Analysis of Twitter Data with the help of Neo4j Graph Database and Python

Asvestopoulos Georgios - 108
Bektsis Evangelos - 113
Kaliakatsos Charilaos - 119
Parousidou Vasiliki - Chrysovalanto - 106

This tutorial aims to analyze the functionality of Neo4j when used in collaboration with python by utilizing it onto a dataset derived from the Twitter API.

This tutorials aims towards explaining how to perform the following tasks in order to analyze graph related data:

- How to convert BSON files coming from MongoDB to CSV files.
- How to use python in order to get a grasp of your data before loading them into your Neo4j graph database.
- How to install and set up your Neo4j Database
- How to create a graph in the Neo4j environment using python programming.
- How to mine useful information concerning a graph by using python and Cypher.

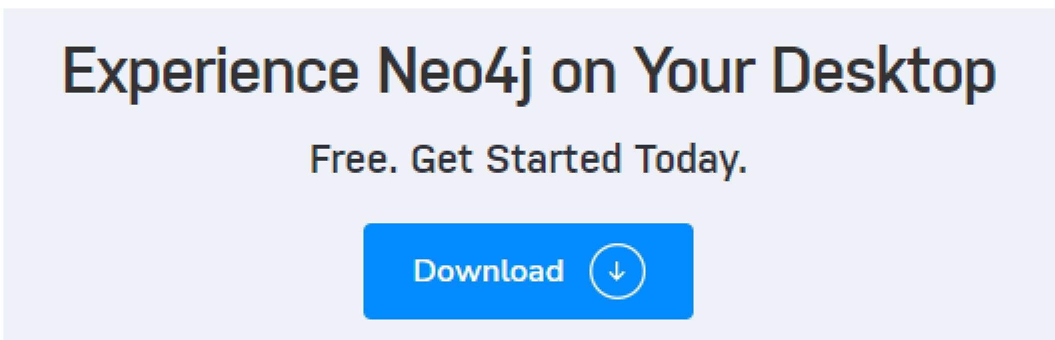
What is Neo4j

Neo4j provides the most trusted and powerful tools for developers and data scientists to swiftly construct today's intelligent apps and machine learning processes. It is offered as a fully managed cloud service or as a self-hosted solution. It's an ACID-compliant transactional database with native graph storage and processing

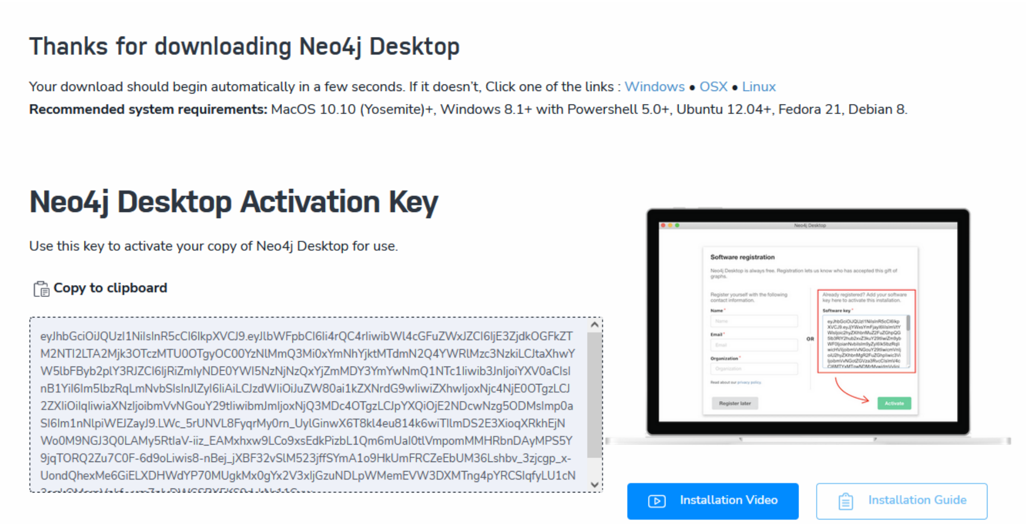
The main elements of a graph database that take on the role of the relational tables are the nodes, edges and the properties that may or may not be attached onto them. These key elements provide a flexible and effective way of interacting with data concerning social networks, supply chain mapping, fraud detection or other types of industry areas that are preferably been analyzed via graph databases rather than RDBMS or NoSQL databases.

How to install Neo4j

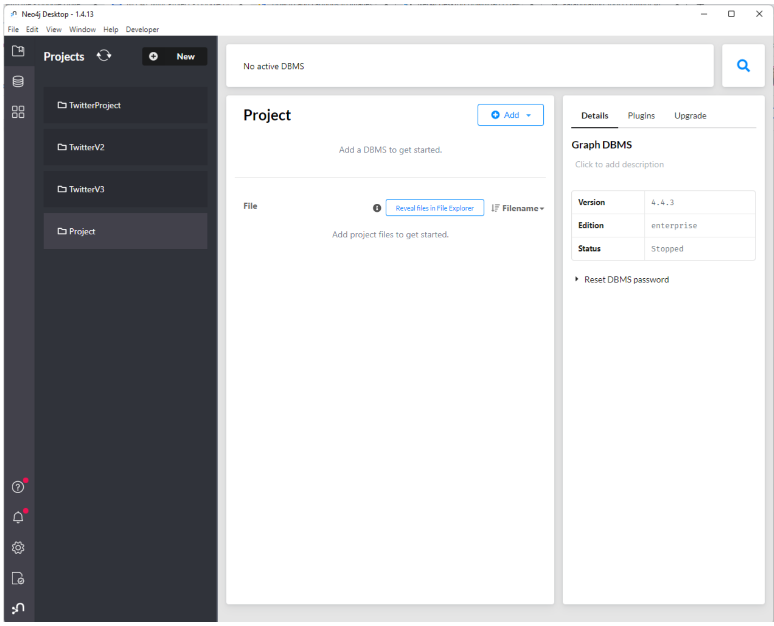
Firstly, download the latest version of [Neo4j] Desktop version.



