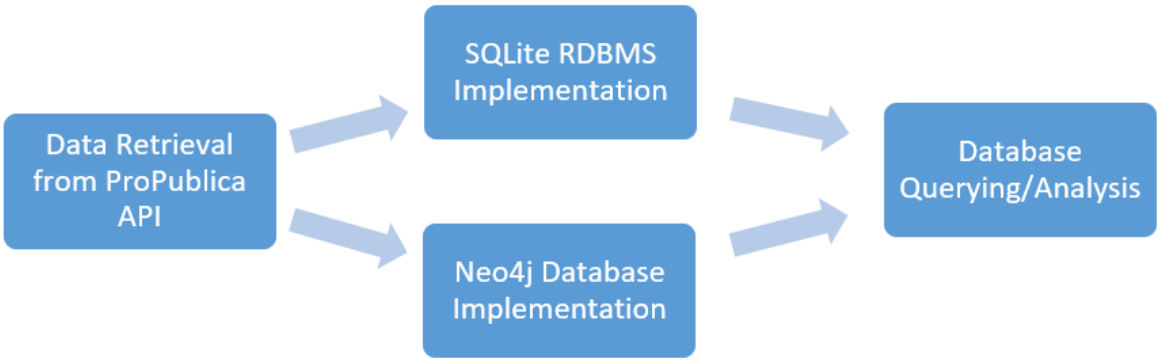
Final Project Report
IST 652: Scripting for Data Analysis
Alex Bailey, Lennart Zeugner

A Database Implementation for U.S. Congress Voting History

Project Overview

Every year, the U.S. Congress takes hundreds of votes to decide collectively on legislative bills, resolutions, nominations or other procedural matters. Often times, voting results are a reflection of the political direction a member of Congress and his or her corresponding party is heading towards. While congressional voting results are a matter of public record, it can be quite challenging to obtain a transparent view of how members of Congress have voted on specific legislative bills. With that said, the objective of this project is to provide politically interested individuals with a tool to capture and follow along with the most recent congressional voting activities. In order to achieve such an objective, a relational SQLite database, as well as a Neo4j graph database containing voting records, is implemented. Furthermore, this project is laying the groundwork for additional social media analysis of each congressional member.

Project Workflow



Data Sources/Acquisition:

The two databases are comprised of data from the following three ProPublica API endpoints.

1) ProPublica Members API
2) ProPublica Bills API
3) ProPublica Votes API

By connecting to the ProPublica Members API, we are able to retrieve personal biographies (name, party, member id, committee membership, and Twitter account) of each member of the Senate. The ProPublica Bills API allows us to obtain summary information of the most recent bills. Summary information include bill name, title, data, sponsoring party, result, etc.. Lastly, the ProPublica votes API provides voting results with the specific position of each voting member. Each of the API's returns data in semi-structured JSON format.

Description of Program:
The program's core functionality is in propublica.py, but has been abstracted to main.py. Data from all of the previously described API's is located in /data and can be refreshed by passing "data_init=True" as a keyword argument to the ProPublica class instantiation in main.py. The Neo4j database was created using docker. Instructions for installing the neo4j docker container can be found here: https://hub.docker.com/_/neo4j. The configuration and initialization for Neo4j is in graph.py. No, additional configuration is necessary to connect the python code to the neo4j instance. The neo4j database can be reinitialized by passing "graph_init=True" to the Neo4j class instantiation in main.py. But before doing so, the current neo4j nodes and edges must be deleted. This can be done by running the following command from the neo4j dashboard located at localhost:7687 -- "MATCH(n) DETACH DELETE(n)". The functionality of the SQLite database is captured in the SQLite notebook.

SQLite Database:
In order to store the acquired data and facilitate simple data retrieval, we have implemented a relational SQLite database using the SQLalchemy library in python. After setting up the database schema and minor adjustments to the JSON structure of the data, we inserted the information in each of the database tables. The below picture exhibits the database schema that was implemented.

