# Assignment 1: Neo4j

**Instructions:**

* Before you start: FINISH THE LAB SESSION !!!
* The first task is to upload a dataset into Neo4j.
* The second task is to write and run different queries on this dataset.
* The third task is to write a reflection on Canvas.

**Deliverables (Summary, Detailed description in the tasks):**

* See task one: In a word document, a screenshot of your browser showing the Database Information and the result of the query mentioned at the end of task one.
* See task two: Write, run, and test your queries in neo4j. Copy and paste queries into the same word document as in task 1. Take a screenshot of the query and result and paste it into the word document under the respective query. Don’t forget the explanation for the last query.
* See task three: Write a reflection by replying to the forum "Assignment Reflection 1"
* Due date see canvas.

**Point Distribution and Grading Notes:**

* Task one: Upload dataset and take screenshot of browser showing Database Information and result of query. Dataset should be loaded in the database. (30 points)
* Task two: Queries should be correct. Query and screenshot for each query provided. Explanation for last query should be provided. (60 points)
* Task three: Post you reflection in the required length. (10 points)

## Explanation of Dataset

The data we are going to use for this, and possible other assignments, is from P2P lending platform Prosper. The data was downloaded in 2010 and it was part of a research project to look at P2P lending and herding behavior. For simplicity, I loaded the dataset into an Access database, later then in a MySQL database, cleaned it, and normalized it. I did not use all attributes available, but choose a subset as shown below in the ERD (Chen-Notation, done in ERDPlus) and in the relational model.

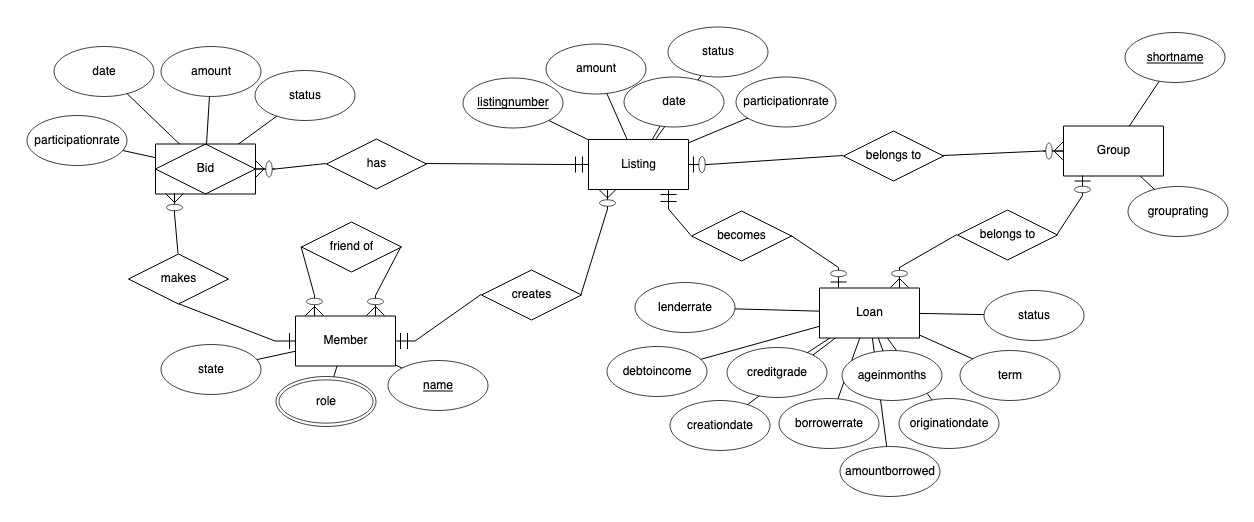
## Background information: P2P lending websites and Prosper

Prosper is an online auction marketplace for people-to-people (P2P) lending which functions similarly to eBay, however, instead of listing and bidding on items, people list and bid on loans using Prosper’s platform. Members (called borrowers) who need a loan can post requests for loans (called listings) on Prosper’s Website, members (called lenders) who are willing to lend their money to earn interests will then make their bids on a listing by specifying the amount they want to contribute to a loan and the interest rate by which they want to get paid. A listing for a loan request remains open for a certain duration during which bids for lending compete by the interest rates they ask. Once a listing gets fully funded, the bids with the lowest interest rates are then combined into a loan. This means, usually several lenders fund one loan.

Please note that the business model and available data changed over the years, also due to legal restrictions. The description pplies to the area before 2010/2011 when Prosper was quite generous with providing data.

