Networks with Neo4j

Author

Table of Contents

[**Introduction** 3](#_Toc449532738)

[**The Foundations of Graph Databases** 3](#_Toc449532739)

[**Installing Neo4j** 5](#_Toc449532740)

[**Getting Started Learning Neo4j** 13](#_Toc449532741)

# **Introduction**

Suppose that an incredibly contagious and unknown disease has appeared. It is known that the disease is spread through human contact, but it is unclear where the disease originated. Though hospital records, a researcher has data on the patients who have contracted the disease. Her goal is identify patient zero.

The key to solving this problem is recognizing that it is the relationships between patients that will lead to finding patient zero. A recent patient must have a connection to a person who contracted the disease previously, and that person must have a similar relationship with another previous patient. One way of narrowing down who patient zero might be is by looking at the tree of all relationships belonging to all patients, and finding the set of people who are upstream from all current patients.

This type of analysis is called Network analysis. Networks analysis is the study of connections in a group. Network analysis is used to solve problems such as disease progression and to evaluate group dynamics and structures. Network analysis has been used to identify the key members of communities, persons who are the most social and most interconnected between groups, so that those persons can be targeted for community outreach. Networks can also show how cliquey a group is. Do Hollywood actors tend to work with the same people over and over again or do they branch out? Networks is not just limited to human relationships. Network analysis can perform a similar appraisal on if Hollywood actors work repeatedly for the same studio, in the same locations, with the same people.

Studying connections requires a data structure that can captures how entities relate to each other. In this lesson we will learn the basics of this data structure, a Graph Database. We will apply what we learn to looking at professors and research grants.

We will start by taking a quick look at the differences between relational databases and graph databases.

# **The Foundations of Graph Databases**

If you are accustomed to using software such as Excel or SQL, you probably think of data in terms of rows and columns. In an excel spreadsheet, data about a group of students might look like this,

|  |  |  |  |
| --- | --- | --- | --- |
| **StudentID** | **Name** | **Year** | **GPA** |
| 10023 | Theodore Jasper Detweiler | FRESHMAN | 2.9 |
| 10245 | Ashley Spinelli | SOPHMORE | 3.1 |
| 45832 | Vincent Pierre LaSalle | JUNIOR | 3.2 |
| 15779 | Gretchen Grundler | FRESHMAN | 4.0 |
| 13664 | Michael Blumberg | FRESHMAN | 3.6 |
| 14589 | Gustav Griswald | SOPHMORE | 2.5 |

Above, each row is a student. We know information about each student by using the information in a row that corresponds to the variable in the columns, such as Name, Year, GPA.

Relational Databases takes this topic to the next level, where we can connect different tables together to tell us how different topics are related. In a relational database, the table above would be a STUDENT Table. The information stored in this table would be information related to individual students. To capture all the data a high schools needs to keep track of, there needs to be additional tables. High school students take multiple classes. Each of these Classes would have features such as topic, time it is offered, and the students enrolled in that class. Therefore there would likely be a CLASS table. Furthermore each class is taught by a Teacher, whom will have her own set of information such as Name, Office Number, Salary, ect. These characteristics would be stored in a TEACHER table. All these tables would be connect to each other so that it would be possible to see what teacher was teaching which classes and the students in those classes. (For more information on relational databases, see \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

Relational Databases such as this are a standard and powerful way of storing large, complex datasets. However, they do have their weaknesses. For one, a relational databases requires a ridged structure. A column cannot be added for one observation without adding it to all observations. It can be a lot of baggage for the database to carry around. Next, Relational databases are not designed to look at connections between units. In this example, students, teachers, classes, ect. A relational Database can tell you what classes were taken by what students, but it has trouble commutating if the same students tend to take classes together, or if students will repentantly take classes with the same teacher.

