

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

Hands-on tutorial on how to create, manage and query/process graphs.



A graph database is one of the NoSQL data storage technologies. There are a few graph database implementations. In this tutorial, we will use the Neo4j graph database combined with Python in order to create a hands-on tutorial on how to create, manage and query/process graphs.

More specifically in this tutorial ,we shall discuss :

- Installing and setting up Neo4j locally
- Modeling our Twitter data
- Populating the Neo4j Graph
- Performing queries to our graph

Useful tools

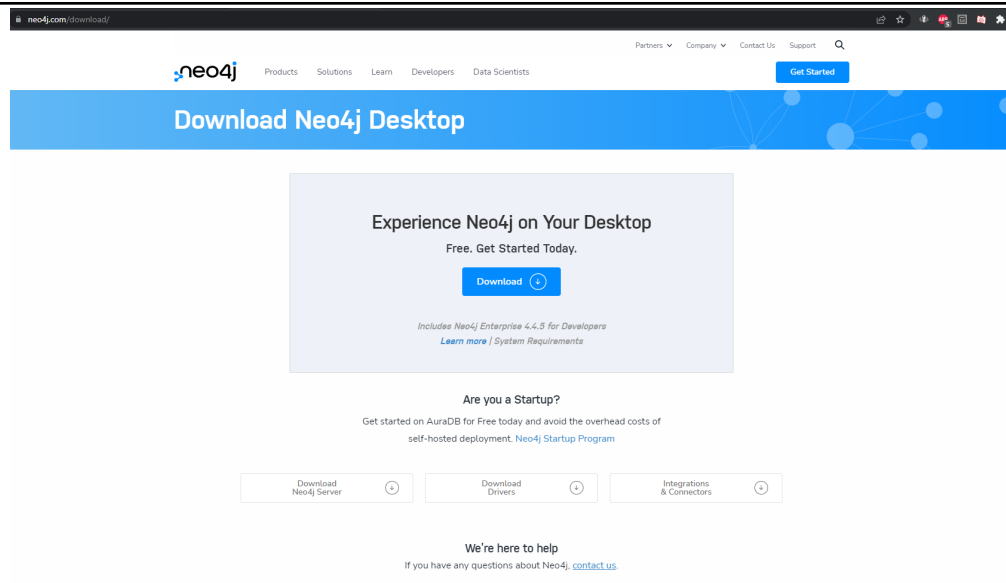
1. Neo4J Desktop
2. MongoDB or pymongo library
3. Jupyter Notebook

Install and setup Neo4J locally

The first thing we will need is to download and install Neo4J for desktop in order to create a local graph in our PC.

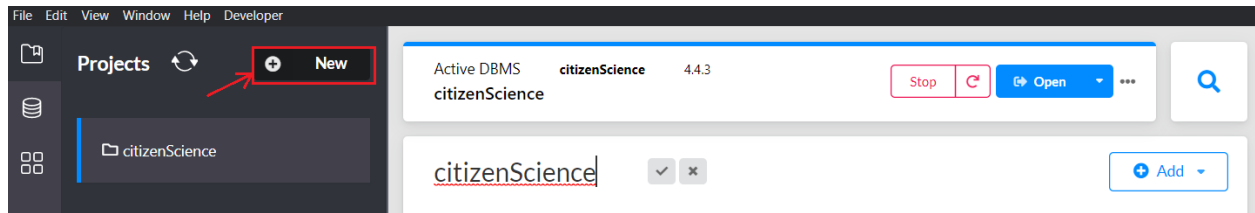
Neo4J Desktop can be found in the link below:

<https://neo4j.com/download/>

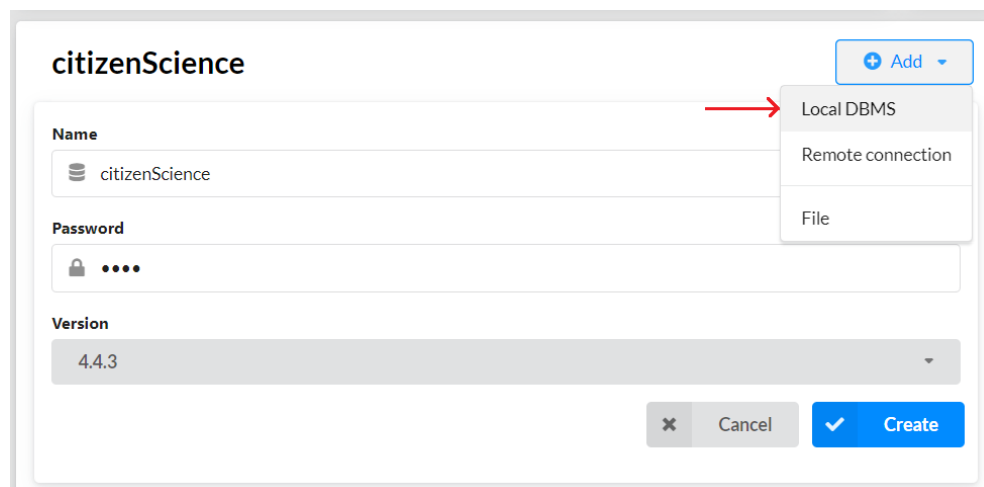


You will have to fill the form in order to download Neo4j on your desktop.

After the installation, we can open Neo4J and create our first project. To do so we press on the “New” button, as shown in the Figure below where we can set the name of our project. In our case we name our project “citizenScience”.



In our newly created project, we need to create a new local database management system (DBMS) that will be used to manage our graph. To achieve that, we press the “Add” button and we choose the DBMS name and password and press “Create”.

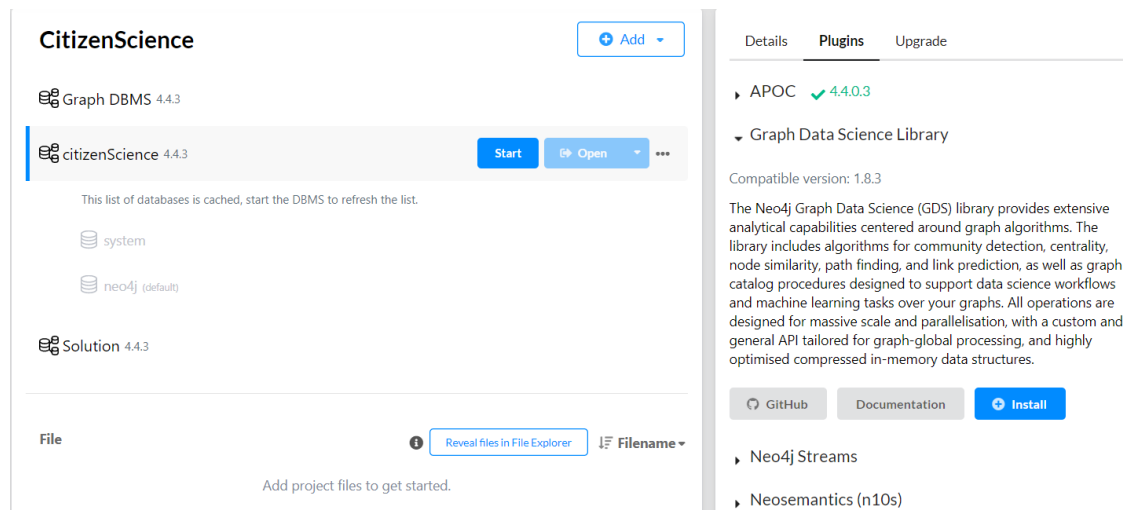


Write down the DBMS name and password as we will need them later in order to connect to the database.

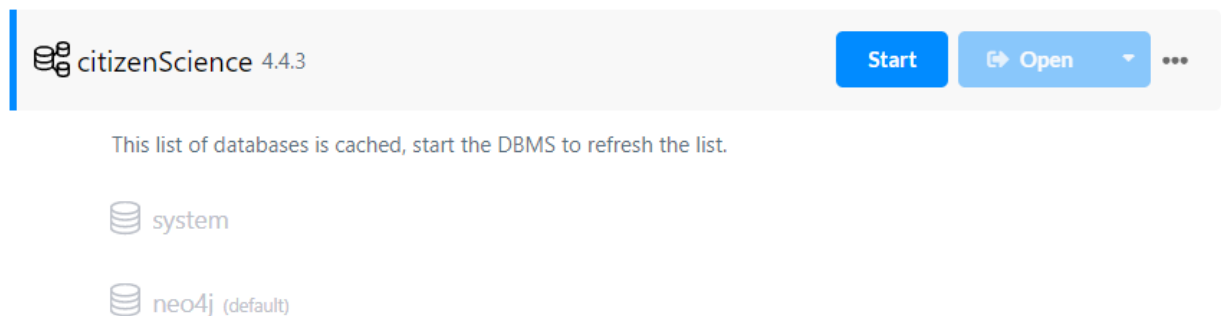
Before we start running the database, it is recommended to install some extra plugins that we will use later.

