

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

A step-by-step tutorial of how to create, manage and query/process graphs in Neo4j.



Photo from [internetdevels](#)

In today's data-driven world, the amount of information generated is growing at an unprecedented rate. Traditional databases are great at handling structured data, but they struggle with the sheer volume, variety, and complexity of today's generated data. This is where graph databases come in. But what exactly are the *graph databases*?

Graph databases are designed to handle highly connected and complex data, making them an ideal choice for applications such as social networks, recommendation engines, and fraud detection systems. In a graph database, each node represents an entity, such as a person or a product, and each edge represents a relationship between those entities, such as a purchase or a friend connection.

Neo4j is a popular graph database that is designed to store and manage data in the form of graphs.

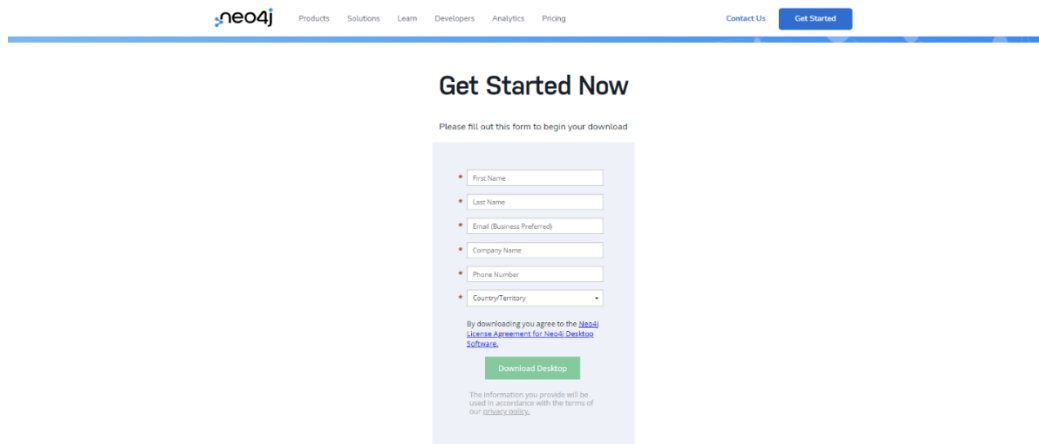
In this tutorial, we will explore how to use Neo4j and Python to analyze Twitter data. Specifically, we will discuss about:

- Installation and setup of Neo4j locally
- Retrieving Twitter data from a MongoDB database using Python
- Build nodes and relationships to populate the Neo4j Graph
- Perform queries to our graph

## Installation and setup of Neo4j

Firstly, let's download and install Neo4j for desktop. You can find the installation link [here](#).

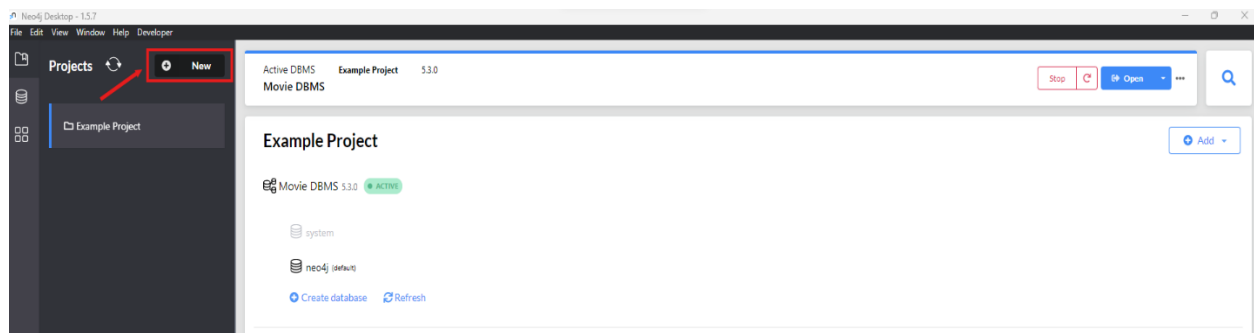
For Neo4j Desktop Edition, you need to fill in the below form and then press Download Desktop.