This is where graph Databases, such as Neo4j, get their edge. Instead of thinking of data in terms of rows and columns, a graph database thinks about data in terms of nodes. A Node is a little cluster of information. In neo4j the row,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **StudentID** | **Name** | **Year** | **GPA** | |
| 10245 | Ashley Spinelli | SOPHMORE | | 3.1 | |

Would be conceptualized like a little ball of data,

In the above circle, our representation of a node, the node has an id (10245) , a node type (Student) and then a series of labels to tell me more about that student such as their name, year and GPA.

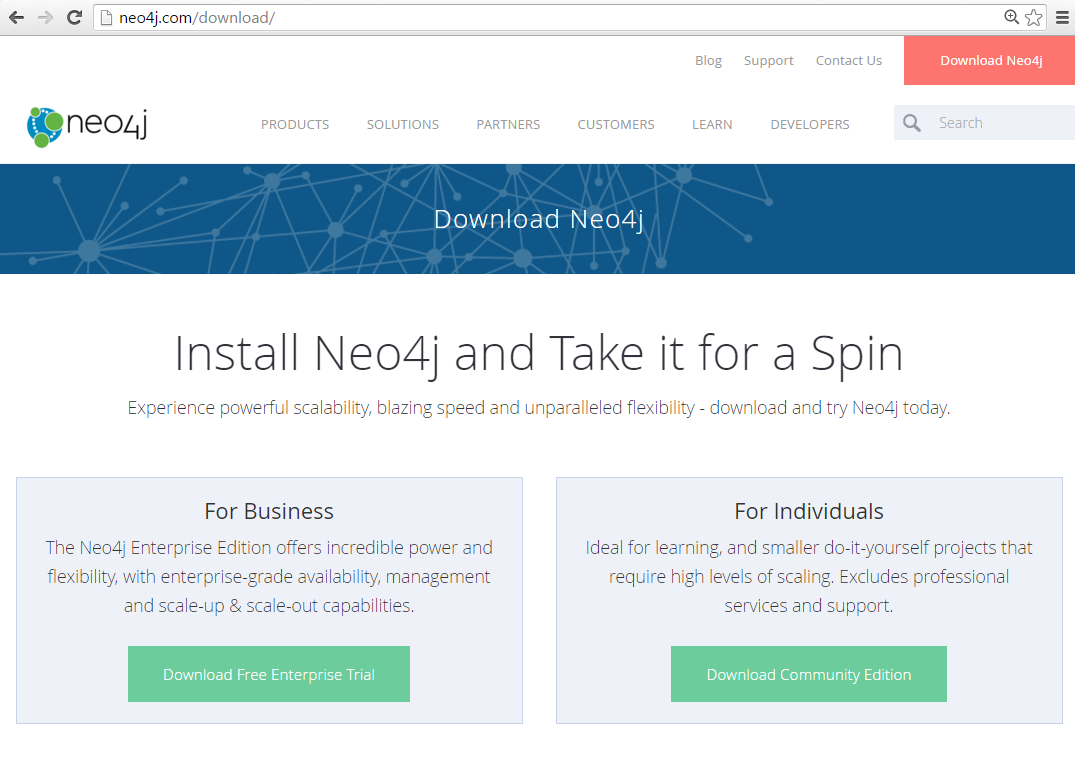
A neo4j databases can have any number of types of nodes. For our high school example, we could have nodes for students as well as teachers and classes. In the diagram below there are three types of nodes. Student, Class and Teacher. Each of these nodes would have labels containing the information we need to know for each individual entity. Because the information in each node is independent of the information in other nodes, we can add labels to one node without affecting the others. For example I can add a label to Mrs. Finster’s node indicating that she will be taking maternity leave next month.

Nodes are connected to each other via relationships. A relationship is a directional data structure that commutates how the nodes are connected. There are two types of relationships in the diagram below. The green arrow is a TEACHES relationship. Mrs. Finster TEACHES math class. The blue arrows are ENROLLED\_IN relationships. The students are ENROLLED\_IN math class. Relationships can also have labels just like nodes. The relationships in this diagram would probably have a label indicating the semester and year that these students and teacher were involved with this math class node. This is important because over the years we would expect students and teachers to have multiple relationships to different classes.