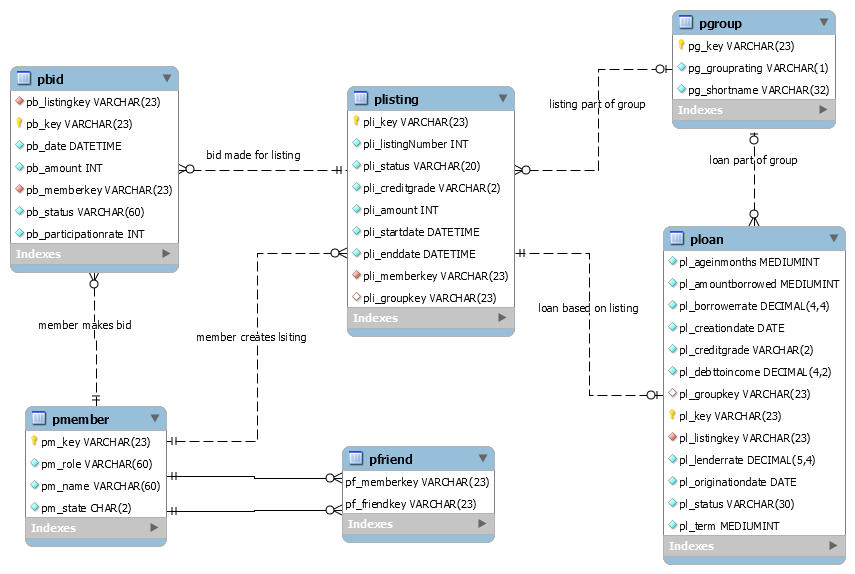
### Domain Data Model (Conceptual Model, database independent model, business view on the ata)

The following ERD shows the domain model in Chen notation (done in ERDPlus). Rectangles are entities and ovals are attributes. Underlined are business identifiers. Double sided ovals are multi-valued attributes (e.g., role could take on different values). A relationship is shown with a diamond with the verb phrase inside the diamond. Minimum and maximum cardinality are shown here similar to IE notation with zero, one, and many. We can see Members who create listings to request money and members who can bid on a listing. Completed listings become loans. Both listings and loans can be part of a group.



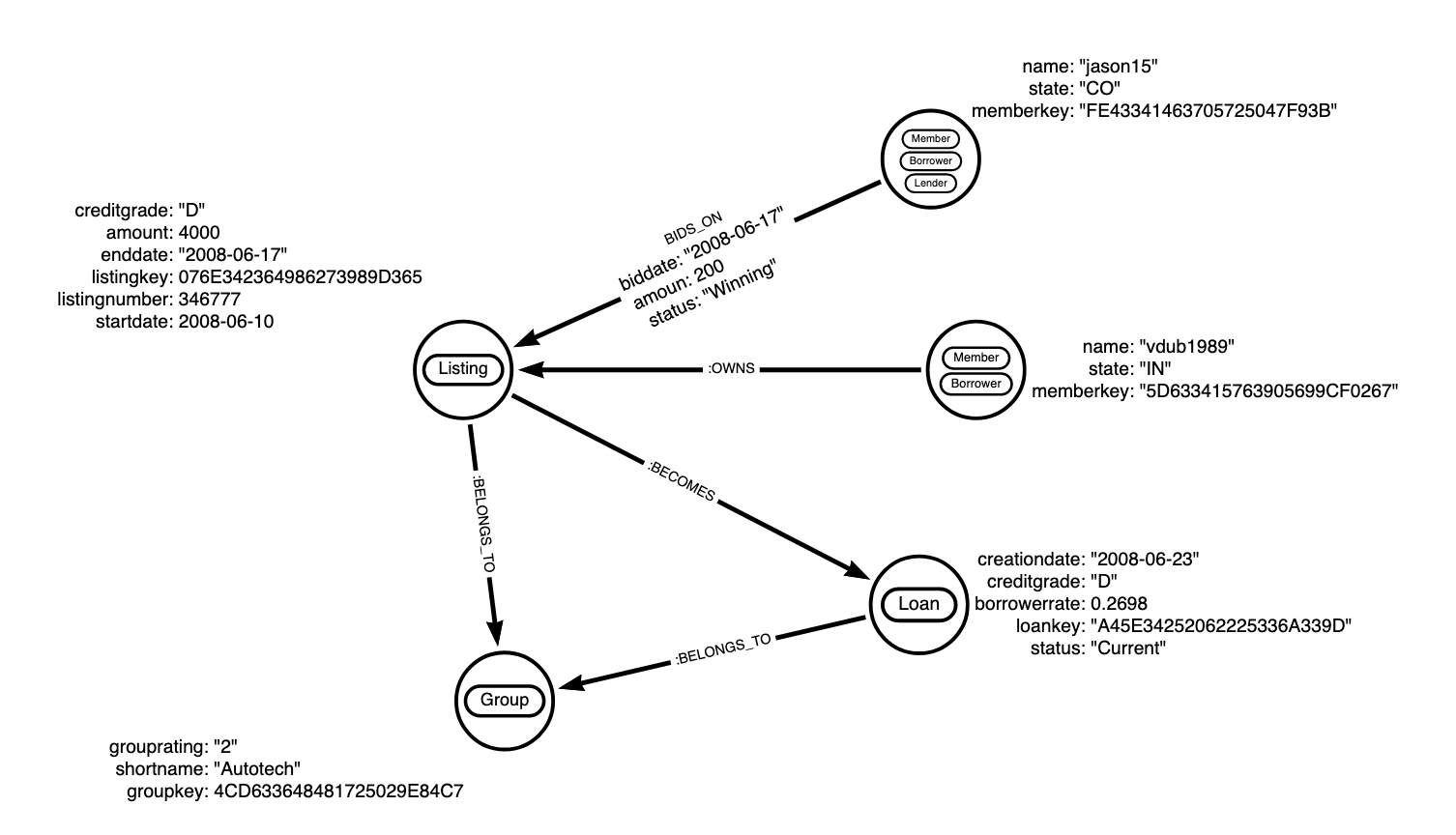
### Relational Model (Logical Model for a RDBMS)

