**­Create a graph database in Neo4j using Python**

From data cleaning to graph population

[CJ Sullivan](https://medium.com/@cj2001?source=post_page-----4172d40f89c4-----------------------------------)

[Feb 9, 2021·8 min read](https://towardsdatascience.com/create-a-graph-database-in-neo4j-using-python-4172d40f89c4?source=post_page-----4172d40f89c4-----------------------------------)

Imagen que contiene luz, pez, iluminado, oscuro

Descripción generada automáticamente

[Martin Grandjean, CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0), via Wikimedia Commons

One of the most common questions I am asked by data scientists taking their first foray into graphs with Neo4j is how to get data into the database. In a [previous post](https://towardsdatascience.com/importing-csv-files-in-neo4j-f3553f1a76cf) I showed how to do this in one of a few different ways using the Neo4j browser UI set up through Docker. In this post I will show how you can use your own data generated with Python to populate the database. I will also show you how to use a different Neo4j database setup using the [Neo4j Sandbox](https://sandbox.neo4j.com/).

A Google Colab notebook with the code for this post can be found[here](https://colab.research.google.com/drive/1J9__HotNoINHpucoipLH-4qWc48GALAk?usp=sharing). (There are instructions in that notebook of how to connect Colab to Kaggle for getting your data downloaded more quickly.)

Necessary tools

1. The Neo4j [Python driver](https://neo4j.com/docs/api/python-driver/current/) (version 4.2 at the writing of this post)
2. [Jupyter Notebook/Lab](http://jupyter.org/) or [Google Colab Notebook](https://colab.research.google.com/notebooks/intro.ipynb#recent=true) (optional)
3. [Pandas](https://pandas.pydata.org/)

Data cleaning with Python

Now we can actually start doing some data munging with Python. For the sake of this post we are going to use the [arXiv Dataset](https://www.kaggle.com/Cornell-University/arxiv" \t "_blank) found on Kaggle, which contains more than 1.7M scholarly STEM papers. (At the writing of this post, it is on Version 18.) You should go ahead and download that data to your local machine.

Moving now into your notebook, we can start thinking about our data. I have loaded the data in via:

Texto

Descripción generada automáticamente

(You don’t have to use tqdm, but I find it helpful to check my progress knowing the file size is over 1.79M entries.)

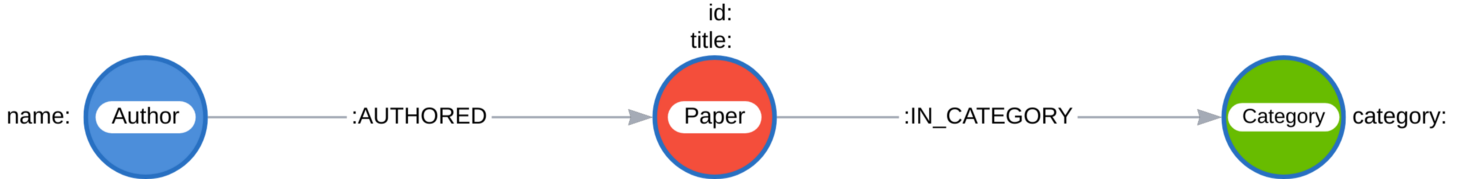
You can see via df.dtypes that our data is structured as:

Tabla

Descripción generada automáticamente

Suppose we want to construct a graph with this dataframe and that we are curious about which authors are publishing which papers and then what categories those papers are associated with. **We will then want to have three different node types corresponding to this: authors, papers, and categories**. Each of these node types is going to have a property or two. For authors it makes sense that we know their name. The papers could have both their ID and their title. And lastly it also makes sense for the categories to have their own name. We also have some relationships: authors author papers and papers are in certain categories.

Therefore, our goal is to have the following data model (drawn with [arrows.app](https://arrows.app/" \t "_blank)):



Data model. (Image by author.)

There are a few columns that are going to be useful for us. For example, I plan on keeping id so we can use it as a unique index for each paper. After that, I want to get an individualized list of each author. Also, the authors\_parsed column gives us a much cleaner listing of all authors. Definitely we will keep the title column as the main property for the paper. Lastly, I want to keep the categories columns.

The next step is to clean up our data a bit so I have one author per row of the dataframe along with a single category per row. For instance, we see that the authors\_parsed column gives a list of lists where the entries each have a superfluous comma following the names. If we simply import this into the database as is, we will wind up with author nodes like (shown for a small sample):

Texto

Descripción generada automáticamente

Since that is not a very pleasant thing to look at (and will result in queries that are not the most elegant), we need to clean that up a bit. We also see that the categories column could have a single category, or several categories that are not in traditional list format (as shown in the last line of this sample):

Texto

Descripción generada automáticamente

We could do that in Cypher, but for the sake of this post we are going to do our cleaning in Python in order to hopefully illustrate that we can do as much or as little as we want using Python.