More specifically, we need to install the APOC library which will help us with string similarity and subgraph visualization and Graph Data Science library, which is a Neo4j library that provides extensive analytical capabilities centered around graph algorithms. In our tutorial, we will use some algorithms like PageRank or Betweenness centrality.

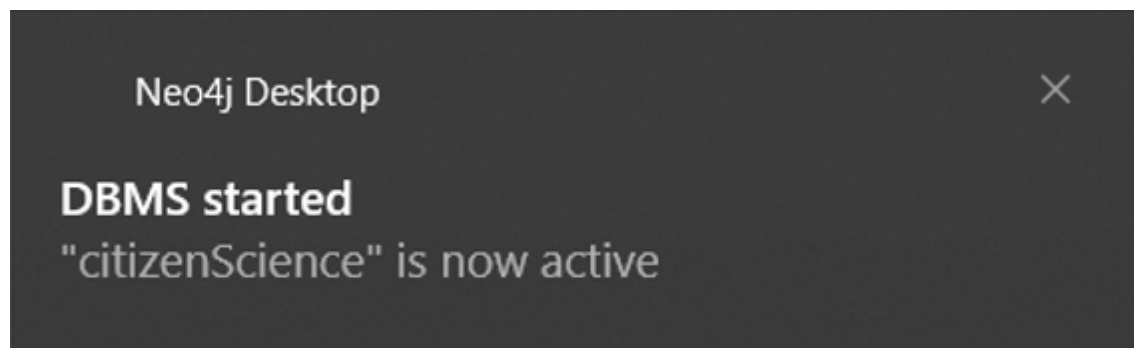
In order to proceed with the installation, we need to go to the Plugins tab, select the “Graph Data Science Library” and simply press install.



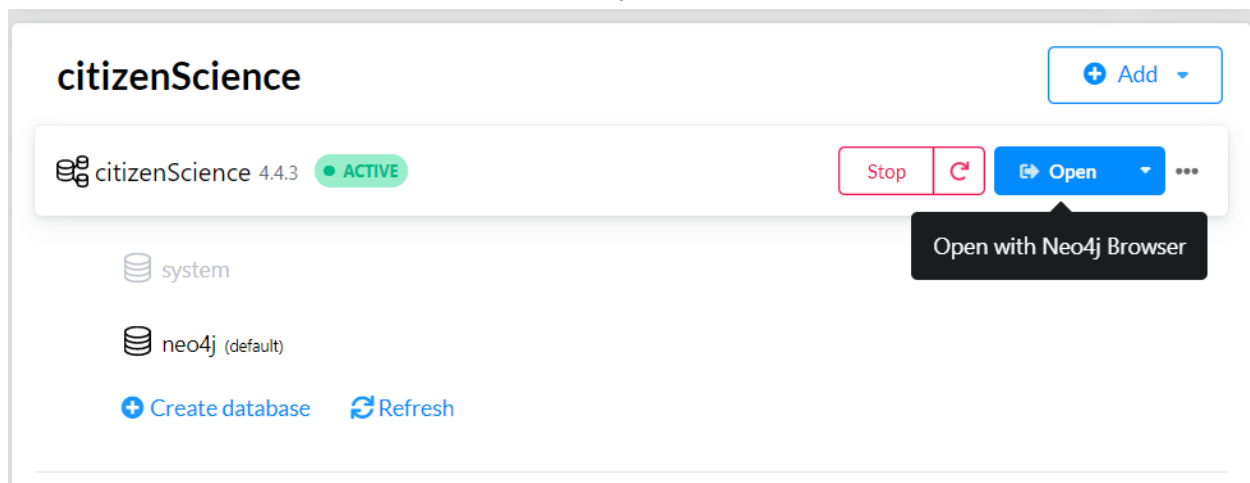
Now we are ready to run the database. We click on the “Start” button and wait some seconds in order for the database to start functioning.



When the DBMS has launched and is ready to be used, a notification will pop up that will inform us that the database is now active.

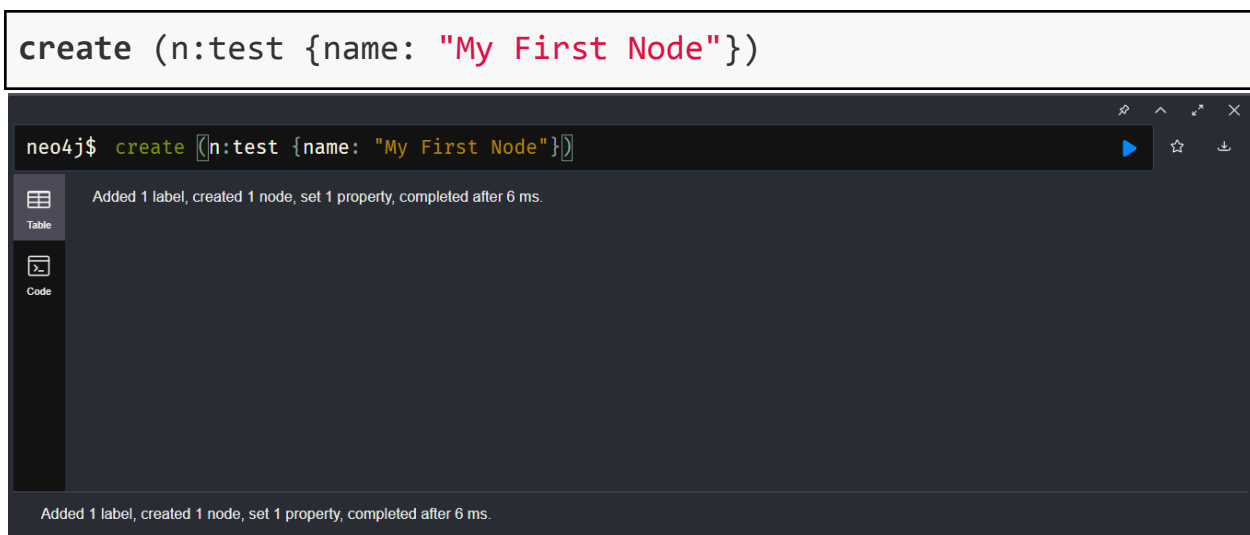


Neo4j offers the Neo4j browser which is a handy tool for developers to interact with the graph and let us perform cypher queries and visualize data results. In order to use it , click on the “Open” button and it will open a new window in your browser.



Let's send some test cypher queries to check our setup. Cypher is a flexible language, you can create nodes, relationships and even both at the same time.

We will create our first node that will have the label “test” and one property “name” with the value “My first node”. To do so we execute the query below:



Now let's retrieve all the nodes with label "test" from our graph, it should return just one result :

```
match (n:test) return n
```

The screenshot shows the Neo4j Desktop application. The query editor at the top contains the command `neo4j$ match (n:test) return n`. Below the editor, the results are shown in a table format. The table has two rows: the first row contains the variable `n`, and the second row contains a JSON object `{ "name": "My First Node" }`. The interface includes a sidebar on the left with icons for Graph, Table, Text, and Code. The status bar at the bottom indicates "MAX COLUMN WIDTH" with a slider.

| |
|-----------------------------|
| n |
| { "name": "My First Node" } |

And finally let's delete all nodes from our graph as those were just some initial testing queries.

```
match (n) delete n
```

The screenshot shows the Neo4j Desktop application. The query editor at the top contains the command `neo4j$ match (n) delete n`. Below the editor, the results are shown in a table format. The table has one row containing the message `Deleted 1 node, completed after 1 ms.`. The interface includes a sidebar on the left with icons for Table and Code. The status bar at the bottom indicates "Deleted 1 node, completed after 1 ms."

| |
|---------------------------------------|
| Deleted 1 node, completed after 1 ms. |
|---------------------------------------|

That's it! Now our Neo4j environment is ready to be filled with data and start producing graphs!

Dataset

Now we can actually start doing some data analysis with Python. For the sake of this project, we are going to use [this dataset](#) , which contains more than 30.000 tweets of citizens.