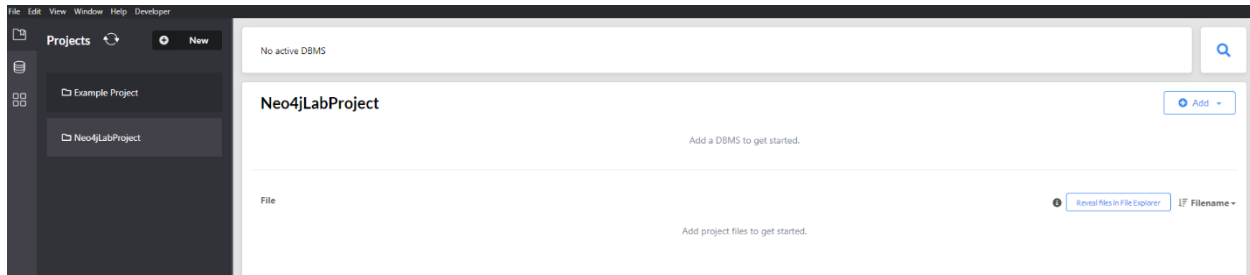
The image shows the Neo4j Desktop download form. At the top, there is a navigation bar with the Neo4j logo and links for Products, Solutions, Learn, Developers, Analytics, Pricing, Contact Us, and a Get Started button. Below the navigation bar, the heading "Get Started Now" is displayed. Underneath, a sub-heading says "Please fill out this form to begin your download". The form itself is a light blue box containing several input fields: First Name, Last Name, Email (Business Preferred), Company Name, Phone Number, and a Country/Territory dropdown menu. Below these fields, there is a link to the Neo4j License Agreement and a green "Download Desktop" button. At the bottom of the form, a small disclaimer states: "The information you provide will be used in accordance with the terms of our privacy policy."

An .exe file will be downloaded inside your Downloads folder. When you open the Neo4j Desktop app, firstly you need to choose a folder to store the application data. After that, we are ready to create our first Neo4j project.

To do so, we press the “New” button and then we choose “Create Project”. We will name our project “Neo4jLabProject”.



This will be the interface of our new project:

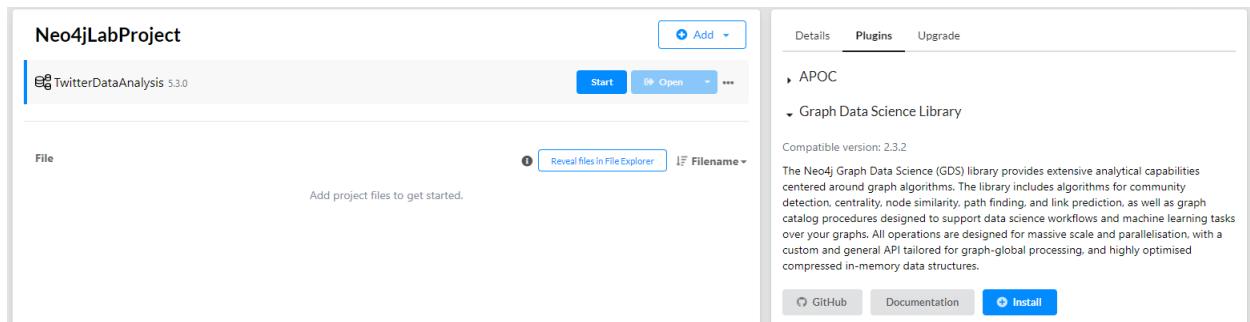


Now, we need to create a new local Database Management System (DBMS) that will be used to manage our graph. To achieve that, press the “Add” button and choose “Local DBMS”. Give a DBMS name and password and then press “Create”.

**Be careful!**

***Write down the DBMS name and password as we will need them later in order to connect to the database.***

It is recommended that we install some extra plugins that will be used later, before running the database. Specifically, we need to install the Graph Data Science library, which is a Neo4j library that provides extensive analytical capabilities based on graph algorithms. To install the library, go to the Plugins tab, select the “Graph Data Science Library” and press install.



Now we are ready to start our database, by pressing the blue “Start” button at the right side.

This may take a few seconds. You will get notified when the database is active.