# **Installing Neo4j**

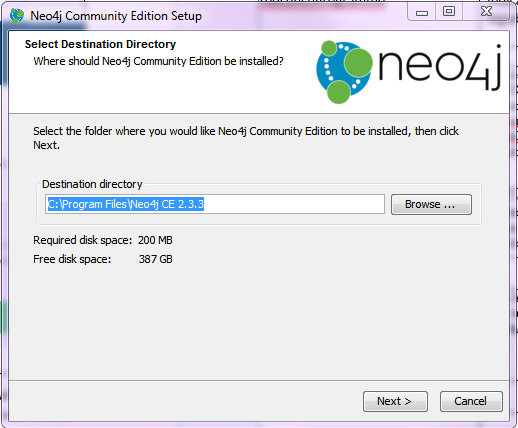
1. Go to the website <http://neo4j.com/>

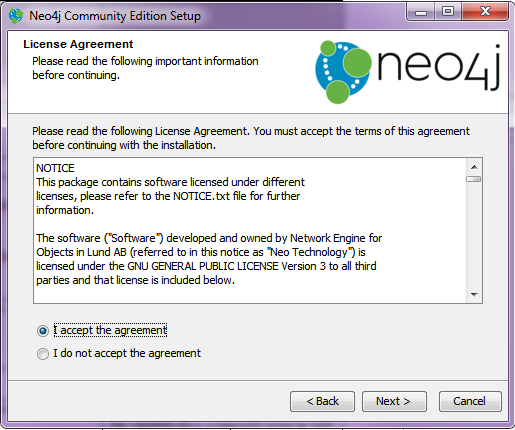
The Download Neo4j link is in the upper left hand corner



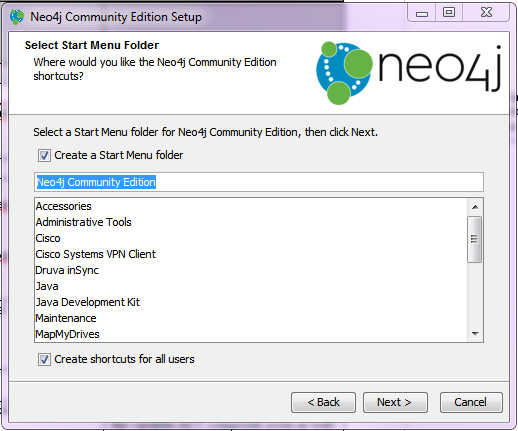
Once you are on this page, click Download Community Edition under for Individuals. This will give you a version of neo 4j that is perfect for learning about graph databases and getting you started with smaller projects. Your download should start automatically.

Pick a directory, under program files is fine



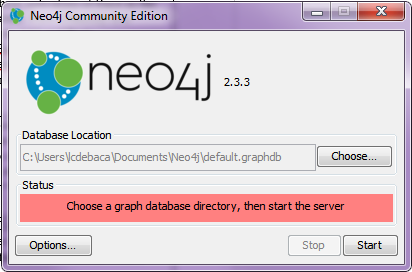


Accept the terms.



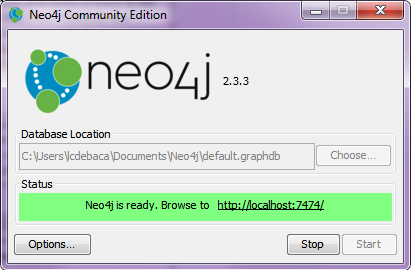
Accept the defaults.

It will prompt you to go ahead and start a server in the default folder of your neo4j graph DB.