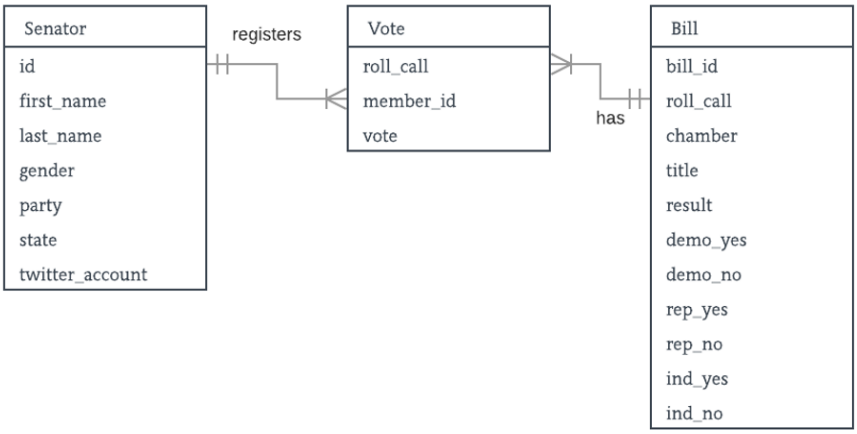
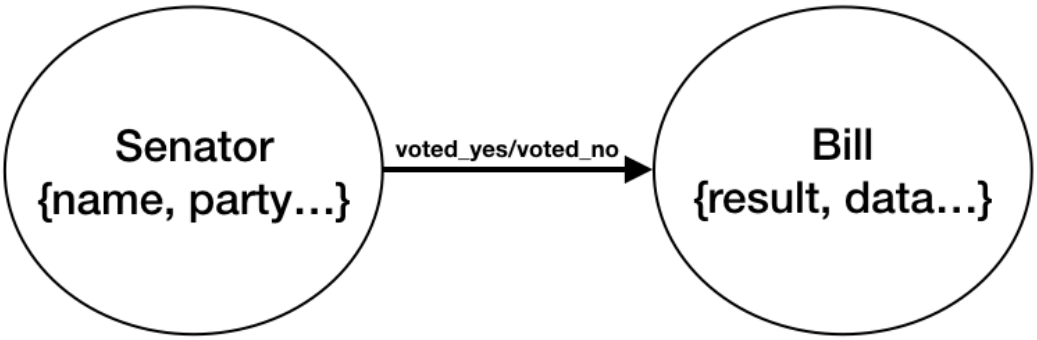


Figure 1: ERD demonstrating database schema

Neo4J Graph Database:
While Neo4j offers similar capabilities to a traditional relational database system in such that it is a fully ACID compliant and transactional database, it also exhibits performance improvements and adds a visual dimension/component to the data by displaying the data in a graphical network structure. The relation modeled in the SQL entity diagram above is very similar to the one in the Neo4j model. The significant difference is that the Vote join table that connects the one to many relationships in the SQL model is represented as an edge in then Neo4j graph model. This edge is a first-class data object and can be queried on its own. The Neo4j graph object model is outlined below:



The Senator and Bill nodes have the same properties as outlined in the relational database. The edge is labeled as either: voted_yes or voted_no. The edge is also directional to indicate in which direction the relationship applies. This graph can be read as “Senator X votes yes for Bill Y.” Each node is indexed by default for Neo4j, this is not the case in a relational database where the database administrator must take great care to select which keys to index in order speed up SQL CRUD operations. Another area where Neo4j shines over its counterpart is with performance of join operations. In SQL, a join operation creates a Cartesian product of both joined tables; thus the asymptotic runtime cost is O(A * B), where A and B are the number of records in each table. In contrast, Neo4j’s runtime cost is O(k); where k is the number of of records in the database that will be searched to find the request data[1].