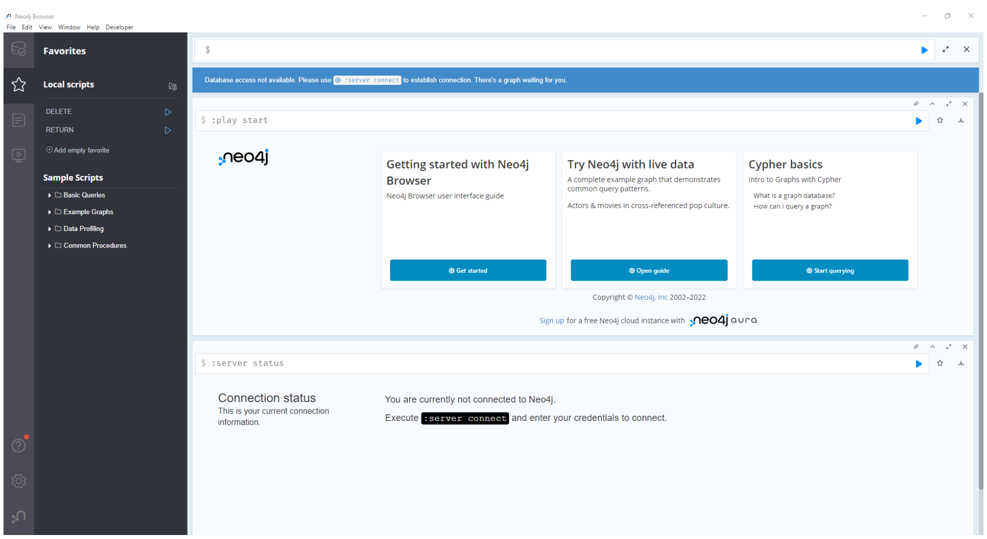
After installing it, you will be prompted to verify your account by using a unique key that is to be provided by the website.



Starting the Neo4j Desktop application you will be greeted with the following window from where you will create your first Project. After that using the Add button you can create your own local graph database which will be used and get populated by your data.



Lastly, when the configuration setup has been completed in the database, you can launch the Neo4j Browser in order to visualize and graph and enjoy the many capabilities that the browser provides.



Converting your data from BSON to CSV

In this tutorial two ways of converting your BSON files to CSV format are to be presented. It's up to you to choose the one that suits you best.

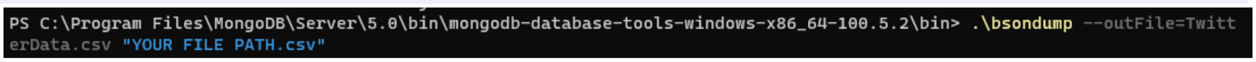
The first way includes the usage of python libraries PyMongo and bson. The following snippet will load your bson file and convert it to a dataframe.

```
import bson
import pandas as pd
data = bson.decode_file_iter(open('YOUR FILE PATH','rb'))
df = pd.DataFrame(data)
```

Even though that's the easiest method to convert your file, it will take time to preprocess the data from your dataframe since some of the columns contain dictionary type data.

The method that was followed for this tutorial is the one presented below. First of all, you need to download the [MongoDB database tools](#) in order to perform the conversion. The command to be executed in the Windows Power Shell is the following:

```
.\bsondump --outFile=TwitterData.csv "YOUR FILE PATH.bson"
```



After performing the commands presented above, your file will be saved in the path your CMD prompt is pointing at with the name given into the "–outFile" field.
This seems to be the best way since the columns that were previously in dictionary format are now expanded and can be easily accessed.

Twitter Data Schema

[Twitter API](#) provides a variety of different objects to the users in order for them to be able to collect information about the tweets and their metadata they are interested in.
The objects are:

- **Tweet object**
  The attributes that the tweet object contains are `id`, `created_at`, and `text`. The tweet object is also the parent to several tweet child objects which include `user`, `entities`, and `extended_entities`.
- **User object**
  The User object contains a variety of metadata concerning the twitter users that are capable of explaining the users actions.
  In general, the values of user metadata are pretty consistent. Some data, such as the user's `id` (given as a string `id_str`) and the date the account was established, never change. Other metadata, such as the `screen name`, `displayname`, `description`, `location`, and other profile parameters, may vary from time to time. Some information, such as the number of Tweets the account has posted `statuses_count` and the number of followers `followers_count`, fluctuate regularly.

- **Entities object**
  Entities give metadata and other contextual information about Twitter posts.
  The `entities` and `extended_entities` sections are both made up of arrays of entity objects.
- **Extended entities object**
  An `extended entities` JSON object will be included in any Tweets with attached photographs, videos, or animated GIFs. A single media array of `media` objects is contained in the extended entities object (see the entities section for its data dictionary). The extended entities section does not include any more entity kinds, such as hashtags or links. The media object in the extended entities section is structurally identical to the one in the `entities` section.
- **Geo object**
  To represent the location associated with a Tweet, two 'root-level' JSON objects are used: `coordinates` and `place`.

Graph Creation

This subsection describes the procedure that is followed in order to populate a Neo4j graph model using the tweets that have been collected in Python. As it was described earlier, a graph consists of different kinds of nodes and edges between them. Moreover, each node and edge can contain some properties. In our case, the nodes and the relationships that are used are presented as follows:

- **Nodes**
  - **User nodes**: Represent users that can be either authors of a tweet/retweet or have been mentioned in some other user's tweet. The properties that they contain are the username, the user id, the date when their account was created and the number of their followers.
  - **Tweet nodes**: Represent tweets. The properties that they contain are the ids of the tweets, their text and the date that they were created.
  - **Hashtag nodes**: Represent hashtags used in tweets.
  - **URL nodes**: Represent URLs used in tweets.