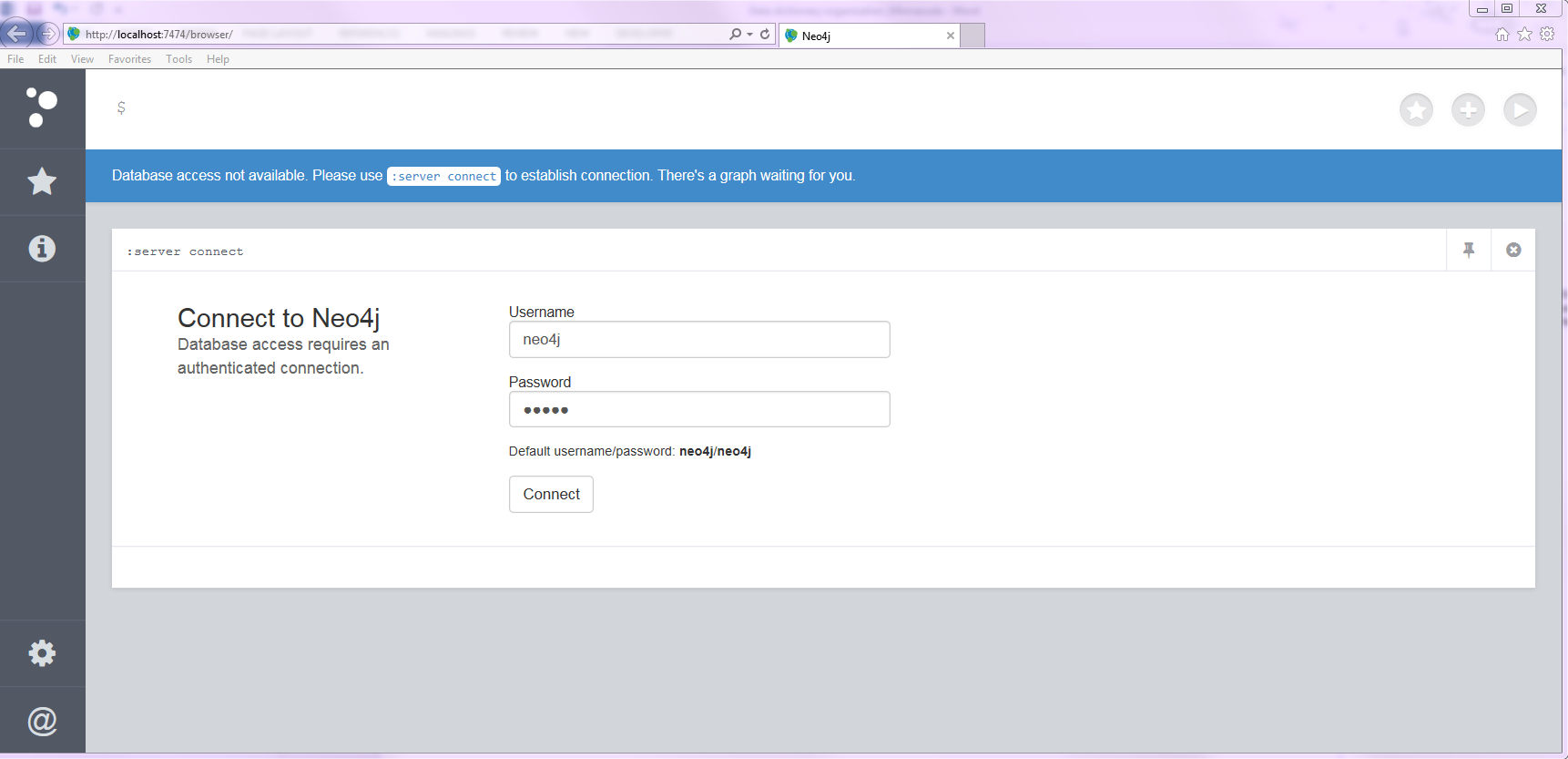
Once the server has been established

Your window will look like this,



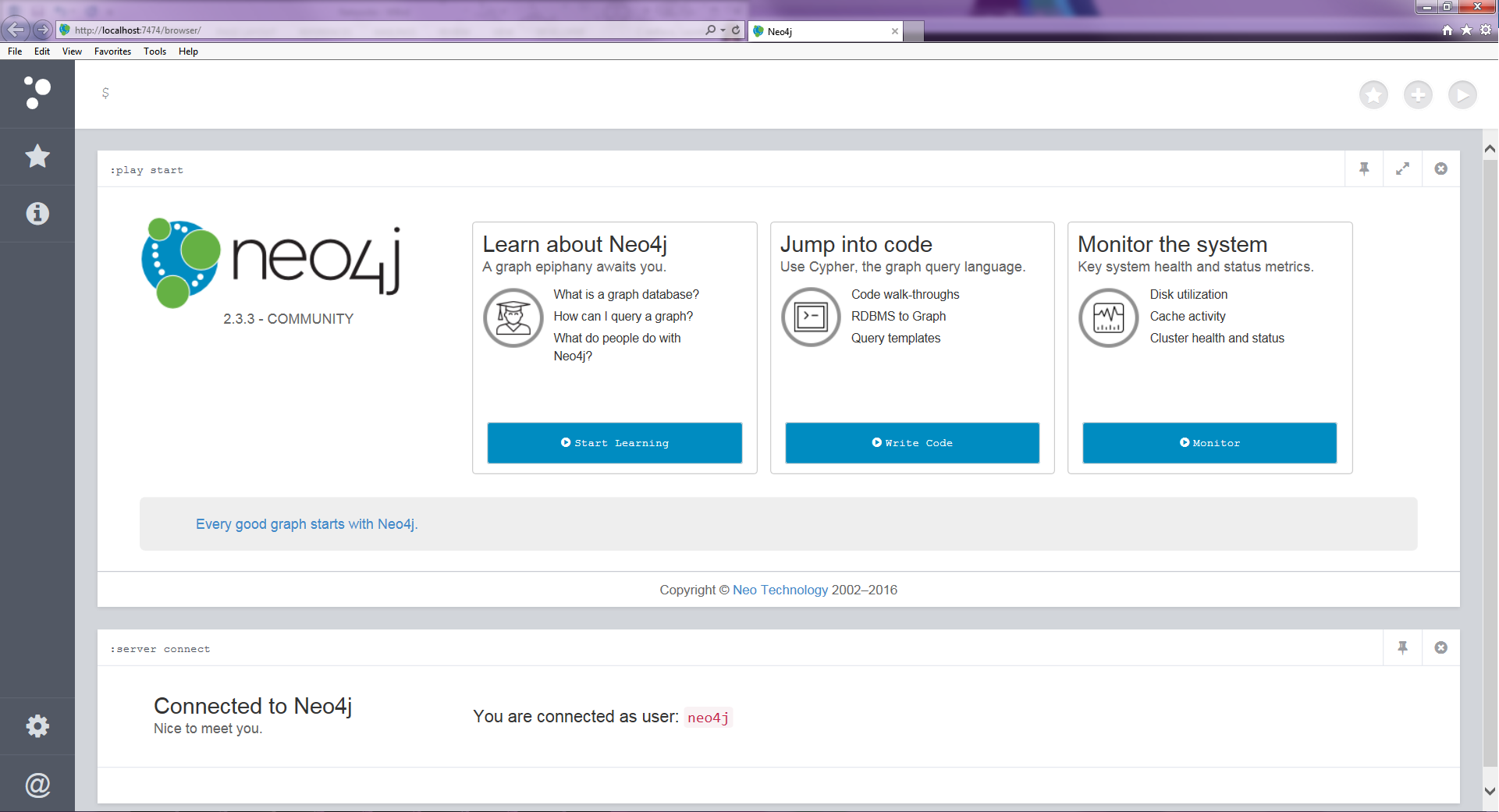
***A quick side note about servers:***  In the simplest terms, servers are pieces of software that are designed to give information. You have probably heard the term server when talking about websites. A websites information is not saved on your computer, it is saved someplace else with dedicated software that maintains and sends information to your computer when it is requested. Your computer then displays that information in your browser, a specialized piece of software that reads information given by a server. When you are in an Ipython Juniper notebook, you are actually using your browser to display information on a local server. By doing this we are able to combine python code, a language which can only be read by the python executable on your computer, with the user friendly interface of cells created by JavaScript and html, languages that normally can only be read by a browser. By using a local server, we can access to both the python language and the user friendly interface of the browser.