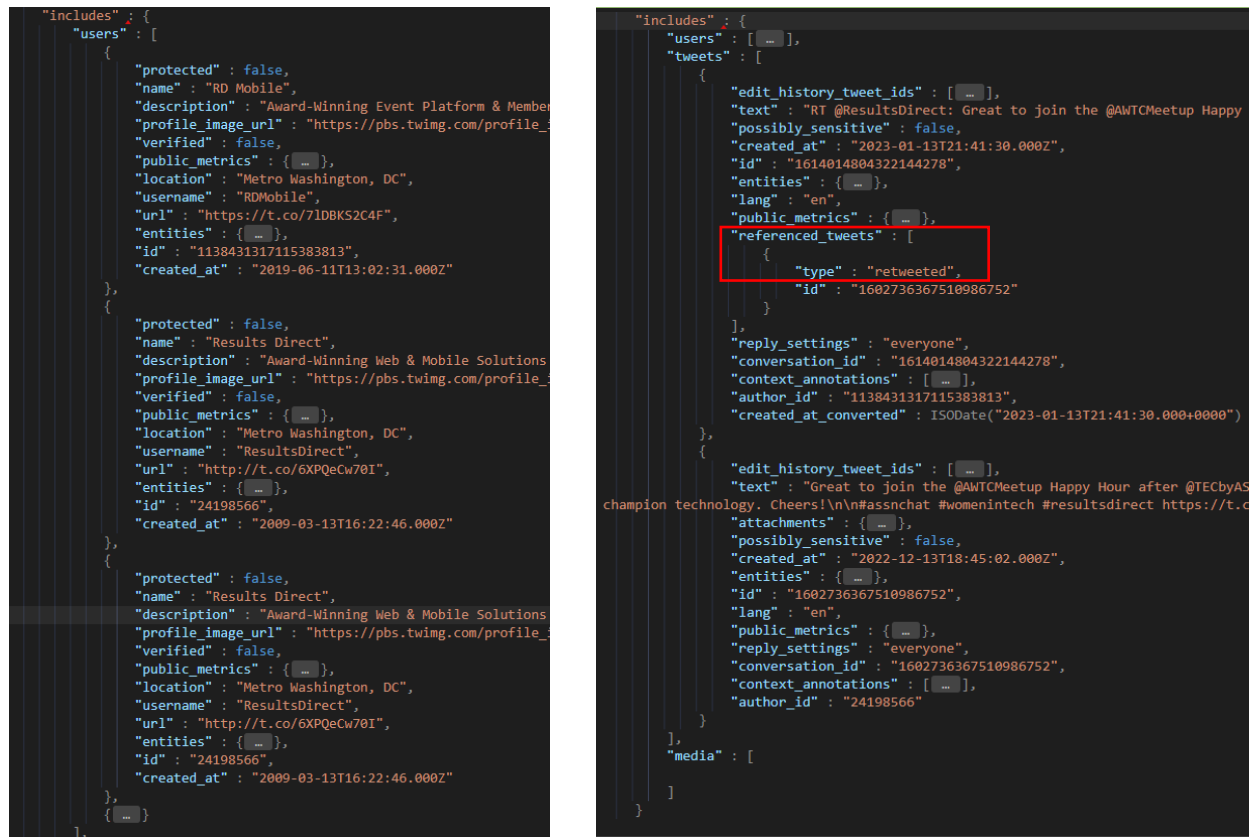


## Dataset

The dataset is a mongoexport JSON file, which contains 33,223 tweets retrieved from Twitter API.

Each document in our collection is either a tweet, a retweet, a quoted tweet or reply to tweet. The information of the type of the tweet can be taken from property “retweeted\_tweets”.

Let’s see an example of a retweet for better understanding:



```

"includes": {
  "users": [
    {
      "protected": false,
      "name": "RD Mobile",
      "description": "Award-Winning Event Platform & Member",
      "profile_image_url": "https://pbs.twimg.com/profile_...",
      "verified": false,
      "public_metrics": { "followers": 11384313, "following": 115383813 },
      "location": "Metro Washington, DC",
      "username": "RDMobile",
      "url": "https://t.co/7lDBKS2c4F",
      "entities": { "url": { "urls": [ { "url": "https://t.co/7lDBKS2c4F", "expanded_url": "https://t.co/7lDBKS2c4F", "display_url": "https://t.co/7lDBKS2c4F" } ] } },
      "id": "1138431317115383813",
      "created_at": "2019-06-11T13:02:31.000Z"
    },
    {
      "protected": false,
      "name": "Results Direct",
      "description": "Award-Winning Web & Mobile Solutions",
      "profile_image_url": "https://pbs.twimg.com/profile_...",
      "verified": false,
      "public_metrics": { "followers": 24198566, "following": 115383813 },
      "location": "Metro Washington, DC",
      "username": "ResultsDirect",
      "url": "http://t.co/6XPQeCw70I",
      "entities": { "url": { "urls": [ { "url": "http://t.co/6XPQeCw70I", "expanded_url": "http://t.co/6XPQeCw70I", "display_url": "http://t.co/6XPQeCw70I" } ] } },
      "id": "24198566",
      "created_at": "2009-03-13T16:22:46.000Z"
    }
  ],
  "tweets": [
    {
      "edit_history_tweet_ids": [ "1614014804322144278" ],
      "text": "RT @ResultsDirect: Great to join the @AWTCMeetup Happy",
      "possibly_sensitive": false,
      "created_at": "2023-01-13T21:41:30.000Z",
      "id": "1614014804322144278",
      "entities": { "mentions": [ { "username": "ResultsDirect", "id": "24198566" } ] },
      "lang": "en",
      "public_metrics": { "retweets": 1, "replies": 0 },
      "retweeted_tweets": [
        {
          "type": "retweeted",
          "id": "1602736367510986752"
        }
      ],
      "reply_settings": "everyone",
      "conversation_id": "1614014804322144278",
      "context_annotations": [ { "domain": "Technology", "title": "AI/ML", "text": "AI/ML" } ],
      "author_id": "1138431317115383813",
      "created_at_converted": "2023-01-13T21:41:30.000+0000"
    },
    {
      "edit_history_tweet_ids": [ "1614014804322144278" ],
      "text": "Great to join the @AWTCMeetup Happy Hour after @TECbyAS",
      "possibly_sensitive": false,
      "created_at": "2022-12-13T18:45:02.000Z",
      "id": "1602736367510986752",
      "entities": { "mentions": [ { "username": "ResultsDirect", "id": "24198566" } ] },
      "lang": "en",
      "public_metrics": { "retweets": 1, "replies": 0 },
      "reply_settings": "everyone",
      "conversation_id": "1602736367510986752",
      "context_annotations": [ { "domain": "Technology", "title": "AI/ML", "text": "AI/ML" } ],
      "author_id": "24198566"
    }
  ],
  "media": [
  ]
}