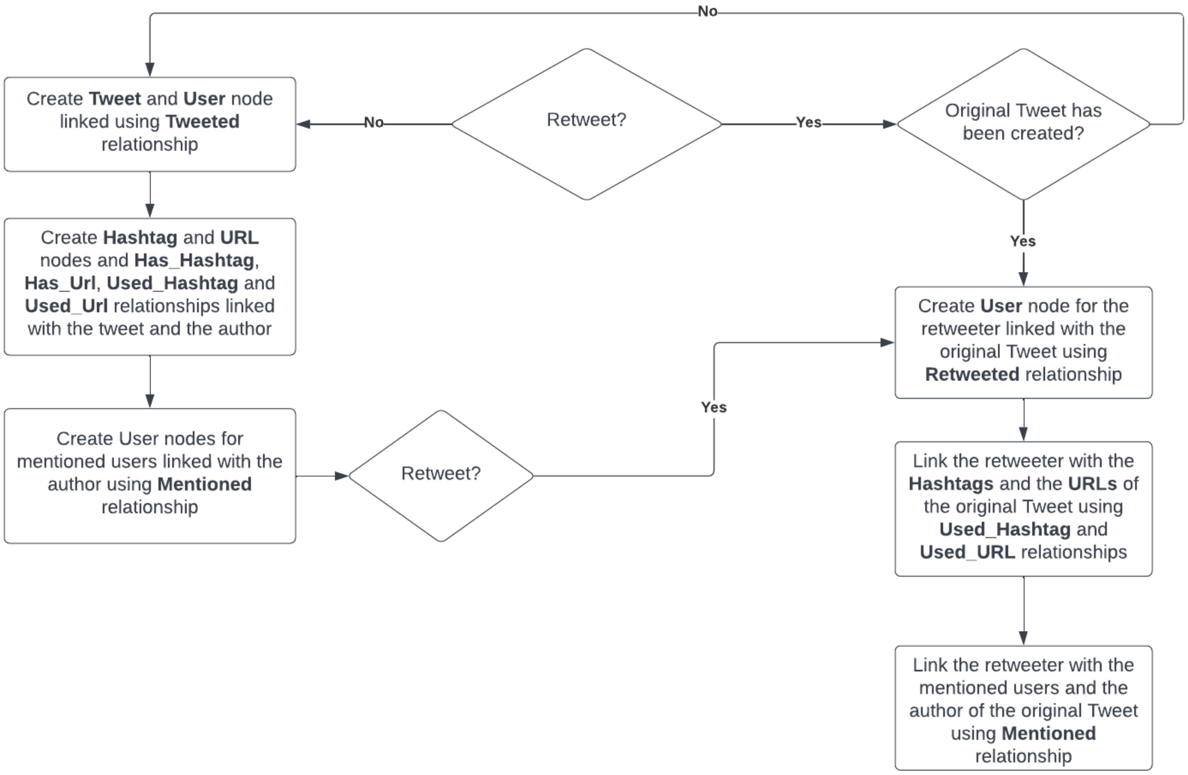
- **Relationships**
  - **Tweeted**: Represents a relationship between a tweet and its author. The properties of this relationship are the creation date of the tweet and the device from which it was tweeted.
  - **Retweeted**: Represents a relationship between an original tweet and the authors that have retweeted it. The properties of this relationship include the date of the retweet and the device from which it was retweeted.
  - **Has_Hashtag**: Represents a relationship between a hashtag and the tweets that have used it.
  - **Has_Url**: Represents a relationship between a URL and the tweets that refer to it.

- **Used_Hashtag**: Represents a relationship between a hashtag and a user that has created either a tweet or a retweet that contains this hashtag.
- **Used_Url**: Represents a relationship between a URL and a user that has created either a tweet or a retweet that contains this URL.
- **Mentioned**: Represents a relationship between a user and the users that he/she has mentioned or has retweeted their tweets.

In order to create all these nodes and relationships, we iterate through the dataset. At each line, which represents a tweet, we first have to check if this is an original tweet or a retweet. If it is an original tweet, we extract the required information so as to create the **tweet** node, the **author's** node (type of "User") and the "**Tweeted**" relationship between them. After that, we have to retrieve information about the **hashtags** and the **URLs** that this tweet contains. When we do so, we have to create a node for each one of them and the required relationships. More specifically, we create a "**Has_Hashtag**" and a "**Has_Url**" relationship between the tweet node and each hashtag and url node, respectively. In addition, we create a "**Used_Hashtag**" and a "**Used_Url**" relationship between the user node (author node) and each hashtag and URL node, respectively. Last but not least, we have to create a "**Mentioned**" relationship between the author and each user that is mentioned in the tweet. In case that there is not a node in the graph about some mentioned user, we create it, as well.

In case that the tweet is a retweet, we have to follow the above procedure for the original tweet if it hasn't been created yet. In order to check that, we extract the ID of the original tweet and check if it is included in a list where we store the ids of the Tweet nodes that have been created. If it is included, we skip this step. Otherwise, we use the information about the original tweet as stored under "retweeted_status" and follow the above procedure. After that, we have to use the information about the retweet in order to create some additional nodes and relationships. More specifically, we have to create a user node containing information about the author of the retweet and link it to the tweet node of the original tweet using "**Retweeted**" relationship. Then, we have to create "**Used_Hashtag**" and "**Used_Url**" relationships between the retweeter (i.e, the author of the retweet) and each hashtag and URL that has been used in the original tweet. This step is important because every time a tweet, which contains some hashtags and URLs, is retweeted, the visibility of these hashtags and URLs is increased and this is captured by adding these relationships. Also, we add "**Mentioned**" relationships between the retweeter and each user that has been mentioned in the original tweet for the same reason as described above. Finally, we add the "**Mentioned**" relationship between the retweeter and the author of the original tweet.