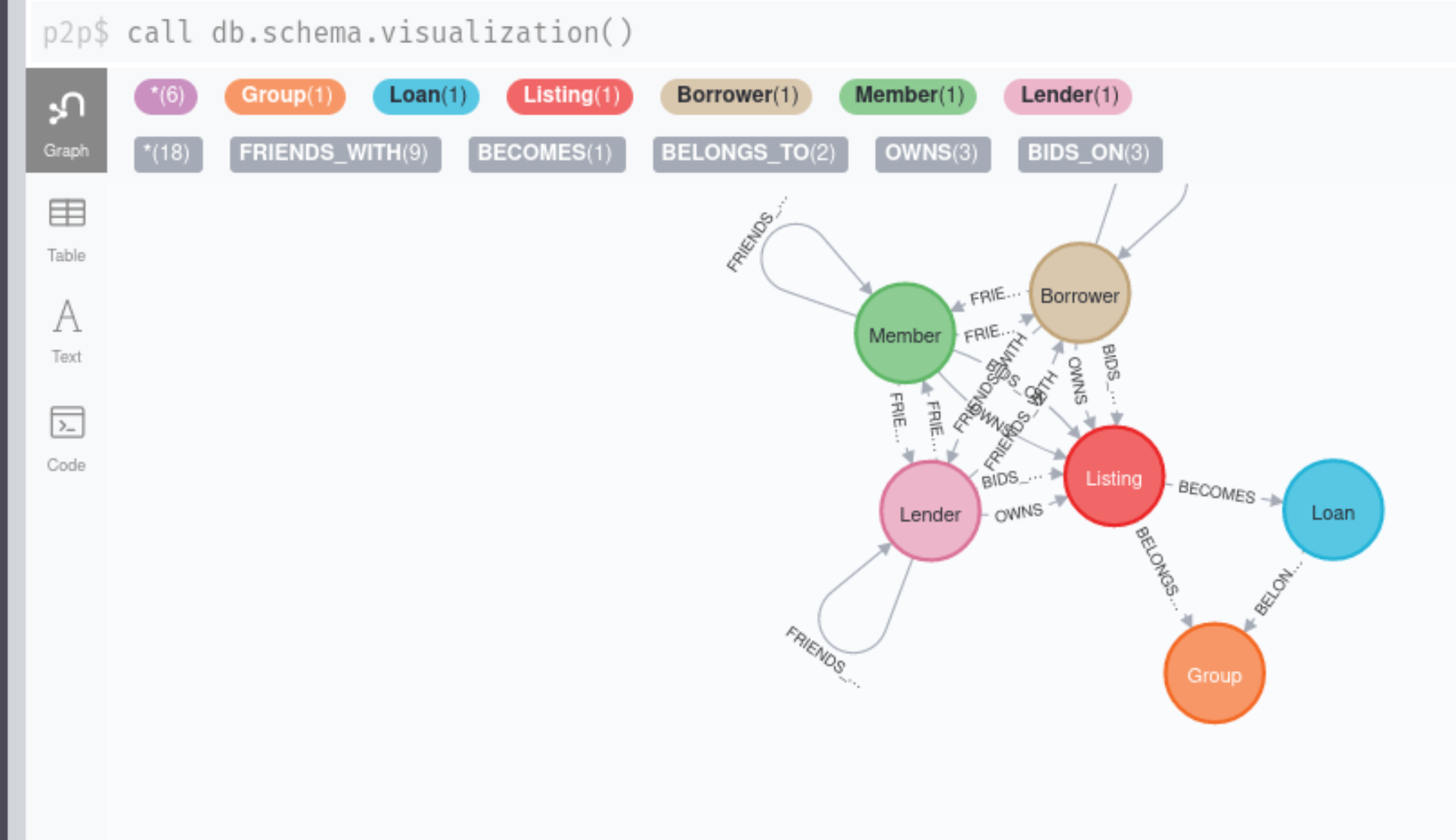
The next figure shows the relational model (reverse engineered from the MySQL database). The main difference is the addition of artificial keys to each of the entities, the foreign keys, and the data types.



