Neo4J

 Neo4j is a new type of NoSQL datastore called a *graph database*.

As the name implies, it stores data as a graph (in the mathematical sense: <https://en.wikipedia.org/wiki/Graph_(mathematics)>).

It’s known for being “whiteboard friendly,” meaning if you can draw a design as boxes and lines on a whiteboard, you can store it in Neo4j.

Neo4j focuses more on the *relationships between* values than on the commonalities *among sets of* values (such as collections of documents or tables of rows).

In this way, it can store highly variable data in a natural and straightforward way.

Neo4j is small enough to be embedded into nearly any application.

On the other end of the spectrum, Neo4j can store tens of billions of nodes and as many edges.

And with its cluster support with master-slave replication across many servers, it can handle most any sized problem you can throw at it.

Neo4J Is Whiteboard Friendly

Imagine you must create a wine suggestion engine where wines have different varieties, regions, wineries, vintages, and designations.

Perhaps you need to keep track of articles by authors describing wines and perhaps you want to let users track their favorites.

A relational model may create a category table and a many-to-many relation- ship between a single winery’s wine and some combination of categories and other data.

But this isn’t quite how humans mentally model data.

Compare these two figures: Figure 30, *Wine suggestion schema in relational UML*, on

page 220 and Figure 31, *Wine suggestion data on a whiteboard*, on page 221.

There’s an old 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 a wine blend has no vintage, instead add a bottle year and point the vintages to the bottle year node.

There is no schema to adjust.

We’ll learn how to interact with Neo4j through a console and then through REST and search indexes.

We’ll work with some larger graphs with graph algorithms.

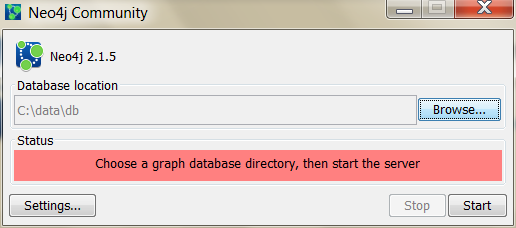
Finally we’ll take a peek at the enterprise tools Neo4j provides for mission-critical applications, from full ACID-compliant transactions to high-availability clustering and incremental backups.

We’ll use the Neo4j 3.2.5 Community Edition though there is also and Enterprise Edition for high availability.

Graphs, Cypher, and CRUD

Let’s start with the web interface to see how Neo4j represents data in graph form and how to walk around that graph.

Download latest Neo4j from <http://neo4j.com/download/> (the latest version 3.2.5 is on the X drive)

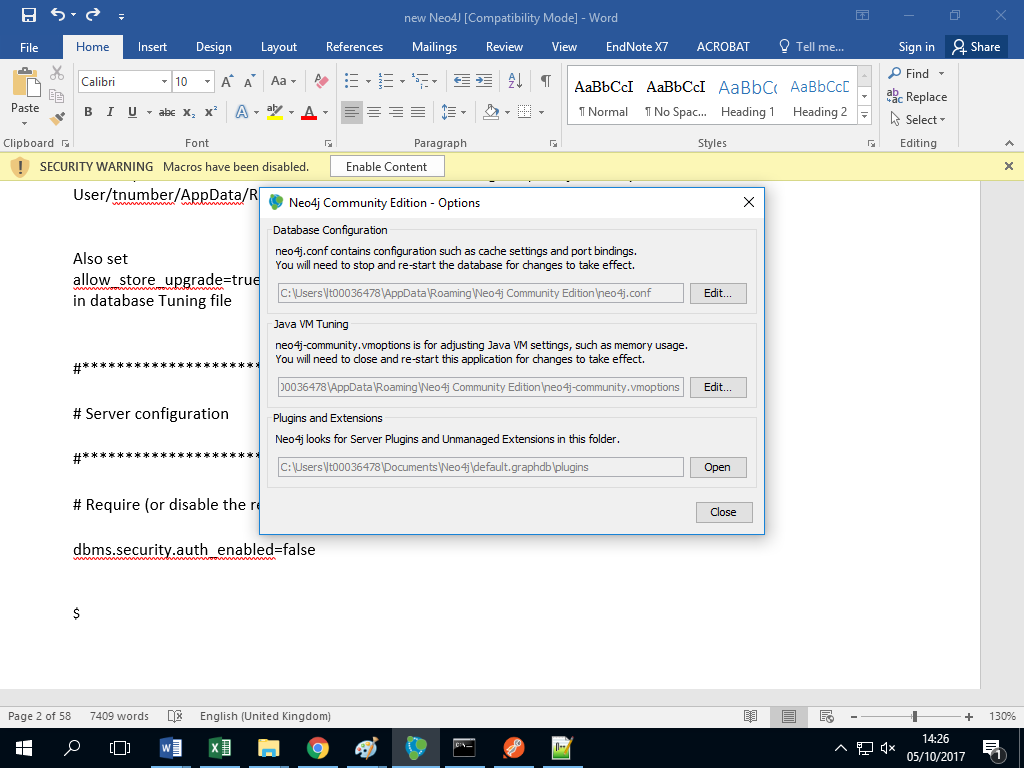


Note where it creates the database.

C:\Users\lt00\Documents\Neo4j\defaults.graphdb

Click Start

To make sure you’re up and running, try curling this URL from a command line prompt or using postman in chrome (but first turn off authentication in Database Tuning file and restart (neo4j.conf in your User/tnumber/AppData/Roaming/Neo4j folder)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Server configuration

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Require (or disable the requirement of) auth to access Neo4j

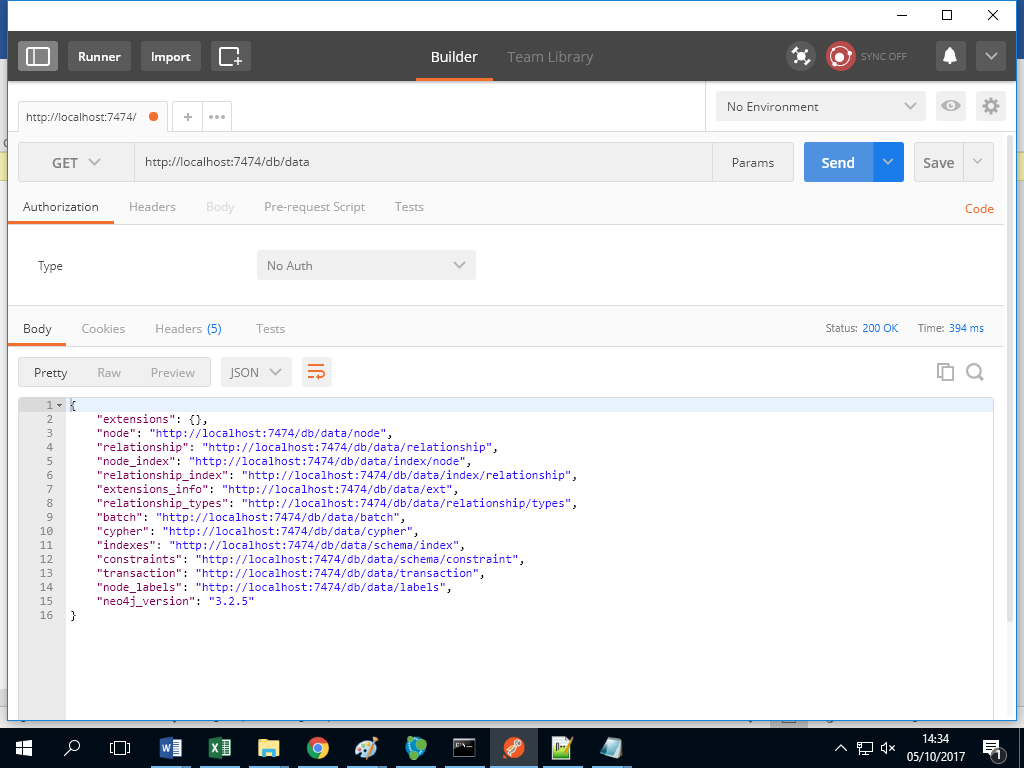
dbms.security.auth\_enabled=false

$

 Also set

allow\_store\_upgrade=true

in database Tuning file



Like CouchDB, the default Neo4j package comes equipped with a substantial web administration tool and data browser, which is excellent for playing with toy commands.

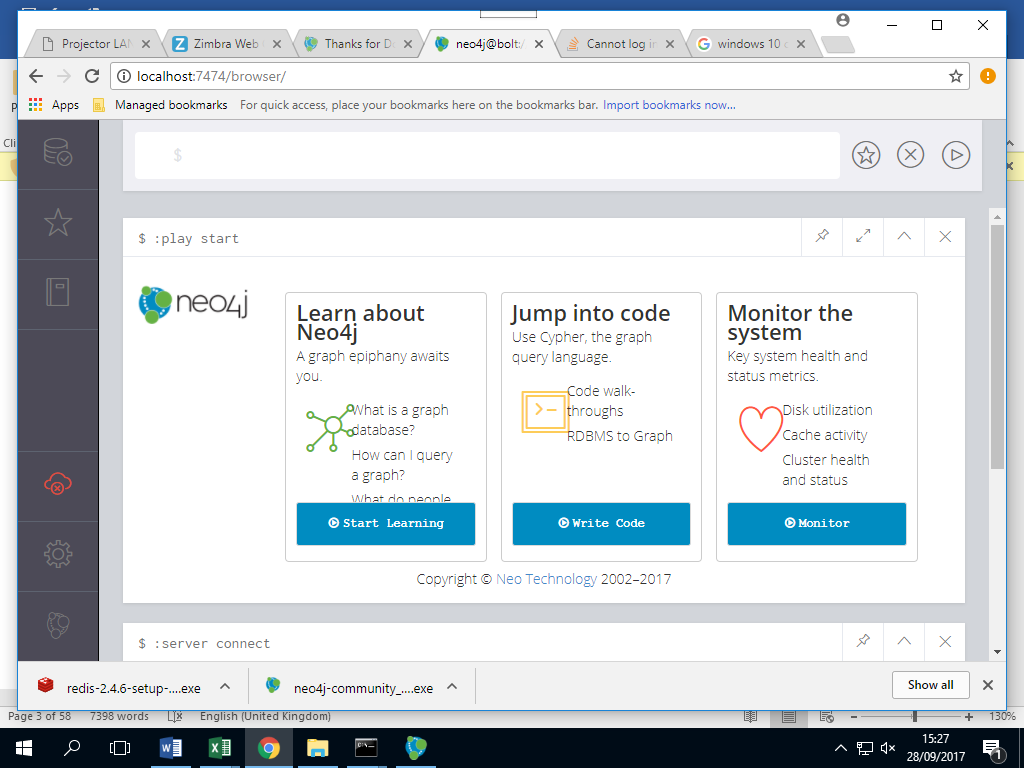
If that weren’t enough, it has one of the coolest graph data browsers we’ve ever seen.