The block diagram below presents the procedure that has been described and the implementation in Python follows.




```
for t in tqdm(tweets.index):
    #Check if a tweet is original in order that the tweet node can be
created or retweet of a original tweet that its node
    #hasn't been created yet.
    if np.isnan(tweets["retweeted_status.id"][t]) or
(tweets["retweeted_status.id"][t] not in tweet_created):
        #In case that the tweet is original
        if np.isnan(tweets["retweeted_status.id"][t]):

            #Get the user's and the tweet's attributes
            user_name = tweets["user.screen_name"][t]
            user_date = tweets["user.created_at"][t]
            user_id = tweets["user.id_str"][t]
            #Number of author's followers
            followers = tweets['user.followers_count'][t]

            date = tweets["created_at"][t]
            #Regex for device retrieval
            device = re.findall('(?<=\>)(.*?)(?=\<)',tweets['source'][t])

            text = tweets["text"][t]
```

```
            tweet_id = tweets["id_str"][t]
            tweet_created.append(tweet_id)

            #Get hashtags from text
            hash_in_text = tweets["entities.hashtags"][t]
            hash_list=[]
            hashtags=json.loads(hash_in_text)
            for k in range(len(hashtags)):
                hash_list.append(hashtags[k]['text'].lower())

            #Get urls from text
            url_in_text = url_in_text_df[t]
            url_list=[]
            if url_in_text != 'NaN':
                urls=json.loads(url_in_text)
                for k in range(len(urls)):
                    url_list.append(urls[k]["url"])
        #In case the tweet is a retweet, the information about the original
tweet is first extracted
        else:
            #Get the user's and the original tweet's attributes
            user_name = tweets["retweeted_status.user.screen_name"][t]
            user_date = tweets["retweeted_status.user.created_at"][t]
            user_id = tweets["retweeted_status.user.id_str"][t]
            #Number of author's followers
            followers = tweets['retweeted_status.user.followers_count'][t]

            #Regex for device retrieval
            device =
re.findall('(?<=\>)(.*?)(?=\<)',tweets['retweeted_status.source'][t])
            date = tweets["retweeted_status.created_at"][t]

            text = tweets["text"][t]
            tweet_id = tweets["retweeted_status.id_str"][t]
            tweet_created.append(tweet_id)

            #Getting hashtags from text
            hash_in_text = tweets["retweeted_status.entities.hashtags"][t]
            hash_list=[]
            hashtags=json.loads(hash_in_text)
            for k in range(len(hashtags)):
                hash_list.append(hashtags[k]['text'].lower())
```

```
            #Getting urls from text
            url_in_text = url_in_text_df[t]
            url_list=[]
            if url_in_text != 'NaN':
                urls=json.loads(url_in_text)
                for k in range(len(urls)):
                    url_list.append(urls[k]["retweeted_status.url"])


        #Create USER's node and merge it to the graph
        user = Node("User", username = user_name, user_id_ = str(user_id),
                    creation_date = user_date, follower = str(followers))

        graph.merge(user, 'User', 'user_id_')

        #Create TWEET's node and merge it to the graph
        tweet_node = Node("Tweet", tweet = str(tweet_id), text = text,
                          creation_date= date)

        graph.merge(tweet_node, 'Tweet', 'tweet')

        #Create relationship TWEETED between the tweet and its author
        POSTS = Relationship.type("TWEETED")
        graph.merge(POSTS(user, tweet_node, creation_date = date, device =
device),
                    ('User','Tweet'),('user_id_','tweet'))

        #Extract mentioned users and create USER nodes for them in case
they haven't been created yet
        if np.isnan(tweets["retweeted_status.id"][t]):
            user_mentioned_json = tweets["entities.user_mentions"][t]
        else:
            user_mentioned_json =
tweets["retweeted_status.entities.user_mentions"][t]
        user_mentioned_loaded= json.loads(user_mentioned_json)
        for screen_names in range(len(user_mentioned_loaded)):
            user_name_mentioned =
user_mentioned_loaded[screen_names]['screen_name']
            user_mentioned_id =
user_mentioned_loaded[screen_names]['id_str']
            mentioned_user = Node("User", username = user_name_mentioned,
                                  user_id_ = str(user_mentioned_id), follower =
str(0))
```

```
            graph.merge(mentioned_user, 'User', 'user_id_')
            #Create a relationship between the author of the tweet and each
mentioned user
            MENTIONED = Relationship.type("MENTIONED")
            graph.create(MENTIONED(user, mentioned_user))

        #Create a HASHTAG node for each hashtag that was used in the tweet
        #Add a relationship HAS_HASHTAG between the hashtag and the tweet
that uses it
        #Add a relationship USED_HASHTAG between the hashtag and the author
of the tweet
        for u in hash_list:
            hashtag = Node("Hashtag", hashtag = u)
            graph.merge(hashtag, 'Hashtag', 'hashtag')
            HAS_HASHTAG = Relationship.type("HAS_HASHTAG")
            USED_HASHTAG = Relationship.type("USED_HASHTAG")
            graph.create(HAS_HASHTAG(tweet_node, hashtag))
            graph.create(USED_HASHTAG(user, hashtag))

        #Create a URL node for each url that was used in the tweet
        #Add a relationship HAS_URL between the url and the tweet that uses
it
        #Add a relationship USED_URL between the url and the author of the
tweet
        for u_r_l in url_list:
            url_node = Node("URL", url = u_r_l)
            graph.merge(url_node, 'URL', 'url')
            HAS_URL = Relationship.type("HAS_URL")
            USED_URL = Relationship.type("USED_URL")
            graph.create(HAS_URL(tweet_node, url_node))
            graph.create(USED_URL(user, url_node))

    # If this tweet is a retweet
    if not np.isnan(tweets["retweeted_status.id"][t]):
        #Retweeter's attributes
        user_name = tweets["user.screen_name"][t]
        user_date = tweets["user.created_at"][t]
        user_id = tweets["user.id_str"][t]
        #Number of author's followers
        followers = tweets['user.followers_count'][t]

        retweet_date = tweets['created_at'][t]
        #Regex for device retrieval
```

```
        device = re.findall('(?<=\>)(.*?)(?=\<)',tweets['source'][t])

        #Create a USER node for the user that created the retweet and merge
it to the graph
        user_ret = Node("User", username = user_name, user_id_ =
str(user_id),
                creation_date = user_date, follower = str(followers))
        graph.merge(user_ret, 'User', 'user_id_')

        #Create MENTIONED relationship between the retweeter and the author
of the original tweet
        MENTIONED = Relationship.type("MENTIONED")
        graph.create(MENTIONED(user_ret, user))

        #Create RETWEETED relationship between the retweeter and the
original tweet
        RETWEETED = Relationship.type("RETWEETED")
        graph.create(RETWEETED(user_ret, tweet_node, creation_date =
retweet_date, dev = device))

        #Create USED_HASHTAG relationship between the author of the retweet
and the hashtags used in the original tweet
        for u in hash_list:
            hashtag = Node("Hashtag", hashtag = u)
            graph.merge(hashtag, 'Hashtag', 'hashtag')
            USED_HASHTAG = Relationship.type("USED_HASHTAG")
            graph.create(USED_HASHTAG(user_ret, hashtag))

        #Create USED_URL relationship between the author of the retweet and
the URLs used in the original tweet
        for u_r_l in url_list:
            url_node = Node("URL", url = u_r_l)
            graph.merge(url_node, 'URL', 'url')
            USED_URL = Relationship.type("USED_URL")
            graph.create(USED_URL(user_ret, url_node))

        #Create MENTIONED relationship between the author of the retweet
and the users that are mentioned in the
        #original tweet
        for screen_names in range(len(user_mentioned_loaded)):
            user_name_mentioned =
user_mentioned_loaded[screen_names]['screen_name']
            user_mentioned_id =
```

```
user_mentioned_loaded[screen_names]['id_str']
            mentioned_user = Node("User", username = user_name_mentioned,
                                  user_id_ = str(user_mentioned_id))
            graph.merge(mentioned_user, 'User', 'user_id_')
            MENTIONED = Relationship.type("MENTIONED")
            graph.create(MENTIONED(user_ret,mentioned_user))
```

Twitter Data Analysis

This subsection presents a number of useful queries that can be utilized for exploratory data analysis of the Neo4j graph data.