### Property Graph Data Model (Logical Model for a Graph Databse)

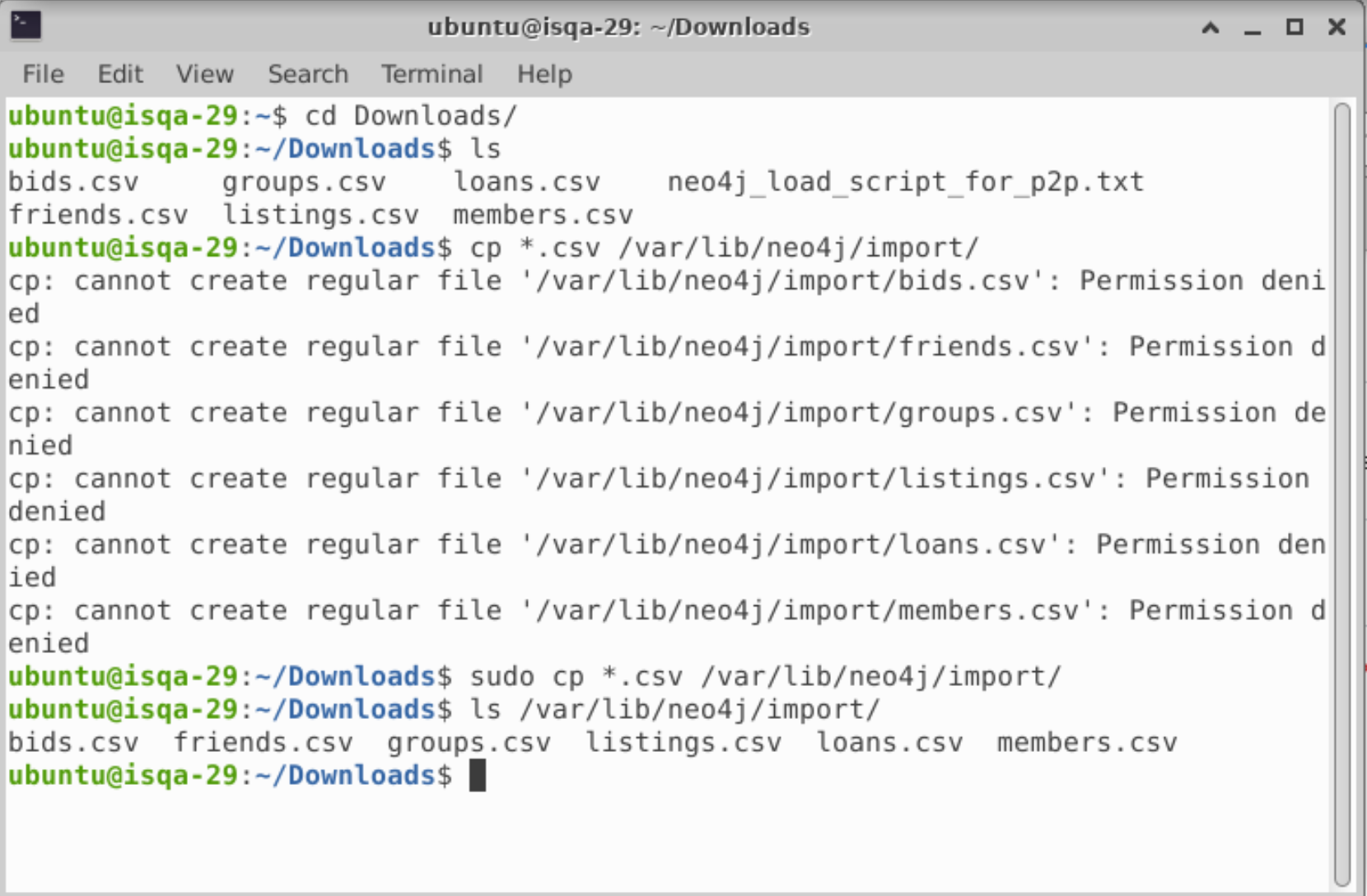
Based on the dataset above, I developed the following property graph data model. The properties show some example data. Please note, that there are several options to design the property graph model, the one below is only one option. FYI: You can review the schema by executing in Neo4k: call db.schema.visualization(), although the visualization is less helpful as multiple labels are represented as separate nodes. A member can take on multiple roles such as Lender and Borrower or both.