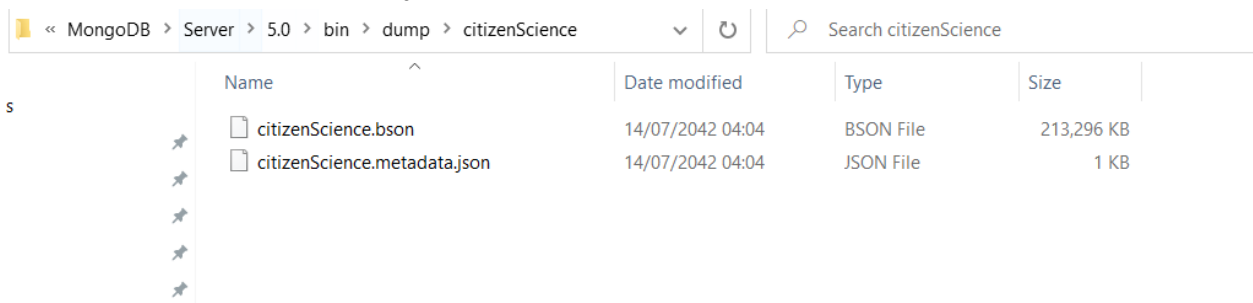
Extract Data

We will present two ways to extract the data from the bson and metadata json files in order to import them on Neo4j.

1. Using MongoDB

The first way is with the help of MongoDB. Since our data is given in bson format we can import them to our Mongo Database and use them by connecting Python with our MongoDB. The steps that need to be followed are :

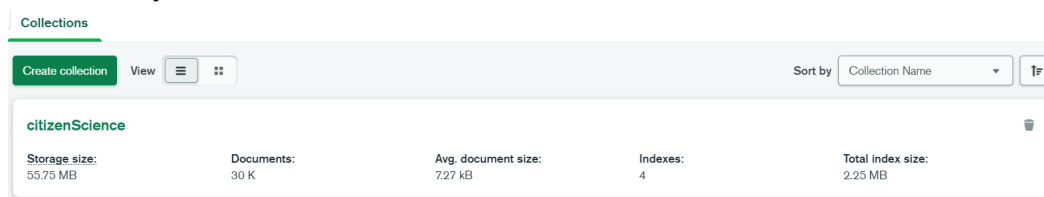
1. Download and install your MongoDB through link [MongoDB Compass | MongoDB](#).
2. Download the MongoDB database tools through link [Download MongoDB Command Line Database Tools | MongoDB](#).
3. Unzip the file and add the files we got from zip to MongoDB's bin folder. Usually we can find it in the path Program Files > MongoDB > 5.0 > bin.
4. In the same folder, create a folder named **dump**. Inside this folder create a file with the name we want our database to have (e.g. "citizenScience") and inside this folder add the bson and metadata json file.



5. Open a command screen with administrator rights and navigate to the bin folder and run "mongorestore"

```
D:\Program Files\MongoDB\Server\5.0\bin>mongorestore
```

That's it! Now, the database should be created with data from bson file and index keys taken from the metadata json file.



After we have successfully imported the data to our MongoDB, in order to use them on our Neo4j, we need to establish a connection between them. This can be achieved using Python as the mediator.

Therefore, using a Jupyter Notebook we install pymongo and import pymongo.MongoClient which will be used to connect to the mongod instance.

Finally we specify the Database name and the name of the collection inside the database and we store the collection in the variable **data** that we will use to access our data.

```
In [ ]: pip install pymongo

from pymongo import MongoClient

In [ ]: #Step 1: Connect to MongoDB - Note: Change connection string as needed
client = MongoClient("mongodb://localhost:27017/")
data=[item for item in client.data.citizenScience.find()]
```

2. Decode bson file

The second way to extract the data is to decode the bson file directly without the need to load the data in mongoDB. To do so we follow the steps below:

- We open a Jupyter Notebook and we install pymongo package with the command bellow
 - **pip install pymongo**
- We import “bson” library contained within the package we installed earlier
 - **import bson**
- Then we need to read the bson file and decode it as shown bellow

Read Data From BSON File

```
In [5]: bson_file = open('data\citizenScience.bson', 'rb')
data = bson.decode_all(bson_file.read())
```

Now we have all our data inside the **data** variable and we can start building our graph

Population of the Neo4j Graph

Before we start the population of our graph, let's explain briefly the format of our data.

Each document in our collection is either a tweet or a retweet. The noticeable difference between the two of them is the property **"retweeted_status"**.

TWEET

```
.id: ObjectId("6213aa701b2e4a22fce6b211")
created_at: "Mon Feb 21 15:06:19 +0000 2022"
id: 1495776905131446283
id_str: "1495776905131446283"
text: "Nice discussion with @ShaniEvenstein about #openscience and #opendata ..."
source: "<a href='\"https://mobile.twitter.com\"' rel='\"nofollow\">Twitter Web App</a...\"
truncated: true
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
user: Object
  geo: null
  coordinates: null
  place: null
  contributors: null
  is_quote_status: false
  extended_tweet: Object
    quote_count: 0
    reply_count: 0
    retweet_count: 0
    favorite_count: 0
  entities: Object
    favorited: false
    retweeted: false
    possibly_sensitive: false
    filter_level: "low"
  lang: "en"
  timestamp_ms: "1645455979031"
  created_at_converted: 2022-02-21T15:06:19.000+00:00
```

RETWEET

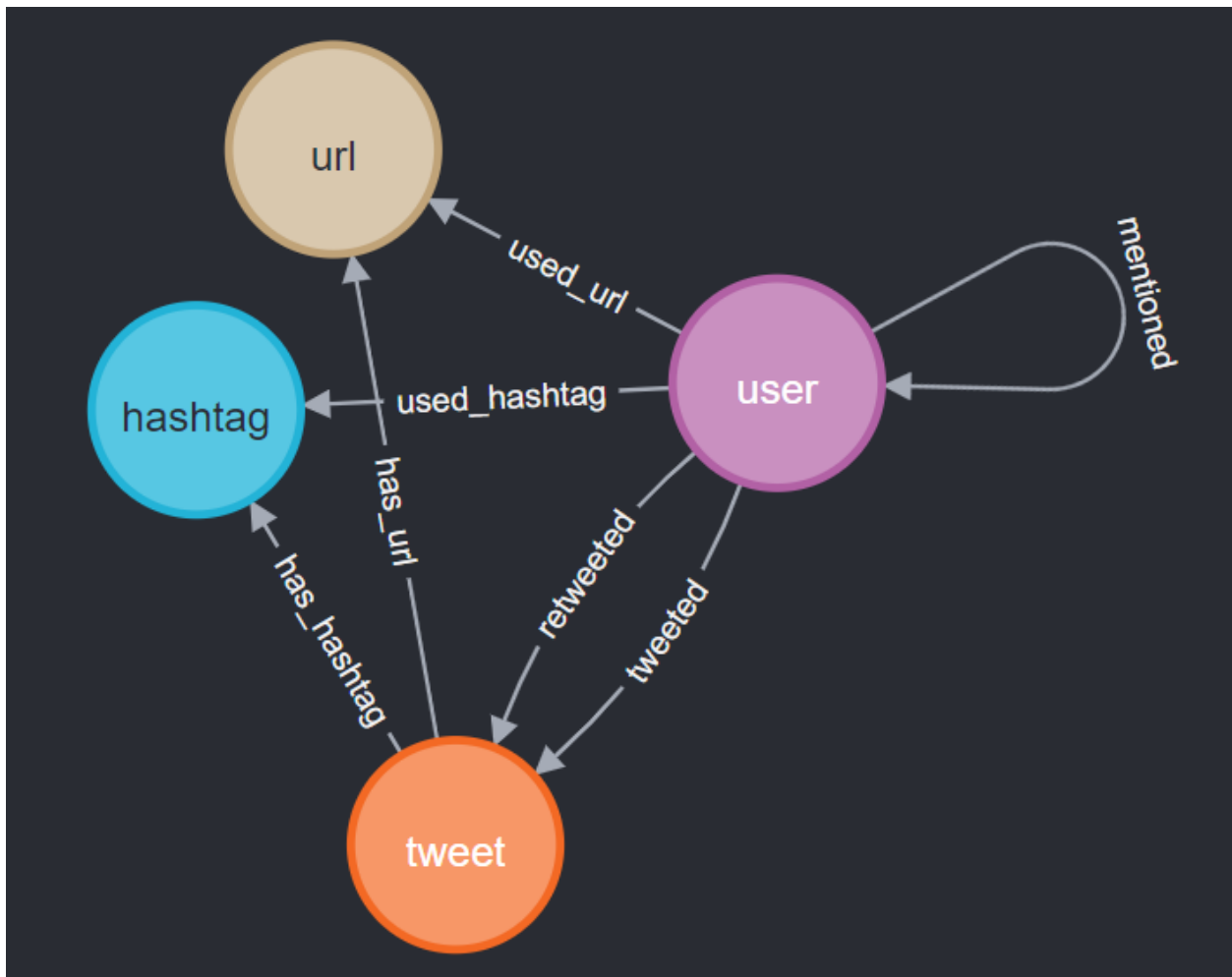
```
.id: ObjectId("6213aa8e1b2e4a22fce6b212")
created_at: "Mon Feb 21 15:06:49 +0000 2022"
id: 1495777032059461632
id_str: "1495777032059461632"
text: "RT @NASAGoddard: A region found by the Galaxy Zoo citizen science proj..."
source: "<a href='\"http://twitter.com/download/android\"' rel='\"nofollow\">Twitter f..."
truncated: false
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
user: Object
  geo: null
  coordinates: null
  place: null
  contributors: null
  retweeted_status: Object ←
  is_quote_status: false
  quote_count: 0
  reply_count: 0
  retweet_count: 0
  favorite_count: 0
  entities: Object
    favorited: false
    retweeted: false
    filter_level: "low"
  lang: "en"
  timestamp_ms: "1645456009293"
  created_at_converted: 2022-02-21T15:06:49.000+00:00
```

More specifically, if it is a retweet, there exists the property **"retweeted_status"**, which contains the original tweet. When we have a retweet, we gather all our data from the property **"retweeted_status"**, which contains a nested document with all the data of a tweet.