1. Get the total number of **tweets**, **hashtags** and **URLs** (case insensitive).

```
node_match = NodeMatcher(graph)
tweets_num = node_match.match("Tweet").count()
hashtag_num = node_match.match("Hashtag").count()
url_num = node_match.match("URL").count()
print(tweets_num, hashtag_num, url_num)
9410 2498 2671
```

- ➔ NodeMatcher is provided by the py2neo library and offers the functionality to match nodes according to certain criteria.
- ➔ The .match() function of the NodeMatcher() takes as a parameter the type of nodes to be returned while the .count() sums up the number of returned.
- ➔ The total number of tweets in our case is 9410 while the hashtags are 2498 and the used urls 2671.

2. Get the total number of **retweets.**

```
relation_match = RelationshipMatcher(graph)
retweets_num = relation_match.match(nodes=None,r_type=RETWEETED).count()
print(retweets_num)
19439
```

- ➔ RelationshipMatcher is provided by the py2neo library and offers the functionality to match relationships according to certain criteria.
- ➔ .match() takes as parameters the type of node (in this case None indicates any type i.e. every type of node is accepted) and the type of the relationship between them.
- ➔ The total number of retweets is 19439.

3. Get the **followers** count of each user

```
followers_num = graph.run('MATCH (n1:User) RETURN n1.username,
```

```
n1.follower')
followers= pd.DataFrame(followers_num)
followers.rename(columns={0:'User',1:'Followers'},inplace=True)
print(followers)
```

- graph.run provides the user with the flexibility to incorporate Cypher queries into his python environment
- As it can be seen above, we match all the User nodes and return their usernames and the number of their followers.
- Then, we insert the data into a dataframe, which is presented below.

	User	Followers
0	gio_iacono_work	261
1	ShaniEvenstein	0
2	NASAGoddard	800156.0
3	NASAHubble	0
4	colnshepp	0
...	...	...
20054	visivoz	2324
20055	Diane_in_SA	0
20056	Pikiran2ku	0
20057	DDHefte	310
20058	StadtwikiDD	0

4. Get the 20 most popular **Hashtags** and **URLs** and the 20 **users** with the most followers in descending order.

```
hashtag_20 = graph.run('MATCH (n1:User)-[r:USED_HASHTAG]->(n2:Hashtag)
RETURN n2.hashtag, count(r) AS n ORDER BY n DESC LIMIT 20')
hashtag_top_20= pd.DataFrame(hashtag_20)
hashtag_top_20.rename(columns={0:'Hashtag',1:'Times used'},inplace=True)
print(hashtag_top_20)
url_20 = graph.run('MATCH (n1:User)-[r:USED_URL]->(n2:URL) RETURN n2.url,
count(r) AS n ORDER BY n DESC LIMIT 20')
url_top_20= pd.DataFrame(url_20)
url_top_20.rename(columns={0:'URL',1:'Times used'},inplace=True)
print(url_top_20)
followers_num = graph.run('MATCH (n1:User) RETURN n1.username,
toInteger(n1.follower) AS n ORDER BY n DESC LIMIT 20')
followers= pd.DataFrame(followers_num)
```

```python
followers.rename(columns={0:'User',1:'Followers'},inplace=True)
print(followers)
```

- ➔ Similar to what we did before, we define what we want to have matched and returned to us.
- ➔ **MATCH** is used in order to declare the relationship(s) and the entities participating in those that we are interested in. For example, in the code above we declare that we want to match the nodes of type `User` with the nodes of type `Hashtag` through the relationship `USED_HASHTAG`. The output of this MATCH will return all the combinations of users along with the respected hashtags they have used.
- ➔ **RETURN** is used in order to declare which information of the MATCH command we wish to be returned. In this case, the attribute hashtag of the hashtags' nodes is selected (hashtag was defined to contain the hashtags text during the nodes creation). The observed n2 is just a shortcut for the Hashtag nodes. These shortcuts are declared in the MATCH phase as seen above. Moreover the count(r) (r is declared as a shortcut for the relationship USED_HASHTAG) is also chosen to be returned and will provide the actual answer to the question "How many times was each hashtag used".
- ➔ The **AS** command can be used to create a shortcut for the selected returned entities. In this case count(r) is defined to be used as "n". This may not be a necessary step for your command to be executed, but it is an excellent technique of making your code easier to handle and more readable.
- ➔ Using the **ORDER BY** command, we can sort the results based on the value of a variable that is stated. The **DESC** command indicates that descending ordering is applied, whereas **ASC** indicates that ascending ordering is applied (in our case the output has to be ordered based on the number or times each hashtag has been used in a descending order). **LIMIT** can be used to set a limit on the number of outputs that are returned. Here only the 20 most used hashtags are chosen to be returned.
- ➔ In a similar fashion we can get the 20 most used URLs in descending order and the 20 users that have the most followers. The only difference in the last case is the type-casting that is used inside the query where "toInteger" is used in order to transform the number of followers from str to int.
- ➔ We convert the data returned from the query to a dataframe and rename the columns with presentable names.
- ➔ The results can be found below. As we can see, there are some hashtags whose usage is excessively higher than the usage of the rest. For example, the first top 2 hashtags have been used almost 3 times more compared to the rest. This indicates that users tend to use the most popular hashtags. Also, it is evident that the most commonly used hashtag has been used more times than the most popular URL, since hashtags are the characteristic feature of Twitter.

	Hashtag	Times used
0	openscience	1832
1	citizenscience	1341
2	crowdsourcing	429
3	earthquake	291
4	openaccess	243
5	ukraine	223
6	scicomm	167
7	opendata	152
8	ai	147
9	raspberryshake	126
10	crowdfunding	122
11	publicengagement	113
12	opensource	113
13	communityscience	104
14	covid19	99
15	openinnovation	97
16	bioinformatics	92
17	datascience	82
18	sb19	74
19	stanworld	68

	URL	Times used
0	https://t.co/Y5O0dgJqJF	155
1	https://t.co/KOU5939WnG	28
2	https://t.co/H9U2dXLnRL	27
3	https://t.co/nw2JiDrJiP	23
4	https://t.co/OdVfEky2oU	23
5	https://t.co/udqjQPNIp7	22
6	https://t.co/XL23iR4BW9	22
7	https://t.co/JMbuOy08Ag	20
8	https://t.co/YH560alI5H	20
9	https://t.co/jIwpLCSh1S	19
10	https://t.co/Zfkiqp65BG	19
11	https://t.co/GB3hkgBfJV	19
12	https://t.co/W0MAI3xTkh	18
13	https://t.co/gSXYE2MhVy	18
14	https://t.co/n9jbuE5UdU	18
15	https://t.co/bxvvZm9duP	18
16	https://t.co/9OxnIC1DAg	18
17	https://t.co/0I4BM6RddZ	17
18	https://t.co/34ulBiiuil	17
19	https://t.co/D2hmXy2eYc	17