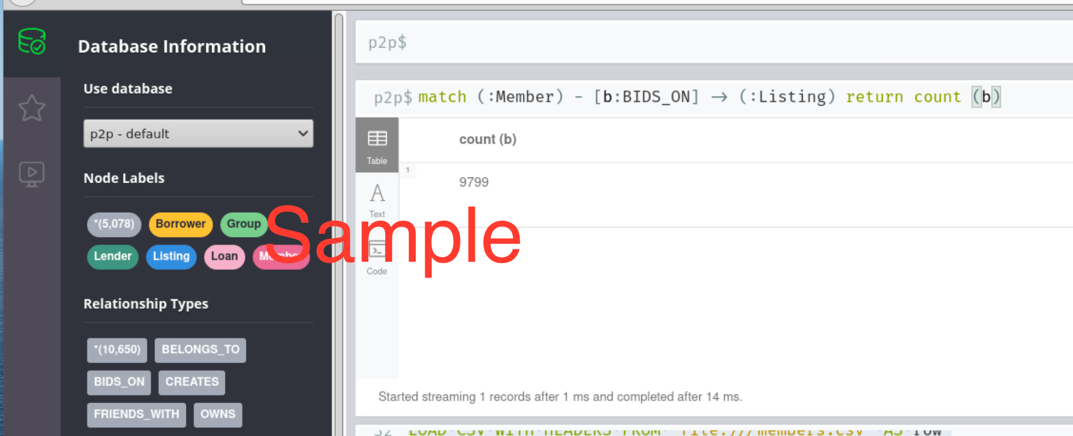
## Task 1: Loading data

1. Please **create and switch to a new database called p2p**. As a reminder, you need to stop the neo4j server, change the active database name to p2p in the neo4j.conf file, and then start neo4j server.
2. Please download the csv files first into your **Downloads** folder.
3. We will load in the data using csv files and neo4j csv file uploader (<https://neo4j.com/developer/guide-import-csv/>). Local files are referenced with a file:/// prefix and with the default setting local files can only be read from theNeo4j import directory (security reasons). In your **terminal window**, switch to the Downloads folder and use "**sudo cp**" to copy the files into the correct directory **/var/lib/neo4j/import** (the import directory is protected, so we need sudo in front of the copy command cp as you can see from the error below when I did not use sudo in front). Necessary commands see below. If the csv files from the assignment are the only csv files, you can use the commands as I did. If not, copy them separately.



1. Open the **neo4j\_load\_script\_for\_p2p.txt** load script available on Canvas (e.g., double-click in Downloads window or use gedit from a terminal). You will see different queries.
2. Copy and paste all queries from the file to your Neo4j prompt and run them at once. The queries will add new nodes and relationships. Some of the queries will also add indexes to improve query performance.

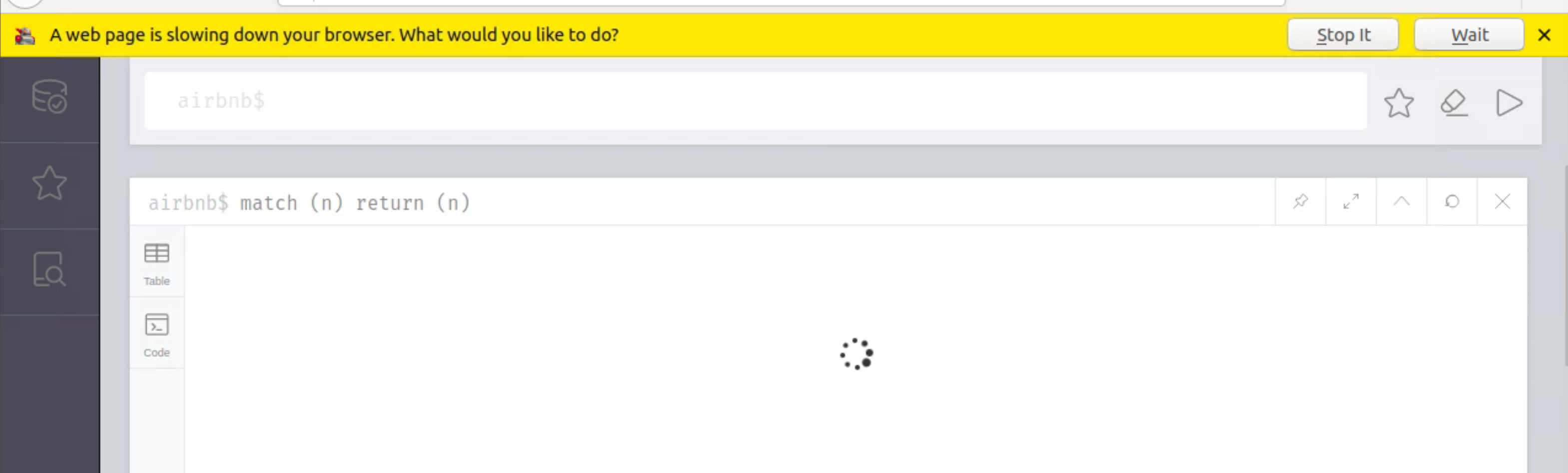
**Deliverables:** Take a screenshot (see example) showing the Database Information and the result of the following query:

match (:Member) – [b:BIDS\_ON] -> (:Listing) RETURN count (b) 

## Task 2: Queries

**Before we start: a note on query optimization**

Before we start: this graph is larger than the one we use in the tutorial. Path traversal can be very computing intense. If your query is not well written optimally, it can result in long running queries. The query may just continue running, or you might see the following:

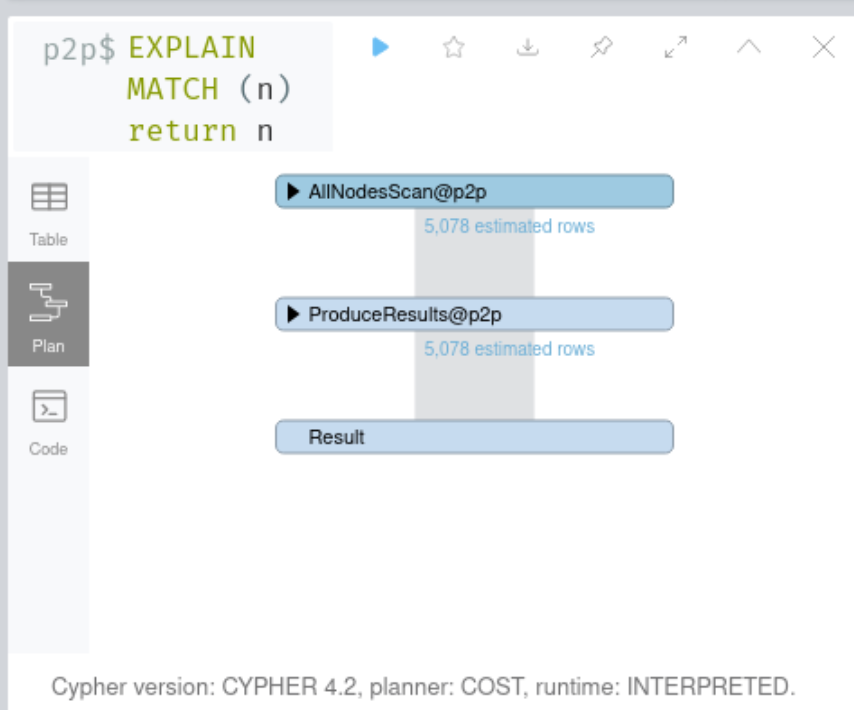
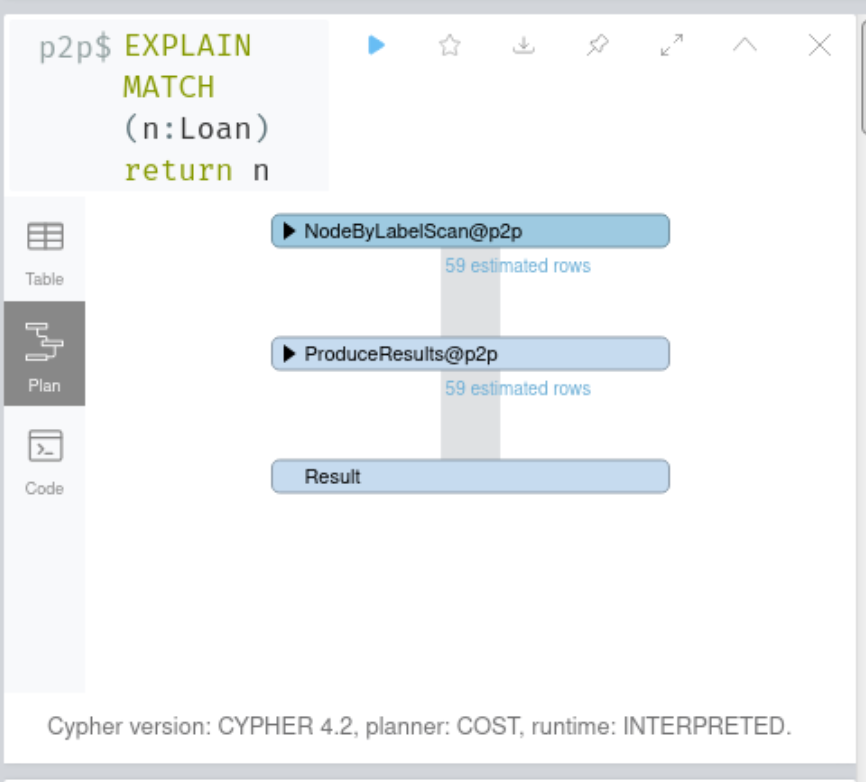


Query management is only implemented in the Enterprise edition of neo4j. In the community edition, there is no effective way to see or stop a query. You could try to stop the server and restart it, but this may result in a corrupt database. Depending on your query, it may be best to sit out the query and return to it at a later time.

One way to avoid running a long-running query is to avoid scanning unnecessary nodes. A useful command is EXPLAIN that you place in front of the query that you want to run. EXPLAIN will show you how neo4j executes the query and shows how many nodes will need to be scanned. Run the following queries:

EXPLAIN MATCH (n) RETURN n

EXPLAIN MATCH (n:Listing) RETURN n

You can see that by limiting the number of nodes to only Listing nodes, we reduced the need to scan all nodes.