```

Users includes the user objects of the users mentioned in this tweet. Users[0] is a user object that always corresponds to the author of the tweet. Tweets[0] is the tweet under investigation (in this case the retweet) and the original tweet is tweet[1].

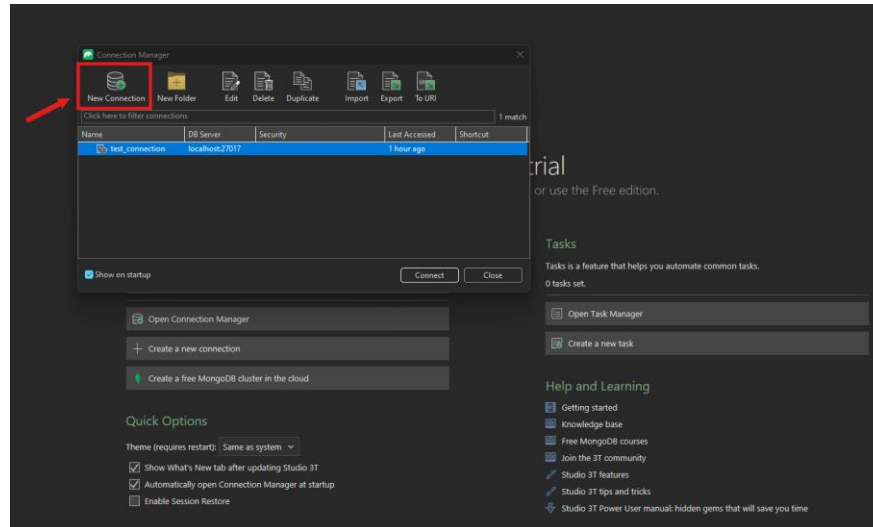
...

## MongoDB

We will use MongoDB to extract the data from the mongoexport JSON file. You can download MongoDB from the link [here](#). We also need a GUI to have a clearer view of the dataset. We use [Studio 3T](#), but also [MongoDB Compass](#) is a quite popular choice.

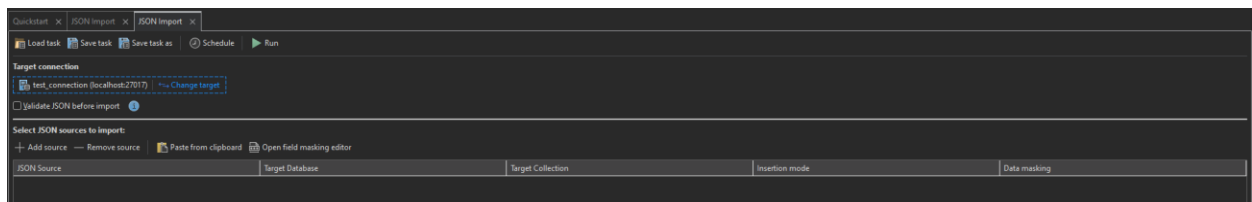
Once the installation has been completed, you need to create a 3T account and then we are ready to create our connection.

The Connection Manager window appears automatically on startup. Click “New Connection” and then select “Manually configure my connection settings”. Use the default properties and just choose a *Connection name*. Click “Save” and our new connection is ready!



Now we are ready to import the dataset.

In the Global toolbar, click “Import” and then choose JSON. Click “Configure” and you will see the below interface:



In the Target Connection, choose the connection that we made before. Then, press “+ Add source” and select the JSON file with the dataset. Then click “Run” and wait until the process finishes. Once it finishes, you will see a new database inside the connection branch, with the name of the folder that JSON is inside.