	User	Followers
0	coinbase	4897983
1	NWS	3048645
2	britishlibrary	1882525
3	BoredElonMusk	1751397
4	TheRickWilson	1333989
5	zeerajasthan_	1261018
6	deray	1027275
7	thidakarn	893280
8	NASAGoddard	800156
9	campbellclaret	784462
10	NASAJuno	763029
11	NASASun	740721
12	UNESCOarabic	721937
13	UNESCOarabic	719042
14	nrc	643790
15	BMWUSA	636192
16	rki_de	595245
17	vulture	513843
18	doctorow	479986
19	the_marketeers	459765

5. Get the number of **tweets** & **retweets** per hour and the **hour** with the most tweets and retweets.

```
per_hour= graph.run('MATCH (n1:User)-[r]->(n2:Tweet) RETURN
r.creation_date')
per_hour_tweets= pd.DataFrame(per_hour)
per_hour_tweets
from datetime import *
time_list = []
time_count = []
for t in range(len(per_hour_tweets[0])):
    time = datetime.strptime(per_hour_tweets[0][t], "%a %b %d %H:%M:%S %z
%Y")
    time_list.append(time.hour)
for i in range(0,24):
    x = time_list.count(i)
    time_count.append(x)
time_count
hours = np.arange(0, 24)
```

```
d={"Number of posts": time_count}
no_of_posts_per_hour=pd.DataFrame(data=d, index=hours)
print(no_of_posts_per_hour)
print(no_of_posts_per_hour['Number of posts'].idxmax())
16
```

- ➔ Here the key difference compared to the previous queries is that we want to gain information about tweeting and retweeting times and, as a result, the interaction between the User nodes and the Tweet nodes can not be exclusively defined. For this purpose we use the plain [r] symbol to indicate that we want to Match all of the relationships between the corresponding nodes. Then, it is stated that the desired returned value is **r.creation_date** which is the attribute that indicates the time of the creation of a tweet or a retweet.
- ➔ Similar to the other queries, we continue by inserting the data returned into a dataframe for further analysis.
- ➔ The first column of the newly created dataframe encapsulates all the date and time information needed. For each row we define the way the datetime is expressed by using the **datetime.strptime(per_hour_tweets[0][t], "%a %b %d %H:%M:%S %z %Y").**
- ➔ The created “time_list” stores each tweet’s hour of creation which is extracted from the `time` variable. The number of tweets/retweets generated each hour of the day are stored into the `time_count` list which is then used for the creation of the corresponding data frame `no_of_post_per_hour`. Finally, the **.idxmax()** function is executed to answer the initial question.
- ➔ The results once again are returned in a dataframe format where the index column represents the hour of the day. It is evident from the dataframe that the hour with the most tweets is 16:00-17:00. Moreover, we can observe that most tweets are created during working hours (9.00-17.00).

	Number of posts
0	736
1	762
2	800
3	782
4	1000
5	779
6	847
7	1076
8	1349
9	1501
10	1425
11	1380
12	1359
13	1442
14	1559
15	1573
16	1720
17	1661
18	1435
19	1398
20	1284
21	1048
22	1071
23	862

6. Get the top **5 devices** that most users post from and the number of users that have been **mentioned** the most (in descending order). Also, get the most **active** users.

```
#Top 5 devices
device = graph.run('MATCH (n1:User)-[r]->(n2:Tweet) RETURN r.device,
count(r.device) AS c ORDER BY c DESC LIMIT 5')
device_num= pd.DataFrame(device)
device_num.rename(columns={0:'Device',1:'Times used'},inplace=True)
print(device_num)
#Most mentioned users in descending order
users_mentioned = graph.run('MATCH (n1:User)-[r]->(n2:User) RETURN
n2.username, count(r) AS c ORDER BY c DESC')
mentioned_users= pd.DataFrame(users_mentioned)
mentioned_users.rename(columns={0:'Username',1:'No. of
Mentions'},inplace=True)
print(mentioned_users)
#The most active users
```

```
users_tweeted = graph.run('MATCH (n1:User)-[r:TWEETED]->(n2:Tweet) RETURN
n1.username, count(r) AS c ORDER BY c DESC')
active_users= pd.DataFrame(users_tweeted)
active_users.rename(columns={0:'Username',1:'No. of Tweets'},inplace=True)
print(active_users)
```

- ➔ In order to get the 5 devices that have been used the most, we MATCH the User nodes with the Tweet ones with any relationship (i.e., both tweeted and retweeted) and return the "**.device**" attribute and the count of each one of them. Then, we sort the results based on the count and keep only the 5 first so that we can get the 5 most used devices.
- ➔ To get the most mentioned users we MATCH all the Users that have any relationship with each other and keep the ones that are on the right side of the relationship (i.e., those who have been mentioned). By RETURNING their names and the counts of their appearances (each User gets 1 count for each time he/she is mentioned in a tweet or retweet) we can find out the most mentioned ones.
- ➔ A relevant query is also used in order to find the most **active** users (users that have posted the most tweets) but with the difference that this time we focus on the users on the left side of the MATCH command, aka the ones who Tweet the tweets.
- ➔ The results are presented below. It is clear that most users prefer the Twitter Web App to post their tweets while the iPhone users are more than the ones with android. The most mentioned user is the user who tweets the most.

	Device	Times used
0	[Twitter Web App]	4112
1	[Twitter for iPhone]	1521
2	[Twitter for Android]	1102
3	[lucht001]	698
4	[TweetDeck]	401

	Username	No. of Mentions
0	Aalst_Waalre	1167
1	OpenSci_News	167
2	raspishakEQ	161
3	openscience	125
4	Primary_Immune	122
...	...	...
7209	heatherg	1
7210	amazonmturk	1
7211	Qualtrics	1
7212	azti_brta	1
7213	benj_ebooks	1

	Username	No. of Tweets
0	Aalst_Waalre	698
1	RobotRrid	101
2	OpenSci_News	98
3	raspishakEQ	90
4	Primary_Immune	66
...	...	...
5393	kekanakalawaia	1
5394	lancerobert	1
5395	TeriFredrick	1
5396	DevinFauxCalf	1
5397	VictoriaWrites3	1

7. Get the **20** most **retweeted tweets** and the users that posted them.

```
tweets_20 = graph.run('MATCH (n1:User)-[r:RETWEETED] -> (n2:Tweet) <-
[t:TWEETED]-(a:User) RETURN a.username, n2.text, count(r) AS c ORDER BY c
DESC LIMIT 20')
tweets_top_20= pd.DataFrame(tweets_20)
tweets_top_20.rename(columns={1:'Tweet',2:'Times
retweeted',0:'Username'},inplace=True)
```

```
print(tweets_top_20)
```

- ➔ This is the first time we use a “double” MATCH. On the left side of the (n2:Tweet), we search the users that have RETWEETED a Tweet, while on the right side we look for the user that has actually TWEETED the tweet.
- ➔ We then declare that we want the usernames of the users who tweeted the tweets returned along with the text of the tweets and the number of times each tweet was retweeted. By ordering these counts in a descending order and limiting the result to 20, we get the 20 most retweeted tweets.
- ➔ The most retweeted tweet was retweeted 48 times while the second one was retweeted only 27 times. It can also be seen that the majority of these tweets are written in english. Nevertheless, tweets in other languages can also be found.