For the neo 4j database, we are also creating a server. For ease of usability Neo4j has a dashboard that can be read in the browser. To access this thing, copy the link <http://localhost:7474/> into your browser.



You will be asked to set a new password.

Congratulations you are connected to neo4j!



# **Getting Started Learning Neo4j**

Before we start using python to interface with neo4j, it is a good idea to get accustomed to what the console has to offer.

The Neo4j console can be used to pass commands to database to create and visualize data. It also has tools for managing the database and learning the use neo4j. I strongly recommend the tutorials that neo4j has made available through the console.

**Cypher**

Neo4j uses the Cypher Query language to communicate with the database. If you are familiar with SQL commands, cypher should not look too much different. For the exercises in this chapter all the cypher commands will be provided, but here is a super quick introduction to the basics of cypher.

Cypher uses asci art to conceptualize data. For instance the symbol **(n)** represents a node named n. (note, n is just a variable name, it can be called anything). The symbol **-->** or **<--** representrelationships. Putting it together the notation **(a) --> (b)** would denote the node **a** having a relationship with node **b**.

Finally, the asci notation is used in conjunction with cypher commands. The most common ones are listed below,

* MATCH
  + Essentially the same thing as SELECT in SQL
* CREATE
  + Create an node or relationship
* MERGE
  + Create a node or relationship if it does not already exist
* RETURN
  + Needed in every query
* LIMIT
  + Same as in SQL

**Loading Data Directly into the Console**

The Data we will be using in the notebooks is from the fictional university, Roke College. Roke College is a research institution where the professors apply to grants to fund their research. The data set we will be using shows the relationship between these awards and the people who work on them.

Currently the data is saved in saved into three csvs, employee\_data.csv, student\_data.csv, and award\_data.csv. Download these data files and place them is an easy to find location on your computer.

The Employee and Student Data look like this,



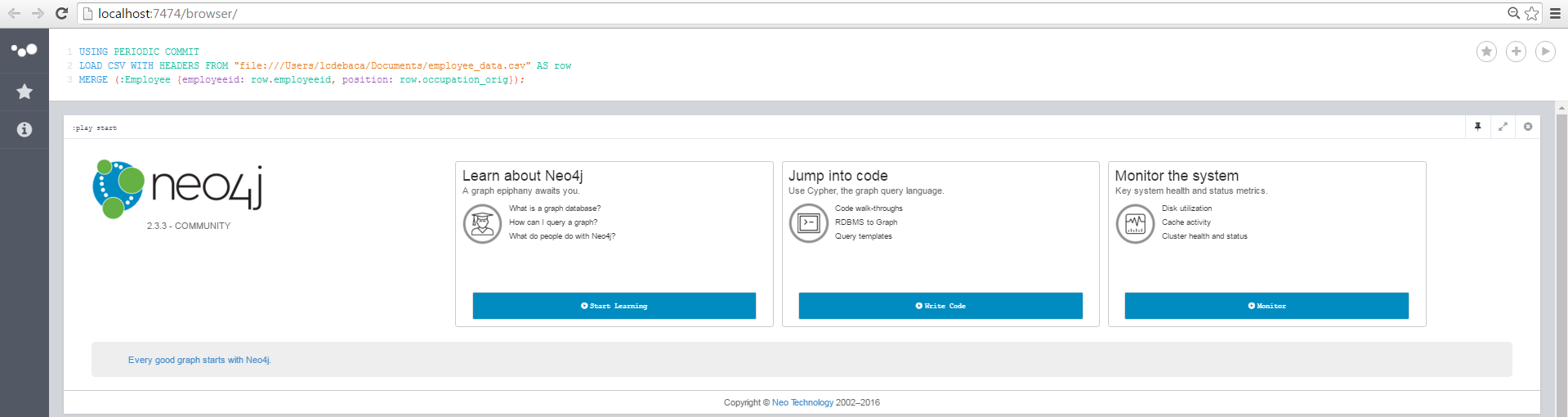
Where each Facility member and student have an employee id and an occupation identifier (FACILTY or STUDENT). Each student and facility member appears only once in their respective lists.