Neo4j’s Web Interface

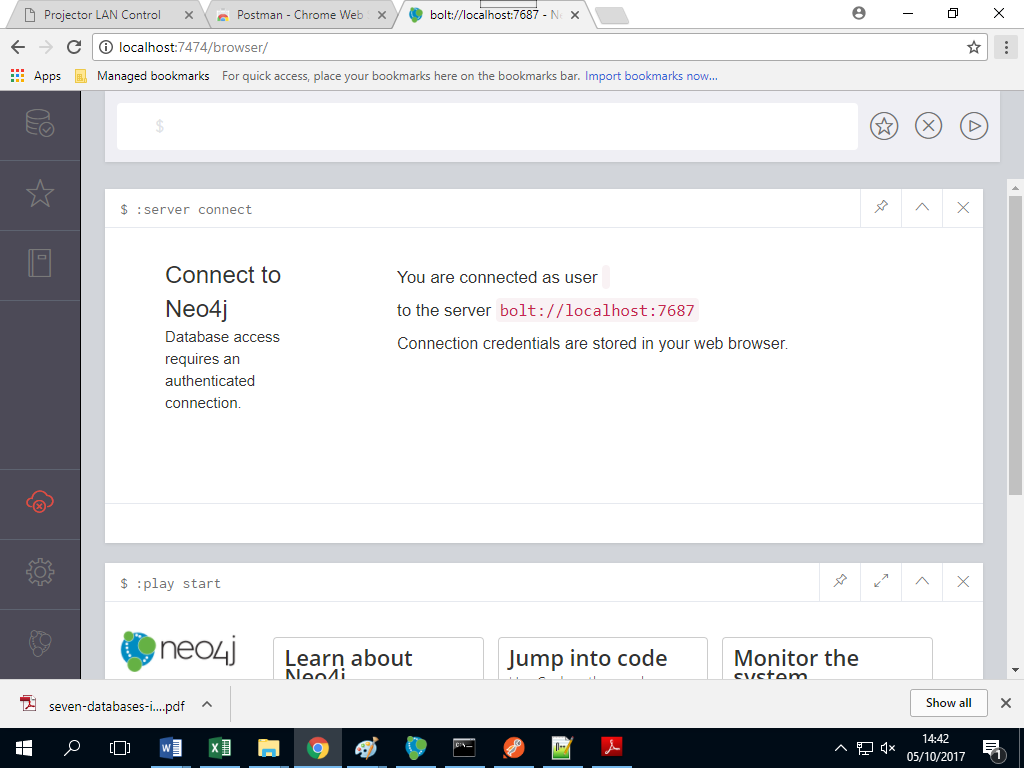
Launch a web browser (IE works better, Chrome doesn’t seem to show graph icon), and navigate to the administration page.

http://localhost:7474/browser

username and password (if authorisation is enabled): neo4j



That will open up a command-line-style interface at the top of the page (distinguished by the $ on the far left). Type in :server connect to connect to the database (or :server status to see the status)



Type :help to see all commands

Click the Database icon option at the top right.

 Neo4j via Cypher

There are several ways that you can interact with Neo4j. In addition to client libraries in a wide variety of programming languages, you can also interact with Neo4j via a REST API, and via two querying languages created with Neo4j exclusively in mind: *Gremlin* and *Cypher*. While Gremlin has some interesting

properties, Cypher is now considered standard.

Developer manual at : https://neo4j.com/docs/developer-manual/current/cypher/clauses/delete/

Cypher is a rich, Neo4j-specific graph traversal language. In Cypher, as in mathematical graph theory, graph data points are called *nodes*. Unlike in graph theory, however, connections between nodes are called *relationships* in Cypher (rather than *vertices*). Statements used to query Neo4j graphs in Cypher typically 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 the graph using MATCH, you can create new nodes and relationships using CREATE, update the values associated with nodes and relationships using UPDATE, and much more.

At the moment, our not-so-exciting Neo4j graph consists of no nodes and no relationships. Let’s get our hands dirty and change that by adding a node for a specific wine to our graph. That node will have a few properties: a name property with a value of Prancing Wolf, a style property of ice wine, and a vintage

property of 2015. To create this node, enter this Cypher statement into the console:

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

In the section of the web UI immediately below the console, you should see output like this.



At the top you’ll see the Cypher statement you just ran. The Tables tile shows you the nodes and/or relationships that you created in the last Cypher statement, and the Code tile provides in-depth information about the action you just completed (mostly info about the transaction that was made via Neo4j’s REST API).

At any time we can access all nodes in the graph, kind of like a SELECT \* FROM entire\_graph statement:

$ MATCH (n)

RETURN n;

At this point that will return just one solitary node. Let’s add some others. Remember that we also want to keep track of wine-reviewing publications in our graph. So let’s create a node representing the publication Wine Expert Monthly:

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

In the last two statements, Wine and Publication were *labels* applied to the nodes, not types. We could create a node with the label Wine that had a completely different set of properties. Labels are extremely useful for querying purposes, as we’ll see in a bit, but Neo4j doesn’t require you to have pre-defined types. If you *do* want to enforce types, you’ll have to do that at the application level.

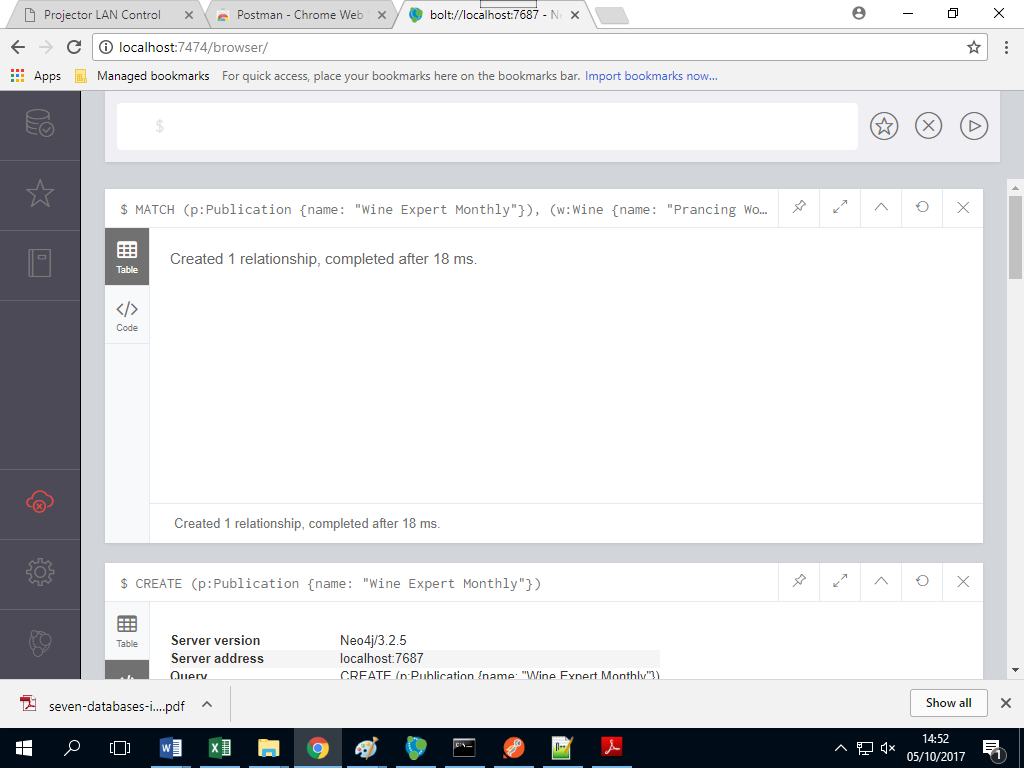
So now we have a graph containing two nodes but they currently have no relationship with one another.

Since Wine Expert Monthly *reports on* this Prancing Wolf wine, let’s create a reported\_on relationship that connects the two nodes:

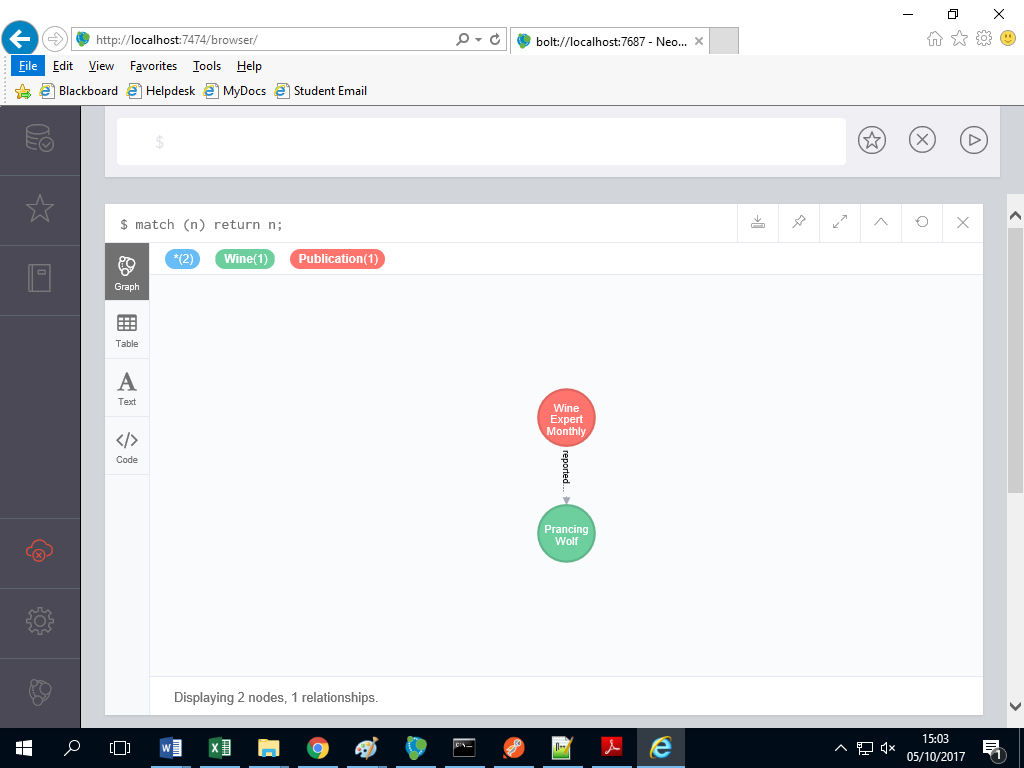
MATCH (p:Publication {name: "Wine Expert Monthly"}),

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

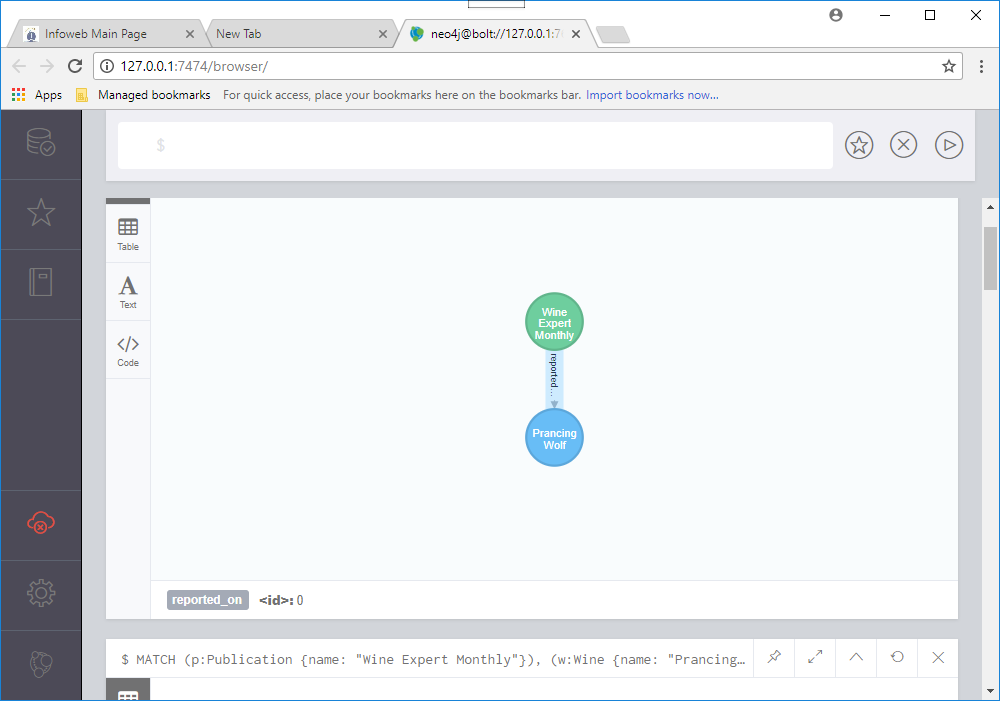
CREATE (p)-[r:reported\_on]->(w)