If the property **"retweeted_status"** doesn't exist, we gather the data from the root of the document because this row is not a retweet but a new original tweet.

In the next steps below, when we refer to "original tweet" we mean the tweet which is either the one gathered from **"retweeted_status"** or the one gathered from the root of the row.

The Data model that we used to represent our Twitter data in Neo4j is as shown in the below figure. We will discuss separately each node and relationship.



We have used the Neo4j browser to generate the picture by using the following command:

```
CALL db.schema.visualization()
```

Nodes

- User – represents a Twitter user
- Tweet – represents a tweet
- Hashtag - represents a hashtag
- URL - represents a URL

Now let's create our nodes. Below we list all the nodes we create:

(Be careful not to create double nodes! Check before entering a new one! :))

User

For the node **User**, we added the properties **screen_name** and **followers_count**, where:

- Screen_name, otherwise known as handle, is the name with which a user identifies himself and it is a unique string
- Followers_count is the number of the followers of the user

In order to populate Neo4j with all the users, we are going to use the following query, where user is the label of our node and inside {} are the properties we want to insert in this node :

```
CREATE (user_name:user {screen_name:$screen_name,
followers_count:$followers_count})
```

Before creating it, you can check if it already exists with this query:

```
MATCH (u:user) WHERE u.screen_name = $screen_name RETURN u.screen_name
```

Of course, remember to substitute \$screen_name and \$followers_count with the correspondent values you get from your data.

For each tweet in our loaded data, there are three different spots to get a new user. New users can be found in the paths:

- **row.retweeted_status.user** : This path exists when the row is a retweet and it contains the user of the original tweet
- **row.user** : If the path mentioned above exists, this path contains the user that did a retweet on the original tweet else this is the user of the original tweet (no retweet at this case)
- **original tweet.entities.user_mentions** : This path contains all the users mentioned in the tweet
- **original tweet.extended_tweet.user_mentions** : This path exists when the tweet is truncated. It contains all the users mentioned, even the truncated.

In the following figure, we display an example row and all the paths where we gather new user nodes :

```

_id: ObjectId("6213ac4a1b2e4a22fce6b223")
created_at: "Mon Feb 21 15:14:12 +0000 2022"
id: 1495778892254257156
id_str: "1495778892254257156"
text: "RT @nairaworkers: Nairaworkers is a source: "<a href="https://mobile.twitter.co
truncated: false
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
> user: Object
  geo: null
  coordinates: null
  place: null
  contributors: null
  retweeted_status: Object
    created_at: "Sun Sep 20
    id: 1307603562206244064
    id_str: "130760356220624
    text: "Nairaworkers is a
    display_text_range: Array
    source: "<a href="http://
    truncated: true
    in_reply_to_status_id: r
    in_reply_to_status_id_s
    in_reply_to_user_id: nul
    in_reply_to_user_id_str
    in_reply_to_screen_name
  > user: Object
    geo: null
  > extended_tweet: Object
    full_text: "Extremely proud to see our
    display_text_range: Array
    entities: Object
      hashtags: Array
      urls: Array
      user_mentions: Array
      > 0: Object
        screen_name: "Academictox"
        name: "Jon Martin"
        id: 2859281312
        id_str: "2859281312"
      indices: Array
      > 1: Object
      > 2: Object
      symbols: Array
  > user: Object
    id: 1441739717758705668
    id_str: "1441739717758705668"
    name: "Prince jidenna"
    screen_name: "prince_jidenna"
    location: null
    url: null
    description: "Born to win"
    translator_type: "none"
    protected: false
    verified: false
    followers_count: 62
    friends_count: 176
    listed_count: 0

```

Tweet

For the node **Tweet**, we added the properties **id**, **text** and **favorites_count**, where:

- Id is the id of the tweet
- Text is the text of tweet
- Favorites_count is the number of favorites of tweet

The tweet nodes consist of only original tweets. For this reason, in case retweeted status exists in an item, we get the tweet data only from there. In case it doesn't exist, it means that our item is an original tweet and so we take our tweet from the item root as explained.

```

created_at: "Fri Feb 18 21:08:29 +0000 2022"
id: 1494780884251250695
id_str: "1494780884251250695"
text: "A region found by the Galaxy Zoo citizen science project, this @NASAHu..."
display_text_range: Array
source: "<a href="https://www.sprinklr.com" rel="nofollow">Sprinklr</a>"
truncated: true
in_reply_to_status_id: null
in_reply_to_status_id_str: null
in_reply_to_user_id: null
in_reply_to_user_id_str: null
in_reply_to_screen_name: null
> user: Object
  geo: null
  coordinates: null
  place: null
  contributors: null
  is_quote_status: false
  > extended_tweet: Object
    quote_count: 8
    reply_count: 8
    retweet_count: 112
    favorite_count: 698
  > entities: Object
    favorited: false
    retweeted: false
    possibly_sensitive: false
    filter_level: "low"
    lang: "en"

```

Hashtag

For the node **Hashtag**, we added the property **text**, where:

- Text is the name of the hashtag

Hashtags are taken from the path original tweet.extended_tweet.entities.hashtags and original tweet.entities.hashtags of the original tweets.

```

  > extended_tweet: Object
    full_text: "The 1st edition of the #ELIXIR "Ready for BioData Management?: Data St..."
    > display_text_range: Array
    > entities: Object
      > hashtags: Array
        > 0: Object
        > 1: Object
        > 2: Object
        > 3: Object
      > urls: Array
      > user_mentions: Array
      > symbols: Array
      > media: Array
    > extended_entities: Object
    quote_count: 0
    reply_count: 0
    retweet_count: 7
    favorite_count: 1
  > entities: Object
    > hashtags: Array
      > 0: Object
    > urls: Array
    > user_mentions: Array
    > symbols: Array
    > favorite: false

```

URL

For the node **Url**, we added the properties **text**, where:

- Url is the name of the URL

Urls are taken from the original tweet.extended_tweet.entities when the extended_tweet property exists. If it doesn't exist, they are taken only from original tweet.entities. This is done because when we have an extended_tweet, in the path original tweet.entities we have the url of the tweet, but this is something that we don't need to gather.

```

  coordinates: null
  place: null
  contributors: null
  is_quote_status: false
  > extended_tweet: Object
    full_text: "The 1st edition of the #ELIXIR "Ready for BioData Management?: Data St..."
    > display_text_range: Array
    > entities: Object
      > hashtags: Array
      > urls: Array
        > 0: Object
          url: "https://t.co/Pk6ku2Xsnx"
          expanded_url: "http://BioData.pt"
          display_url: "BioData.pt"
          > indices: Array
        > user_mentions: Array
        > symbols: Array
        > media: Array
      > extended_entities: Object
    quote_count: 0
    reply_count: 0
    retweet_count: 7
    favorite_count: 1
  > entities: Object
    > hashtags: Array
    > urls: Array
      > 0: Object
        url: "https://t.co/sj5neVZUDQ"
        expanded_url: "https://twitter.com/i/web/status/1495758827098095620"
        display_url: "twitter.com/i/web/status/1_"
        > indices: Array
      > user_mentions: Array
      > symbols: Array

```

Note: It is important to check the existence of the node before we proceed with its creation. In case of existence, an update needs to be performed in its properties. Also we need to iterate on the data in chronological order in order to gather the latest values in our graph. Lastly, it is important to get the expanded_url and not the url (which is the expanded url shorted) because sometimes two different short urls lead us to the same expanded url.

Relationships

Now that we have a plan for our nodes, we need to create the relationships between them. Relationships are created in a slightly different way than nodes, for example:

```
MATCH (u:user), (t:tweet)
WHERE u.screen_name = $screen_name AND t.id = $tweet_id
CREATE (u)-[r:tweeted {created_at:$created_at, device:$device}]->(t)
```

This query first finds the user node with screen_name = \$screen_name and the tweet node with id = \$id then creates a directed relationship from tweet node to hashtag node with label tweeted and properties created_at and device. Of course, if we don't want any properties, we can skip them.