Cypher Query Language:
Neo4j ships with its own domain specific query language called Cypher. It was created to be expressive and human readable. Here is a simple Cyber query:

```
MATCH (b:Boy {name:Bob, age: 25})-[:marries]->(g:Girl {name:Sue, age: 26)
RETURN b.name, g.name

>> Bob, Sue  # output
```

Parenthesis “(...)” are for nodes. Within the parenthesis, “b:Boy” is a label that can be dynamically added/removed to partition nodes into logical groups; A node can have one or more labels. Attributes of the node are denoted as “{name:Bob}”. The relationship/edge between the nodes is annotated by “-[:marries]->”. All relationships must be directional. The keyword MATCH is synonymous with SELECT from SQL

[1] https://dzone.com/articles/why-are-native-graph-databases-more-efficient-than

Analysis (Queries):

Depending on the needs as well as the given skillset of a user, data can be queried from the traditional relational database (SQLite) with python or the Neo4j database utilizing the cypher querying language. The below queries contrast the syntax of the respective querying languages and introduce example use cases of the databases.

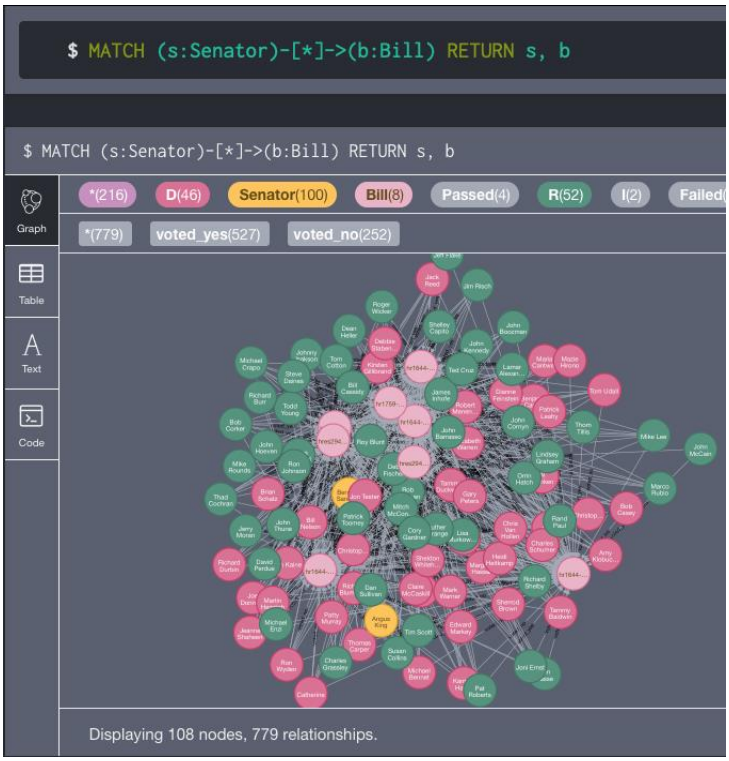
1) Return the voting results of all Senators for all bills (limit 5)

SQL:

```
#1) Return the voting results of all Senators for all bills
query = db.select([senators.c.first_name, senators.c.last_name,votes.c.roll_call, votes.c.vote,
                   bills.c.bill_id, bills.c.result])
query = query.where(and_(bills.c.roll_call == votes.c.roll_call, votes.c.member_id == senators.c.id ))
results = connection.execute(query)
df_all_bills = pd.DataFrame(results)
df_all_bills.columns = results.keys()
df_all_bills.head(5)
```

	first_name	last_name	roll_call	vote	bill_id	result
0	Lamar	Alexander	160	Yes	hres294-116	Passed
1	Lamar	Alexander	161	Yes	hres294-116	Passed
2	Lamar	Alexander	162	Yes	hr1759-116	Passed
3	Lamar	Alexander	163	Yes	hr1644-116	Agreed to
4	Lamar	Alexander	164	Yes	hr1644-116	Agreed to

Neo4j:



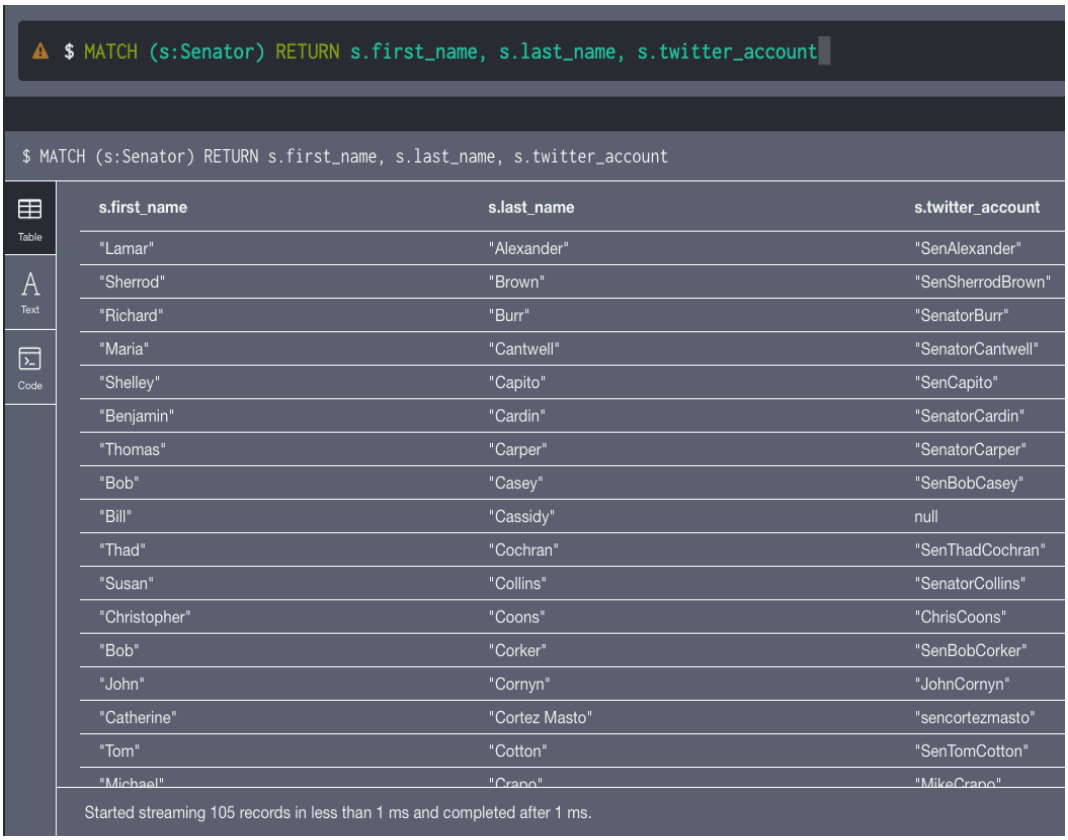
2) Return the Twitter accounts of all senators (limit 5)

SQL:

```
#2) Twitter accounts of all senators
s = select([senators.c.first_name,senators.c.last_name, senators.c.twitter_account])
results = conn.execute(s)
df_twitter = pd.DataFrame(results)
df_twitter.columns = results.keys()
df_twitter.head()
```

	first_name	last_name	twitter_account
0	Lamar	Alexander	SenAlexander
1	Tammy	Baldwin	SenatorBaldwin
2	John	Barrasso	SenJohnBarrasso
3	Michael	Bennet	SenBennetCo
4	Richard	Blumenthal	SenBlumenthal

Neo4j:



3) Return all the democrats that have voted "Yes" for the bill with a roll call of 165