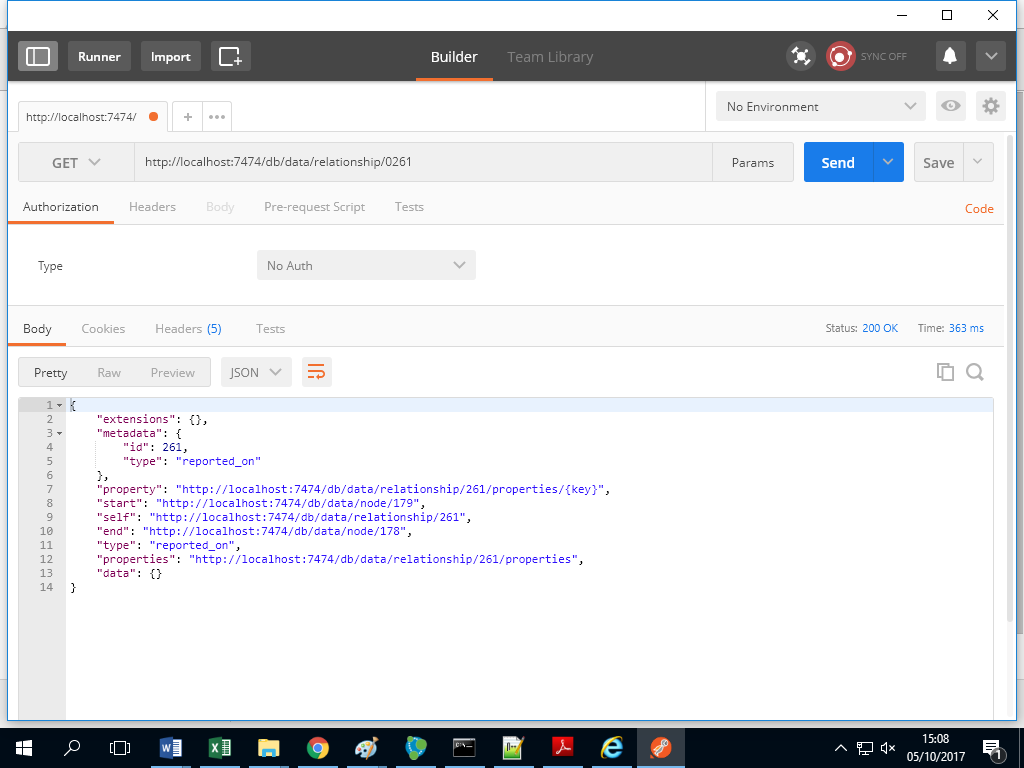


In this statement we’ve MATCHed the two nodes that we want to connect via their labels (Wine and Publication) and their name property, created a reported\_on relationship and stored that in the variable r, and finally RETURNed that relationship.



If you click on the relationship between the nodes in the web UI, you can see that the ID of the relationship is 0

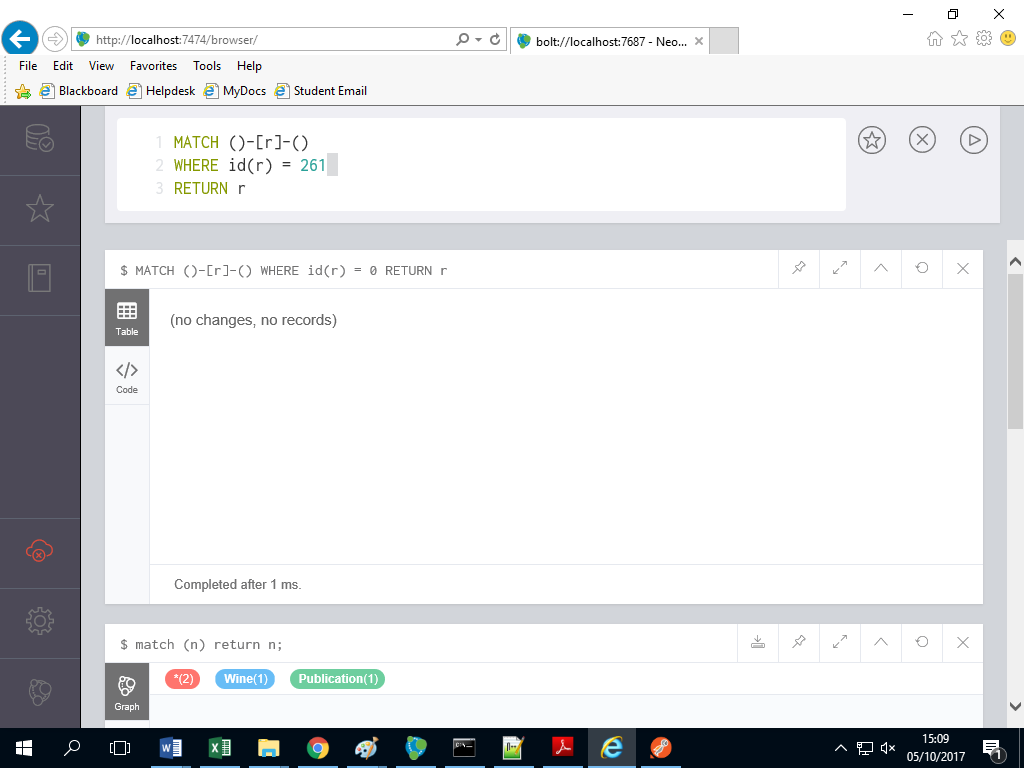
You can use Neo4j’s REST interface to access information about the relationship at http://localhost:7474/db/data/relationship/0

or via Cypher by running:

MATCH ()-[r]-()

WHERE id(r) = 0

RETURN r



Relationships, like nodes, can contain properties and can be thought of as objects in their own right. After all, we don’t want to know simply *that* a relationship exists; we want to know *what* constitutes that relationship.

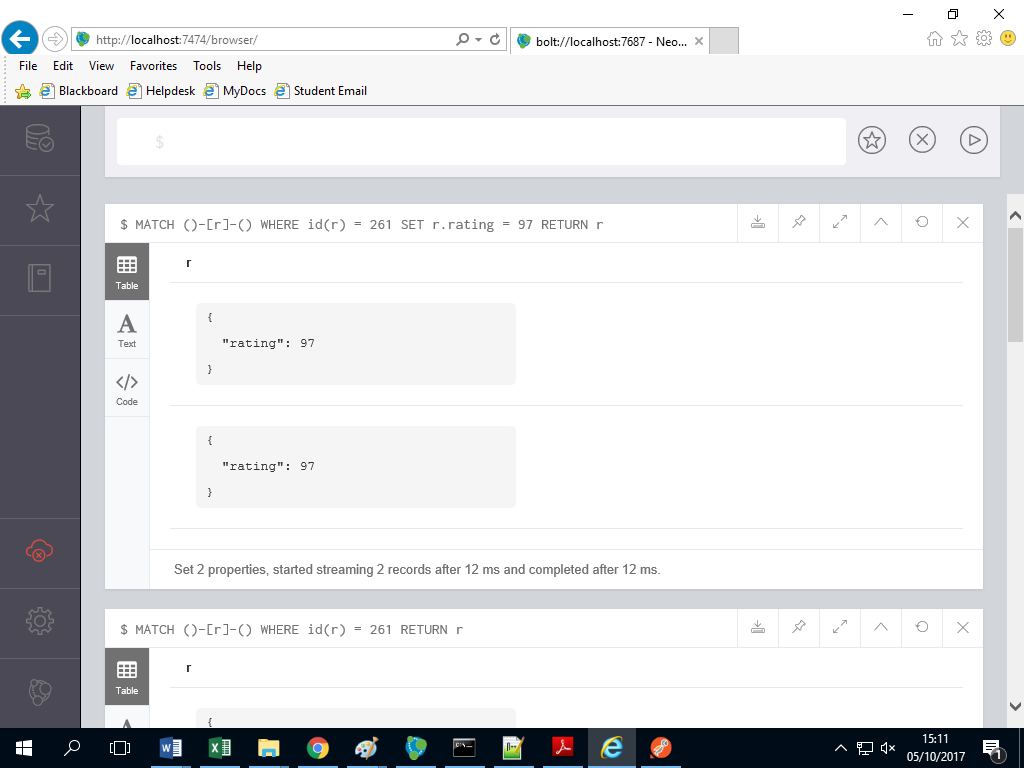
Let’s say that we want to specify which score Wine Expert Monthly gave the Prancing Wolf wine. We can do that by adding a score property to the relationship 0 that we just created.

MATCH ()-[r]-()

WHERE id(r) = 0

SET r.rating = 97

RETURN r



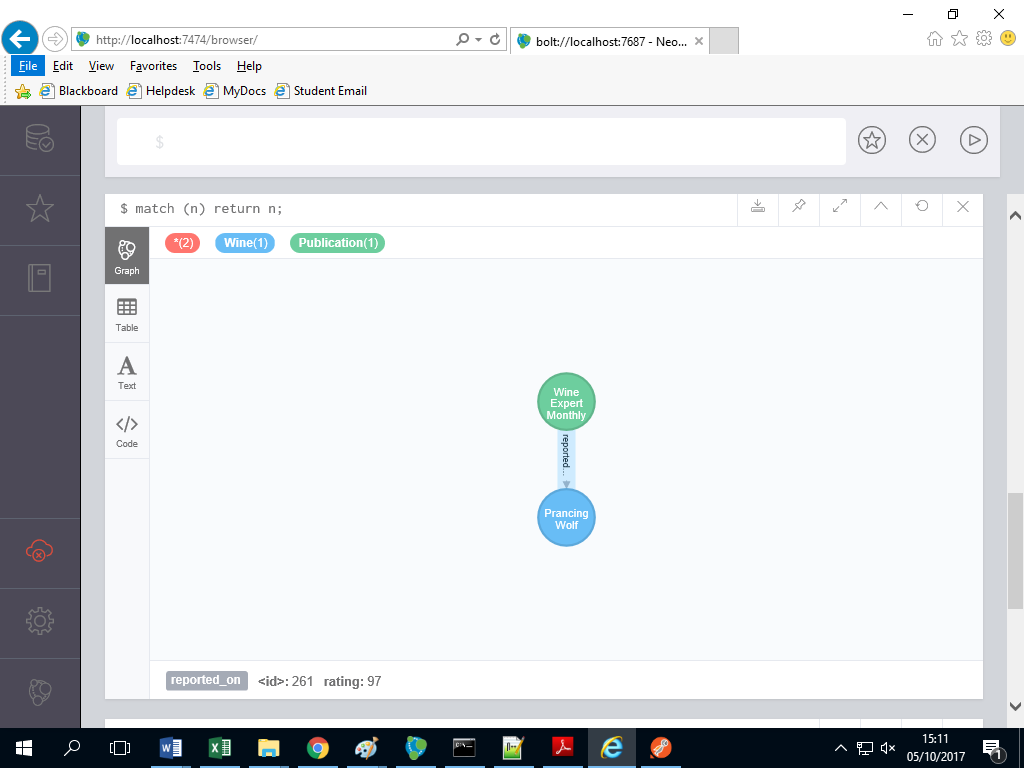
We also could’ve specified the rating when creating the relationship, like this:

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

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

CREATE (p)-[r:reported\_on {rating: 97}]->(w)

At this point, if you display the entire graph again using MATCH (n) RETURN n; and click on the relationship, you’ll see that rating: 97 is now a property of the reported\_on relationship.