We have constructed relationships below:

Tweeted

- **(user)-[tweeted]->(tweet) :**
 - These relationships are built between the original tweets and its users. It has the property "timestamp" which is the value of the property "created_at" of the original tweet converted into the format "YYYYMMDDhhmmss".

Retweeted

- **(user)-[retweeted]->(tweet) :**
 - These relationships are built between the original tweet and the users that retweeted this tweet. It has the property "timestamp" which is the value of the property "created_at" of the original tweet converted into the format "YYYYMMDDhhmmss".

Has_hashtag

- **(tweet)-[has_hashtag]->(hashtag)**
 - These relationships are built between the original tweets and the hashtags that they use

Has_url

- **(tweet)-[has_url]->(url)**
 - These relationships are built between the original tweets and the urls that they use

Used_hashtag

- **(user)-[used_hashtag]->(hashtag)**
 - These relationships are built between the users and the hashtags that they use in their tweets (not including retweets)

Used_url

- **(user)-[used_url]->(url)**
 - These relationships are built between the users and the urls that they use in their tweets (not including retweets)

Mentioned

- **(user)-[mentioned]->(user)**
 - These relationships are built between the users and the users they mention in their tweets. When a user retweets another's user tweet, this is also counted as a mention.

In order to create the graph, we can use the Neo4j python library or the py2neo library. Below is an example on how to connect to Neo4j and create nodes and relationships with **py2neo library** :

```
from py2neo import Node, Relationship, Graph

#Connect to neo4j
graph = Graph(neo4j_url, username=user, password=pass)

#Create a node objects
node1 = Node("node_label", property_1 = value11, property_2 = value12, ...)
node2 = Node("node_label", property_1 = value21, property_2 = value22, ...)
#Create the nodes in graph from the node objects
graph.create(node1)
graph.create(node2)
#Create a relationship object
Rel = Relationship(node1, "relationship_label", node2, properties)
#Create the relationship from the relationship objects
graph.create(rel)
```

Below is an example on how to connect to Neo4j and create nodes and relationships with **Neo4j library** :

```
from neo4j import GraphDatabase

#Connect to neo4j :
session = GraphDatabase.driver(uri, auth=(username, password)).session()

#Create nodes and relationship through cypher commands, example :
session.run("CREATE (node:node_label {property_1:value_1,
property_2:value_2})")
```

Otherwise, we can get the data from our mongodb to Neo4j without the use of python and create our data with the help of apoc.mongodb for example with the following query we get all the users

```
CALL apoc.mongodb.get('mongodb://localhost:27017', 'action',
'citizenScience', {})
YIELD value
RETURN value.user AS user
```

That's it! Now that we have all the tools we need to create our graph.
On this [github link](#) you may find an example code in order to reproduce.

Queries

You can follow along by running the queries in your Neo4J app. Remember, you should have already created the graph in a way mentioned in previous steps.

1. Get the total number of tweets

```
MATCH (tweets:tweet)
RETURN count(tweets) AS Number_of_tweets
```

Result:

```
[{'Number_of_tweets': 9409}]
```

2. Get the total number of retweets

```
MATCH (:user)-[retweets:retweeted]->(:tweet)
RETURN count(retweets) AS Number_of_retweets
```

Result:

```
[{'Number_of_retweets': 20745}]
```

3. Get the total number of hashtags (case insensitive)

```
MATCH (hashtags:hashtag)
RETURN count(toLower(hashtags.text)) AS Number_of_hashtags
```

Result:

```
[{'Number_of_hashtags': 5324}]
```

4. Get the 20 most popular hashtags (case insensitive) in descending order

```
MATCH (hashtags:hashtag)
RETURN DISTINCT toLower(hashtags.text) AS hashtag ,
size((:user)-[:used_hashtag]->(hashtags)) AS indegree
ORDER BY indegree DESCENDING LIMIT 20
```

Result:

```
[{'hashtag': 'openscience', 'indegree': 4492},
{'hashtag': 'citizenscience', 'indegree': 2506},
{'hashtag': 'openaccess', 'indegree': 641},
{'hashtag': 'scicomm', 'indegree': 562},
{'hashtag': 'opendata', 'indegree': 433},
{'hashtag': 'publicengagement', 'indegree': 427},
{'hashtag': 'crowdsourcing', 'indegree': 385},
{'hashtag': 'opensource', 'indegree': 369},
{'hashtag': 'datascience', 'indegree': 299},
{'hashtag': 'bioinformatics', 'indegree': 267},
{'hashtag': 'sb19', 'indegree': 257},
{'hashtag': 'stanworld', 'indegree': 255},
{'hashtag': 'westanpeace', 'indegree': 255},
{'hashtag': 'research', 'indegree': 255},
{'hashtag': 'westanlove', 'indegree': 255},
{'hashtag': 'citsci', 'indegree': 253},
{'hashtag': 'ukraine', 'indegree': 245},
{'hashtag': 'machinelearning', 'indegree': 234},
{'hashtag': 'covid19', 'indegree': 231},
{'hashtag': 'biodiversity', 'indegree': 222}]
```

5. Get the total number of URLs

```
MATCH (urls:url)
RETURN count(urls) AS Number_Of_URLs
```

Result:

```
[{'Number_Of_URLs': 3915}]
```

6. Get the 20 most popular URLs in descending order

```
MATCH (urls:url)
RETURN urls.url AS url, size((:user)-[:used_url]->(urls)) AS indegree
ORDER BY indegree DESCENDING LIMIT 20
```

Result:

```
[{'indegree': 433, 'url': 'http://apne.ws/CCCiUpB'},
{'indegree': 212, 'url': 'http://www.nairaworkers.com'},
{'indegree': 163, 'url': 'https://www.cisa.gov/shields-up'},
{'indegree': 105, 'url': 'https://osf.io/preprints/metaarxiv/zry2u'},
```

```

{'indegree': 95,
 'url':
'https://plantfunctionaltraitscourses.w.uib.no/pftc6-norway-sign-up-now/'},
{'indegree': 75, 'url':
'https://www.nature.com/articles/d41586-022-00402-1'},
{'indegree': 72,
 'url':
'https://ceh-online-surveys.onlinesurveys.ac.uk/pollinator-citizen-science-
across-europe'},
{'indegree': 64,
 'url': 'https://employment.ku.edu/postdoctoral-researcher/21278br'},
{'indegree': 64,
 'url': 'https://apnews.com/article/f2c4960e48b8022a567780f3602b54e2'},
{'indegree': 60,
 'url':
'https://www.atlanticcouncil.org/blogs/ukrainealert/new-crowdsourcing-campa
ign-can-help-save-ukraine/'},
{'indegree': 58,
 'url':
'https://www.humboldt-foundation.de/fileadmin/Bewerben/Programme/Philipp-Sc
hwartz-Initiative/PSI_Special_provisions_Ukraine_25_Feb._2022.pdf'},
{'indegree': 51, 'url': 'https://osf.io/preprints/metaarxiv/b9qaw'},
{'indegree': 51, 'url': 'https://raspberrysake.org'},
{'indegree': 50,
 'url':
'https://www.ajtmh.org/view/journals/tpmd/aop/article-10.4269-ajtmh.21-1010
/article-10.4269-ajtmh.21-1010.xml?rskey=63sqBe&result=2'},
{'indegree': 49, 'url': 'https://bit.ly/3kihpmP'},
{'indegree': 49,
 'url':
'https://www.outreachy.org/blog/2022-02-04/may-2022-initial-applications-op
en/'},
{'indegree': 48, 'url': 'https://osec2022.eu/program'},
{'indegree': 44, 'url': 'http://dx.doi.org/10.1098/rsos.211042'},
{'indegree': 43, 'url': 'https://www.ecsa.sa.gov.au/enrolment'},
{'indegree': 43, 'url': 'https://rdcu.be/cltoT'}]

```

7. Get the followers count of each user

```
MATCH (users:user)
RETURN sum(users.followers_count) AS Total_followers_count
```

```
MATCH (users:{1_user})
RETURN users.screen_name AS User, users.followers_count AS followers
```

Result:

```
[{'Total_followers_count': 118603522}]
[{'User': 'AndGenomics', 'followers': 2369},
 {'User': '_lewtun', 'followers': 2195},
 {'User': 'Raamana_', 'followers': 2997},
 {'User': 'JayHeltzer', 'followers': 1305},
 {'User': 'Canada_CEBCEM', 'followers': 456},
 {'User': 'OutTeachEd', 'followers': 2222},
 {'User': 'zehavoc', 'followers': 4667},
 {'User': 'EcoPol_Arg', 'followers': 642},
 {'User': 'AvilaLovera', 'followers': 266},
 {'User': 'seed_ball', 'followers': 52582},
 {'User': 'bigmeadowsearch', 'followers': 465},
 {'User': 'LicenceProDist', 'followers': 205},
 . . . . . ]
```

