**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.



Photo from [internetdevels](https://internetdevels.com/blog/using-neo4j-graph-database-part1)

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**

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
* Retrieve 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 link [here](https://neo4j.com/download/).

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

Graphical user interface, application, Word

Description automatically generated

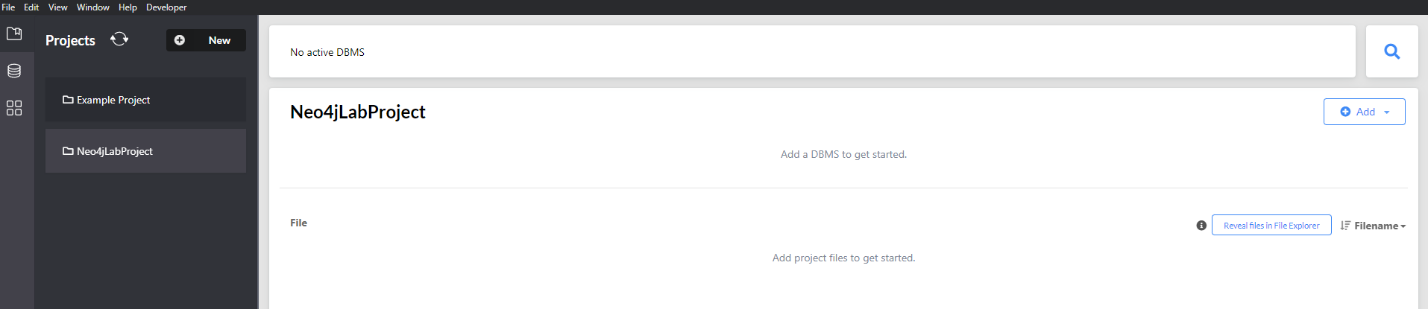
An .exe file will be downloaded inside your Downloads folder. When you open the Neo4j Desktop app, firstly you need to choose a folder to store the application data. After that, we are ready to create our first Neo4j project.

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

A picture containing text

Description automatically generated

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, we press the “Add” button and we choose “Local DBMS”. We give a DBMS name and password and then we 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. In this tutorial, we will use algorithms such as PageRank. To install the library, go to the Plugins tab, select the “Graph Data Science Library” and press install.

Graphical user interface, application

Description automatically generated

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.

Now, we can examine our dataset.

● ● ●

**Dataset**

The dataset is a mongoexport JSON file, which contains 33,223 tweets retrieved from Twitter API. Each tweet document has a unique *id* and 2 main fields, which contain more fields:

* *data* (contains 14 more fields)
* *includes* (contains 3 fields: *users, tweets, media*, which also contain more fields)

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:

A picture containing graphical user interface

Description automatically generated Timeline

Description automatically generated with medium confidence

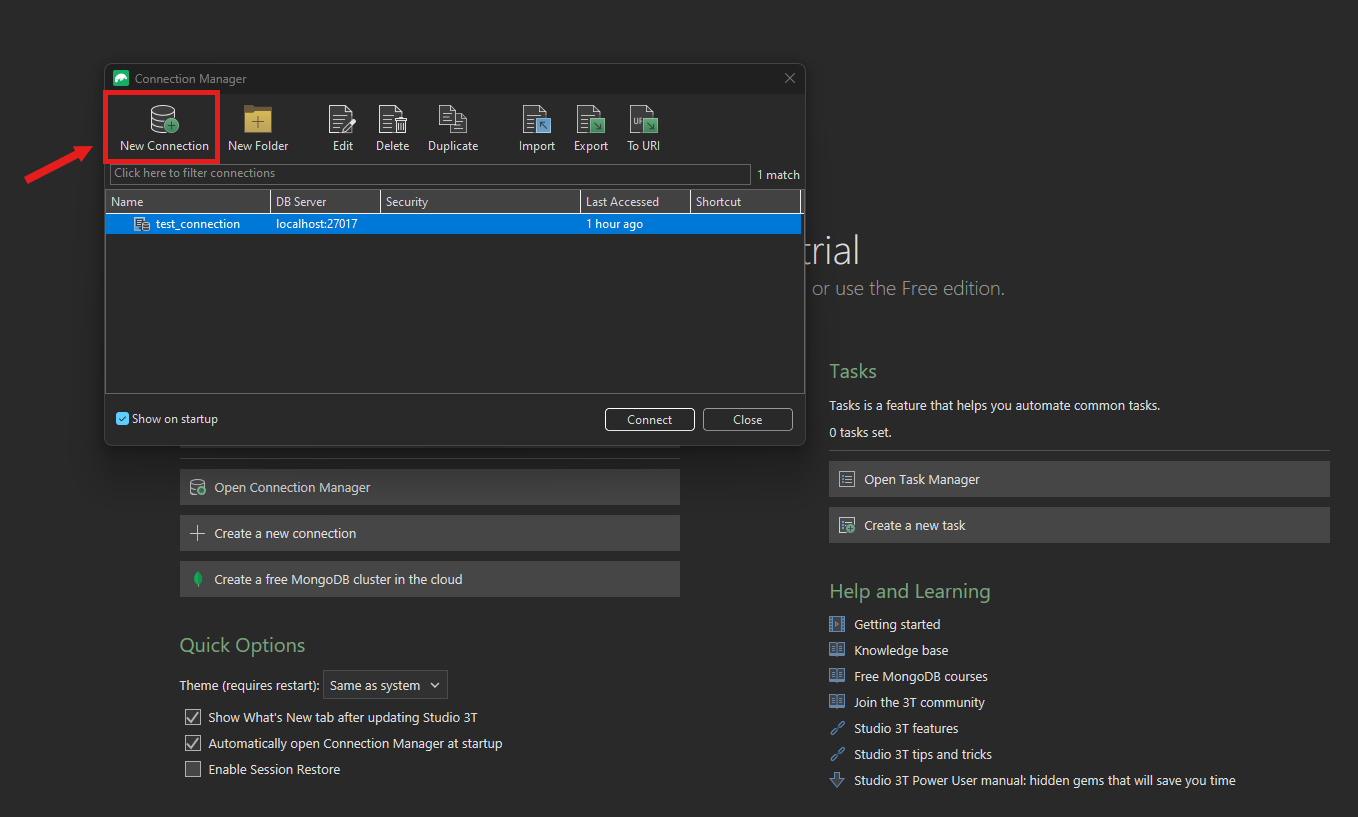
Users includes the user objects of the users mentioned in this tweet. Users[0] is a user object that always corresponds to the author of the tweet. Tweets[0] is the tweet under investigation (in this case the retweet) and the original tweet is tweet[1].

**MongoDB**

We will use MongoDB to extract the data from the JSON file. You can download MongoDB from the link [here](https://www.mongodb.com/try/download/community). We also need a GUI to have a clearer view of the dataset. We use [Studio 3T](https://studio3t.com/), but also [MongoDB Compass](https://www.mongodb.com/products/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 windows 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!

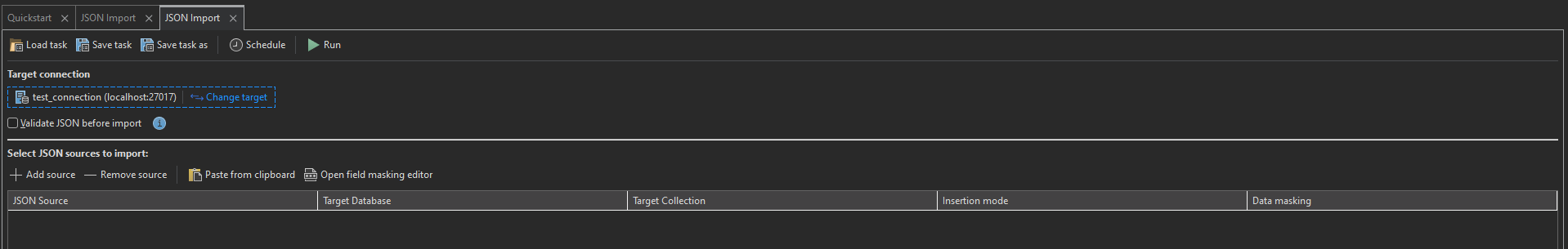


A screenshot of a computer

Description automatically generated with medium confidence

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**

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

A screenshot of a computer

Description automatically generated

When you press “Configure”, you will see a similar interface like before, where we chose “Import”. In this interface, 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.



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

Graphical user interface, text, application, email

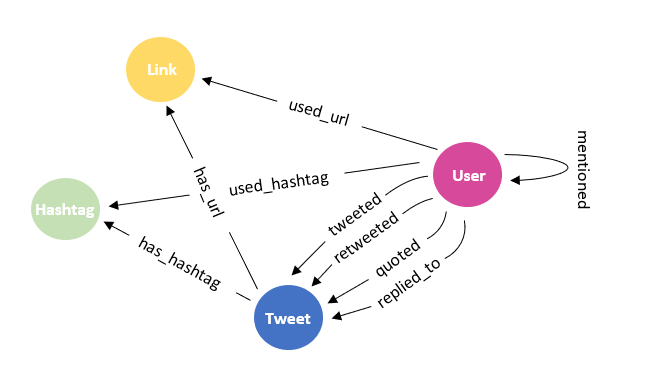
Description automatically generated

Now that we have stored our twitter data into the data variable, 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.