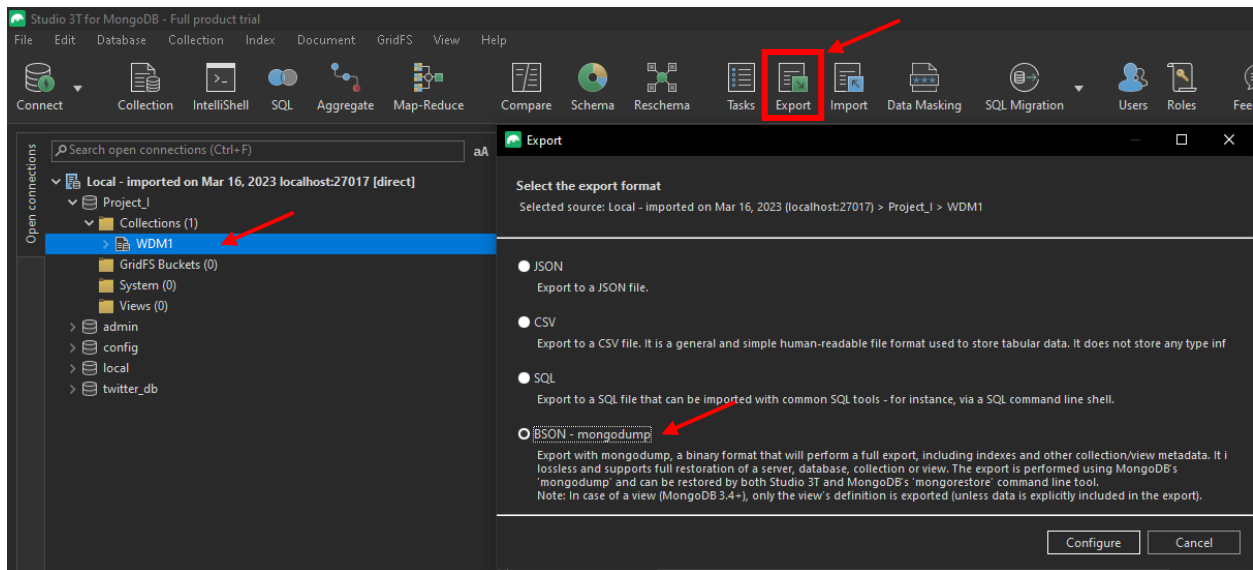
...

## Populate Graph

Awesome, now that we imported the data to MongoDB, let's start building our graph!

In order to populate the Neo4j graph using Python, firstly we chose to extract the dataset into a .bson file and then use this file for the creation of the neo4j database.

Open the dropdown menu of the new database that we made before, and inside the “Collections” folder, choose the JSON file (dataset). Then click the “Export” button in the Global toolbar and select BSON.



When you press “Configure”, you can choose the destination path of the .bson file. Then, click “Run” and wait until the process finishes.

To load the data in Python you need to decode the .bson file. To do so, open a Jupyter Notebook and install pymongo package.

```
!pip install pymongo
```

Then import “bson” library and read the .bson file and decode it as shown:

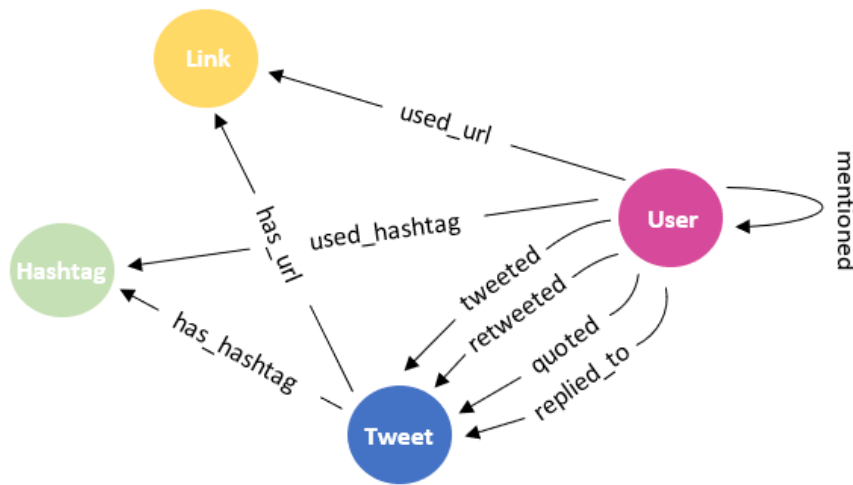
```
import bson
```

```
# Load tweet data in bson format
bson_file = open('WDM1.bson', 'rb')
data = bson.decode_all(bson_file.read())
bson_file.close()
```

Now, it’s about time to introduce the Nodes and the Relationships of our graph!

## Nodes & Relationships

The data model that we used to represent our Twitter data in Neo4j is depicted in the figure below. Each node and relationship will be discussed separately.



## Nodes

Now let's create our nodes. Always be careful not to create double nodes!

To populate Neo4j with all the nodes, we use py2neo Python library.

```
!pip install py2neo
```

Firstly, we iterate over the different documents of the data, we retrieve the information needed from every document and store it into sets.

