neo4j as our Graph Database, we decided to test it with some information we can create and visualize it. We thought that it can help us to understand how both nodes and relationships relate each other. For this purpose, we choose our department's undergraduate course prerequisite tree whose flowchart representation, created by Sevki Bekir KOCADAG, can be seen as follows:


Middle East Technical University
Computer Engineering Department
Undergraduate Curriculum Prerequisite Flowchart
First Year
Fall:
CENG100
CENG111
MATH119
ENG101
PHYS105
CHEM107
Spring:
CENG140
MATH260
MATH120
ENG102
PHYS106
Second Year
Fall:
CENG213
EE281
MATH219
ENG211
CENG223
HIST2205
HIST2201
Spring:
CENG232
EE282
CENG242
CENG222
CENG280
HIST2206
HIST2202
Third Year
Fall:
CENG331
CENG315
CENG351
CENG300
TURK###
1 Restricted
1 Non-Tech.
Spring:
CENG334
CENG336
CENG350
CENG384
TURK###
1 Non-Tech.
Fourth Year
Fall:
CENG400
CENG435
CENG491
CENG477
Tech. Elec.
1 Non-Tech
Spring:
Tech. Elec.
Tech. Elec.
CENG492
Tech. Elec.
Free Elec.
Powered by Şevki Bekir KOCADAĞ
V2.1
Updated Date: 19.02.2017

In order to form exactly the same representation in graph database, I developed a model that each node represents a course and relationships represents the prerequisite relation between the courses. Starting point of the relationship tells that, ‘this course’ is a prerequisite course of ‘that course’ which is at the ending point of the relationship. Also, I labeled each node, course, according to the department they belong to. For example, since ‘CHEM107’ course belongs to Chemistry Department, it is labeled as ‘Chemistry’.

To access the database with a programming language, I chose Python due to the lack of complexity in scripting. Also, Neo4j has several library options. Since we are still in learning process, I chose ‘py2neo’ library for its easy-to-use feature. At the end of data importing, our model is visualized as follows:



In addition to this graph representation, I tested the graph model by writing a program that returns prerequisite courses for a specific course or returns all of the courses in a specific period of a time. In order to give an example how the script requested related information from the database, I share a function for getting prerequisite information of a course in the next picture. Related information is requested by a Cypher language query.

```
# ********************* Getting Prerequisite Info ************************
'''
This function firstly checks if the database connection is established.
If so, runs a query that finds the node specified with the course_code
and its relationships. In order to get prerequisite chains, hops are
defined as variable length that changes between 1 and 10.
After finding the prerequisite courses these are returned in a table
shaped format.
'''
def get_prerequisite_information(course_code=None):
    global isOpened
    if (isOpened == 1):
        if (course_code == None):
            print ("Please specify a course code!")
            return
        global graph
        query = "MATCH (course {code:'" + course_code + "'})"\
                "<-[:PREREQUISITE_OF*1..10]-(prerequisite_course) "\
                "RETURN prerequisite_course.year AS Year, "\
                "prerequisite_course.semester AS Semester, "\
                "prerequisite_course.code AS CourseCode, "\
                "prerequisite_course.name AS CourseName, "\
                "prerequisite_course.credit AS Credit "\
                "ORDER BY prerequisite_course.year, "\
                        "prerequisite_course.semester, "\
                        "prerequisite_course.code"
        result = graph.run(query).to_table()
        if (not result):
            print ("Course " + course_code + " has no prerequisite courses.")
            return
        print ("Course " + course_code + " has this prerequisite courses:\n")
        print (result)

    else:
        print ("Please check your database connection!")
# ************************************************************************
```

Example outputs for both prerequisite information (CENG492) and courses in a specific time (second year fall semester) can be shown as:

```
Course CENG492 has this prerequisite courses:

 Year | Semester | CourseCode | CourseName                    | Credit
------|----------|------------|-------------------------------|--------
    1 | spring   | CENG140    | C PROGRAMMING                 |      4
    2 | fall     | CENG213    | DATA STRUCTURES               |      4
    3 | spring   | CENG350    | SOFTWARE ENGINEERING          |      3
    4 | fall     | CENG491    | COMPUTER ENGINEERING DESIGN I |      4
```

```
 Year | Semester | CourseCode | CourseName                          | Credit
------|----------|------------|-------------------------------------|--------
    2 | fall     | CENG213    | DATA STRUCTURES                     |      4
    2 | fall     | CENG223    | DISCRETE COMPUTATIONAL STRUCTURES   |      3
    2 | fall     | EE281      | ELECTRICAL CIRCUITS                 |      3
    2 | fall     | ENG211     | ACADEMIC ORAL PRESENTATION SKILLS   |      3
    2 | fall     | HIST2201   | PRINCIPLES OF KEMAL ATATURK I       |      0
    2 | fall     | MATH219    | INTRODUCTION TO DIFFERENTIAL EQUATIONS |      4
```

At the end of this phase, as a project group we decided to continue with project related data. In our case, these are news data.