Let’s create a couple of helper functions for cleaning up those two columns:

Texto

Descripción generada automáticamente

Our resulting dataframe now looks like:

Interfaz de usuario gráfica, Texto, Aplicación, Tabla

Descripción generada automáticamente

Now we have something we can work with!

Creating a Neo4j Sandbox

The [Neo4j Sandbox](http://neo4j.com/sandbox) is a nice, free way to get to tinker with Neo4j. You can spin up an instance that will persist for 3 days and get to work!

For the purposes of this article, when you get to the Sandbox, you will create a basic, blank sandbox like this:

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Creating a blank sandbox. (Image by author.)

There are plenty of other useful Sandboxes, as you can see from the creation window, but we are going to select this option since we are going to populate the database with our own data. Sit back for a couple of minutes while it creates it. Once that happens, you will get your connection information, as shown below:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Web connection information for our Sandbox. (Image by author.)

There are a few things you will need with this window. First, you will note the Bolt URL, complete with its port number. You will need this to establish the connection via Python. Next, you will also need the password (“difficulties-pushup-gaps” in this case). This will be needed to authenticate into this instance. I would point out that after 3 days when this instance gets taken down, this information is no longer valid, so no worries about trying to scoop my Sandbox and its data!

Connecting to Neo4j and populating the database

Now we need to establish the connection between our local machine (or wherever we have our Python code) and the Sandbox database. This is where those BOLT URL and passwords will come in.

I have created a helper class for doing this, which is very slightly modified from [this Medium post](https://towardsdatascience.com/neo4j-cypher-python-7a919a372be7):

class Neo4jConnection:

def \_\_init\_\_(self, uri, user, pwd):

self.\_\_uri = uri

self.\_\_user = user

self.\_\_pwd = pwd

self.\_\_driver = None

try:

self.\_\_driver = GraphDatabase.driver(self.\_\_uri, auth=(self.\_\_user, self.\_\_pwd))

except Exception as e:

print("Failed to create the driver:", e)

def close(self):

if self.\_\_driver is not None:

self.\_\_driver.close()

def query(self, query, parameters=None, db=None):

assert self.\_\_driver is not None, "Driver not initialized!"

session = None

response = None

try:

session = self.\_\_driver.session(database=db) if db is not None else self.\_\_driver.session()

response = list(session.run(query, parameters))

except Exception as e:

print("Query failed:", e)

finally:

if session is not None:

session.close()

return response

conn = Neo4jConnection(uri="bolt://52.87.205.91:7687",

user="neo4j",

pwd="difficulties-pushup-gaps")

Cool. Now we can start populating the database. We start by creating some constraints within the database that will ensure the nodes are not duplicates while also setting up some indexing:

conn.query('CREATE CONSTRAINT papers IF NOT EXISTS ON (p:Paper) ASSERT p.id IS UNIQUE')

conn.query('CREATE CONSTRAINT authors IF NOT EXISTS ON (a:Author) ASSERT a.name IS UNIQUE')

conn.query('CREATE CONSTRAINT categories IF NOT EXISTS ON (c:Category) ASSERT c.category IS UNIQUE')

Now we create three functions for creating dataframes for the category and author nodes, which we will use to populate each into the database separately:

def add\_categories(categories):

# Adds category nodes to the Neo4j graph.

query = '''

UNWIND $rows AS row

MERGE (c:Category {category: row.category})

RETURN count(\*) as total

'''

return conn.query(query, parameters = {'rows':categories.to\_dict('records')})

def add\_authors(rows, batch\_size=10000):

# Adds author nodes to the Neo4j graph as a batch job.

query = '''

UNWIND $rows AS row

MERGE (:Author {name: row.author})

RETURN count(\*) as total

'''

return insert\_data(query, rows, batch\_size)

def insert\_data(query, rows, batch\_size = 10000):

# Function to handle the updating the Neo4j database in batch mode.

total = 0

batch = 0

start = time.time()

result = None

while batch \* batch\_size < len(rows):

res = conn.query(query,

parameters = {'rows': rows[batch\*batch\_size:(batch+1)\*batch\_size].to\_dict('records')})

total += res[0]['total']

batch += 1

result = {"total":total,

"batches":batch,

"time":time.time()-start}

print(result)

return result

These functions take each column into the variable $rows and those columns are of list format. The UNWIND command takes each entity of the list and adds it to the database. After this we use a secondary function to update the database in batch mode, which can be helpful when you are dealing with more than about 50k uploads.

