# COMP-3704-2
# NoSQL Databases

Lecture 14
Neo4j Intro, Graphs, Cypher and CRUD

Daniel Pittman, Ph.D., CISSP
05/21/2019

# Project 1 Questions

# Neo4j is Whiteboard Friendly

- To begin our coverage of Neo4j let's see what the book has to say:

*A bungee cord is a helpful tool because you can use it to tie together the most*

*disparate of things, no matter how awkwardly shaped or ill fitting they may*

*be. In a lot of ways, Neo4j is the bungee cord of databases, a system intended*

*not so much to store information about things as to tie them together and*

*record their connections with each other.*

# Neo4j is Whiteboard Friendly

- Neo4j is a member of the family of databases referred to as **graph databases**
  - Because it stores data in a graph!
- Neo4j is known to be “whiteboard friendly” because virtually any diagram that you can draw using boxes and lines on a whiteboard can be stored in Neo4j
- Neo4j focuses more on the **relationships between** values rather than on the **commonalities among sets** of values
  - Such as collections of documents or tables of rows
- This allows for Neo4j to store highly variable data in a natural and straightforward way

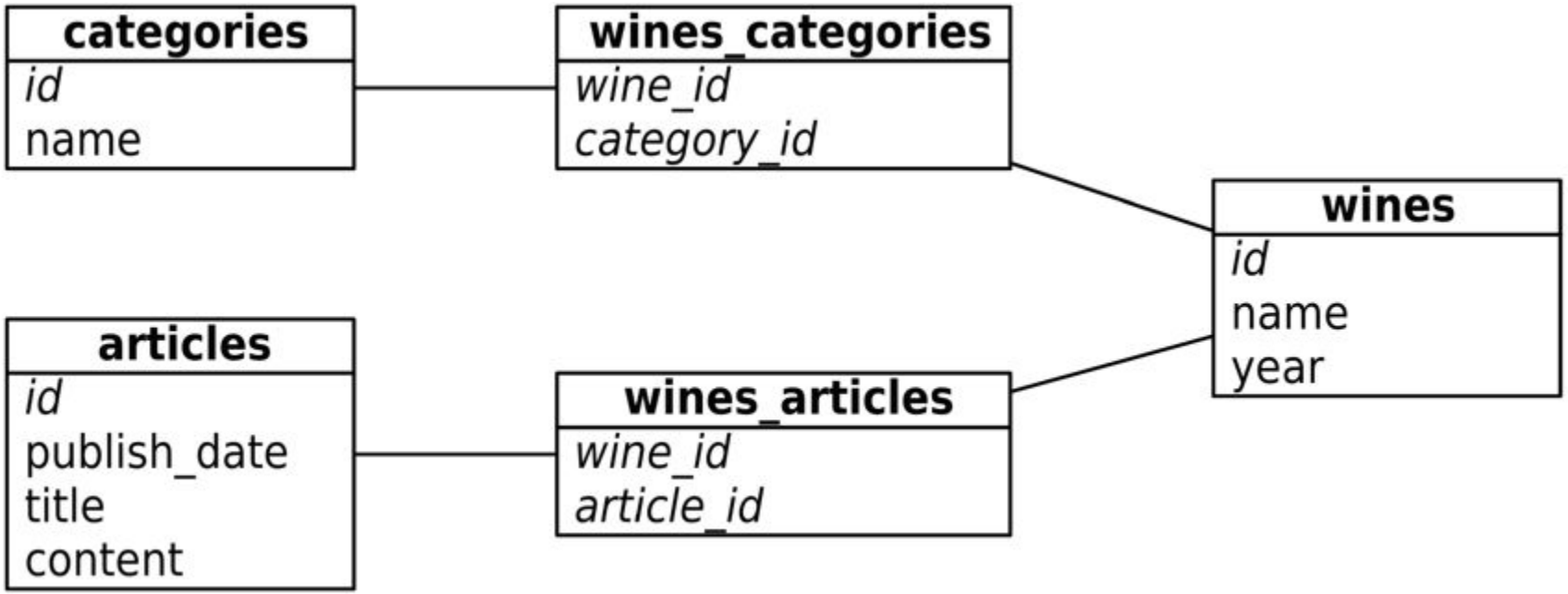
# Neo4j is Whiteboard Friendly

- In terms of scale, Neo4j is small enough to be embedded into any application
- It can also run in large cluster of servers using master-slave replication and store tens of billions of nodes and as many relationships!
- Neo4j can handle just about any size problem you can throw at it!