Another bit of info that we want to note is that the Prancing Wolf wine is made from the *Riesling* grape. We *could* insert this info by adding a grape\_type: Riesling property to the Prancing Wolf node, but let’s do things in a more Neo4j-native fashion instead by creating a new node for the Riesling grape type and adding relationships to wines of that type:

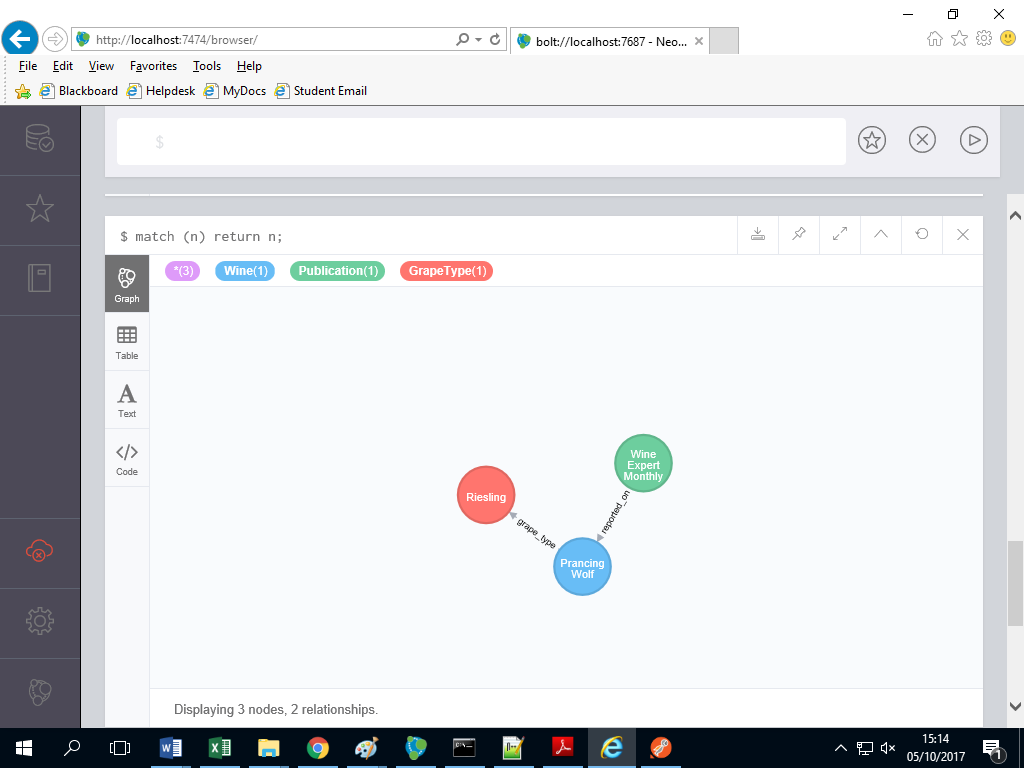
$ CREATE (g:GrapeType {name: "Riesling"})

Let’s add a relationship between the Riesling node and the Prancing Wolf node using the same method:

$ 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.



So far we’ve created and updated both nodes and relationships. You can also delete both from a graph.

Below are three Cypher statements that will create a new node, establish a relationship between that node and one of our existing nodes, delete the relationship, and then delete the node (you can’t delete a node that still has relationships associcated with it):

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

Our wine graph is now back to where it was before creating the short lived node.

Speaking of deletion, if you ever want to burn it all down and start from scratch with an empty graph, you can use the following command at any time to delete all nodes and relationships. But beware! This command will delete the entire graph that you’re working with, so run it only if you’re sure that you’re ready to move on from a graph’s worth of data for good.

MATCH (n)

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

DELETE n, r

Now that we know how to start from scratch, let’s continue building out our wine graph. Wineries typically produce more than one wine. To express that relationship in an RDBMS we might create a separate table for each winery and store wines that they produce as rows.

The most natural way to express this in Neo4j would be—you guessed it—to represent wineries as nodes in the graph and create relationships between wineries and wines. Let’s create a node for Prancing Wolf Winery and add a relationship with the Prancing Wolf wine node that 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—and also create produced relationships and specify that all of the Prancing Wolf wines are Rieslings.

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"})

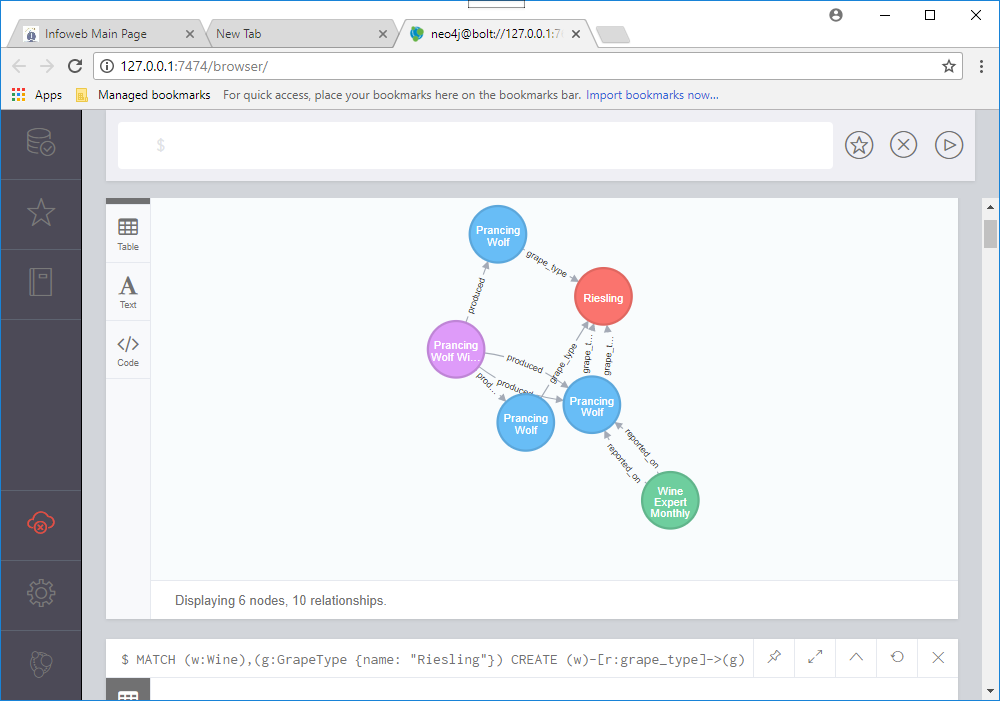
CREATE (wr)-[r:produced]->(w)

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

CREATE (w)-[r:grape\_type]->(g)

This will result in a graph that’s fully fleshed out, like the one shown here.

MATCH (n) return n;



Schemaless Social

In addition to knowing about wines, wineries, and publications, we want our wine graph to have a social component, i.e. we want to know about the *people* affiliated with these wines and their relationships with one another. To do that, we just need to add more nodes.

Suppose that we want to add three people, two who know each other and one stranger, each with their own wine preferences.

Alice has a bit of a sweet tooth and so she’s a big fan of ice wine.

CREATE (p:Person {name: "Alice"})

MATCH (p:Person {name: "Alice"}),

(w:Wine {name: "Prancing Wolf", style: "ice wine"})

CREATE (p)-[r:likes]->(w)

Tom likes Kabinett and ice wine and trusts anything written by *Wine Expert Monthly*.

CREATE (p: Person {name: "Tom"})

MATCH (p:Person {name: "Tom"}),(w:Wine {name: "Prancing Wolf", style: "ice wine"})

CREATE (p)-[r:likes]->(w)

MATCH (p:Person {name: "Tom"}),

(pub:Publication {name: "Wine Expert Monthly"})

CREATE (p)-[r:trusts]->(pub)

Patty is friends with both Tom and Alice but is new to wine and has yet to choose any favorites.

CREATE (p:Person {name: "Patty"})

MATCH (p1:Person {name: "Patty"}),

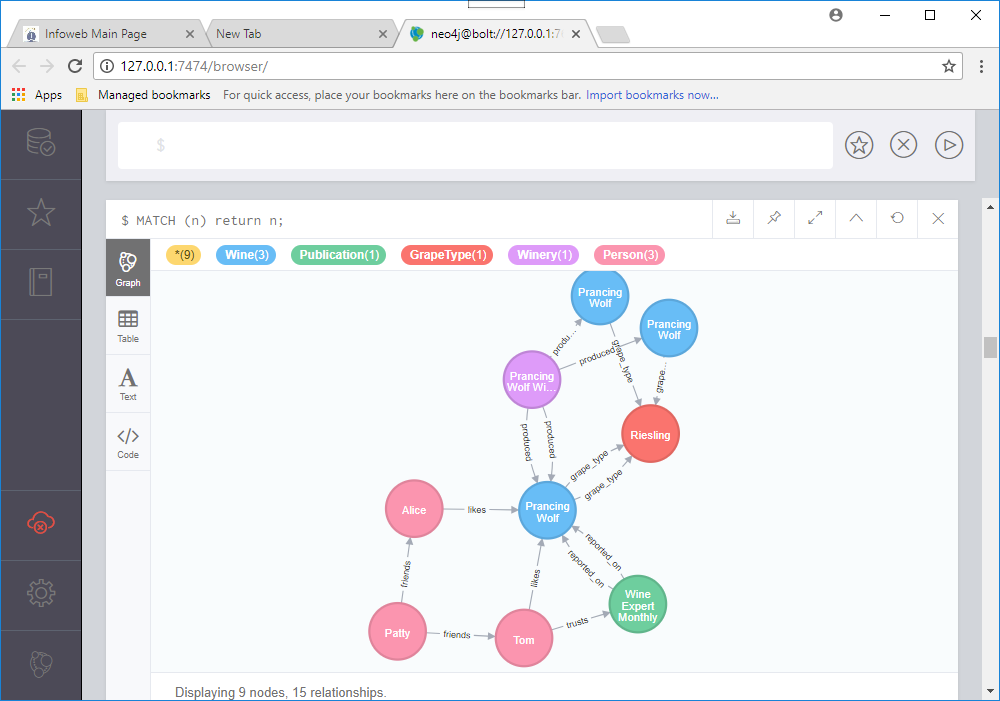
(p2:Person {name: "Tom"})

CREATE (p1)-[r:friends]->(p2)

MATCH (p1:Person {name: "Patty"}),

(p2:Person {name: "Alice"})

CREATE (p1)-[r:friends]->(p2)



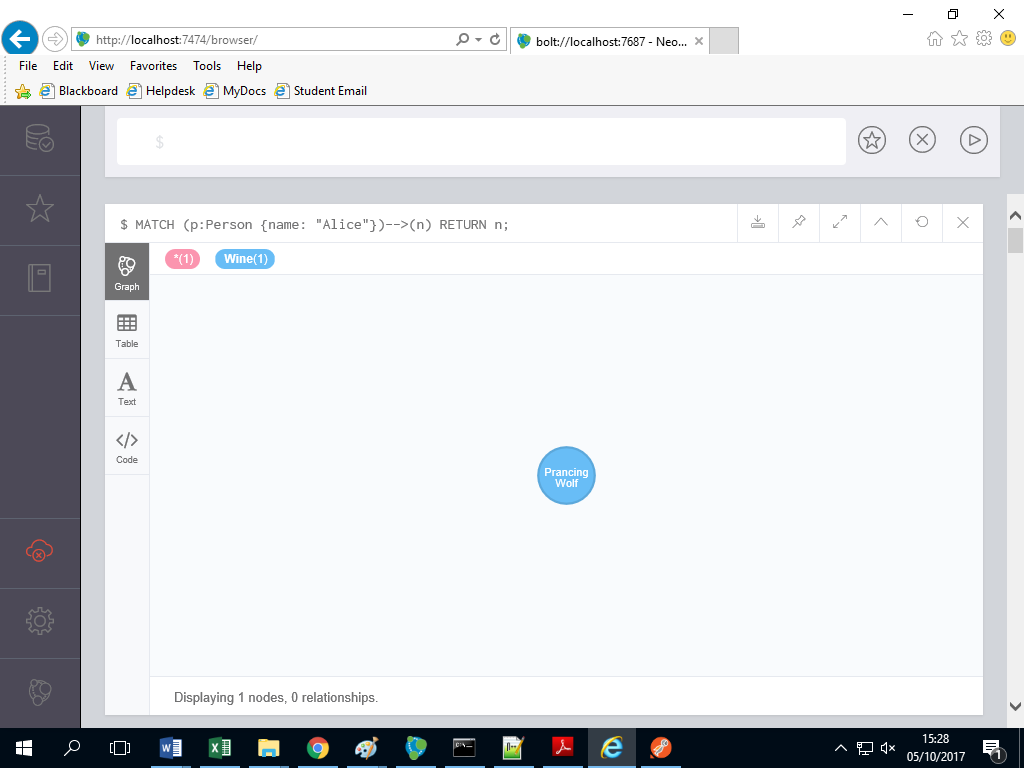
Stepping Stones

Thus far we’ve mostly been performing simple, almost CRUD-like operations using Cypher. You can do *a lot* with these simple commands, but let’s dive in and see what else Cypher has to offer. First, let’s explore Cypher’s syntax for querying all relationships that a node has with a specific type of node.