**Hashtag:** For the node Hashtag, we only added the property **tag**, where tag is the name of the hashtag. All tags are converted to lowercase for our queries to be case insensitive. Hashtags are taken from the `includes.tweets[0].entities.hashtags.tag`, when the hashtags and tag properties exist.

**Link:** For the node Link, we only added the property **url**, where url is the name of the url. Urls are taken from the `includes.tweets[0].entities.urls.expanded_url`, when the urls and expanded\_url properties exist.

**Tweet:** For the node Tweet, we added the **properties id, created\_at, reply\_count, type** and **author\_id**. Type and author\_id properties are used later to facilitate creating the relationships between users and tweets. The tweet nodes only consist of `Tweets[0]`, which is the tweet under investigation as stated above. Tweet data are taken from the following paths:

- `includes.tweets[0].id`
- `includes.tweets[0].created_at`
- `includes.tweets[0].author_id`
- `includes.tweets[0].public_metrics.reply_count`
- `includes.tweets[0].referenced_tweets[0].type`

**User:** For the node User, we added the properties **id**, **username** and **followers\_count**. The user nodes consist of User[0], which is the author of the tweet and of the users mentioned in the tweet. User data are taken from the following paths:

- includes.users[0].id
- includes.users [0].username
- includes.users[0].public\_metrics.followers\_count

Mentioned users are taken at a second step from:

- data.entities.mentions.id
- data.entities.mentions.username

After storing the information of the nodes into sets (for uniqueness) , we use py2neo to generate neo4j nodes. The merge function in py2neo is used to either create a new node or update an existing one based on some specified criteria. In this case, the merge function is used to avoid creating duplicate nodes in the graph. The property1 parameter is used to specify the unique identifier of the node.

```
for tweet in unique_tweets:
    node = Node("Tweet", property1=tweet["id"], property2=tweet["created_at"],...)
    graph.merge(node, "Tweet", "property1")
```

## Relationships

Having our nodes, we can create the relationships between them. Firstly, we create 4 different relationships between User and Tweet to represent the kind of Tweet based on the property Tweet.type (tweet, retweet, reply or quote).

```
MATCH (u: User),(t: Tweet)
WHERE u.id = t.author_id AND t.type<> 'retweeted' AND t.type<> 'quoted' AND t.type<> 'replied_to'
MERGE (u)-[r:TWEETED]->(t)
RETURN count(r)
```

```
MATCH (u: User),(t: Tweet)
WHERE u.id = t.author_id AND t.type = 'retweeted'
MERGE (u)-[r:RETWEETED]->(t)
RETURN count(r)
```

Similarly, to create 'quoted' and 'replied\_to' relationships, modify the query in the above figure, so that t.type equals 'quoted' and 'replied\_to' accordingly.

In addition, we iterate over the documents to create relationships between User, Tweet, Hashtag and Link.

**HAS\_HASHTAG:** represents hashtag that is included on tweet.

**USED\_HASHTAG:** represents hashtags that is used by user.



**HAS\_URL:** represents url that is included on tweet.

**USED\_URL:** represents urls that is used by user.

**MENTIONS:** refers to user that is mentioned in tweet by another user. If the mentioned user does not exist in the graph, we will generate a new node for the user. It is important to check if the mentioned user does not already exist in the network and then add a new User node.

```
if 'entities' in item['data'] and 'mentions' in item['data']['entities']:
    mentions = item['data']['entities']['mentions']
    for i in range(len(mentions)):
        if 'id' in mentions[i]:
            mentions_id = mentions[i]['id']
            mentions_username = mentions[i]['username']
            # if user does not already exist in the network, add User node
            userExists = graph.run("MATCH (u:User {id: $id1}) return count(u.id)", id1 = mentions_id).data()
            if userExists[0]['count(u.id)'] == 0:
                node = Node("User", id=mentions_id, username=mentions_username)
                graph.merge(node, "User", "id")
            # User - MENTIONS - User
            mentionRel = graph.run(
                "MATCH (u1:User),(u2:User) WHERE u1.id = $user1Id and u2.id = $user2Id MERGE (u1)-[r:MENTIONS]->(u2) return count(r)"
                , user1Id = user["id"], user2Id = mentions_id).data()
```

• • •

## Queries

You can now run some cypher queries on the created Neo4J database. Keep in mind that you must have already generated the graph using the methods described in the previous sections. Here are some examples of queries that we have applied on our database.

