COSC 416A: Neo4j Assignment

Installation. [0]

You can access Neo4j on GPU1 at `/srv/neo4j/bin/neo4j-shell` via ssh, however, installing neo4j locally should be very easy. The installation guide is at http://www.neo4j.org/install -- on Linux, your package manager may have an appropriate package called neo4j; on Mac OS, if you use a package manager like brew, there is probably a neo4j package. It is up to you how you interact with neo4j.

You may also want to consult the chapter on autoindexing: http://docs.neo4j.org/chunked/stable/auto-indexing.html -- this allows you to START at nodes indicated by an index (with node_auto_index()). We have already enabled autoindexing for the fields in question 2.

Question 1. [10]

For this question, read the Cypher documentation found as part of the Neo4 manual. It can be viewed online at http://docs.neo4j.org/chunked/snapshot/. The relevant chapter is chapter 14.

a. [1] What START clause should you use to make a query where the starting point is every node in graph?

b. [1] Assuming proper indexes exist, which START clause should be used to make all nodes with the "name" property equal to "dude" starting points for a Cypher query?

c. [8] Modify the match clause in the Cypher query

```
START a=node(*)
MATCH a-->b
RETURN a, b
```

to match these relationships:

i. All relationships from b to a
ii. All relationships from a to b of type "isA"
iii. All relationships from a to b of type "wants" or "bought" or "sells"
iv. All relationships from a to b of depth at least 2
v. All relationships from a to b of depth no more than 5
vi. All relationships from a to b of any depth
vii. All relationships between a and b (in any direction) of depth exactly 3, naming the relationship "r"
viii. All relationships between a and b of depth between 2 and 5

(inclusive) and type “wants.”

Question 2. [10]

For this question, either connect to GPU1 and launch neo4j-shell with the test data with the following command or download the test Twitter data from the example data website:

```
/srv/neo4j/bin/neo4j-shell -path /srv/neo4j/examples/twitter/
-readonly
```

This will load in the example data set. Launch the cypher shell within the neo4j-shell by typing `cypher`. Please make sure you have the -readonly flag, because it is pretty easy to wreck things if you start writing to the dataset (and everyone needs this dataset!)

You should get in the habit of visualizing “tables” in the traditional relational sense as hierarchies. That is, “tuples” are connected to a “category node” which organizes the record set in to tables. node(0) is, by convention, the root of the organization tree.

Note that neo4j error messages are awful and this is not your fault. If you see “SyntaxException: Versions supported are 1.7, 1.8 and 1.9” there’s a really good chance you did NOT make a syntax error, but rather, nothing was returned from your query.

a. [3] Write a query that returns all the users in the Twitter database. To do this, note that node 2 has “category” USERS. *Hint: you should get more than 300 nodes returned, check your answer!*

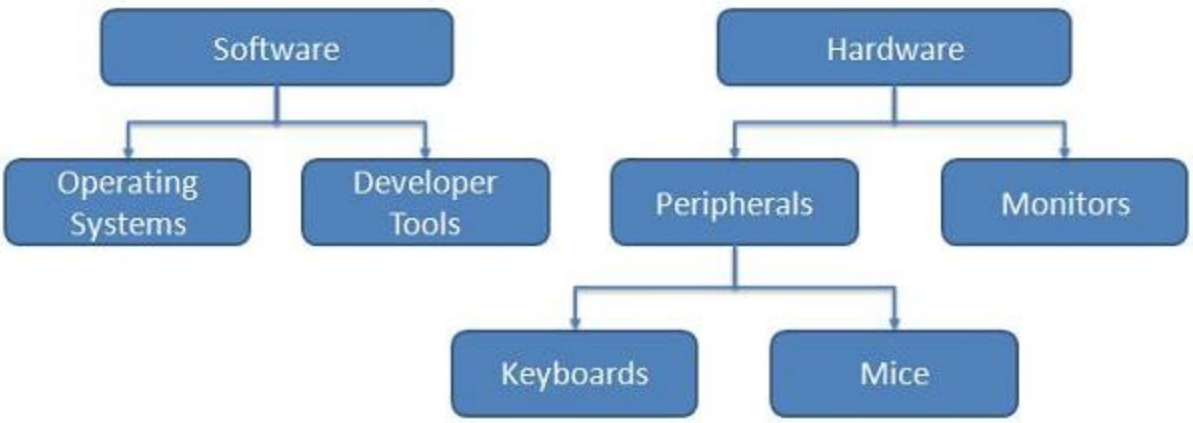
b. [7] In this data set, there are a number of types of relationships. KNOWS, USED, TWEETED, MENTIONS. TWEETED is a relation from a user to a Tweet that he sent. Return the text of the two tweets sent by the user with twid “nosqlweekly”. *Hint: make sure your match directions are correct, you cannot “WHERE” on fields that don’t exist in every single matched record!*