2.3. COMPARISON OF DATA IMPORTING WAYS

Since the news data is gathered from a known economical news website in SQL file format, in order to parse and get the information from the file, all data is transferred to a MySQL database. There are 32734 news entry in the file. After import to MySQL database is done, the data is pulled from that database, formatted and transferred to Neo4j database with a Python script. Again, in this script, for the communication between the script and database, ‘py2neo’ library is used. For the data share between the script and MySQL database ‘mysqlclient’ library is used. Further inspections on the scripting progress will be done in subgroups.

2.3.1. Transferring News Data to MySQL Database

Since our SQL file has all data as SQL commands, it was easy to import it to MySQL database by command line file forwarding. After giving file as input to the database, all tables in the database was created by MySQL itself. While transferring data, table columns are determined as news entry id, news entry date, news entry title, news entry's first paragraph and news entry content. A new test user is created for safety reasons (with just reading privileges) due to testing reasons.

After this point, following stages will be handled by the python script. For the connection, as it is said earlier, 'mysqlclient' library has chosen.

```
26  DROP TABLE IF EXISTS `HaberlerYil`;
27
28  CREATE TABLE `HaberlerYil` (
29    `ID` varchar(255) CHARACTER SET utf8mb4 NOT NULL DEFAULT '',
30    `Date` varchar(255) CHARACTER SET utf8mb4 DEFAULT NULL,
31    `Title` varchar(10000) CHARACTER SET utf8mb4 DEFAULT NULL,
32    `Content` text CHARACTER SET utf8mb4,
33    `FirstParagraph` text CHARACTER SET utf8mb4
34  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
35
36  LOCK TABLES `HaberlerYil` WRITE;
37  /*!40000 ALTER TABLE `HaberlerYil` DISABLE KEYS */;
38
39  INSERT INTO `HaberlerYil` (`ID`, `Date`, `Title`, `Content`, `FirstParagraph`)
40  VALUES
41      ('267286','2015-01-01','Çin\'de kutlama faciaya dönüştü: 35 ölü! | Dünya','Çin\'in res
42      ('267288','2015-01-01','2014'ün yıldız borsaları Çin ve Arjantin! | Dünya','DIŞ HABERL
43      ('267289','2015-01-01',''Zirai ilaç kalıntı oranı AB standardının altına indi' | Tarım
44      ('267294','2015-01-01','Yeni yılın ilk ayı önemli gelişmelere tanık olacak | Ekonomi',
```

2.3.2. Transferring News Data From MySQL to Neo4j Database

As our project requires fast data importing and exporting, it is essential to try and compare different ways of data transfer to Neo4j. MySQL Database is not a part of our main purpose. It is only used because our news data has come with SQL command format in a sql file. Therefore, main comparison between the ways of importing data to Neo4j is calculated without MySQL export session. We have found 3 different ways. Our main comparison is how fast they are.

2.3.2.1. Row-by-row Approach

First way of importing data is getting each news entry data one by one from MySQL database and then importing them to Neo4j one by one again. This approach uses the functions of the course prerequisite tree script.

In order to form a suitable graph model for the data we have, which we are going to use it for all approaches, we determined each news entry data as nodes. Nodes' labels are given according to their publication date. Each news entry is connected to every news entry that is published on the next day by a 'FOLLOWED_BY' type relationship. This model resulted with 731 labels, 32.734 nodes and 1.668.265 relationships.

With row-by-row creating node and the relationships approach, import operation is took 3456 seconds (57 minutes 36 seconds) in total.

2.3.2.2. MySQL to CSV and CSV to Neo4j with LOAD CSV Command Approach

Second approach for our model mentioned in the first part is partially the same with the first one. Just like the first one, news entry data is gathered one by one from MySQL database. However, this time one row data is written to a CSV file by using 'csv' python library. As a next step, for the relationships and nodes different CSV files are created. Column fields remained unchanged, but for each node starting from 1 an identification number assigned rather than the news id. These newly created identification numbers are used in relationship CSV file to determine which news has a relationship with other news.