### 1. Get the total number of tweets

```
MATCH (t:Tweet)
RETURN count(t) AS total_tweets
```

```
total_tweets
-----
Result:      33223
```

### 2. Get the total number of hashtags (case insensitive)

```
MATCH (h:Hashtag)
RETURN count(h) AS total_hashtags
```

```
total_hashtags
-----
Result:      11404
```

### 3. Get the 20 most popular URLs in descending order

```
MATCH (:User)-[r:USED_URL]->(link:Link)
RETURN link.url AS url, COUNT(r) AS url_count
ORDER BY url_count DESC
LIMIT 20
```

Result:

[illegible]

#### 4. Get the followers count of each user

```
MATCH (u:User)
WHERE u.followers_count is not null
AND u.followers_count <> "nan"
RETURN u.username AS username, u.followers_count AS followers_count
ORDER BY u.followers_count DESC
```

Result (not all users are presented):

username	followers_count
thekiranbedi	12185146
UNHumanRights	4022080
UNESCO	3675313
UN_women	2242549
bsindia	2232593
glamourmag	1302202
ELLEINDIA	1235302
CNBCTV18News	1089824
BLACKPINKGLOBAL	1000438
khaleejtimes	974472
UNGeneva	899576
thebetterindia	796657
iownjd	795479
UN_News_Centre	709651
OECD	694189
MPTourism	642233
threadreaderapp	636172

## 5. Get the number of tweets & retweets per hour

```
MATCH (u:User)-[:TWEETED]->(t:Tweet)
WHERE t.created_at IS NOT NULL
WITH datetime(t.created_at).hour AS hour, count(t) AS total_tweets
ORDER BY hour
WITH hour, total_tweets

MATCH (u:User)-[:RETWEETED]->(t:Tweet)
WHERE t.created_at IS NOT NULL
WITH hour, total_tweets, datetime(t.created_at).hour AS retweet_hour,
count(t) AS total_retweets
WHERE hour = retweet_hour
RETURN hour, total_tweets, total_retweets,
(total_tweets + total_retweets) AS total
ORDER BY hour
```

Result:

hour	total_tweets	total_retweets	total
0	258	743	1001
1	192	673	865
2	193	588	781
3	274	732	1006
4	228	708	936
5	276	729	1005
6	303	838	1141
7	304	807	1111
8	362	924	1286
9	291	943	1234
10	356	1004	1360
11	370	1138	1508
12	434	1137	1571
13	470	1214	1684
14	550	1260	1810
15	612	1371	1983
16	538	1178	1716
17	462	1192	1654
18	425	1127	1552
19	369	1059	1428
20	335	916	1251
21	349	932	1281
22	236	833	1069
23	290	841	1131

## 6. Get the user with the most replies

```
MATCH (u:User)-[r:REPLIED_TO]->(t:Tweet)
RETURN u.username , count(r) as reply_count
ORDER BY count(r) DESC LIMIT 1
```

Result:

u.username	reply_count
FatmaHasimm	54

## 7. Get the top-20 hashtags that co-occur with the hashtag that has been used the most

```
MATCH (:Tweet)-[r1:HAS_HASHTAG]->(h1:Hashtag)
WITH h1, COUNT(r1) AS count
ORDER BY count DESC
LIMIT 1

MATCH (h1)<-[r1:HAS_HASHTAG]-(:Tweet)-[r2:HAS_HASHTAG]->(h2:Hashtag)
WHERE h1 <> h2
WITH h1, h2, COUNT(r2) AS co_occur_count
ORDER BY co_occur_count DESC
LIMIT 20
RETURN h1.tag AS most_popular_hashtag,
collect({hashtag: h2.tag, co_occur_count: co_occur_count}) AS top20_list
```

### Result:

Most popular hashtag: #womenintech

Top-20 hashtags that co-occur with hashtag #womenintech:

co_occur_count	hashtag
1042	'100daysofcode'
1009	'womenwhocode'
705	'coding'
700	'python'
693	'teach'
673	'programming'
614	'javascript'
484	'codenewbie'
471	'technology'
465	'blacktechtwitter'
448	'100devs'
441	'womeninstem'
434	'devcommunity'
422	'ai'
269	'blockchain'
262	'datascience'
247	'techiewomen'
236	'ces2023'
226	'machinelearning'
224	'technie'

## 8. Get the most “important” user in the dataset. Use PageRank algorithm. Apply the algorithm in the mention network

For this query, you need to have installed the Graph Data Science Library on neo4j Desktop. We create a graph called “mentionGraph” based on the “MENTIONS” relationship.

```
CALL gds.graph.project.cypher(
  'mentionGraph',
  'MATCH (u:User) RETURN id(u) AS id',
  'MATCH (u:User)-[e:MENTIONS]->(m:User)
  RETURN id(u) AS source, e.weight AS weight, id(m) AS target')
```

```
CALL gds.pageRank.stream('mentionGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).username AS name, score
ORDER BY score DESC LIMIT 1
```