# Neo4j is Whiteboard Friendly

- Imagine that you need to create a wine suggestion engine in which wines are categorized by different varieties, regions, wineries, vintages, and designations
- Imagine that you also need to keep track of things like articles describing those wines, each of which is written by various authors
- What if you also wanted to allow users to track their favorite wines?
- Let’s compare a relational database UML diagram to what you would draw on a whiteboard:

# Neo4j is Whiteboard Friendly



# Neo4j is Whiteboard Friendly

# Neo4j is Whiteboard Friendly

- There is a saying in the relational database world: *on a long enough timeline, all fields become optional*
- Neo4j handles this implicitly by providing values and structure only where necessary
- If certain data is not available for a given node (i.e. a wine doesn't have a vintage available), then don't create that link!
- In graph databases such as Neo4j there are no schemas to adjust, the relationships are all that matter

# Installing Neo4j

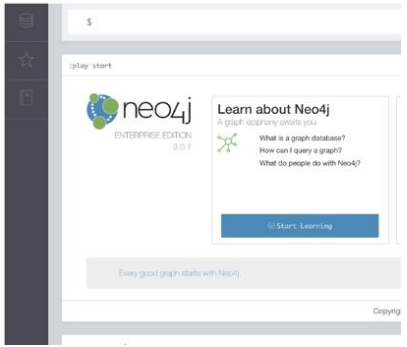
- There are two versions of Neo4j, the community (free) edition and the enterprise edition
- For the first two classes we will be using the community edition, but on day 3 we will switch to the enterprise edition to show some of the replication and HA features of the database
- For now let’s get Neo4j community edition installed!

# Installing Neo4j

- Download neo4j.zip from Canvas and unzip it into your docker folder
- Open a terminal and navigate to the docker/neo4j folder
- Run the following commands to start CouchDB:
  - **docker-compose build**
  - **docker-compose up**
- Attach to the container by running the following command in a new terminal:
  - **docker exec -it nosql-neo4j /bin/bash**

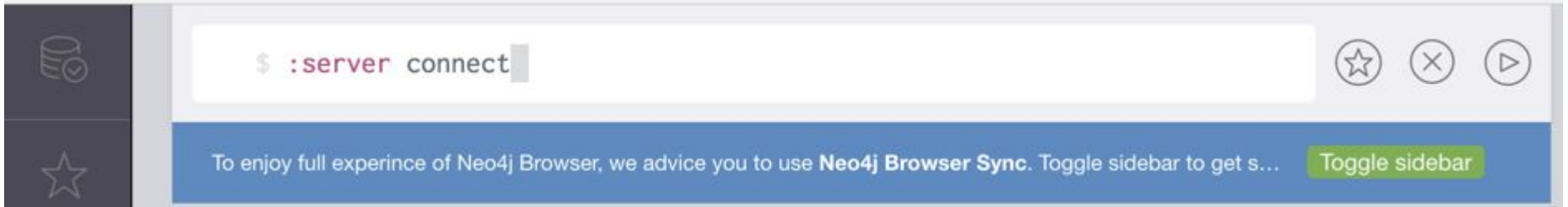
# Neo4j's Web Interface

- To make sure you have Neo4j running correctly, run the following cURL command from inside your container:
  - **curl http://localhost:7474/db/data/**
- We will explore using Neo4j via cURL more later, but for now let's look at its super awesome bundled web administration tool and data browser!
  - It also has a great graph data browser which will really help when getting started with graph traversal
- Open your browser to [http://localhost:7474/browser/](http://localhost:7474/browser/)
  - Remember, Windows 10 Home users will have a different URL!