News are transferred in 1000 batches (which is default by Neo4j LOAD CSV command) in order to make transfer faster. Although after increasing or decreasing batch number, there is no significant change in import time to Neo4j Database. For all batches, node import operation took approximately 7 minutes. On the other hand, even if we waiting for 2 hours for importing relationships, it didn't finished in several tries. It took so much time due to the nature of the Cypher Query, which searches nodes among all of the nodes to create a relationship. It is getting slower and slower while searching these nodes when new nodes are created in the database.

```
163     node_query  = "USING PERIODIC COMMIT 1000 "\
164                   "LOAD CSV WITH HEADERS FROM "\
165                   "'file:///news_nodes.csv' AS csv "\
166                   "FIELDTERMINATOR ',' "\
167                   "MERGE (node:NewsEntry "\
168                   "{ date:coalesce(csv.date, 'NULL'), "\
169                   "title:coalesce(csv.title, 'NULL'), "\
170                   "firstParagraph:coalesce(csv.firstParagraph, 'NULL'), "\
171                   "content:coalesce(csv.content, 'NULL'), "\
172                   "nodeId:toInt(csv.nodeId) })"
173     rel_query   = "USING PERIODIC COMMIT 2000 "\
174                   "LOAD CSV WITH HEADERS FROM "\
175                   "'file:///news_rels.csv' AS csv2 "\
176                   "FIELDTERMINATOR ',' "\
177                   "MATCH (fromNode:NewsEntry { nodeId:toInt(csv2.fromNodeId) }) "\
178                   "MATCH (toNode:NewsEntry { nodeId:toInt(csv2.toNodeId) }) "\
179                   "MERGE (fromNode)-[r:FOLLOWED_BY { relId:toInt(csv2.relationshipId) }]->(toNode)"
```

As a result, I didn't want to spend more time in this approach. Row-by-row Approach already has better performance from this one. Next, we tried a different option.

2.3.2.3. MySQL to CSV and CSV to Neo4j with Neo4j Import Tool Approach

Last approach for our model partially includes the second one. On this approach, I only used MySQL to CSV part of the LOAD CSV way. For Neo4j side importing, I used a tool which is created by Neo4j

in order to import huge amount of data.

CSV files' headers are changed according to the format required by the import tool. For example, identification number of nodes' column header is changed to ':ID' and label header is changed to ':LABEL'. Moreover, for relationships, starting point of the relationship header is changed to ':START_ID' and ending point of the relationship header is changed to ':END_ID'. For the type header, it is changed to ':TYPE'.

```
args           ="./neo4j-admin import "\
                 "--database=" + str(database_name)+ " "\
                 "--id-type=INTEGER "\
                 "--nodes=../import/news_hyperedges.csv "\
                 "--nodes=../import/news_nodes.csv "\
                 "--relationships=../import/news_rels.csv "\
                 "--multiline-fields=true"
args           = args.split()
restart_server ="./neo4j restart".split()

print ("Copying CSV files to import directory of the Neo4j Database.")
copyfile("./csv_files/news_nodes.csv", str(import_dir) + "/news_nodes.csv")
copyfile("./csv_files/news_hyperedges.csv", str(import_dir) + "/news_hyperedges.csv")
copyfile("./csv_files/news_rels.csv", str(import_dir) + "/news_rels.csv")
print ("Copy process is completed. Adjusting index properties.\n")

print ("Import operation is starting...")
popen = subprocess.Popen(args, cwd=bin_dir , stdout=subprocess.PIPE)
popen.wait()
output = popen.stdout.read()
```

With this syntax, Neo4j Import Tool is capable of generating nodes and forming relationships between them with arbitrary unique numbers.

After running the script several times (approximately 50 times), average time for all import operation was 40,5 seconds. It was at most 50 seconds and at least 26 seconds in total. It took 18-20 seconds to export CSV files from MySQL files and importing CSV files to Neo4j

Database took 4-7 seconds. The remaining time is spent to deleting all the existing data in the Neo4j Database.