The --> operator let’s us do that (or --). First, let’s see all nodes associated with Alice:

MATCH (p:Person {name: "Alice"})-->(n)

RETURN n;



Now let’s see all of the people that Alice is friends with, except let’s return only the name property of those nodes:

MATCH (p:Person {name: "Alice"})--(other:Person)

RETURN other.name;

That should result in 1 returned values: Patty. Now let’s say that we

want to see which nodes with the label Person are in the graph, but excluding

Patty (boo, Patty!). Note the <> operator, which is used instead of != in Cypher:

MATCH (p:Person)

WHERE p.name <> 'Patty'

RETURN p;

Thus far, all of our queries have sought out nodes adjacent to one another.

But we also said at the beginning of the chapter that Neo4j is an extremely

scalable database capable of storing tons of nodes and relationships.

Cypher is absolutely up to the task of dealing with far more complex relationships

that the ones we’ve seen thus far. Let’s add some nodes that aren’t directly

related to Patty (for Alice’s friend Ahmed and Tom’s friend Kofi) and then

query for a relationship.

CREATE (p1:Person {name: "Ahmed"}), (p2:Person {name: "Kofi"});

MATCH (p1:Person {name: "Ahmed"}),(p2:Person {name: "Alice"})

CREATE (p1)-[r:friends]->(p2);

MATCH (p1:Person {name: "Kofi"}),(p2:Person {name: "Tom"})

CREATE (p1)-[r:friends]->(p2);

Friends of friends of Alice?

MATCH (fof:Person)-[:friends]-(f:Person)-[:friends]-(p:Person {name: "Alice"})

RETURN fof.name;

As expected, this returns one values: Tom.

 REST, Indexes, and Algorithms

We’ll create nodes and relationships using REST and then use REST to index and execute a full-text search.

 Taking a REST

Just like Riak, HBase, Mongo, and CouchDB, Neo4j ships with a REST interface.

One of the reasons all of these databases support REST is because it allows language-agnostic interactions in a standard connection interface.

We can connect to Neo4j—which requires Java to work—from a separate machine with no trace of Java whatsoever

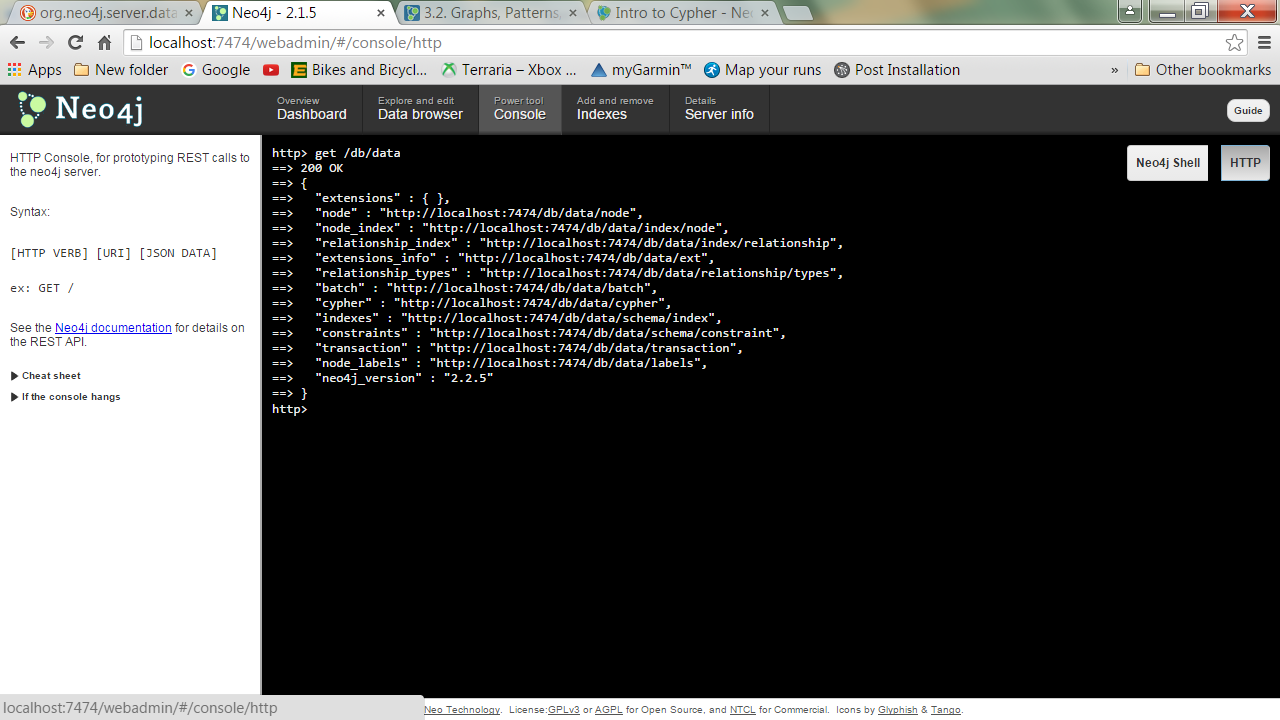
We’ll see how to gain the power of its terse query syntax over REST.

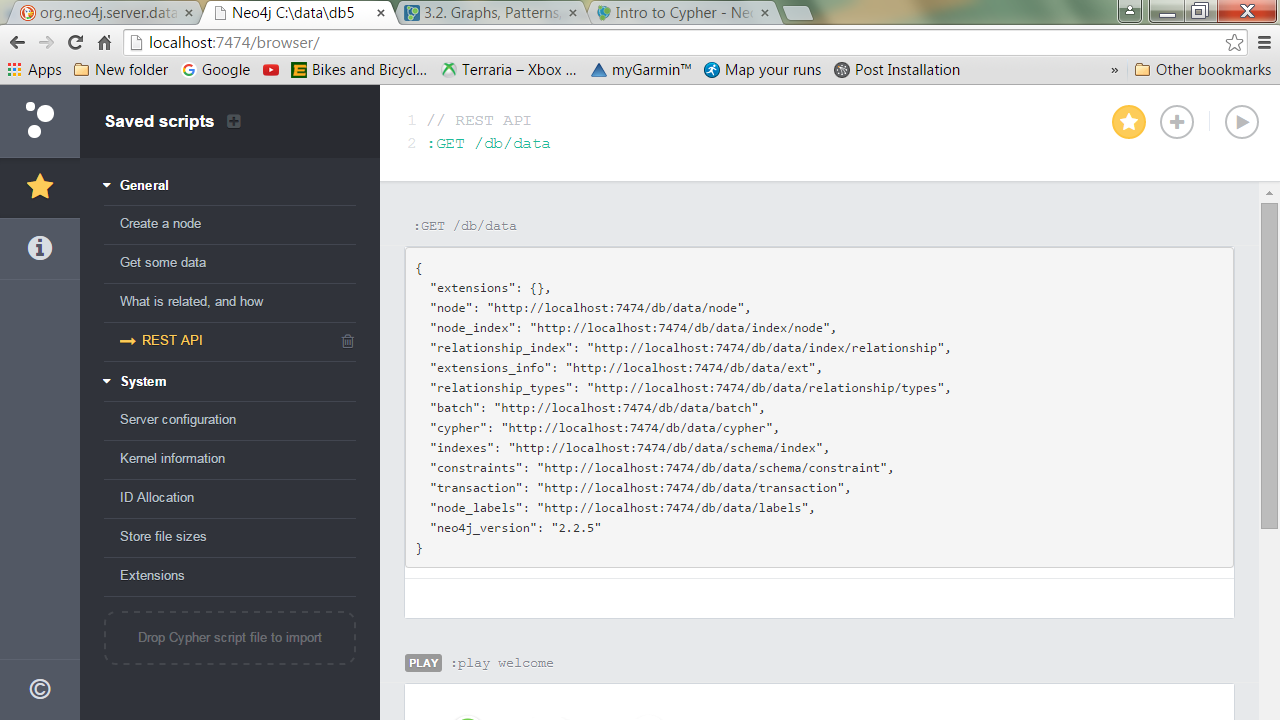
First you might want to check that the REST server is running by issuing a GET against the base URL, which retrieves the root node.

You can do this through the classic Neo4j interface. In the console tab you will see a HTTP tab

Click it and enter

GET /db/data



| 

The REST server It runs on the same port as the web admin tool , at the /db/data/ path.

We’ll can also use our trusty friend curl to issue the REST commands

You need to start up the Neo4J Server this time. We will create a new graph so create a folder called c:\data\new to use and point the Server at this folder

**$ curl** [**http://localhost:7474/db/data/**](http://localhost:7474/db/data/)

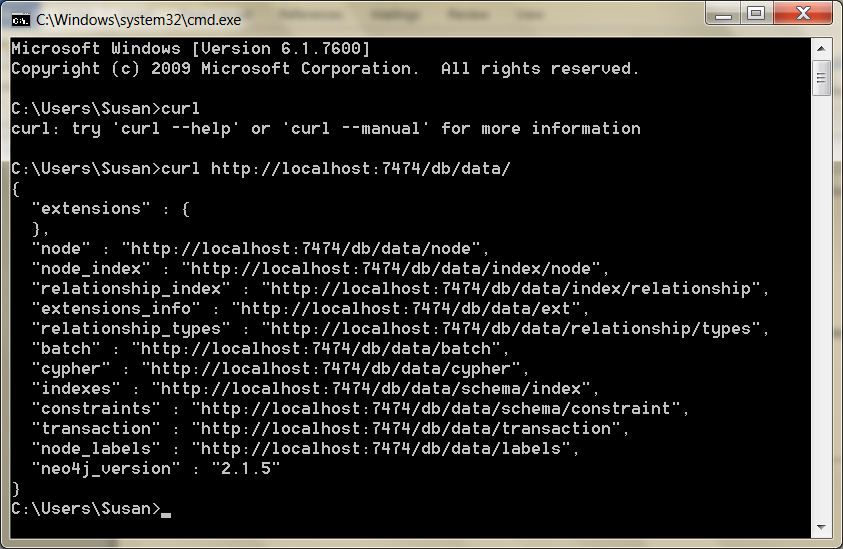
{

"relationship\_index" : "<http://localhost:7474/db/data/index/relationship>",  
 "node" : "<http://localhost:7474/db/data/node>",  
 "relationship\_types" : "<http://localhost:7474/db/data/relationship/types>",  
 "extensions\_info" : "<http://localhost:7474/db/data/ext>",

"node\_index" : "<http://localhost:7474/db/data/index/node>",  
 "extensions" : {  
 }

}

It will return a nice JSON object describing the URLs of other commands, like node actions or indices.