# Neo4j's Web Interface

- At the top of the page type **:server connect** to connect to the database
  - Normally you'd need to login first to do this, but we've turned that off!
- You can enter **:help commands** at any time for an in-depth explanation of existing commands
- **:help cypher** will bring up a help page with instructions for specific Cypher commands
  - Cypher is the querying language we will be using throughout the web interface



# Aside: A Note on Terminology

## Nodes and Relationships: A Note on Terminology

A *node* in a graph database is not entirely unlike the nodes we talked about in prior chapters. Previously, when we spoke of a node, we meant a physical server in a network. If you viewed the entire network as a huge interconnected graph, a server node was a point, or *vertex*, between the server *relationships*, or *edges*.

In Neo4j, a node is conceptually similar: It's a vertex between edges that may hold data. That data is stored as a set of key-value pairs (as in many other non-relational databases we've talked about).

# Neo4j via Cypher

- There’s a lot of different ways to use Neo4j out there:
  - Client libraries in multiple programming languages
  - REST API
  - Querying languages:
    - Gremlin
    - Cypher (Pretty much the standard query language at this point)
- Cypher is a rich, Neo4j-specific graph traversal language
- Graph data points are referred to as **nodes**
- Graphs in Cypher consist of **nodes** rather than **vertices**
- Connections between nodes are called **relationships** (rather than edges)

# Neo4j via Cypher

- A Cypher statement to query Neo4j graphs might look something like this:

$ MATCH [some set of nodes and/or relationships]

WHERE [some set of properties holds]

RETURN [some set of results captured by the MATCH and WHERE clauses]

- In addition to querying using **MATCH**, you can create new nodes and relationships using **CREATE**, update the values associated with nodes and relationships using **UPDATE**, and more!

# Neo4j via Cypher

- Our database is pretty boring at the moment, so let's fix that by adding a node for a specific wine to our graph:

```
CREATE (w:Wine {name:"Prancing Wolf", style: "ice wine", vintage: 2015})
```

- The UI will show the result of the statement you just ran, including the nodes and/or relationships you just created
- The code tile will show detailed information about the action just performed (including the request that was made to Neo4j's REST API)
- If you want to access all nodes in the graph (like a SELECT *), run:

```
MATCH (n) RETURN n;
```

# Neo4j via Cypher

- Let’s add some more nodes!
- Since we are wanting to keep track of wine-reviewing publications in our graph, let’s create a node for the publication **Wine Expert Monthly**

```
CREATE (p:Publication {name: "Wine Expert Monthly"})
```

- In our last two CREATE statements **Wine** and **Publication** were **labels** applied to the nodes, not types
- Two nodes with the same label can have completely different sets of properties
  - Neo4j doesn’t require you to have predefined types!
  - If you want to enforce types you must do that at the application level
- Labels are very useful for querying, however, and that is how we will use them!

# Neo4j via Cypher

- OK! We now have two nodes, but with no relationships to one another
  - Let's fix that!
- Since **Wine Expert Monthly** reports on **Prancing Wolf** wine, let's create a new relationship **reported_on** that connects the two nodes:

```
MATCH (p:Publication {name: "Wine Expert Monthly"}),

(w:Wine {name: "Prancing Wolf", vintage: 2015})

CREATE (p)-[r:reported_on]->(w)
```

- In the statement above, we MATCHed the two nodes that we wanted to connect via their labels (Wine and Publication) and their name property
- We then created a **reported_on** relationship stored in the variable **r** and then RETURNed that relationship

# Neo4j via Cypher

- You can now see on the UI the relationship that was created
  - If you click on the relationship you'll see the ID of it is 0
- We can access information about the relationship either via REST:
  - curl http://localhost:7474/db/data/relationship/0
- Or via Cypher:

```
MATCH ()-[r]-()

WHERE id(r) = 0

RETURN r
```

- Relationships, like nodes, can contain properties
  - They can be thought of as objects in their own right!
- This makes sense since you don't just want to know a relationship exists, but you want to know **what** constitutes that relationship!