2.3.2.4 Results

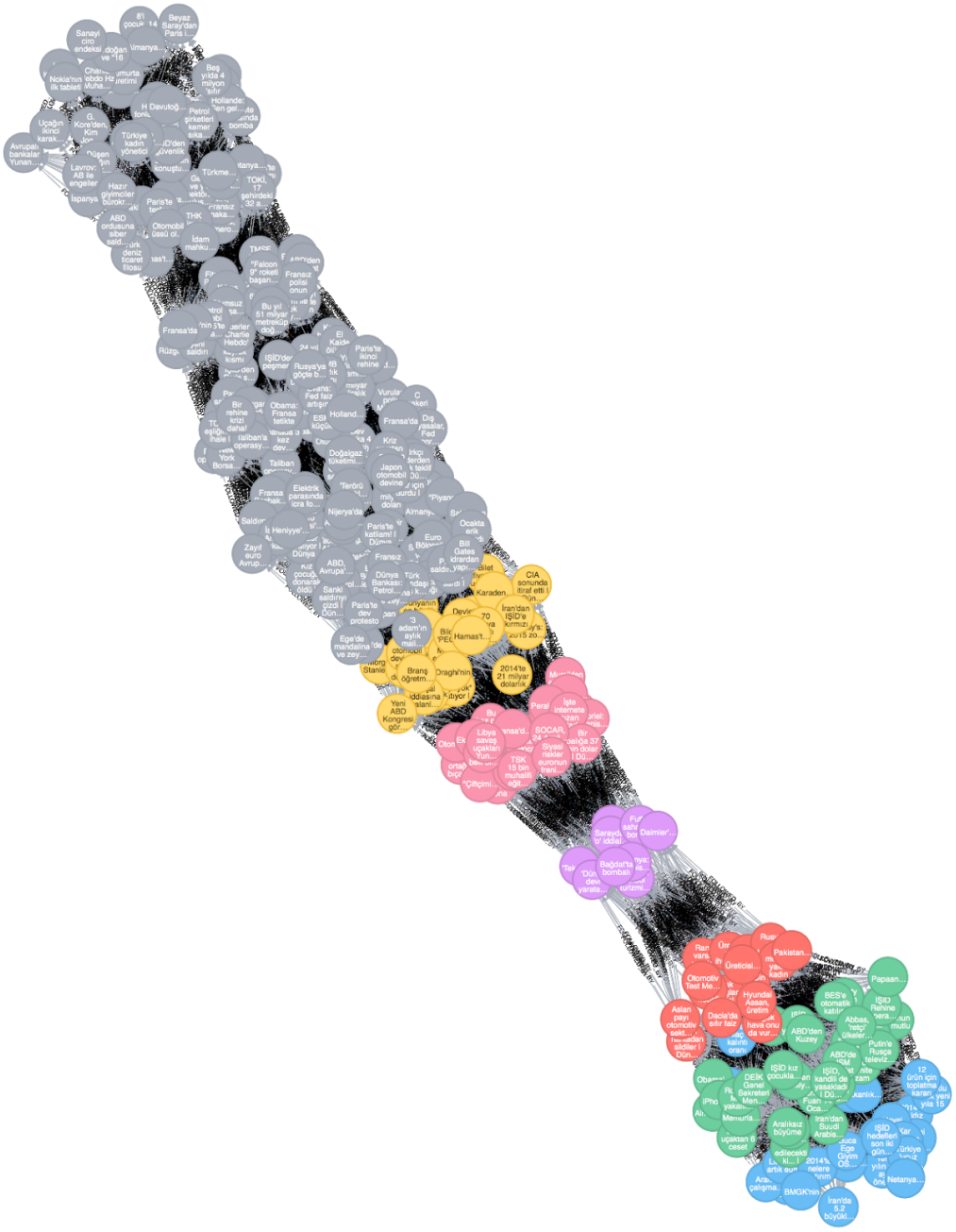
In conclusion, the last approach with Neo4j's own import tools is the best by far when the comparison and easy-to-use features are considered. After this moment in the project, it is decided to be used for import approach if there is no extraordinary need for others.