### Result:

Most important user: crazydollsnft
------------------------------------

9. For the 5th most important user, get the list of hashtags and URLs that have been posted (if no hashtags or URLs - check another user e.g. 6th, 7th , etc..)

```
CALL gds.pageRank.stream('mentionGraph')
YIELD nodeId, score
WITH gds.util.asNode(nodeId).username AS name, score
ORDER BY score DESC SKIP 4
LIMIT 1

MATCH (urls:Link)<-[[:USED_URL]]-(u:User {username: name})-
[:USED_HASHTAG]->(h:Hashtag)
RETURN name AS username, collect(DISTINCT urls.url) AS urls,
       collect(DISTINCT h.tag) AS hashtags
```

Result:

```
5th most important user: conscientious10
urls: ['https://twitter.com/conscientious10/status/1612372707936718849/photo/1']
hashtags: ['unitedkingdom', 'womenempowerment']
```

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

Next, we apply the Louvain algorithm to the "mentionGraph". This algorithm generates a new property called "communityId," which indicates the community to which a user belongs.

```
CALL gds.louvain.write(
  'mentionGraph',
  {
    writeProperty: 'communityId'
  }
)
```

Result: Total number of communities: 4606

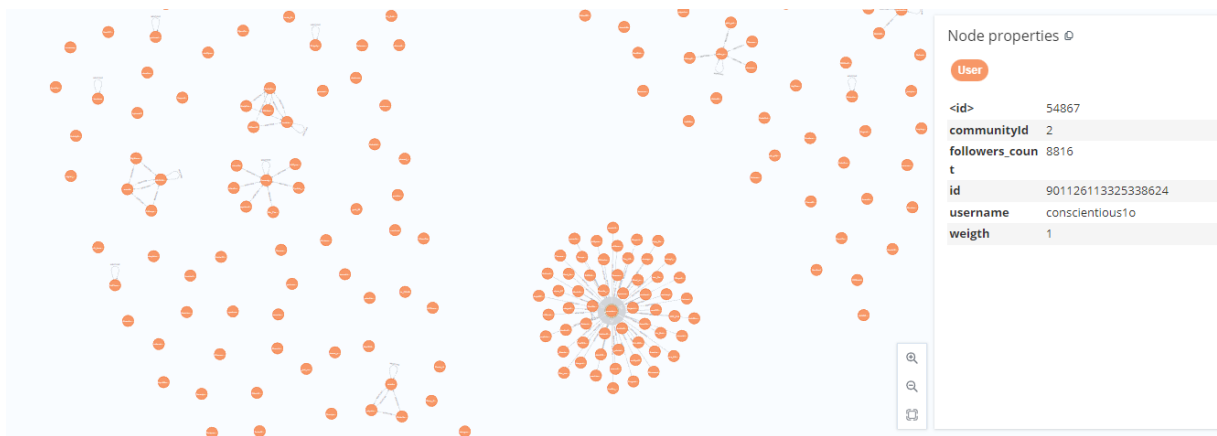
Total number of communities with multiple users: 2882

To visualize communities, we run the following query in neo4j:

```
MATCH (u1:User)-[:MENTIONS]->(:User)
RETURN u1.communityId AS community, u1 AS user
ORDER BY community
```

Result:

The displayed image is a segment of the final graph, as the actual outcome is considerably larger.



## 11. Get the most popular hashtag for each community

```
MATCH (:User)-[:MENTIONS]-(u:User)-[:USED_HASHTAG]->(h:Hashtag)
WITH u.communityId AS communityId, h.tag AS hashtag, COUNT(h.tag) AS count
ORDER BY communityId, count DESC

WITH communityId, COLLECT({tag: hashtag, count: count})[0] AS topHashtag
RETURN communityId, topHashtag.tag AS hashtag, topHashtag.count AS count
ORDER BY count DESC
```

Result (not all communities are presented):

communityId	most popular hashtag	count
20746	ai	3730
2592	giveaway	2974
1287	womenempowerment	2188
21281	womenintech	1228
964	womenintech	554
12721	genderequality	553
21221	womenempowerment	431
1505	genderequality	353
8926	live	349
371	womenempowerment	252
20994	thephotoshopartist	165
20845	womenintech	163
21361	genderequality	152
18727	genderequality	139
2078	womenintech	115
20097	womenempowerment	109
7509	justice_for_1158	108

## 12. Get the top 10 users who have tweeted the highest number of tweets that contain at least one hashtag and one URL

```
MATCH (u:User)-[:TWEETED]->(t:Tweet)-[:HAS_HASHTAG]->(h:Hashtag),
      (t)-[:HAS_URL]->(l:Link)
WITH u, count(DISTINCT t) AS numTweets
RETURN u.username AS username, numTweets
ORDER BY numTweets DESC
LIMIT 10
```

Result:

username	numTweets
-----	-----
bcsn_official	445
CourseOp	170
Lemetellusthg	142
Mahsamoulavi	124
Financedata1	124
Global1Event	123
WeSabio	98
DipsLab3	83
KerenaNicole	72
VivianAaron5	71

*I hope you found this information useful and thanks for reading!*