YOU MUST EXTEND THE PREVIOUS SOLUTION to answer this question. You cannot use auto indexes that do not exist. START from node(2), just like in part (a). There is no index on twid to rely on. :-)

Question 3. [30]

For accessing Neo4j in PHP or Java (or your programming language of choice), consult Chapter 5 of the Neo4j documentation.

A major advantage of using a graph datastore is it allows you to solve problems that a relational database requires "outside help" in solving. Consider, for example, a series of hierarchical categories:



We will write a small webpage that classifies video games by genre. It need not be complicated, but can be (if you so desire) incorporated into your lab 1 project.

We will use the following genre hierarchy:

Games
- Role Playing Games
  - Adventure
  - Massively Online-Multiplayer
- Strategy
  - RTS
  - Turn-based
- Shooters
  - First person
    - Multiplayer
  - Third Person

a. [15] Develop a relational schema that could be used by a SQL database (say, MySQL) that could be used to output nested lists of the categories. Provide both the schema and the code to produce these lists in your language of choice. (Hint: standard SQL won't get

you all the way through the problem on its own. You will need to use make your query “iterative” -- i.e., run a query in a loop or recursion)

You do not have to do anything fancy here, a single page that outputs the nested category hierarchy is sufficient. If you would like to, modify your lab 1 project to assign games to hierarchical categories.

Please make sure that your categories are in the “correct” order: that is, subcategories of a given category appear immediately under their parent category, nested appropriately to indicate that they are subcategories.

Bonus [+5]: use the SQL “WITH RECURSIVE” statement instead of running your query in a loop. You will need to use a relational database that supports this functionality (read: NOT MySQL, but either PostgreSQL or SQL Server).

b. [15] Reusing as much code as possible from part (a), make a new version of the category outputting webpage that uses Neo4j as the datastore instead of a relational database. If you would rather not do web development here, feel free to generate the hierarchical output with a command line application (nest child categories with an increasing number of hyphens).

If you would like to CREATE THE DATA LOCALLY, you may use the following CREATE statements to make your data. Note that you must run each create statement one at a time! PLEASE DO NOT CREATE THE DATA AGAIN ON gpu1!

```
create n = {category : 'Games'};

start n = node(*)
where n.category! = 'Games'
create n-[:parent_of]->(a{category:'Role Playing Games'});

start n = node(*)
where n.category! = 'Games'
create n-[:parent_of]->(a{category:'Strategy'});

start n = node(*)
where n.category! = 'Games'
create n-[:parent_of]->(a{category:'Shooters'});

start n = node(*)
where n.category! = 'Role Playing Games'
create n-[:parent_of]->(a{category:'Adventure'});

start n = node(*)
where n.category! = 'Role Playing Games'
create n-[:parent_of]->(a{category:'Massively Online-Multiplayer'});
```

```
start n = node(*)
where n.category! = 'Strategy'
create n-[:parent_of]->(a{category:'RTS'});

start n = node(*)
where n.category! = 'Strategy'
create n-[:parent_of]->(a{category:'Turn-based'});

start n = node(*)
where n.category! = 'Shooters'
create n-[:parent_of]->(a{category:'First person'});

start n = node(*)
where n.category! = 'Shooters'
create n-[:parent_of]->(a{category:'Third person'});

start n = node(*)
where n.category! = 'First person'
create n-[:parent_of]->(a{category:'Multiplayer'});
```