If you are just interested in seeing one node, you need to limit your result. For example, if you only want to get one member node and see the relationships to this node, you could use:

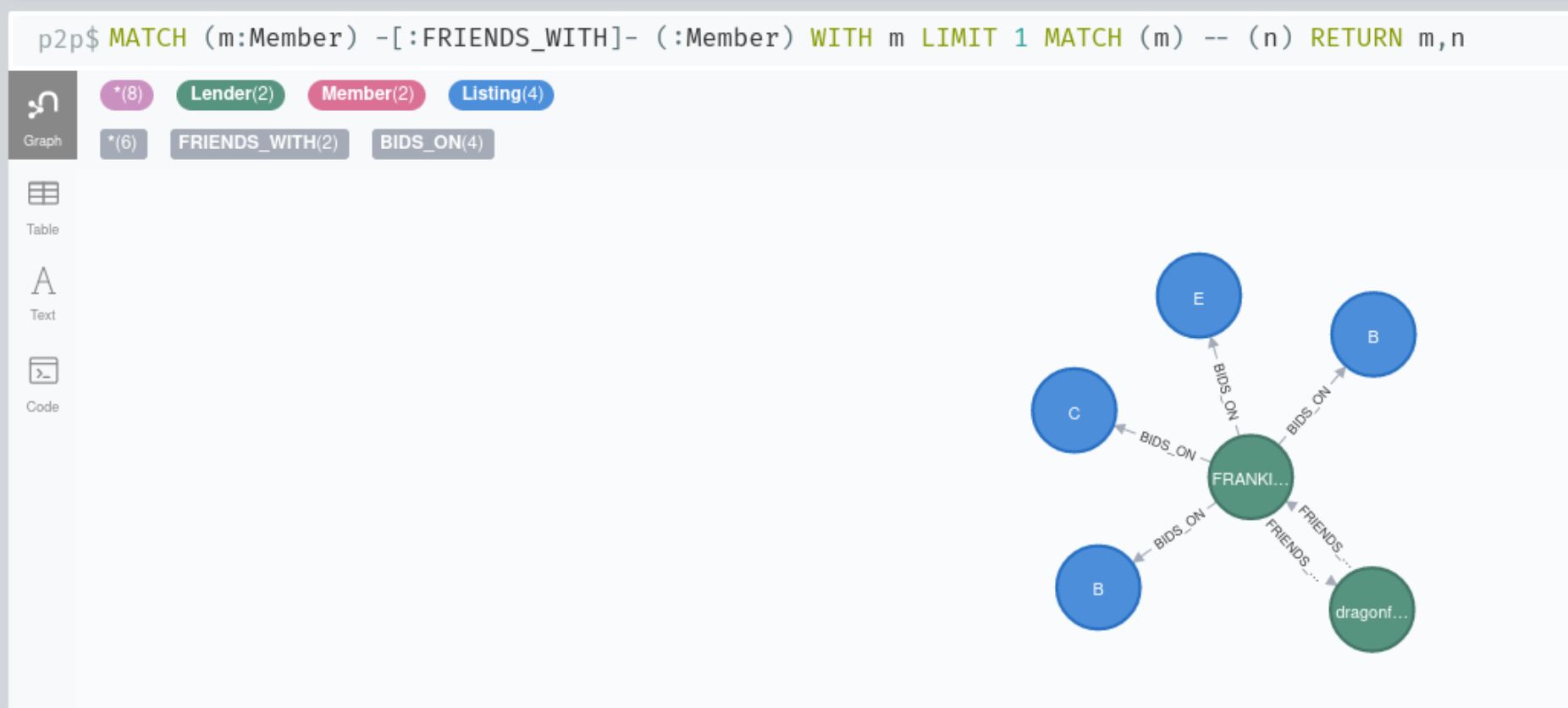
MATCH (m:Member) WITH m LIMIT 1 MATCH (m) -- (n) RETURN m,n

With this query, we first search for a Member node and pipe them to the rest of the query limiting it to one node. We then search for the pattern that this node is connected to another node. We return the one member node and all nodes it is connected to.

Or, if you want to use a member In the friendship subnetwork as a sample and see all relationships that this node has, a simple query would be:

MATCH (m:Member) -[:FRIENDS\_WITH]- (:Member) WITH m LIMIT 1 MATCH (m) -- (n) RETURN m,n

With this query, we first search for Member nodes that are in a friends relationship with another Member nodes and pipe them to the rest of the query limiting them to one node. We then search for the pattern that this node is connected to another node. We return the one member node and all nodes it is connected to. In short, try to anchor your queries in some way.



Path traversals are also very process intense. Like discussed in the tutorial, limit the path traversal by limiting the number of hops and/or making the type of relationship explicit.

**Let’s get started**

Let’s start by taking on a lender perspective. A lender wants to find a new listing to bid on.

1. First, let’s find a lender to anchor our queries to. A lender has three attributes memberkey, name, and state. Select a US state of your liking and then write a query that finds all lenders from this state but only if the lender name starts with the first letter of your name. Return the nodes.