The Awards data looks like this,



Where the employeeid of the Student or facility member who worked on an award appears next to the award number for that award. An award and a person can appear multiple times in this file.

To load the data type the following commands in the top portion of the console (the box highlighted below),



Commands

1. Load in the Employee Nodes from the employee\_data.csv. (Note, you will have to change the path of the csv to match the location of where you saved the data on your machine.) This commands will create a node for each row in the data and give the node the properties of employeeid and position based on the data in that row.

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM "file:///<PATH TO DATA>/employee\_data.csv" AS row

MERGE (:Employee {employeeid: row.employeeid, position: row.occupation\_orig});

1. Load in the Award Nodes from the award\_data.csv. (Note, you will have to change the path of the csv to match the location of where you saved the data on your machine.) This commands will create a node for each row in the data if it does not already exist and gives the node the prosperity of award\_num based on the data in that row.

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM "file:/// <PATH TO DATA>/award\_data.csv" AS row

MERGE (:Award {award\_num: row.uniqueawardnumber});

1. Create relationships between Facility members and the Awards they worked on. This command will go back through the award\_data.csv file and for each row, each worked on relationship, find the existing nodes that are part of that relationship, then formally create that relationship. These will be names WORKED\_ON relationships.

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM "file:/// <PATH TO DATA>/award\_data.csv" AS row

MATCH (a:Award {award\_num: row.uniqueawardnumber})

MATCH (e:Employee {employeeid: row.employeeid})

MERGE (e)-[r:WORKED\_ON]->(a);

1. Create relationships between Facility members who worked on the same awards. This Command finds the pattern where two nodes have a WORKED\_ON relationship with the same award node. If a WORKED\_WITH relationship does not already exist, the command creates the relationship, WORKED\_WITH between those nodes.

MATCH (n1)-[:WORKED\_ON]->(a:Award)<-[:WORKED\_ON]-(n2)

MERGE (n1)-[r:WORKED\_WITH]-(n2);

At this point you have successfully create a neo4j database. We did not load the Student’s data in the above commands. This will be done later in the notebooks.

**Visualization with the Console**

The Neo4j console is a convent platform to visually explore data.

Try running the following commands to see a few of the patterns and relationships in your database.

* MATCH ()-[r:WORKED\_ON]->() RETURN r LIMIT 25
* MATCH ()-[r:WORKED\_WITH]->() RETURN r LIMIT 25
* MATCH ()-[r]->() RETURN r LIMIT 25

You can always return to the console to experiment with commands or to visually explore your data. If you think you are comfortable with the console, you can move on to the notebooks where we will learn how to use python to interface with our data set.

**Appendix: Deleting Data**

If you find yourself in a situation where you would like to delete all relationships and nodes from the database and start over, the following command will remove all entities from the database. You can then run the commands in the section above to repopulate the data from the CSVs

* MATCH (n)

DETACH DELETE n