8. Get the 20 users with most followers in descending order

```
MATCH (users:user)
RETURN users.screen_name AS user, users.followers_count AS followers
ORDER BY followers DESCENDING LIMIT 20
```

Result:

```
[{'followers': 17519323, 'user': 'Forbes'},
 {'followers': 15630637, 'user': 'AP'},
 {'followers': 4897983, 'user': 'coinbase'},
 {'followers': 4286962, 'user': 'marcorubio'},
 {'followers': 3048702, 'user': 'NWS'},
 {'followers': 2940260, 'user': 'verge'},
 {'followers': 1882525, 'user': 'britishlibrary'},
 {'followers': 1751550, 'user': 'BoredElonMusk'},
 {'followers': 1333989, 'user': 'TheRickWilson'},
 {'followers': 1262006, 'user': 'zeerajasthan'},
 . . . . . ]
```

```
{'followers': 1227496, 'user': 'NSF'},
{'followers': 1027275, 'user': 'deray'},
{'followers': 893280, 'user': 'thidakarn'},
{'followers': 802165, 'user': 'NASAGoddard'},
{'followers': 784534, 'user': 'campbellclare'},
{'followers': 763029, 'user': 'NASAJuno'},
{'followers': 744498, 'user': 'NASASun'},
{'followers': 721944, 'user': 'UNESCOarabic'},
{'followers': 643790, 'user': 'nrc'},
{'followers': 636812, 'user': 'BMWUSA']}
```

9. Get the number of tweets & retweets per hour

First we get the total number of tweets:

```
MATCH ()-[r:tweeted]->()
RETURN DISTINCT count(r) AS tweets
```

Then we get the earliest and latest timestamp from these tweets:

```
MATCH (:user)-[r:tweeted]->()
RETURN min(toInteger(r.timestamp)) AS min, max(toInteger(r.timestamp)) AS max
```

Our timestamp is in format YYYYMMDDhhmmss. For our next query we should change it to format YYYY-MM-DDThh:mm:ss.000. (ex. 20220301123811 -> 2022-03-01Thh:mm:ss)

And we execute the below query with the new min and max timestamps

```
UNWIND [ duration.inSeconds(localdatetime(min_date),
localdatetime(max_date)) ] AS aDuration
RETURN aDuration
```

The duration we get is in format P[nY][nM][nW][nD][T[nH][nM][nS]] (P is prefix for duration) and we can translate it to hours with the below python function

```
def duration_in_hours(duration):
    # Unit-based form: P[nY][nM][nW][nD][T[nH][nM][nS]] (P is prefix for
    duration)
    years=months=weeks=days=hours=minutes=seconds = 0
    duration=str(duration)
    if "T" in duration:
        x,y = duration.split("T",1)
    else:
        x = str(duration)
    _,x = x.split("P",1)
    if 'Y' in x:
        years,x = x.split("Y",1)
```

```

if 'M' in x:
    months,x = x.split("M",1)
if 'W' in x:
    weeks,x = x.split("W",1)
if 'D' in x:
    days,x = x.split("D",1)

if y:
    if 'H' in y:
        hours,y = (y.split("H",1))
    if 'M' in y:
        minutes,y = (y.split("M",1))
    if 'S' in y:
        seconds,y = (y.split("S",1))
if int(minutes) > 30:
    return (int(years)*8760) + (int(months)*730) + (int(weeks)*168) +
(int(days)*24) + int(hours) + 1
else:
    return (int(years)*8760) + (int(months)*730) + (int(weeks)*168) +
(int(days)*24) + int(hours)

```

We follow the same procedure to calculate retweets per hour.

Result:

| |
|---|
| Tweets per hour: 0.23174306051575083 Retweets per hour: 58.436619718309856 |
|---|

Notice: Tweets per hour are significantly less than Retweets per hour. This happens because even though the data collection happened between February and March of 2022, someone retweeted a tweet that had been posted in 2017.

10. Get the hour with the most tweets and retweets

We will execute this query for every hour of the day. For example prevHour = 00 and currentHour = 01.

```

MATCH (:user)-[relationships]->(:tweet)
WHERE substring(relationships.timestamp,8,6) >= '{prevHour}' and
substring(relationships.timestamp,8,6) <= '{currentHour}'
RETURN count(relationships) AS c

```

Result:

```
[{'hour': '00-01', 'tweets': 765},
 {'hour': '01-02', 'tweets': 749},
 {'hour': '02-03', 'tweets': 807},
 {'hour': '03-04', 'tweets': 809},
 {'hour': '04-05', 'tweets': 1005},
 {'hour': '05-06', 'tweets': 828},
 {'hour': '06-07', 'tweets': 880},
 {'hour': '07-08', 'tweets': 1151},
 {'hour': '08-09', 'tweets': 1441},
 {'hour': '09-10', 'tweets': 1632},
 {'hour': '10-11', 'tweets': 1531},
 {'hour': '11-12', 'tweets': 1455},
 {'hour': '12-13', 'tweets': 1422},
 {'hour': '13-14', 'tweets': 1520},
 {'hour': '14-15', 'tweets': 1710},
 {'hour': '15-16', 'tweets': 1669},
 {'hour': '16-17', 'tweets': 1777},
 {'hour': '17-18', 'tweets': 1755},
 {'hour': '18-19', 'tweets': 1478},
 {'hour': '19-20', 'tweets': 1445},
 {'hour': '20-21', 'tweets': 1318},
 {'hour': '21-22', 'tweets': 1076},
 {'hour': '22-23', 'tweets': 1098},
 {'hour': '23-00', 'tweets': 0}]
```

Hour with most tweets & retweets : 16-17

11. Get the device that most users are tweeting from (top 5 devices)

```
MATCH (:{user})-[tweeted]->(:tweet)
RETURN DISTINCT tweeted.source AS device, count(tweeted.source) AS
times_used
ORDER BY times_used DESCENDING LIMIT 5
```

```
[{'device': 'Twitter Web App', 'times_used': 9508},
 {'device': 'Twitter for Android', 'times_used': 5975},
 {'device': 'Twitter for iPhone', 'times_used': 5962},
 {'device': 'OpenSciTalk', 'times_used': 1852},
 {'device': 'TweetDeck', 'times_used': 841}]
```

12. Get the users, in descending order, that have been mentioned the most

```
MATCH ()-[r:mentioned]->(u:user)
RETURN u.screen_name AS user, count(r) AS mentions
ORDER BY count(r) DESCENDING LIMIT 20
```

```
[{'mentions': 257, 'user': 'SB190Official'},
 {'mentions': 217, 'user': 'TheRickWilson'},
 {'mentions': 116, 'user': 'doctorow'},
 {'mentions': 116, 'user': 'zitrtrain'},
 {'mentions': 110, 'user': 'CitSciOZ'},
 {'mentions': 96, 'user': 'PFTCourses'},
 {'mentions': 95, 'user': 'raspishake'},
 {'mentions': 79, 'user': 'NIH'},
 {'mentions': 75, 'user': 'FORRTproject'},
 {'mentions': 70, 'user': 'inaturalist'},
 {'mentions': 68, 'user': 'OPERASEU'},
 {'mentions': 68, 'user': 'Flavio_Azevedo_'},
 {'mentions': 67, 'user': 'cOAlitionS_OA'},
 {'mentions': 64, 'user': 'AGUecohydro'},
 {'mentions': 64, 'user': 'LandonMarston'},
 {'mentions': 61, 'user': 'UniLeipzig'},
 {'mentions': 61, 'user': 'SueReviews'},
 {'mentions': 61, 'user': 'AutismINSAR'},
 {'mentions': 59, 'user': 'AvHStiftung'},
 {'mentions': 58, 'user': 'ScienceEurope'}]
```

13. Get the most active users (users that have posted most tweets)

```
MATCH (u:user)-[r:tweeted]-()
RETURN count(r) AS tweets , u.screen_name AS user
ORDER BY count(r) DESCENDING LIMIT 20
```

```
[{'tweets': 698, 'user': 'Aalst_Waalre'},
 {'tweets': 101, 'user': 'RobotRrid'},
 {'tweets': 98, 'user': 'OpenSci_News'},
 {'tweets': 90, 'user': 'raspishakEQ'},
 {'tweets': 66, 'user': 'Primary_Immune'},
 {'tweets': 65, 'user': 'DG_Rand'},
 {'tweets': 42, 'user': 'moneynetlink'}]
```



```
{'tweets': 38, 'user': 'DocCrenau'},
{'tweets': 33, 'user': 'egonwillighagen'},
{'tweets': 32, 'user': 'AlanSheehan18'},
{'tweets': 32, 'user': 'citizenskies'},
{'tweets': 30, 'user': 'HeidiProject'},
{'tweets': 30, 'user': 'MDDelahunty'},
{'tweets': 28, 'user': 'for_designer'},
{'tweets': 28, 'user': 'CreativeSage'},
{'tweets': 22, 'user': 'Treadstone71LLC'},
{'tweets': 22, 'user': 'pivottwistdev'},
{'tweets': 22, 'user': 'CitieSHealthEU'},
{'tweets': 21, 'user': 'BitcoinORama'},
{'tweets': 21, 'user': 'fdmhildesheim']}
```