Creating Nodes and Relationships Using REST

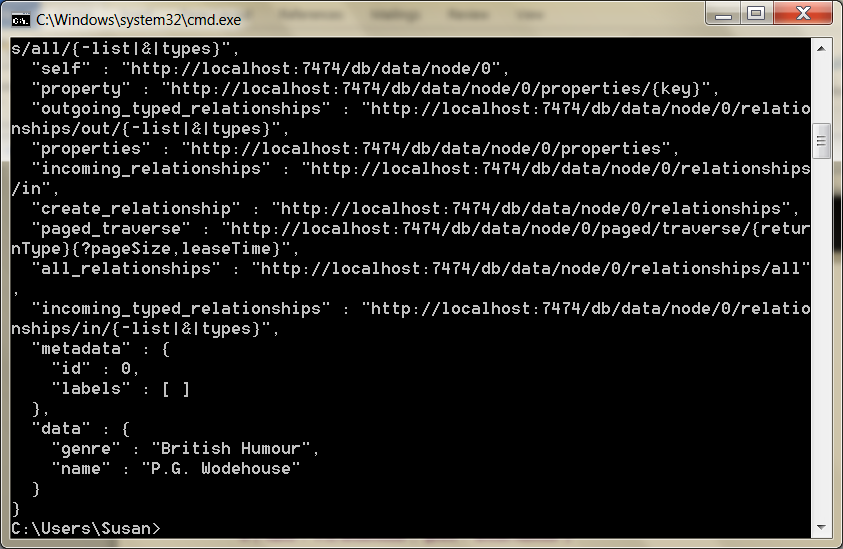
It’s as easy to create nodes and relationships in Neo4j REST as in CouchDB or Riak.

Creating a node is a POST to the /db/data/node path with JSON data.

As matter of convention, it pays to give each node a name property.

This makes viewing any node’s information easy: just call name.

**curl -i -X POST** [**http://localhost:7474/db/data/node**](http://localhost:7474/db/data/node) **^  
 -H "Content-Type: application/json" ^  
 -d "{\"name\":\"P G Wodehouse\", \"genre\": \"British Humour\"}"**



When posted, you’ll get the node path in the header and a body of metadata about the node (both are truncated here for brevity).

All of this data is retrievable by calling GET on the given header Location value (or the self property in the metadata).

HTTP/1.1 201 Created  
Location: http://localhost:7474/db/data/node/0  
Content-Type: application/json

{  
 "outgoing\_relationships" :

"<http://localhost:7474/db/data/node/0/relationships/out>",  
 "data" : {

"genre" : "British Humour",

"name" : "P.G. Wodehouse"  
 },

"traverse" : "[http://localhost:7474/db/data/node/0/traverse/{returnType}](http://localhost:7474/db/data/node/0/traverse/%7breturnType%7d)",  
 "all\_typed\_relationships" :

"[http://localhost:7474/db/data/node/0/relationships/all/{-list|&|types}](http://localhost:7474/db/data/node/0/relationships/all/%7b-list|&|types%7d)",  
 "property" : "[http://localhost:7474/db/data/node/0/properties/{key}](http://localhost:7474/db/data/node/0/properties/%7bkey%7d)",  
 "self" : "http://localhost:7474/db/data/node/0",  
 "properties" : "<http://localhost:7474/db/data/node/0/properties>",  
 "outgoing\_typed\_relationships" :

"[http://localhost:7474/db/data/node/0/relationships/out/{-list|&|types}](http://localhost:7474/db/data/node/0/relationships/out/%7b-list|&|types%7d)",  
 "incoming\_relationships" :

"<http://localhost:7474/db/data/node/0/relationships/in>",  
 "extensions" : {  
 },  
 "create\_relationship" : "<http://localhost:7474/db/data/node/0/relationships>",  
 "paged\_traverse" :

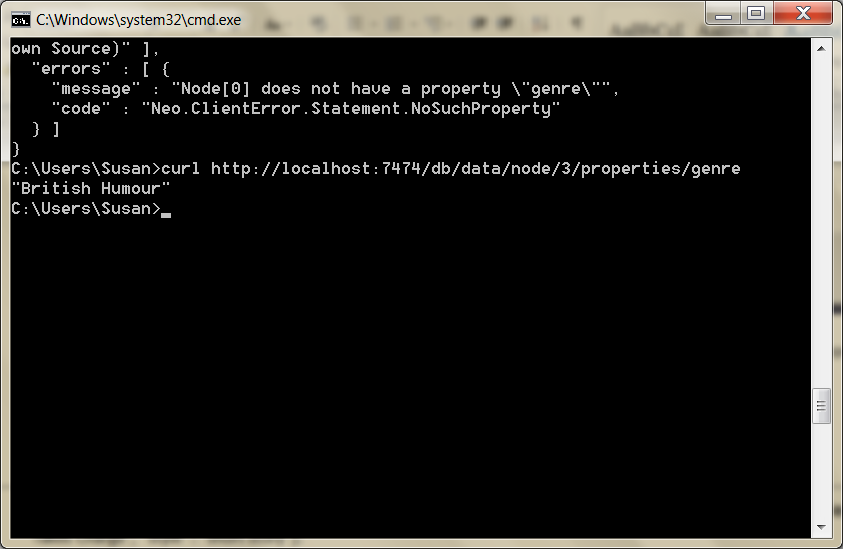
"[http://localhost:7474/db/.../{returnType}{?pageSize,leaseTime}](http://localhost:7474/db/.../%7breturnType%7d%7b?pageSize,leaseTime%7d)",  
 "all\_relationships" : "<http://localhost:7474/db/data/node/0/relationships/all>",  
 "incoming\_typed\_relationships" :

"[http://localhost:7474/db/data/node/0/relationships/in/{-list|&|types}](http://localhost:7474/db/data/node/0/relationships/in/%7b-list|&|types%7d)"  
}

If you just want the node properties (not the metadata), you can GET that by appending /properties to the node URL or even an individual property by further appending the property name.

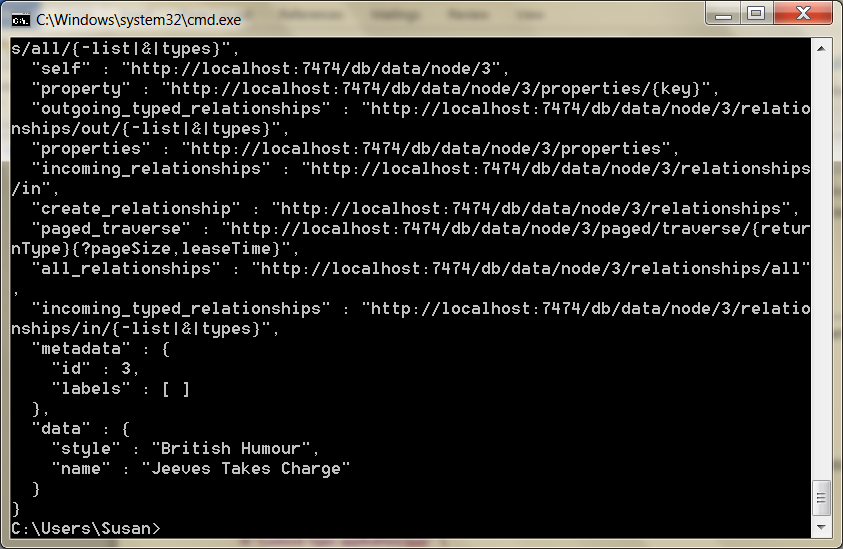
**$ curl** [**http://localhost:7474/db/data/node/0/properties/genre**](http://localhost:7474/db/data/node/0/properties/genre)

"British Humour"



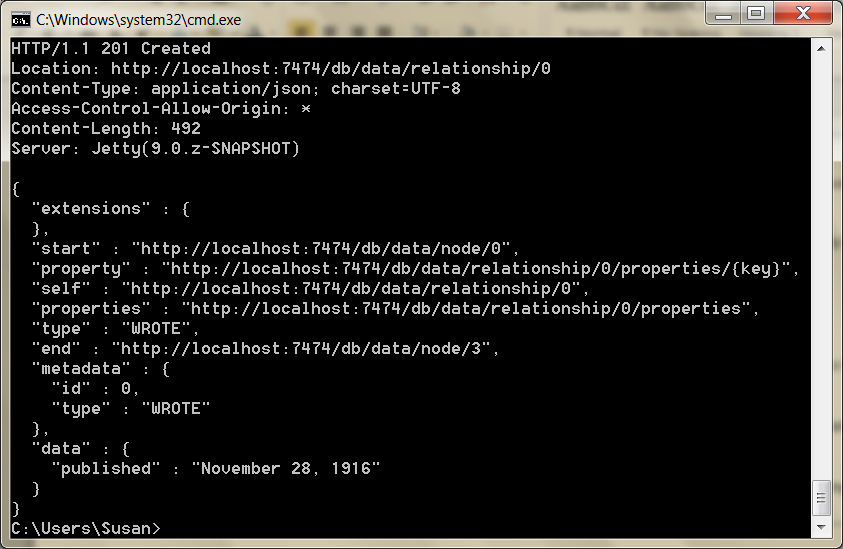
One node doesn’t do us much good, so go ahead and create another one with the properties ["name" : "Jeeves Takes Charge", "style" : "short story"].

**curl -i -X POST** [**http://localhost:7474/db/data/node**](http://localhost:7474/db/data/node) **^  
 -H "Content-Type: application/json" ^  
 -d "{\"name\":\"Jeeves Takes Charge\", \"style\": \"British Humour\"}"**



Since P.G. Wodehouse wrote the short story “Jeeves Takes Charge,” we can make a relationship between them.

**curl -i -X POST** [**http://localhost:7474/db/data/node/0/relationships**](http://localhost:7474/db/data/node/0/relationships) **^  
-H "Content-Type: application/json" ^  
-d "{\"to\":\"http://localhost:7474/db/data/node/1\",\"type\":\"WROTE\",**\"data\":{\"published\":\"November 28, 1916\"}}**"**



A nice thing about the REST interface is that it actually reported on how to create a relationship early in the body metadata’s create\_relationship property.

In this way, the REST interfaces tend to be mutually discoverable.

Finding Your Path

Through the REST interface, you can find the path between two nodes by posting the request data to the starting node’s /paths URL.

The POST request data must be a JSON string denoting the node you want the path to, the type of relationships you want to follow, and the path-finding algorithm to use.

For example, here we’re looking for a path following relationships of the type WROTE from node 9 using the shortestPath algorithm and capping out at a depth of 10.

**curl -X POST** [**http://localhost:7474/db/data/node/9/paths**](http://localhost:7474/db/data/node/9/paths) **^  
-H "Content-Type: application/json" ^  
-d "{\"to\":\"http://localhost:7474/db/data/node/10\",**\"relationships\":{\"type\":\"WROTE\"},\"algorithm\":\"shortestPath\",\"max\_dept\":10}**"**

[{

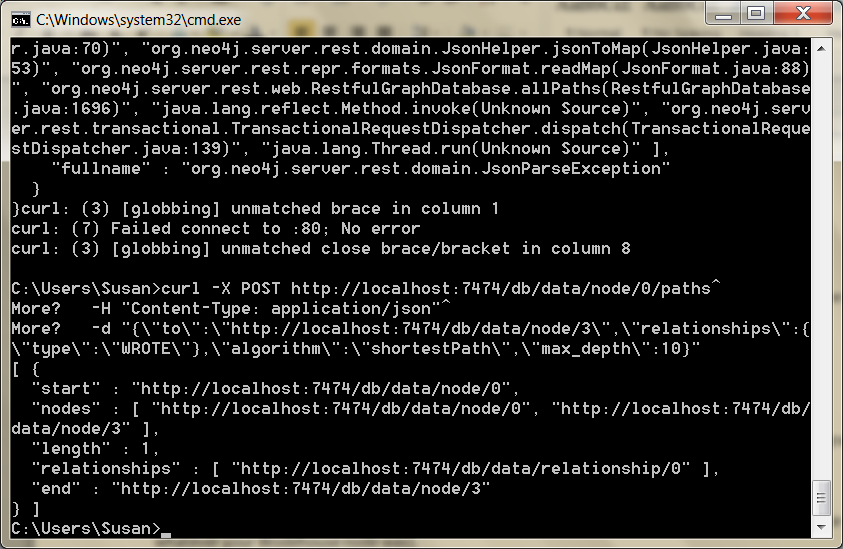
"start" : "<http://localhost:7474/db/data/node/9>",  
 "nodes" : [

"<http://localhost:7474/db/data/node/9>",

"<http://localhost:7474/db/data/node/10>"  
 ],

"length" : 1,  
 "relationships" : [ "<http://localhost:7474/db/data/relationship/14>" ],  
 "end" : "<http://localhost:7474/db/data/node/10>"

}]



The other path algorithm choices are allPaths, allSimplePaths, and dijkstra.