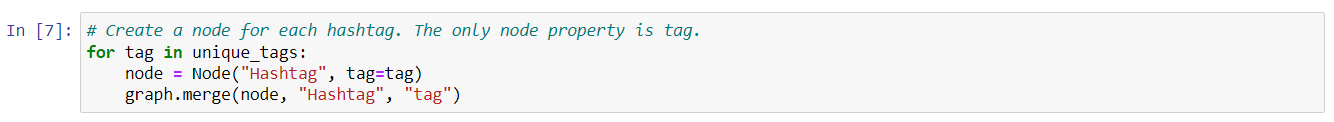


**Nodes**

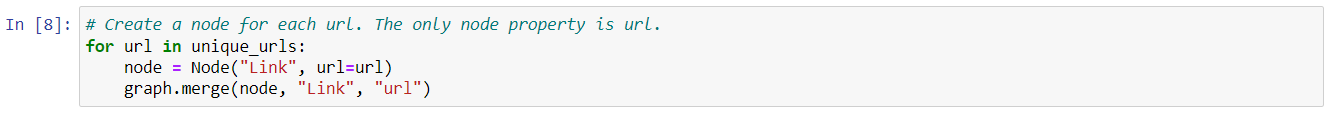
Now let's create our nodes. Always be careful not to create double nodes!

To populate Neo4j with all the nodes, we use py2neo Python library. Firstly, we iterate over the different documents of the data, and we retrieve the information needed from every document. We create 4 different sets, to store the different unique nodes and their properties, and we populate these sets in every iteration.

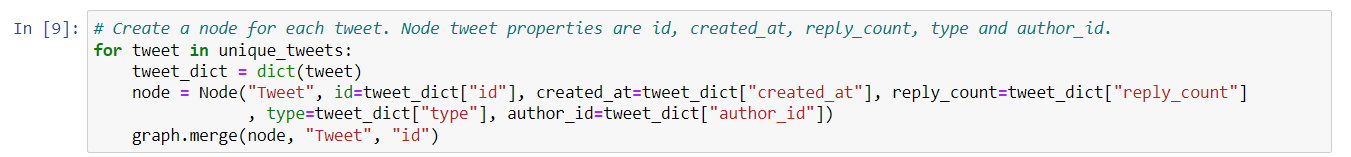
* **Hashtag**: For the node Hashtag, we only added the property tag, where tag is the name of the hashtag. All tags are converted to lowercase in order for our queries to be case insensitive. Hashtags are taken from the includes.tweets[0].entities.hashtags.tag when the hashtags and tag properties exist.



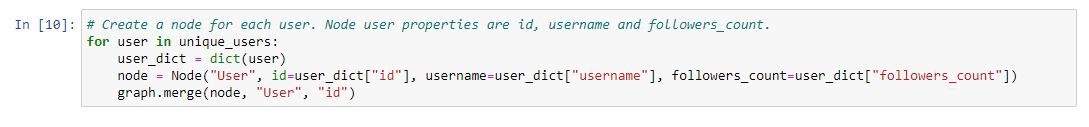
* **Link**: For the node Link, we only added the property url, where url is the name of the url. Urls are taken from the includes.tweets[0].entities.urls.expanded\_url when the urls and expanded\_url properties exist.



* **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



Data from data.entities.mentions are taken at a second step:

* + data.entities.mentions.id
  + data.entities.mentions.username

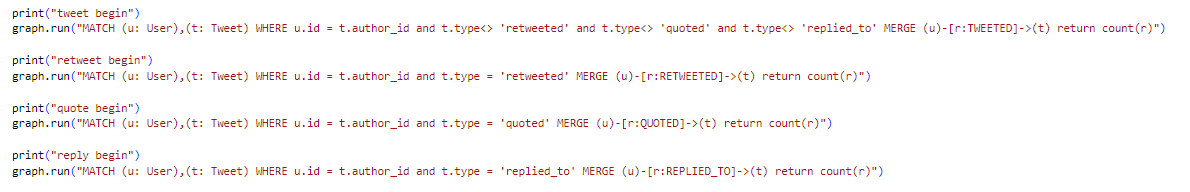
We check if the mentioned user does not already exist in the network and then we add a new User node.