14. Get the top 20 tweets that has been retweeted the most and the persons that posted them

```
MATCH (tweet:tweet)<-[:retweeted]-(users:user)
RETURN DISTINCT tweet, users.screen_name AS user,
size((tweet)<-[:retweeted]-()) AS times_retweeted
ORDER BY times_retweeted DESCENDING LIMIT 20
```

Results:

```
[{'times_retweeted': 934,
  'tweet_id': '1499064794829131779',
  'user': 'PigsAndPlans'},
{'times_retweeted': 428,
  'tweet_id': '1499959375536001025',
  'user': 'AP'},
{'times_retweeted': 252,
  'tweet_id': '1500684178114760706',
  'user': 'acetwtts'},
{'times_retweeted': 243,
  'tweet_id': '1499579773382967296',
  'user': 'commissionsbyk'},
{'times_retweeted': 212,
  'tweet_id': '1307603562206244864',
  'user': 'nairaworkers'},
{'times_retweeted': 203,
  'tweet_id': '1497580495533711362',
  'user': 'paulUKcoder'},
  . . . . . ]
```

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

First we have to get the hashtag that has been used the most.

```
MATCH ()-[r:used_hashtag]->(h:hashtag)
RETURN h.text AS most_popular, count(r)
ORDER BY count(r) DESC LIMIT 1
```

Then we get the 20 top hashtags that co-occur with that.

```
MATCH (t:tweet)-[r:has_hashtag]->(h:hashtag)
WHERE EXISTS{{
MATCH (t)-[r2:{has_hashtag}]->(h2:{hashtag})
WHERE h2.text = '{most_popular}' }}
AND h.text <> '{most_popular}'
RETURN h.text AS hashtag, count(r) AS count
ORDER BY count DESC LIMIT 20
```

Result:

```
Most used hashtag : openscience
[{'count': 305, 'hashtag': 'openaccess'},
 {'count': 145, 'hashtag': 'opendata'},
 {'count': 137, 'hashtag': 'scicomm'},
 {'count': 101, 'hashtag': 'opensource'},
 {'count': 96, 'hashtag': 'datascience'},
 {'count': 93, 'hashtag': 'bigdata'},
 {'count': 79, 'hashtag': 'bioinformatics'},
 {'count': 67, 'hashtag': 'python'},
 {'count': 63, 'hashtag': 'genomics'},
 {'count': 60, 'hashtag': '100daysofcode'},
 {'count': 60, 'hashtag': 'research'},
 {'count': 51, 'hashtag': 'rstats'},
 {'count': 49, 'hashtag': 'coding'},
 {'count': 48, 'hashtag': 'machinelearning'},
 {'count': 46, 'hashtag': 'covid19'},
 {'count': 45, 'hashtag': 'immunology'},
 {'count': 45, 'hashtag': 'serverless'},
 {'count': 41, 'hashtag': 'fairdata'},
 {'count': 39, 'hashtag': 'linux'},
 {'count': 39, 'hashtag': 'citizenscience'}]
```

16. Get the most “important” user in the dataset (use Graph algorithms: Pagerank, Betweenness centrality, etc.). You will apply these algorithms in the mention network (which includes retweets)

To find the most important users, we use the PageRank algorithm.

We are also using the followers count of each user as a weight between the relationship between two users. We give more weight to inbound relationships of users with most followers because theoretically the most followers a user has the most connections with other users it has. Another reason why we used the followers count as a weight for the calculation of the most important user, is because we don't have the followers count for some mentioned users as we have already discussed.

So the first thing we do it to find the user with most followers in order to normalize our weight:

```
MATCH (users:user)
RETURN max(users.followers_count) AS max
```

Then we create a new 'weight' property to all “mentioned” relationships. The value of this property is the followers count of the mentioned user divided by the maximum followers count found above.

```
MATCH (:user)-[r:mentioned]->(users:user)
SET r.weight = (users.followers_count * 1.0 + 0.1) / max
```

Note: We add 0.1 on followers count in order to have some, non zero, weight even for users without followers.

Now that we have weighed all the user to user relationships, we can save a graph of users in the catalog and execute the weighted PageRank algorithm.

```
CALL gds.graph.create(
  'userMentionsGraph',
  'user',
  'mentioned',
  {nodeProperties: 'followers_count', relationshipProperties:'weight' }
)
```

After saving our graph, we can use the PageRank algorithm in order to get the top user and also the top 5 users :

```
CALL gds.pageRank.stream('userMentionsGraph',
  { relationshipWeightProperty: 'weight' })
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).screen_name AS user, score
ORDER BY score DESC, user ASC LIMIT 5
```

Results:

```
Most important user: inaturalist
Top 5 users: ['inaturalist', 'doctorow', 'SueReviews', 'WHCEQ', 'PLOSONE']
```

17. For the 5th most important user, get the list of hashtags and URLs that have been posted

We get the 5th most important user from the results of question 16 and gather all hashtags and urls that he used.

```
MATCH (user:user)-[r]->(nodes)
WHERE (nodes:hashtag or nodes:url) and user.screen_name = 'PLOSONE'
RETURN nodes.text AS hashtag, nodes.url AS url
```

Results:

```
5th Most important user: PLOSONE
Hashtags used: ['openscience']
Urls used: []
```

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

For this question we get the most important user found in question 16 and search for the hashtags it used.

```
MATCH (u:user)-[r:used_hashtag]->(h:hashtag)
WHERE u.screen_name = 'inaturalist'
RETURN h.text AS hashtags
```

For each hashtag we gather with the query above, execute the query below in order to get the users that used similar hashtags (we look for 80% similarity and above):

```
MATCH (u)-[r:used_hashtag]->(h:hashtag)
WITH apoc.text.sorensenDiceSimilarity('{hashtag}', h.text) AS similarity,
      u.screen_name AS screen_name,
      h.text AS hashtag
WHERE similarity > 0.8 AND similarity <> 1
RETURN screen_name, hashtag, AS similarity LIMIT 5
```

Results:

```
[['User', 'Hashtag', 'Hashtag used by important user', 'Similarity'],
 ['KeithPiccard', 'scienceed', 'science', 0.8571428571428571],
 ['nettie087', 'sciences', 'science', 0.9230769230769231],
```

```
[ 'OpenSciTalk', 'sciences', 'science', 0.9230769230769231],
[ 'amarois', 'sciences', 'science', 0.9230769230769231],
[ 'oxygenases', 'myscience', 'science', 0.8571428571428571],
[ 'CitSciOZ', 'citizensciencekits', 'citizenscience', 0.8666666666666667],
[ 'CitSciWA', 'citizensciencekits', 'citizenscience', 0.8666666666666667],
[ 'SciStarter', 'citizensciencekits', 'citizenscience',
0.8666666666666667],
[ 'GrundyLibrary', 'citizensciencekits', 'citizenscience',
0.8666666666666667],
[ 'citnatchallenge', 'citizenscienceforall', 'citizenscience', 0.8125],
[ 'CitSciOZ', 'citizensciencekits', 'citizenscience', 0.8666666666666667],
[ 'CitSciWA', 'citizensciencekits', 'citizenscience', 0.8666666666666667],
[ 'SciStarter', 'citizensciencekits', 'citizenscience',
0.8666666666666667],
[ 'GrundyLibrary', 'citizensciencekits', 'citizenscience',
0.8666666666666667],
[ 'citnatchallenge', 'citizenscienceforall', 'citizenscience', 0.8125],
[ 'kyliesoanes', 'urbanbiodiversity', 'biodiversity', 0.8148148148148148],
[ 'nettie087', 'bibliodiversity', 'biodiversity', 0.88],
[ '_open_science_', 'bibliodiversity', 'biodiversity', 0.88],
[ 'OpenSciTalk', 'bibliodiversity', 'biodiversity', 0.88],
[ 'ChristopheDony', 'bibliodiversity', 'biodiversity', 0.88],
[ 'kyliesoanes', 'urbanbiodiversity', 'biodiversity', 0.8148148148148148],
[ 'nettie087', 'bibliodiversity', 'biodiversity', 0.88],
[ '_open_science_', 'bibliodiversity', 'biodiversity', 0.88],
[ 'OpenSciTalk', 'bibliodiversity', 'biodiversity', 0.88],
[ 'ChristopheDony', 'bibliodiversity', 'biodiversity', 0.88],
[ 'kyliesoanes', 'urbanbiodiversity', 'biodiversity', 0.8148148148148148],
[ 'nettie087', 'bibliodiversity', 'biodiversity', 0.88],
[ '_open_science_', 'bibliodiversity', 'biodiversity', 0.88],
[ 'OpenSciTalk', 'bibliodiversity', 'biodiversity', 0.88],
[ 'ChristopheDony', 'bibliodiversity', 'biodiversity', 0.88]]
```