# Neo4j via Cypher

- Let’s say we wanted to specify which score **Wine Expert Monthly** gave the **Prancing Wolf** wine
- We can do that by adding a **rating** property to the relationship we just created:

```
MATCH ()-[r]-()

WHERE id(r) = 0

SET r.rating = 97

RETURN r
```

- We could have also done that at the time we created the relationship:

# Neo4j via Cypher

```
MATCH (p:Publication {name: "Wine Expert Monthly"}),

(w:Wine {name: "Prancing Wolf"})

CREATE (p)-[r:reported_on {rating: 97}]->(w)
```

- Now if we display the entire graph again using **MATCH (n) RETURN n;** and click on the relationship we'll see that **rating:97** is now a property of the **reported_on** relationship
- We also want to note that **Prancing Wolf** wine is made from **Riesling** grapes
- We **could** insert the info by adding a **grape_type: Riesling** to the **Prancing Wolf** node, but the more Neo4j thing to do is to create a new node for the Riesling grape type and add a relationship to it!

# Neo4j via Cypher

$ CREATE (g:GrapeType {name: "Riesling"})

$ MATCH (w:Wine {name: "Prancing Wolf"}),(g:GrapeType {name: "Riesling"})

CREATE (w)-[r:grape_type]->(g)

- Now we have a three-node graph: a wine, a type of grape, and a publication!
- But what if you want to delete nodes or relationships from a graph?
- The following statements will:
  - Create a new node
  - Establish a relationship between that node and our **Prancing Wolf** wine node
  - Delete the relationship
  - Delete the node
- Note: you can’t delete a node that still has relationships associated to it

# Neo4j via Cypher

```
$ CREATE (e: EphemeralNode {name: "short lived"})

$ MATCH (w:Wine {name: "Prancing Wolf"}),

        (e:EphemeralNode {name: "short lived"})

CREATE (w)-[r:short_lived_relationship]->(e)

$ MATCH ()-[r:short_lived_relationship]-()

DELETE r

$ MATCH (e:EphemeralNode)

DELETE e
```

# Neo4j via Cypher

- If you want to delete everything in your database and start from an empty graph, the following command will burn it all down!

```
MATCH (n)

OPTIONAL MATCH (n)-[r]-()

DELETE n, r
```

- Wineries typically produce more than one wine
- In an RDBMS you might create a separate table for each winery and store wines that they produce as rows
- In Neo4j the most natural thing to do is represent wineries as nodes in the graph, and create relationships between wineries and wines!

# Neo4j via Cypher

- Let’s create a node for **Prancing Wolf Winery** and add a relationship with the **Prancing Wolf** wine node we created earlier:

$ CREATE (wr:Winery {name: "Prancing Wolf Winery"})

$ MATCH (w:Wine {name: "Prancing Wolf"}),

(wr:Winery {name: "Prancing Wolf Winery"})

CREATE (wr)-[r:produced]->(w)

- We’ll also add two more wines produced by **Prancing Wolf Winery**: a **Kabinett** and a **Spätlese** as well as a **produced** relationship that specifies all of the wines are Rieslings

# Neo4j via Cypher

$ CREATE (w:Wine {name:"Prancing Wolf", style: "Kabinett", vintage: 2002})

$ CREATE (w:Wine {name: "Prancing Wolf", style: "Spätlese", vintage: 2010})

$ MATCH (wr:Winery {name: "Prancing Wolf Winery"}),(w:Wine {name: "Prancing Wolf"})

CREATE (wr)-[r:produced]->(w)

$ MATCH (w:Wine),(g:GrapeType {name: "Riesling"})

CREATE (w)-[r:grape_type]->(g)

- This results in a graph that’s fully flushed out, like this one:

# Neo4j via Cypher