	Username	Tweet	Times retweeted
0	SoftieJoshTin	UP\n\n@SB19Official #SB19\n#STANWORLD\n#WeStan...	48
1	marcorubio	DANGER\n\nWe can't stop the crowdsourcing of r...	27
2	theneurolander	"Doc, will my loved one wake up?"\n\n#NeuroT...	26
3	AP	"A self-organizing swarm." Formed in a fury to...	26
4	RMU_Ukraine	Перелік українських наукових журналів, які інд...	22
5	GiselaFrauke	Must watch as it shows the interest to include...	22
6	CzechEmbassyDC	@PetrVTuma, Czech Visiting Fellow @AtlanticCou...	22
7	kev_kevs_kevin	Crowdsourcing lang daw sabi ni kumareng Kurt h...	22
8	bauhws	@openscience @fairdata @fair4rs many instituti...	21
9	MKaanCihan	#Ukraine "A self-organizing swarm." Formed in ...	21
10	TrendNewsBLOG1	テレワークが浸透し始めている今、\nクラウドソーシングが益々注目されています。\n副業として...	19
11	Aroguden	"A self-organizing swarm." Formed in a fury to...	19
12	NejBusinessCZ	Očekávání zákazníků ohledně rychlejšího a flex...	19
13	drjenndowd	21/ The analyses are fully reproducible with s...	19
14	NSF	Researchers can harness the power of #OpenData...	18
15	ourcommoncode	We will be presenting at the 3/24 @internetofh...	18
16	drpaulwinston	Canada USA New Zealand and Chile. Can we invit...	18
17	Aalst_Waalre	Crikey! That's not good, the level is below 20...	18
18	tehbb	Er what?	17
19	CibeleHdoAmaral	Hey, come work with us at @EarthLabCU! #innova...	17

8. Get the **20 hashtags** that **co-occur** with the one that has been used the most.

```
co_occurence = graph.run(f"MATCH (h1:Hashtag)<- [u1:HAS_HASHTAG] - (t:
Tweet) - [u2:HAS_HASHTAG]-> (h2:Hashtag) WHERE h1.hashtag =
'{hashtag_top_20['Hashtag'][0]}' AND h1.hashtag <> h2.hashtag RETURN
h2.hashtag, count(t) AS c ORDER BY c DESC LIMIT 20")
co_occ_20= pd.DataFrame(co_occurence)
co_occ_20.rename(columns={0:'Hashtag',1:'Co-occurence'},inplace=True)
print(co_occ_20)
```

- ➔ This query may seem a little trickier but when breaking it down it is not that complicated. First of all, we create the relationship we want to **MATCH**. More specifically, we want to find a combination of 2 hashtags that co-exist in any tweet.
- ➔ In order to keep only those that co-occur with the most used hashtag, we use the **WHERE** command. WHERE (just like in any SQL query) inserts a boolean equation that is used to filter the data returned by the match statement. In this case, we define that the h1.hashtag (the hashtag's name) in the MATCH command has to be the first item found in the \`hashtag_top_20\` data frame's column \`Hashtag\` created in query 3 (i.e., the most used one). Furthermore h1 and h2 have to be different in order to avoid duplicates.
- ➔ We return the names of the **Hashtags** (h2) that co-occur with the most frequent one as well as the count of the tweets in which they appeared together. We order the results based on this count and **LIMIT** the results to 20, in order to keep only the first 20 hashtags.
- ➔ It is clear that the hashtags that co-occur the most times with #openscience are #openaccess, #opendata, #opensource, which have similar meaning with "#openscience". Furthermore, some hashtags are the concatenation of others seen in the list, thus indicating that many users prefer to use single word hashtags while others use multi word hashtags.

	Hashtag	Co-occurence
0	openaccess	60
1	opendata	40
2	opensource	27
3	researchdata	17
4	scicomm	13
5	rpa	13
6	100daysofcode	13
7	bioinformatics	13
8	compchem	12
9	science	11
10	openresearch	11
11	datascience	11
12	citizenscience	10
13	ml	9
14	fairdata	8
15	fair	8
16	rdm	8
17	peerreview	8
18	research	8
19	machinelearning	8

9. Use **PageRank** to get the most important user in the dataset.

```
co_occurence = graph.run("CALL gds.pageRank.stream({nodeProjection:
'User',relationshipProjection: 'MENTIONED'})")
most_important_user = pd.DataFrame(co_occurence)
most_important_user.rename(columns={0:'Username',1:'Importance'},inplace=Tr
ue)
influencers = most_important_user.sort_values(ascending = False,
by=['Importance'])
influencers.reset_index(inplace=True)
get_name = graph.run(f"MATCH (n:User) WHERE ID(n) =
{influencers['Username'][0]} RETURN n.username")
print(get_name)
[{'n.username': 'Aalst_Waalre'}]
```

➔ In order to find the most important user of the graph we use the well known **PageRank** algorithm which is included in the “Graph Data Science Library” Plugin. The algorithm returns a score value along with the node_id of the user who the score corresponds to. In similar fashion to the previous queries we convert the data in a dataframe in descending order and then we use a new query to correlate the node_id to their username.

➔ The username that is returned is “Aalst_Waalre”, a familiar username since it belongs to the user with the highest number of mentions and tweets.