2.4. Improving Graph Model in the Future

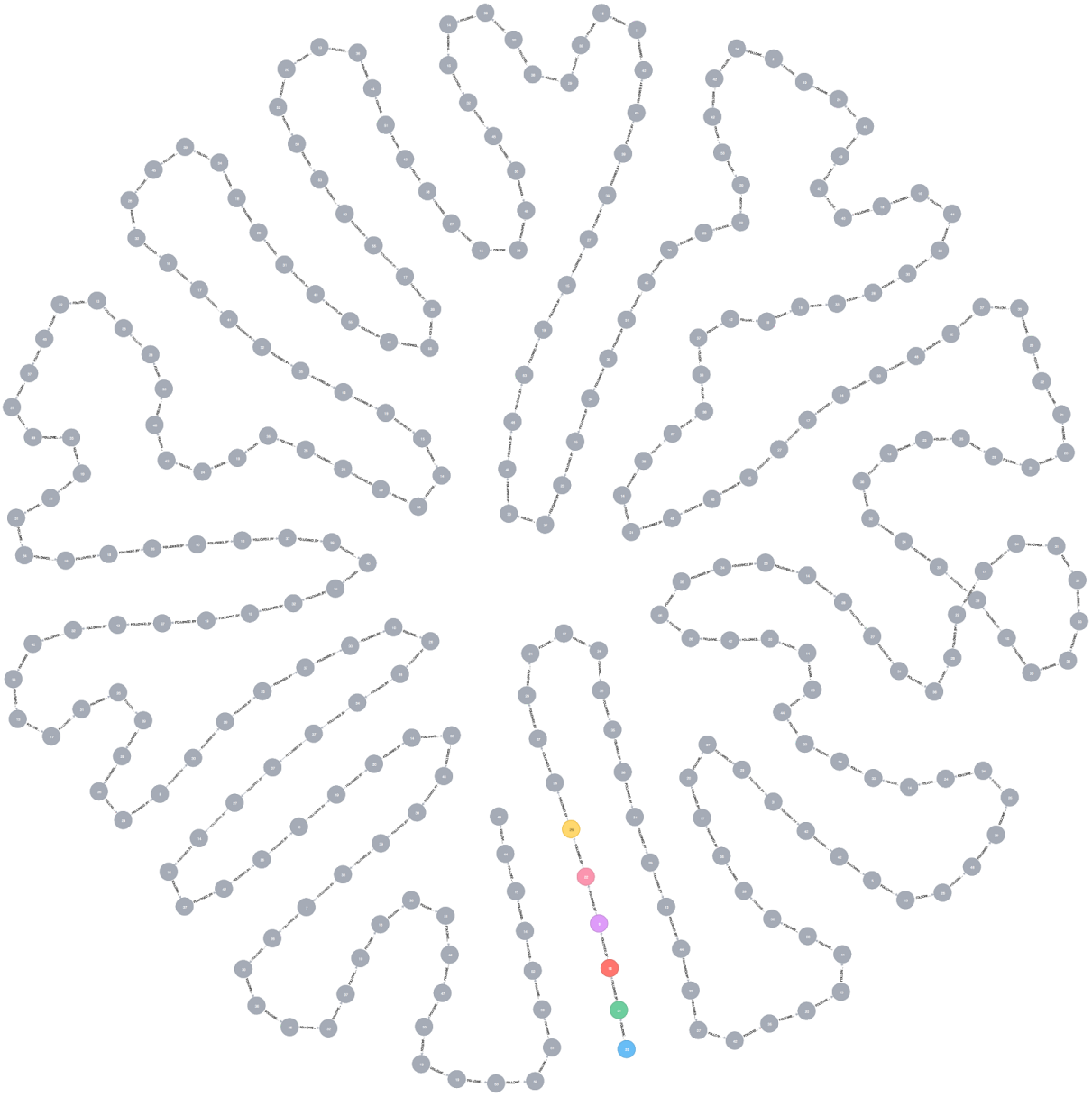
The very first model of our news data is consist of just connecting nodes which are successive in publishing dates. It requires a lot of relationships to be created. Therefore, it both slows entire process and requires more storage space. In order to increase the performance, we come up with another mathematical concept which is hypergraphs. This model is also tested with the last approach of importing data.

Since our earlier model has showed us that categorization of nodes can be done via the dates, we asked ourselves 'Why don't we create hyperedges that represent dates and categorize news nodes with hyperedges?'. With this question, a new model is created. We are still going to have our 32.734 news entry as nodes, but with an extra 731 nodes that will represent hyperedges (since there are 731 different publication dates for news) in our graph. The relationships based on dates are created between this hyperedges like they are forming a linked list. Categorization is handled by created relationships between the news entries (normal nodes) and the hyperedges (date nodes).

At the end of this implementation, node numbers are increased to 33.465 and relationship numbers are decreased to 33.464. It is a significant decrease in relationship numbers. Therefore, it is also decided that hypergraph model is more suitable for our future purposes.



Representation of the first model of our graph data



Representation of hyperedges of the second model of our graph data