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

We store a graph with our users in the catalog using the cypher command below:

```
CALL gds.graph.create(
  'GraphforLouvain',
  'user',
  {mentioned:
    {orientation: 'UNDIRECTED'}}
```

```

    },
    { nodeProperties: 'followers_count' }
  )

```

Then with the use of the Louvain algorithm, we group our users in communities by storing their community id in a property called 'community'.

```

CALL gds.louvain.write(
  'GraphforLouvain',
  {writeProperty: 'community'}
)

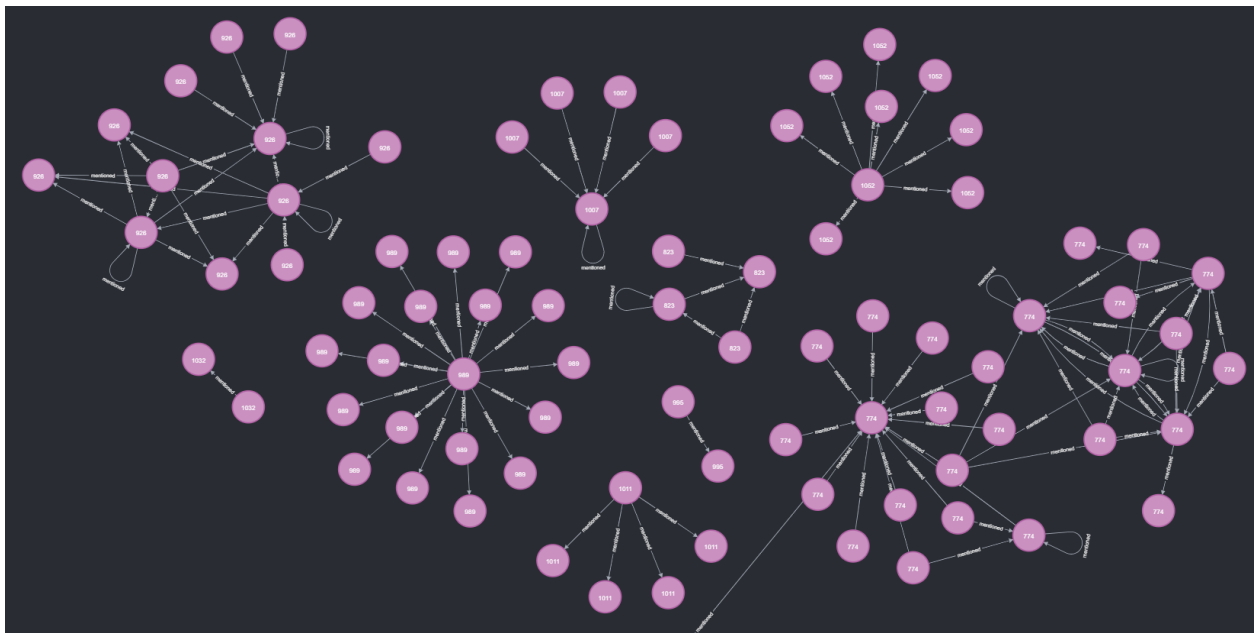
```

To visualize the communities we execute the query below in neo4J Browser.

```

MATCH (users:user)-[:mentioned]-(:user)
RETURN users.community AS communityID, users AS user ORDER BY communityID

```



The image shown is a part of the resulting graph because the actual result is much bigger.

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

First we get the most common hashtag using the query below:

```

MATCH (hashtags:hashtag)
RETURN DISTINCT toLower(hashtags.text) AS hashtag, size()-[]->(hashtags))

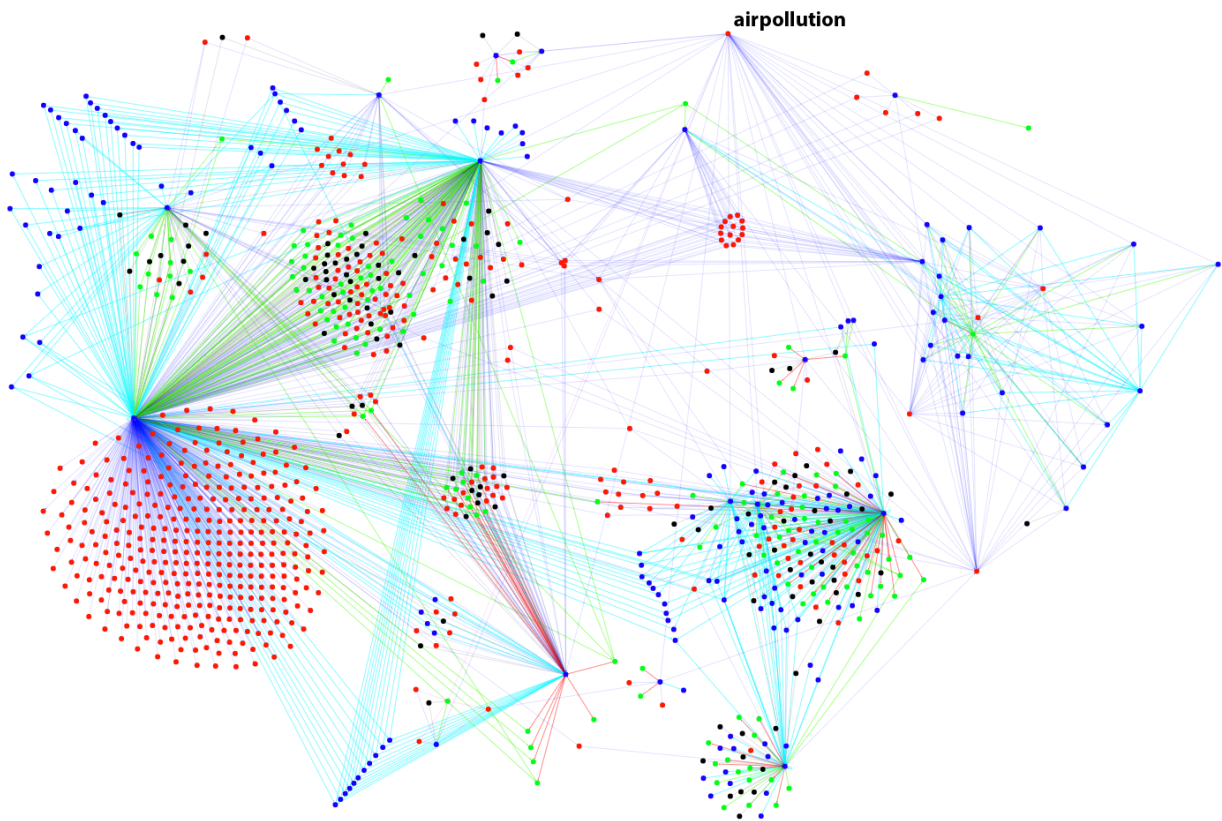
```












```
AS indegree
ORDER BY indegree DESCENDING LIMIT 5
```

With the query above, we find that the 5th most common hashtag is 'airpollution'. We then execute the command below to get the subgraph of the users that used the hashtag 'airpollution':

```
MATCH (users:user)-[:used_hashtag]-(hashtags:hashtag)
WHERE hashtags.text = 'airpollution'
CALL apoc.path.subgraphAll(users, {maxLevel: 1}) YIELD nodes, relationships
RETURN nodes,relationships
```

Using neo4J Bloom, we can visualize the produced subgraph, the result can be seen below.



|  | has_hashtag | 4 | | |
|---|--------------|-----|---|-------------|
|  | has_url | 1 | | |
|  | mentioned | 357 | | |
|  | retweeted | 221 | | |
|  | tweeted | 56 | | |
|  | used_hashtag | 965 | | |
|  | used_url | 183 | | |
| | | | STYLE | SELECTION |
| | | |  | hashtag 560 |
| | | |  | tweet 153 |
| | | |  | url 104 |
| | | |  | user 183 |

We hope you found this information useful and thanks for reading!

Authors

1. Chatziara Dimitra, 98*
2. Kalyvas Emmanouil, 93*
3. Grigoroudis Efstratios, 74*

** All authors contributed equally to this tutorial*