10. Get the **hashtags** and **URLs** the **5th most important** user has posted.

```
fifth_name = graph.run(f"MATCH (n:User) WHERE ID(n) =
{influencers['Username'][4]} RETURN n.username ").data()
print(fifth_name[0]['n.username'])
get_from_fifth_urls = graph.run(f"MATCH (n1:User)-[u1:USED_URL]->(u:URL)
WHERE n1.username = '{fifth_name[0]['n.username']}' RETURN u.url").data()
display(pd.DataFrame(get_from_fifth_urls))

get_from_fifth_hash = graph.run(f"MATCH
(n1:User)-[h1:USED_HASHTAG]->(h:Hashtag) WHERE n1.username =
'{fifth_name[0]['n.username']}' RETURN h.hashtag").data()
display(pd.DataFrame(get_from_fifth_hash))
```

➔ In this query, we first **MATCH** the 5th most important user’s username given the ID of his/her node as it was found above.

➔ Then we use two additional statements to find the URLs and the hashtags this user has used. We define that we want the URL and the Hashtag nodes that are connected with the user with relationships of type ‘**USED_URL**’ and ‘**USED_HASHTAG**’.

➔ The results are presented below where the user is **‘cem_wave’** and the URLs and hashtags he used follow.

	u.url
0	https://t.co/Kul6UZ95ff
1	https://t.co/k1SxbKSARB

	h.hashtag
0	h2020
1	events
2	ceramics
3	cmc
4	materials
5	composites
6	steel

11. Get the **users** that post tweets with **hashtags most similar** to those used by the most important user.

```
influenced = graph.run(f"MATCH
(u1:User)-[h1:USED_HASHTAG]->(h2:Hashtag)<-[h3:USED_HASHTAG]-(u2:User)
WHERE u2.username = '{get_name[0]['n.username']}' RETURN u1.username,
count(h1) AS c ORDER BY c DESC").data()
influenced_users = pd.DataFrame(influenced)
influenced_users.rename(columns={'u1.username':'Username','c':'Hashtags'},i
nplace=True)
influenced_users
```

- This is a type of a query involving an indirect form of relationship. In order to get only the hashtags used by the most important user, we define that the user on the right side of the **USED_HASHTAG** relationship has to be the first one from the “get_name” list where the most important users are stored. All of the other commands used in this query follow the same principles as before. Ultimately, we have the names of the users that used these hashtags returned along with the number of the common hashtags they used with the most important user.
- Some of the users that have posted similar tweets with the most important user are presented below.

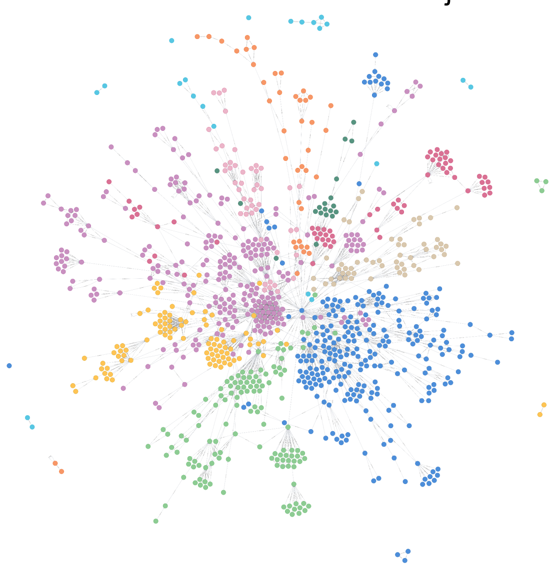
	Username	Hashtags
0	skbsacky	1
1	IsmaelAyobami	1
2	Primary_Immune	1
3	ClaudiaSittner	1
4	brightnesseu	1
5	llindamaher	1
6	m4bcn	1
7	jwwijnen	1
8	craigw94533	1
9	AmreiBahr	1
10	CoderRetweet	1

12. Get the user communities that have been created based on the users’ interactions and visualize them (**Louvain** algorithm).

```
pre_louvain = graph.run("CALL gds.louvain.write({nodeProjection:
'User',relationshipProjection:'MENTIONED',writeProperty:'community'})")
louvain = graph.run("CALL gds.louvain.stream({nodeProjection:
'User',relationshipProjection:'MENTIONED'})").data()
give_labels = graph.run("MATCH (n:User) WITH DISTINCT toString(n.community)
AS group, collect(DISTINCT n) AS persons CALL
```

```
apoc.create.addLabels(persons, [apoc.text.upperCamelCase(group)]) YIELD
node RETURN *")
communities =  pd.DataFrame(louvain)
len(communities['communityId'].unique())
communities['intermediateCommunityIds']=1
communities
```

- ➔ The **Louvain** method is an algorithm used for the detection of communities inside large networks. It follows the hierarchical clustering logic in terms of continuously merging communities together according to the parameters given.
- ➔ The parameters here define from which nodes we want the communities to be created and the type of relationship according to which we want to have our nodes separated into different groups. Louvain offers a lot of more complex ways to separate your nodes and thus it would be really useful for anyone dealing with such tasks to further review the documentations for the Louvain method on their own.
- ➔ The Louvain algorithm which was used to identify the community each node belongs to, returns the results in the format seen below. There are some communities that consist of a lot of users while others contain just a small number of participants.



	nodeId	communityId	intermediateCommunityIds
0	9372	1	1
1	9374	1	1
2	9380	3	1
3	9382	3	1
4	9383	3	1
...	...	...	...
20054	44003	20056	1
20055	44005	20056	1
20056	44006	20056	1
20057	44007	20058	1
20058	44009	20058	1

.

13. Try to visualize the **subgraph** of users that have used the **5th** most common hashtag.

```python
from neo4j import GraphDatabase
import networkx as nx

driver = GraphDatabase.driver('bolt://localhost:7687', auth=("neo4j",
"0000"))
query =f"MATCH p= (u1:User)-[h:USED_HASHTAG]->(h1:Hashtag) WHERE h1.hashtag
= '{hashtag_top_20['Hashtag'][4]}' RETURN p"
results = driver.session().run(query)
G = nx.MultiDiGraph()
nodes = list(results.graph()._nodes.values())
for node in nodes:
      G.add_node(node.id, labels=node._labels, properties=node._properties)
rels = list(results.graph()._relationships.values())
for rel in rels:
      G.add_edge(rel.start_node.id, rel.end_node.id, key=rel.id,
type=rel.type, properties=rel._properties)
nx.draw(G)
```

- First, we MATCH the users' nodes that have used the 5th common hashtag along with the "USED_HASHTAG" relationship connected with the hashtag.
- Then, we utilize the "GraphDatabase" library in order to create and visualize a **subgraph** using only the returned nodes and relationships.
- The subgraph that connects the users that have used the **5th most common hashtag** is shown below. It is obvious that the hashtag node is the one the left side since the subgraph visualizes just the connections between the users and the hashtag and not between the users.