Indexing

Like other databases we’ve seen, Neo4j supports fast data lookups by constructing indexes.

There is a twist, though. Unlike other database indexes where you perform queries in much the same way as without one, Neo4j indexes have a different path.

This is because the indexing service is actually a separate service.

The simplest index is the key-value or hash style.

You key the index by some node data, and the value is a REST URL, which points to the node in the graph.

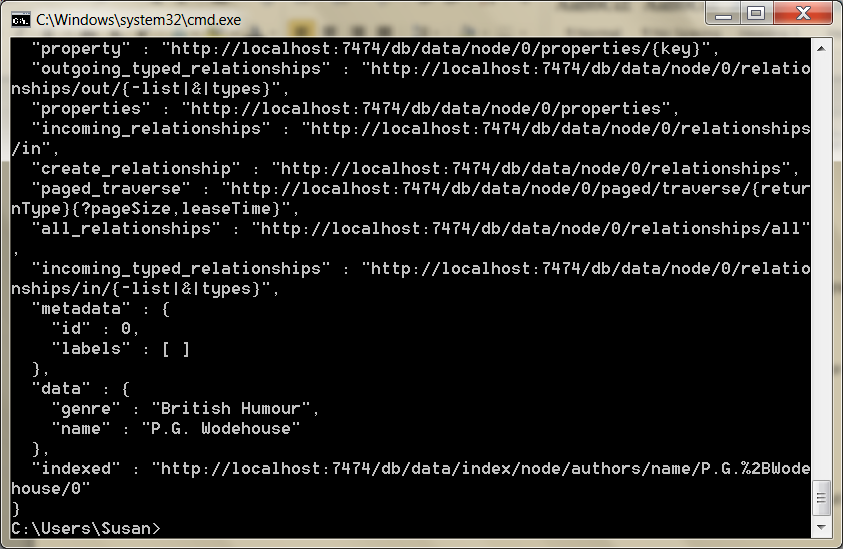
You can have as many indexes as you like, so we’ll name this one “authors.”

The end of the URL will contain the author name we want to index and pass in node 0 as the value (or whatever your Wodehouse node was).

**curl -X POST** [**http://localhost:7474/db/data/index/node/authors ^**](http://localhost:7474/db/data/index/node/authors%20%5e)

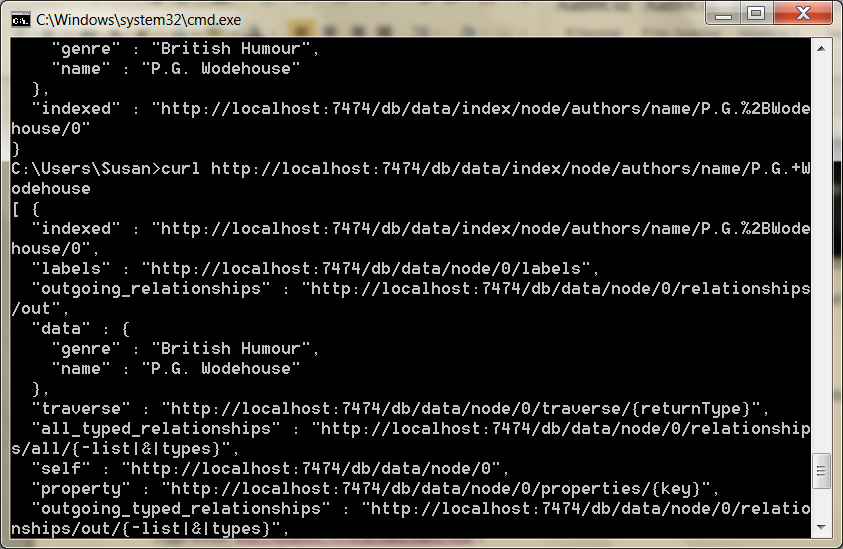
**-H "Content-Type: application/json" ^**

**-d "{\"uri\":\"http://localhost:7474/db/data/node/0\",\**"key\":\"name\", \"value\":\"P.G.+Wodehouse\"}**"**



Retrieving the node is simply a call to the index, which you’ll notice doesn’t return the URL we set but instead the actual node data.

**curl** [**http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse**](http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse)

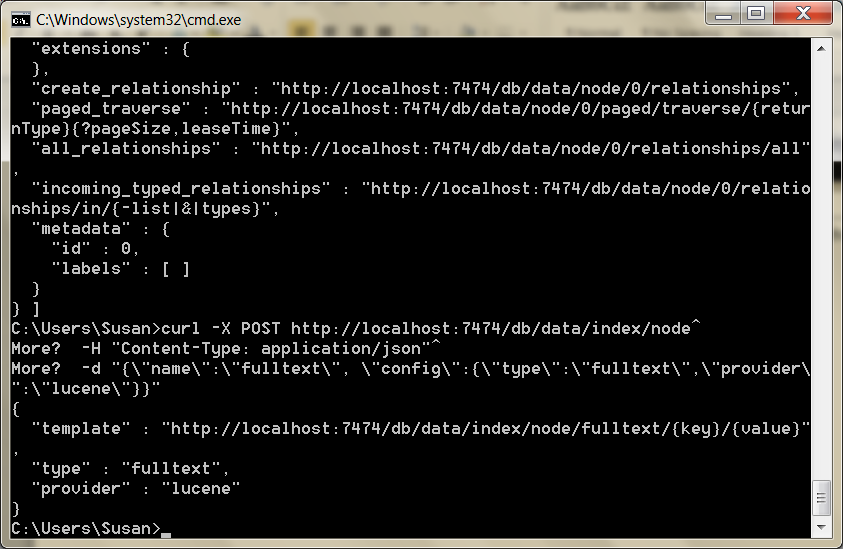


Besides key-value, Neo4j provides a full-text search inverted index, so you can perform queries like this: “Give me all books that have names beginning with ’Jeeves.’”

To build this index, we need to build it against the entire dataset, rather than our one-offs earlier.

Like Riak, Neo4j incorporates Lucene to build our inverted index (http://en.wikipedia.org/wiki/Lucene)

**curl -X POST** [**http://localhost:7474/db/data/index/node**](http://localhost:7474/db/data/index/node) **^  
-H "Content-Type: application/json" ^  
-d "{\"name\":\"fulltext\", \"config\":{\"type\":\"fulltext\",\"provider\":\"lucene\"}}"**



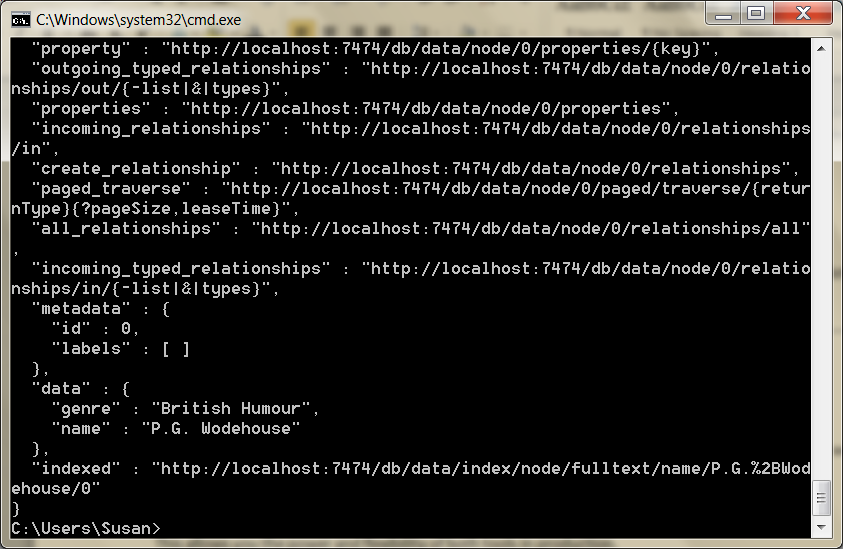
The POST will return a JSON response containing information about the newly added index.

{  
 "template" : "[http://localhost:7474/db/data/index/node/fulltext/{key}/{value}](http://localhost:7474/db/data/index/node/fulltext/%7bkey%7d/%7bvalue%7d)",  
 "provider" : "lucene",  
 "type" : "fulltext"

}

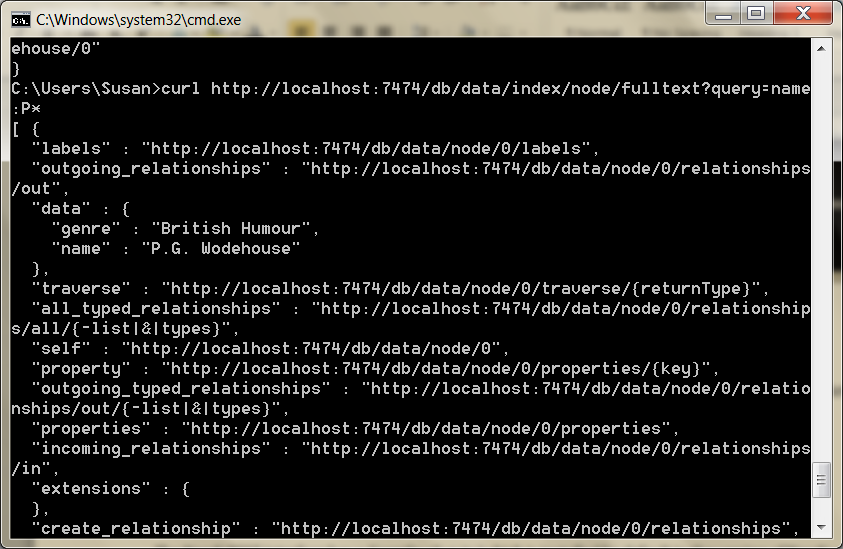
Now if we add Wodehouse to the full-text index, we get this:

curl -X POST <http://localhost:7474/db/data/index/node/fulltext> ^  
-H "Content-Type: application/json" ^  
-d "{\"uri\":\"http://localhost:7474/db/data/node/0\",\"key\":\"name\",\"value\":\"P.G.+Wodehouse\"}"



Then a search is as easy as a Lucene syntax query on the index URL.

**curl** [**http://localhost:7474/db/data/index/node/fulltext?query=name:P\***](http://localhost:7474/db/data/index/node/fulltext?query=name:P*)



Indexes can also be built on relationships like earlier; just replace the instances of *node* in the URLs with *relationship*, for example <http://localhost:7474/db/data/index/relationship/published/date/1916-11-28>.

Distributed High Availability

We’re going to wrap up our Neo4j investigation by learning how to make Neo4j more attuned to mission-critical uses.