A picture containing text

Description automatically generated

**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).



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

* **HAS\_HASHTAG:** represents hastag that is included on tweet. Hashtag is retrieved from includes.tweets[0].entities. hashtags.[i].tag
* **USED\_HASHTAG:** represents hastags that is used by user. Similar to the HAS\_HASHTAG relationship, hashtag is retrieved from includes.tweets[0].entities. hashtags.[i].tag
* **HAS\_URL:** represents url that is included on tweet. Url is retrieved from includes.tweets[0].entities. urls.[i]. expanded\_url
* **USED\_URL:** represents urls that is used by user. Similar to the HAS\_URL relationship, url is retrieved from includes.tweets[0].entities. urls.[i]. expanded\_url
* **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. Information is obtained by accessing the paths listed below:
  + data.entities.mentions.id
  + data.entities.mentions.username

**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**



Result:

A picture containing diagram

Description automatically generated

1. **Get the total number of hashtags (case insensitive)**



Result:

A picture containing shape

Description automatically generated

1. **Get the 20 most popular URLs in descending order**

Text

Description automatically generated with medium confidence

Result:

Text

Description automatically generated

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

A picture containing Word

Description automatically generated

Result (not all users are presented):

Table

Description automatically generated

1. **Get the number of tweets & retweets per hour**

Graphical user interface, text, application

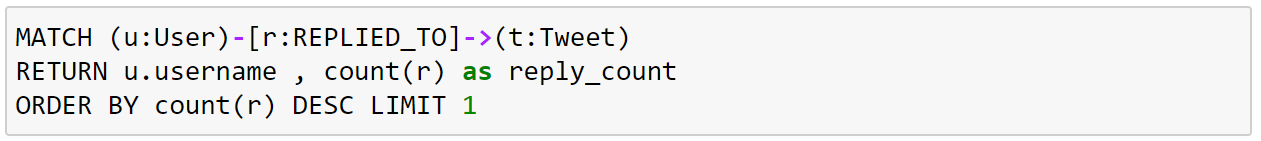
Description automatically generated

Result:

Table

Description automatically generated

1. **Get the user with the most replies**



Result:

A screenshot of a computer

Description automatically generated with low confidence

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

Graphical user interface, text, application, email

Description automatically generated

Result:

Text, letter

Description automatically generated

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

Firstly, we install the Graph Data Science Library on neo4j Desktop. We create a graph called “communityGraph” based on the “MENTIONS” relationship.

A picture containing text

Description automatically generated

Graphical user interface, text

Description automatically generated

Result:



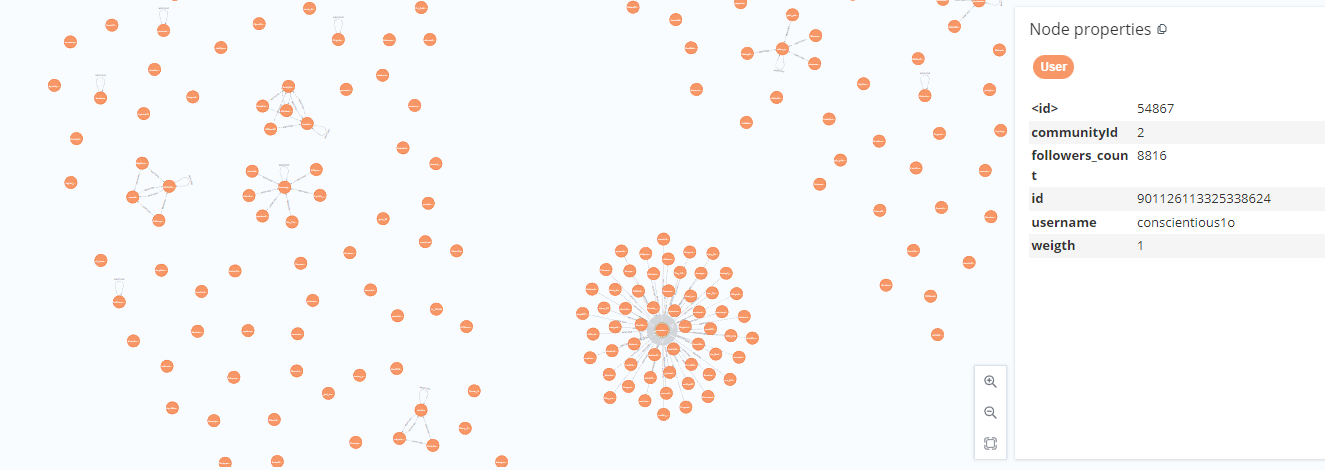
**19.** **Get the user communities that have been created based on the users’ interactions and visualise them (Louvain algorithm)**

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



To visualize communities, we run the following query:

MATCH (u1:User)-[:MENTIONS]->(:User) return u1.communityId as community, u1 as user order by community



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