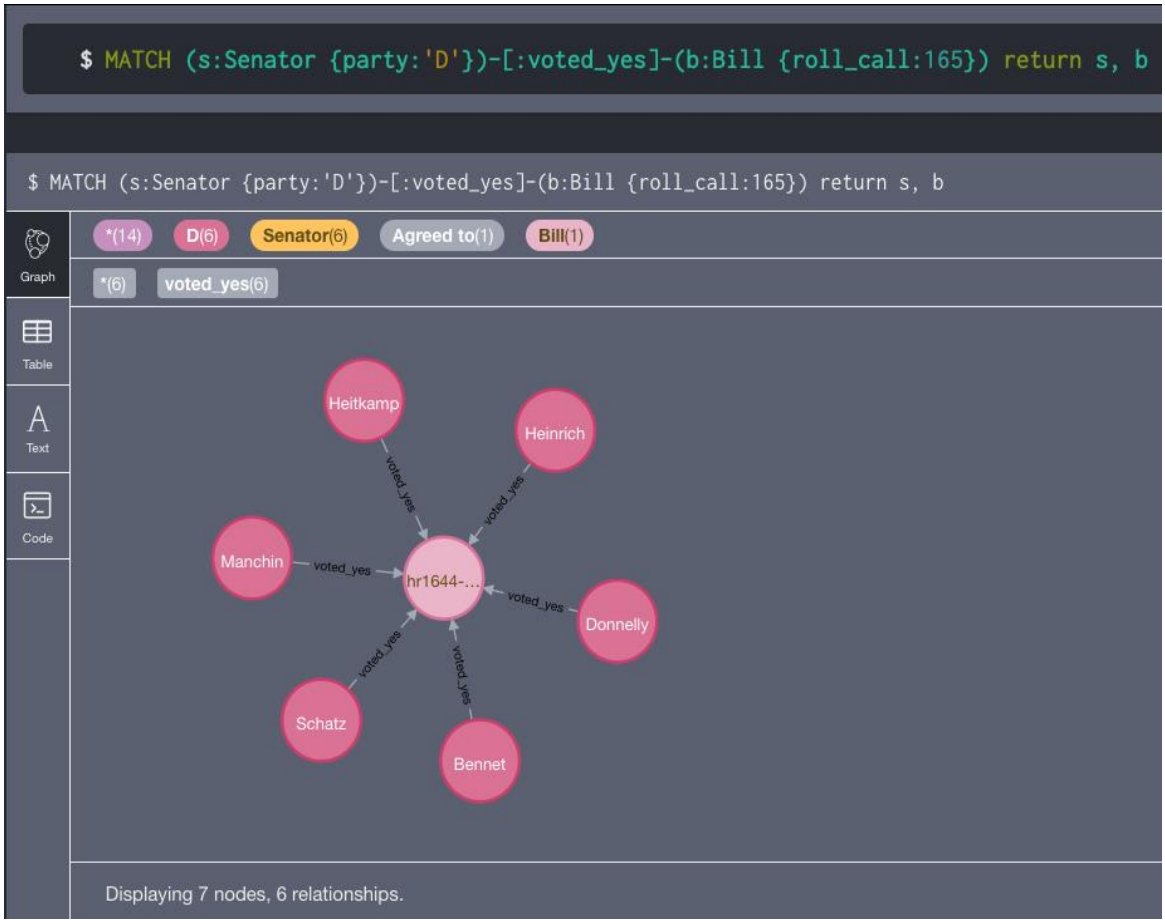
SQL:

```
#3) Return all the democrats that have voted "Yes" for bill with rollcall of 165
query = db.select([senators.c.first_name, senators.c.last_name, bills.c.bill_id])
query = query.where(and_(bills.c.roll_call == votes.c.roll_call,
        votes.c.member_id == senators.c.id, votes.c.roll_call == 165, senators.c.party == 'D', votes.c.vote == 'Yes'))
results = connection.execute(query)
df1 = pd.DataFrame(results)
df1.columns = results.keys()
df1

```

	first_name	last_name	bill_id
0	Michael	Bennet	hr1644-116
1	Joe	Donnelly	hr1644-116
2	Martin	Heinrich	hr1644-116
3	Heidi	Heitkamp	hr1644-116
4	Joe	Manchin	hr1644-116
5	Brian	Schatz	hr1644-116

Neo4j:



4) Return all the bills Luther Strange voted “Yes” for

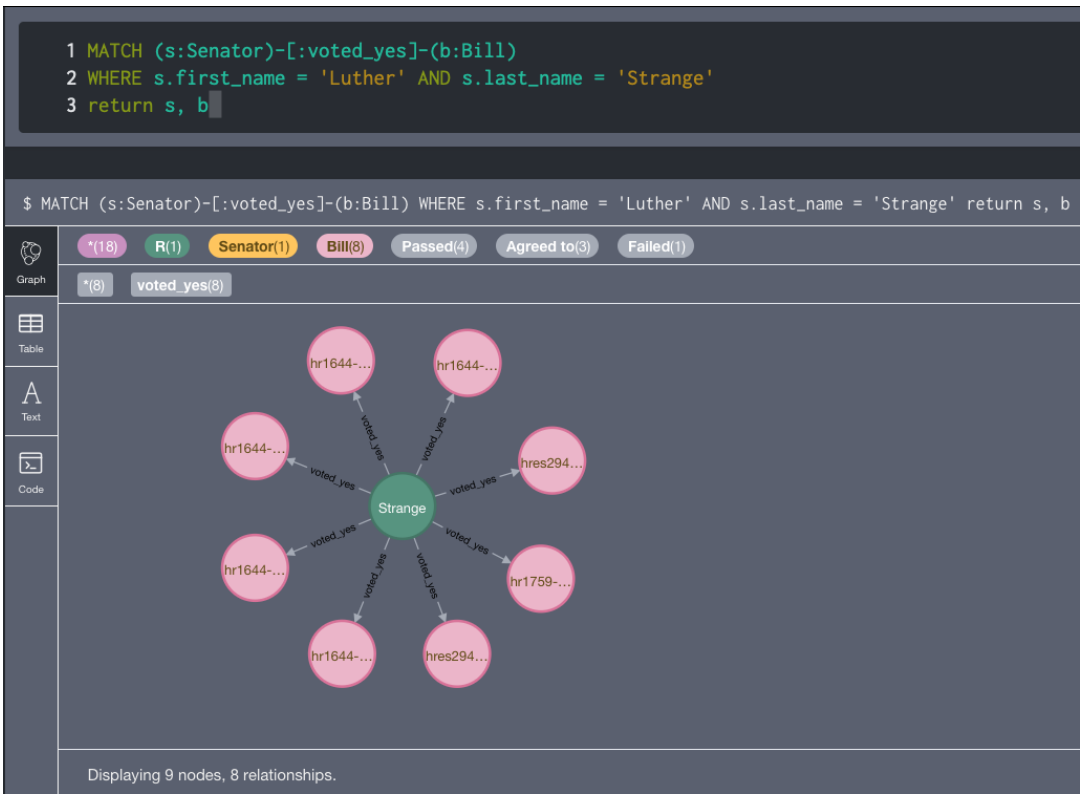
SQL:

```
#4) Return all the bills Luther Strange voted "Yes" for
query = db.select([senators.c.first_name, senators.c.last_name, bills.c.bill_id])
query = query.where(and_(bills.c.roll_call == votes.c.roll_call,
        votes.c.member_id == senators.c.id, senators.c.last_name == 'Strange', votes.c.vote == 'Yes'))
results = connection.execute(query)
df_Strange = pd.DataFrame(results)
df_Strange.columns = results.keys()
df_Strange

```

	first_name	last_name	bill_id
0	Luther	Strange	hres294-116
1	Luther	Strange	hres294-116
2	Luther	Strange	hr1759-116
3	Luther	Strange	hr1644-116
4	Luther	Strange	hr1644-116
5	Luther	Strange	hr1644-116
6	Luther	Strange	hr1644-116

Neo4j:



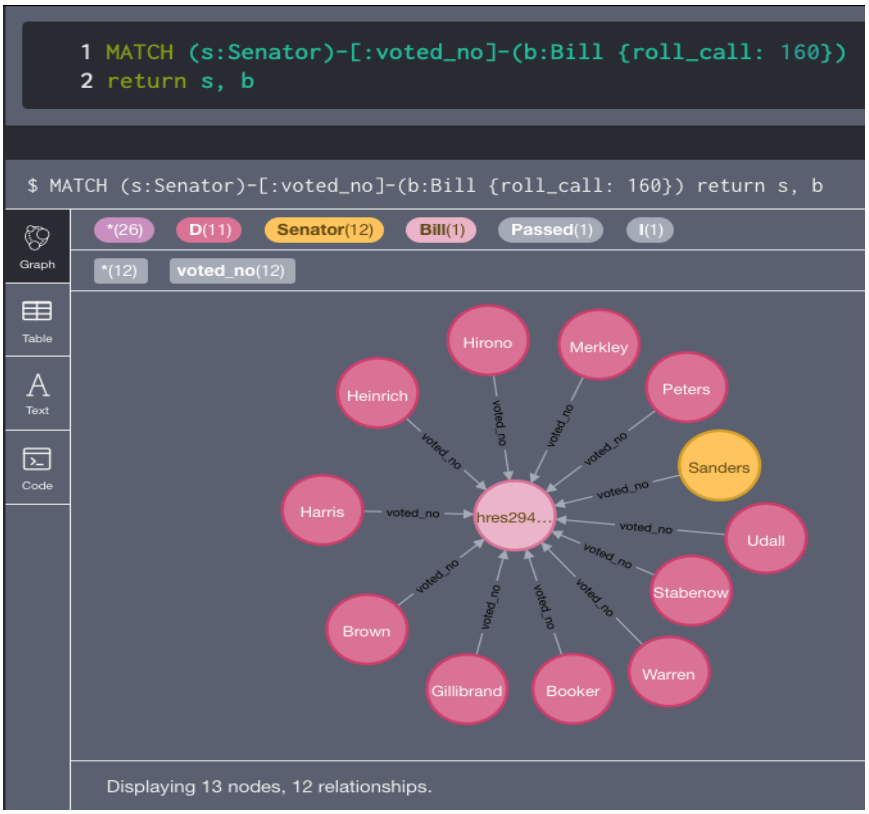
5) Return all senators that have voted “No” for the bill with a roll call of 160

SQL:

```
#5) All the senators that have voted 'No' for roll_call 160
query = db.select([senators.c.first_name, senators.c.last_name,votes.c.roll_call,
                   votes.c.vote, bills.c.bill_id, bills.c.result])
query = query.where(and_(bills.c.roll_call == votes.c.roll_call,
                         votes.c.member_id == senators.c.id, votes.c.roll_call ==160, votes.c.vote == 'No'))
results = connection.execute(query)
df_bill160_no = pd.DataFrame(results)
df_bill160_no.columns = results.keys()
df_bill160_no
```

	first_name	last_name	roll_call	vote	bill_id	result
0	Cory	Booker	160	No	hres294-116	Passed
1	Sherrod	Brown	160	No	hres294-116	Passed
2	Kirsten	Gillibrand	160	No	hres294-116	Passed
3	Kamala	Harris	160	No	hres294-116	Passed
4	Martin	Heinrich	160	No	hres294-116	Passed
5	Mazie	Hirono	160	No	hres294-116	Passed
6	Jeff	Merkley	160	No	hres294-116	Passed
7	Gary	Peters	160	No	hres294-116	Passed
8	Bernard	Sanders	160	No	hres294-116	Passed
9	Debbie	Stabenow	160	No	hres294-116	Passed
10	Tom	Udall	160	No	hres294-116	Passed
11	Elizabeth	Warren	160	No	hres294-116	Passed

Neo4j:



Conclusion:

Our project provides a granular and expressive way to collect data from publicly available APIs for information about U.S. Congress. We provide two techniques for storing as well as visualizing the data in the form of traditional SQL tabular results as well as using a graph database. This project could easily be extended to collect more relevant data (i.e tweets) to potentially look for correlations between sentiment expressed in tweets and voting results. For example, from query number three we understand that only six democrats voted “Yes” for a given bill, which implies that the other 44 democratic Senators voted “No”. For potential future analysis, it would be interesting to dive into the social media activities of those Senators that voted against the majority of their party.