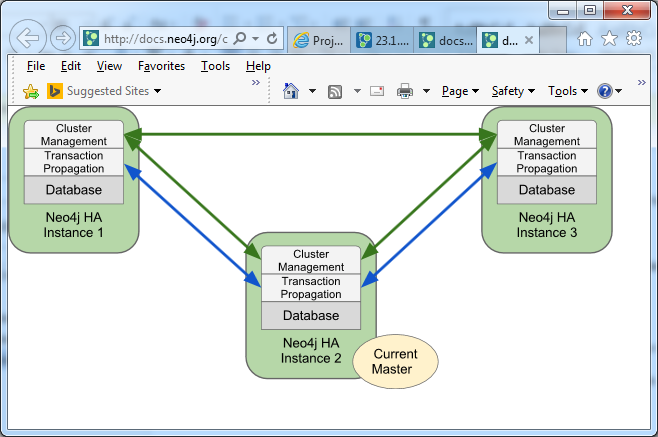
We’ll see how Neo4j keeps data stable via ACID-compliant transactions.

Then we’ll install and configure a Neo4j high availability (HA) cluster to improve availability when serving high-read traffic.

Then we’re going to look into backup strategies to ensure our data remains safe.

High Availability

High availability mode is Neo4j’s answer to the question, “Can a graph database scale?” Yes, but with some caveats.



When running Neo4j in HA mode there is always a single master and zero or more slaves. Compared to other master-slave replication setups Neo4j HA can handle write requests on all machines so there is no need to redirect those to the master specifically.

A write to one slave is not immediately synchronized with all other slaves, so there is a danger of losing consistency (in the CAP sense) for a brief moment (making it eventually consistent).

HA will lose ACID-compliant transactions.

It’s for this reason that Neo4j HA is touted as a solution largely for increasing capacity for reads.

Just like Mongo, the servers in the cluster will elect a master that is the gold copy of data.

Unlike Mongo, however, slaves accept writes.

Slave writes will synchronize with the master node, which then propagates those changes to the other slaves.

HA Cluster

To use Neo4j HA, we must first set up a cluster.

You can download Neo4j Enterprise from the website for your operating system (be sure you select the correct edition )and then unzip it and create two more copies of the directory. You can find the executable in this weeks folder.

Normally you would unpack one copy per server and configure the cluster to be aware of the other servers.

But since we’re running them locally, we’ll instead run them on different directories using different ports.

Each node contains a

conf/neo4j.conf configuration file. At the top of that file in the folder for node 1, neo4j-1.local, add this:

dbms.mode=HA

dbms.memory.pagecache.size=200m

dbms.backup.address=127.0.0.1:6366

dbms.backup.enabled=true

ha.server\_id=1

ha.initial\_hosts=127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003

ha.host.coordination=127.0.0.1:5001

ha.host.data=127.0.0.1:6363

dbms.connector.http.enabled=true

dbms.connector.http.listen\_address=:7474

dbms.connector.bolt.enabled=true

dbms.connector.bolt.tls\_level=OPTIONAL

dbms.connector.bolt.listen\_address=:7687

dbms.security.auth\_enabled=false

Copy and paste the same thing into the other two nodes’ config files, except increase the following values by 1 *for each node* (producing three separate values for each, for example 7474, 7475, and 7476 for dbms.connector.http.listen\_ address):

Backup.address

• ha.server\_id

• ha.host.coordination

• ha.host.data

• dbms.connector.http.listen\_address  
dbms.connector.bolt.listen\_address

# HTTPS Connector. There can be zero or one HTTPS connectors.

dbms.connector.https.enabled=true

dbms.connector.https.listen\_address=:7471, 7472, 7473

In config file must also change https connector

dbms.connector.https.listen\_address=:7471

dbms.connector.https.listen\_address=:7472

dbms.connector.https.listen\_address=:7473

# HTTPS Connector. There can be zero or one HTTPS connectors.

dbms.connector.https.enabled=true

dbms.connector.https.listen\_address=:7471, 7472, 7473

So, ha.server\_id should have values of 1, 2, and 3 on the different nodes,

respectively, and so on for the other configs. This is to ensure that the nodes

aren’t attempting to open up the same ports for the same operations. Now

we can start each node one by one (the order doesn’t matter):

Latest details here: https://neo4j.com/docs/operations-manual/current/installation/windows/

Put path to powershell in PATH

C:\Windows\System32

C:\Users\lt00036478\AppData\Local\Microsoft\WindowsApps

C:\Windows\system32\WindowsPowerShell\v1.0

Install the service with bin\neo4j install-serviceand start it with bin\neo4j start

Start the Neo4j servers as normal. Note the startup order does not matter.

**$ neo4j-1.local/bin/neo4j start**

**$ neo4j-2.local/bin/neo4j start**

**$ neo4j-3.local/bin/neo4j start**

Then

neo4j console

in each window

Now, you should be able to access the 3 servers and check their HA status:

<http://127.0.0.1:7474/webadmin/#/info/org.neo4j/High%20Availability/>

<http://127.0.0.1:7475/webadmin/#/info/org.neo4j/High%20Availability/>

<http://127.0.0.1:7476/webadmin/#/info/org.neo4j/High%20Availability/>

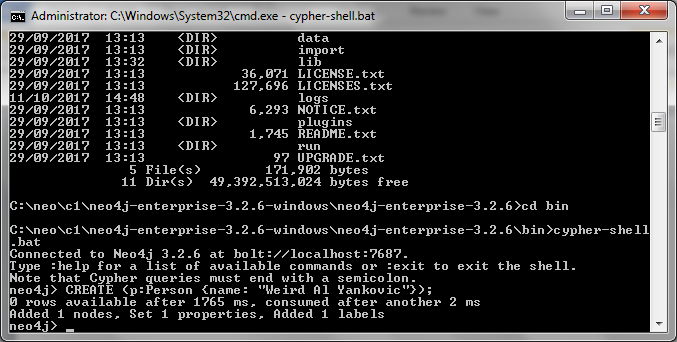
Jump into the Cypher shell for the first node:

**$ neo4j-1.local/bin/cypher-shell**

Now let’s write a new data node to our cluster and exit the Cypher shell for

node 1...

CREATE (p:Person {name: "Weird Al Yankovic"});



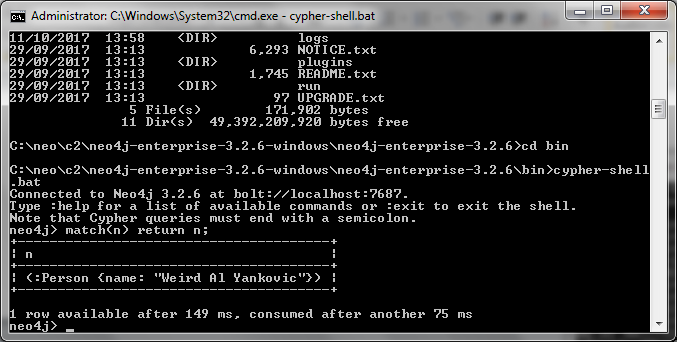
:exit

...and then open up the shell for node 2...

**$ neo4j-2.local/bin/cypher-shell**

...and finally see which data nodes are stored in the cluster:

MATCH (n) RETURN n;



 And there we have it: our data has been successfully replicated across nodes.

You can try the same thing on node 3 if you’d like.

Master Election

In HA Neo4j clusters, master election happens automatically. If the master server goes offline, other servers will notice and elect a leader from among themselves. Starting the previous master server again will add it back to the cluster, but now the old master will remain a slave (until another server goes

down).

High availability allows very read-heavy systems to deal with replicating a graph across multiple servers and thus sharing the load. Although the cluster as a whole is only eventually consistent, there are tricks you can apply to reduce the chance of reading stale data in your own applications, such as

assigning a session to one server. With the right tools, planning, and a good setup, you can build a graph database large enough to handle billions of nodes and edges and nearly any number of requests you may need. Just add regular backups, and you have the recipe for a solid production system.

Backups

Backups are a necessary aspect of any professional database use. Although backups are effectively built in when using replication in a highly available cluster, periodic backups—nightly, weekly, hourly, etc.—that are stored offsite are always a good idea for disaster recovery.

It’s hard to plan for a server room fire or an earthquake shaking a building to rubble. Neo4j Enterprise offers a tool called neo4j-admin that performs a wide variety of actions, including backups

The most powerful method when running an HA server is to craft a full backup command to copy the database file from the cluster to a date-stamped file on a mounted drive. Pointing the copy to every server in the cluster will ensure you get the most recent data available. The backup directory created

is a fully usable copy. If you need to recover, just replace each installation’s data directory with the backup directory, and you’re ready to go.

You must start with a full backup. Here we back up our HA cluster to a directory that ends with today’s date (using the \*nix date command). The neo4j-admin command can be run from any server and you can choose any server in the cluster when using the --from flag. Here’s an example command:

**$ neo4j-1.local/bin/neo4j-admin backup \**

**--from 127.0.0.1:6366 \**

**--name neo4j-`date +%Y.%m.**

**--backup-dir /mnt/backups**

Once you have done a full backup, you can choose to do an incremental backup by specifying an existing .db database directory as the target directory.

But keep in mind that incremental backups only work on a fully backed-up directory, so ensure the previous command is run on the same day or else the directory names won’t match up.

Wrap-Up

Neo4j is a top open source implementation of the (relatively rare) class of graph databases.

Graph databases focus on the relationships between data, rather than the commonalities among values.

Modeling graph data is simple.

You just create nodes and relationships between them and optionally hang key-value pairs from them.

Querying is as easy as declaring how to walk the graph from a starting node using Cypher.

Neo4j’s Strengths

Neo4j is one of the finest examples of open source graph databases.

Graph databases are perfect for unstructured data, in many ways even more so than document datastores.

Not only is Neo4j typeless and schemaless, but it puts no constraints on how data is related.

It is, in the best sense, a free-for-all. Currently, Neo4j can support 34.4 billion nodes and 34.4 billion relationships, which is more than enough for most uses (Neo4j could hold more than 42 nodes for each of Facebook’s 800 million users in a single graph).

The Neo4j distributions provide several tools for fast lookups with Lucene and easy-to-use (if sometimes cryptic) language extensions like Cypher (and Gremlin) and the REST interface.

Beyond ease of use, Neo4j is fast. Unlike join operations in relational databases or map-reduce operations in other databases, graph traversals are constant time.

Like data is only a node step away, rather than joining values in bulk and filtering the desired results—as most of the databases we’ve seen operate.

It doesn’t matter how large the graph becomes; moving from node A to node B is always one step if they share a relationship. Finally, the Enterprise edition provides for highly available and high read- traffic sites by way of Neo4j HA.

Neo4j’s Weaknesses

Neo4j does have a few shortcomings.

Relationships (Edges) in Neo4j cannot direct a vertex back on itself.

We also found its choice of nomenclature (*node* rather than *vertex*, and *relationship* rather than *edge*) to add complexity when communicating

Although HA is excellent at replication, it can only replicate a full graph to other servers.

It cannot currently shard subgraphs, which still places a limit on graph size (though, to be fair, that limit measures in the tens of billions).

Finally, if you are looking for a business-friendly open source license, Neo4j may not be for you.

Where the Community edition is GPL, if you want to run a production environment using the Enterprise tools (which includes HA and backups), you’ll probably need to purchase a license.

Neo4j on CAP

If you choose to distribute, the name “high availability” cluster should give away their strategy.

Neo4j HA is available and partition tolerant (AP).

Each slave will return only what it currently has, which may be out of sync with the master node temporarily.

Although you can reduce the update latency by increasing a slave’s pull interval, it’s still technically **eventually** consistent.

This is why Neo4j HA is recommended for read-mostly requirements.

Parting Thoughts

Neo4j’s simplicity/complexity can be off-putting if you’re not used to modeling graph data.

It provides a powerful open source API with years of production use and yet still has relatively few users.

We chalk this up to lack of knowledge, since graph databases mesh so naturally with how humans tend to conceptualize data.

We imagine our families as trees, or our friends as graphs; most of us don’t imagine personal relationships as self-referential datatypes.

For certain classes of problems, like social networks, Neo4j is an obvious choice.

But you should give it some serious consideration for nonobvious problems as well— it just may surprise you how powerful and easy it is.

Reference: Seven databases in seven weeks