(Tip 1: If you do not get a node, and assuming your query is correct, choose another state. You could write a query that counts the number of lenders for each state to see which state has the most lenders, of course …)

(Tip 2: Please note that the search strings are case sensitive. If you want to run a case insensitive search, you can apply toLower() to the attribute you are searching for, e.g., toLower(*attribute*))

1. The previous query probably returned quite a lot of lenders. Let’s find out who has the most bids. Revise the first query so that we count the number of bids a lender in the state and with the starting letter as in (1) has. Return the memberkey, the name, and the count of bids. Order the result by bid count descending. (Hint: we did this in the tutorial under “Count the number of relationships”). The result should be in table format: memberkey, name, count (you may have slightly different headings depending on the variables you chose)

(Tip: If you revise the query, don’t revise it in the same panel as the previous one. Copy and paste the query into the prompt and then revise it. That allows you to keep all queries in different panels, so that if you find an error or you need to choose another anchored lender, you don’t have to write each query out again. In other words, at the end of the this tutorial, you should have ten different panels open, one query for each question.)

1. From the previous result, choose the lender with the highest bid count. Copy the memberkey of this lender by highlighting the memberkey number and copying it (Ctr+c or Cmd+C). We will use the memberkey to anchor our following queries to this lender (from this on, we’ll reference this lender “anchored lender”).

Write a query that finds all listings that your chosen lender bid. Return the anchored lender and the listings. Show them in a graph.

1. Revise the previous query that it also optionally shows all associated loans with a bid (note: optionally because not all listing resulted in a loan). Show the graph with the anchored lender, the listings, and any loans.

(Tip: If your query does not return any loans, click on a listing note to verify that the status is indeed not “Completed”. Completed listings should have a loan associated with them. Other statuses (“Expired”) will not.)

1. Over the next few queries, our goal will be to develop a simple recommendation pattern.

We start by writing a query that finds all “Completed” listings (status: “Completed”) that our anchored lender bid on and also all “other lenders” that bid on the same completed listings. Return the graph that shows the anchored lender, the completed listings our anchored lender bid on, and all other lenders that bid on the same completed listings than our anchored lender.

(Tip: Be careful with the variable names you give. You cannot give the anchored lender the same variable name as the other lender nodes. I use numbers like in the tutorial when I explain the Recommendation pattern)

(Tip: If your anchored lender does not have completed listings, chose another lender)

1. Let’s build on this query. Revise the query so that it also shows all listings the other lenders (who bid on the same completed listings as our anchored lender) bid on. From this remove all listings that our anchored lender bid on. In other words, we are finding all lenders that bid on the same completed listing than our anchored lender and then search for other listings that our other lenders have also bid on but not our anchored lender. Return the listingnumber, amount, and credit grade of the listing that the anchored lender bid on, as well as the listing number, amount, and credit grade of the listings that the anchored lender has not bid on (our recommendations to him). Your table columns should be listingnumber, amount, credit grade, listingnumber, amount, credit grade where the first three columns are from the common listings, and the last three columns from the recommended listings.

(Tip: This is very similar to the first recommendation pattern explained in the tutorial)

(Tip: Again, be careful how you name your variables. You cannot give the same variable name to the common listings and the listings that the other lenders bid on)

1. Let’s do a final revision. Revise the previous query that it only recommends listings if the common listing and the recommended listing have the same amount and the same credit grade. The display will be the same as in the previous query 6.
2. Let’s switch gears. Write a query that displays all member nodes that are part of the friends network ( the FRIENDS WITH relationship ). The direction does not matter, btw, the relationship is bi-directional.
3. When you zoom out the graph, you may see a cluster of members all connected together and many pairs of members. Let’s see if our anchored lender is connected to someone in this cluster. Click on an arbitrary member in the cluster and copy the memberkey.

Write a query that gives you the shortestpath between this member of the cluster and the lender that you chose in query 3. Limit the path to a max of six hops. The types of relationships or directions of relationships do not matter.

(Tip: if an empty result is returned then this may not be wrong, it may just mean that your anchored lender is not connected to a member in the cluster at all or further separated than six hops. You can try out these two member numbers to test your query: A49A34100715607952C1209 and A55B33760335861723F81D3. If you do not get a result with these two, your query is not correct.)

1. A query of your choice. Please write down the query **and an explanation** what kind of question your query answers.

**Deliverables:** Write down each query and take a screenshot showing the pane that shows the result of the query. Write down an explanation for query 10 (query of your choice).

## Task 3: Reflection

Write a reflection (150 to 250 words) to the respective discussion board. Example questions you may answer:

* What kind of other questions could you answer with this graph?
* What refinements to the property graph could I suggest? (additions, changes, removals)
* What questions could this graph help answering that the relational model in the RDBMS could not?
* What did I like best when going through the tutorial and assignment? What did I dislike most?
* What challenges did I face going through the assignment? How did I solve the challenge?
* What open questions do I still have?
* How could I relate what I learned in class to my current or past work?

Each reflection needs to be written in your own words. There is absolutely no copying and pasting from other sources. If you reference other sources, you need to incorporate it in your entry completely in your own words (i.e., paraphrasing and not merely changing a few words). Repeating what other say is not what the reflection is about. The reflection is about your thoughts, your subjective view on a topic, your analysis, and your ideas.

**Deliverables:** Write and post your reflection to the discussion board "Assignment 1 Reflection”.