After we have these nodes loaded in, we will add the paper nodes as well as all relationships with the following functions: def add\_papers(rows, batch\_size=5000):

# Adds paper nodes and (:Author)--(:Paper) and

# (:Paper)--(:Category) relationships to the Neo4j graph as a

# batch job.

query = '''

UNWIND $rows as row

MERGE (p:Paper {id:row.id}) ON CREATE SET p.title = row.title

// connect categories

WITH row, p

UNWIND row.category\_list AS category\_name

MATCH (c:Category {category: category\_name})

MERGE (p)-[:IN\_CATEGORY]->(c)

// connect authors

WITH distinct row, p // reduce cardinality

UNWIND row.cleaned\_authors\_list AS author

MATCH (a:Author {name: author})

MERGE (a)-[:AUTHORED]->(p)

RETURN count(distinct p) as total

'''

return insert\_data(query, rows, batch\_size)

So in a similar fashion to the category and author nodes, we create each paper and then connect it via the :AUTHORED or :IN\_CATEGORY relationships for each row of the dataframe (UNWINDing the lists as needed). Note that there is much more data moving through the pipe in this function, so it might help to decrease the batch size to prevent time out errors.

Also, for this step we might be tempted to use a method like explode.() on the complete dataframe to get one row for each element of each list and processing the entire dataframe into the database that way. That can work and that is exactly what we will do below for small amounts of data like a deduplicated category and author node list in the categories and authors dataframes. However, for larger datasets it is not a very efficient way to load data into Neo4j. Because Neo4j is a transactional database, we would be creating one and executing one single statement per row of the dataframe, which will result in a very slow loading process. It also could exceed the available memory. Sandbox instances have approximately 500 MB of heap memory and 500 MB of page cache. So this further motivates updating the database as a batch process.

Executing all of these functions to populate the graph, we have:

categories = pd.DataFrame(df[['category\_list']])

categories.rename(columns={'category\_list':'category'},

inplace=True)

categories = categories.explode('category') \

.drop\_duplicates(subset=['category'])

authors = pd.DataFrame(df[['cleaned\_authors\_list']])

authors.rename(columns={'cleaned\_authors\_list':'author'},

inplace=True)

authors=authors.explode('author').drop\_duplicates(subset=['author'])

add\_categories(categories)

add\_authors(authors)

add\_papers(df)

Great! We now have a populated database! Here is what a subsample of that graph should look like (obtained via MATCH (a:Author)-[:AUTHORED]->(p:Paper)-[:IN\_CATEGORY]->(c:Category) RETURN a, p, c LIMIT 300:

Gráfico, Gráfico de dispersión

Descripción generada automáticamente

Subsample of the graph, visualized in the Sandbox with Neo4j Bloom. The blue nodes are authors, green nodes are papers, and pink nodes are categories. (Image by author.)

Let’s make sure it has what we want in it…

Querying the database to get some answers back

Here is a tip: when you have a populated database, you should let Neo4j handle as many calculations as you can before bringing the answers back into Python (if you even need to). In this case, suppose we want to calculate the in-degree of each category and return the categories of the top 20 categories. Obviously we could just do this simplistic thing in Python, but let’s do it in Neo4j. At some point you will want to do more complicated calculations (such as node centrality, path finding, or community detection), which can and should all be done in Neo4j before downloading those results back into Python.

To do this in Cypher, we could use any number of several approaches, but here is a quick one that works:

query\_string = '''

MATCH (c:Category)

RETURN c.category\_name, SIZE(()-[:IN\_CATEGORY]->(c)) AS inDegree

ORDER BY inDegree DESC LIMIT 20

'''

top\_cat\_df = pd.DataFrame([dict(\_) for \_ in conn.query(query\_string)])

top\_cat\_df.head(20)

This should give us:

Tabla

Descripción generada automáticamente con confianza media

Top 20 categories by in-degree. (Image by author.)

The distribution of the in-degree for the above subset of data looks like this:

Gráfico, Gráfico de barras

Descripción generada automáticamente

Distribution of in-degree for subsample of graph. (Image by author.)

So this shows that the database is both populated and how we can get results back. For what it is worth, another way you could get those sa result = conn.query(query\_string)

for record in result:

print(record['c.category'], record['inDegree'])me results returned in list form would be:

Summary

We have shown how to connect to a Neo4j Sandbox from Python and upload our data once we are satisfied that it contains what we want. Like anything else in coding, there are a ton of different ways to do this and the interested user is encouraged to explore doing the above demonstration predominantly in Cypher rather than Python. But by using the Neo4j Python connector it is easy to go back and forth between Python and your Neo4j database, just as it is for any other major database. This will open up all kinds of exciting possibilities in data science and machine learning, such as automated node classification, link prediction, and node clustering. Stay tuned for future blog posts on those subjects!