Neo4j

Downloading, Installing, and Running Neo4j - Supplementary Resources

Download Neo4j at:

http://www.neo4j.com/

Check your system requirements at:

http://neo4j.com/docs/stable/deployment-requirements.html

Useful Links

- Ask questions at Stack Overflow
- Discuss Neo4j on Google Groups
- Visit a local Meetup Group
- Contribute code on Github

Getting Started With Neo4j

Pseudocode to create our 'Toy' Network

==============

Five Nodes

N1 = Tom

N2 = Harry

N3 = Julian

N4 = Michele

N5 = Josephine

Five Edges

e1 = Harry 'is known by' Tom

e2 = Julian 'is co-worker of' Harry

e3 = Michele 'is wife of' Harry

e4 = Josephine 'is wife of' Tom

e5 = Josephine 'is friend of' Michele

==============

A simple text description of a graph

N1 - e1 -> N2

N2 - e2 -> N3

2 - e3 -> N4

N1 - e4 -> N5

N4 - e5 -> N5

==============

A more technical text description of a graph

N1:ToyNode - e1 -> N2:ToyNode

N2 - e2 -> N3:ToyNode

N2 - e3 -> N4:ToyNode

N1 - e4 -> N5:ToyNode

N4 - e5 -> N5

==============

Even more technical pseudo-code

N1:ToyNode - ToyRelation -> N2:ToyNode

N2 - ToyRelation -> N3:ToyNode

N2 - ToyRelation -> N4:ToyNode

N1 - ToyRelation -> N5:ToyNode

N4 - ToyRelation -> N5

==============

Pseudo-code approximating CYPHER code

N1:ToyNode {name: 'Tom'} - ToyRelation {relationship: 'knows'} -> N2:ToyNode {name: 'Harry'}

N2 - ToyRelation {relationship: 'co-worker'} -> N3:ToyNode {name: 'Julian', job: 'plumber'} N2 - ToyRelation {relationship: 'wife'}-> N4:ToyNode {name: 'Michele', job: 'accountant'}

N1 - ToyRelation {relationship: 'wife'} -> N5:ToyNode {name: 'Josephine', job: 'manager'}

N4 - ToyRelation {relationship: 'friend'} -> N5

==============

The actual CYPHER code to create our 'Toy' network

create (N1:ToyNode {name: 'Tom'}) - [:ToyRelation {relationship: 'knows'}] -> (N2:ToyNode {name: 'Harry'}),

(N2) - [:ToyRelation {relationship: 'co-worker'}] -> (N3:ToyNode {name: 'Julian', job: 'plumber'}), (N2) - [:ToyRelation {relationship: 'wife'}] -> (N4:ToyNode {name: 'Michele', job: 'accountant'}),

(N1) - [:ToyRelation {relationship: 'wife'}] -> (N5:ToyNode {name: 'Josephine', job: 'manager'}),

(N4) - [:ToyRelation {relationship: 'friend'}] -> (N5)

;

More code examples

==============

View the resulting graph

match (n:ToyNode)-[r]-(m) return n, r, m

==============

Delete all nodes and edges

match (n)-[r]-() delete n, r

==============

Delete all nodes which have no edges

match (n) delete n

==============

Delete only ToyNode nodes which have no edges

match (n:ToyNode) delete n

==============

Delete all edges

match (n)-[r]-() delete r

==============

Delete only ToyRelation edges

match (n)-[r:ToyRelation]-() delete r

//Selecting an existing single ToyNode node

match (n:ToyNode {name:'Julian'}) return n

Adding to and Modifying a Graph

//Adding a Node Correctly

match (n:ToyNode {name:'Julian'})

merge (n)-[:ToyRelation {relationship: 'fiancee'}]->(m:ToyNode {name:'Joyce', job:'store clerk'})

//Adding a Node Incorrectly

create (n:ToyNode {name:'Julian'})-[:ToyRelation {relationship: 'fiancee'}]->(m:ToyNode {name:'Joyce', job:'store clerk'})

//Correct your mistake by deleting the bad nodes and edge

match (n:ToyNode {name:'Joyce'})-[r]-(m) delete n, r, m

//Modify a Node's Information

match (n:ToyNode) where n.name = 'Harry' set n.job = 'drummer'

match (n:ToyNode) where n.name = 'Harry' set n.job = n.job + ['lead guitarist']

Cypher Scripts for Importing Data Into Neo4j

//One way to "clean the slate" in Neo4j before importing (run both lines):

match (a)-[r]->() delete a,r

match (a) delete a

//Script to Import Data Set: test.csv (simple road network)

//For Windows use something like the following

//[NOTE: replace any spaces in your path with %20, "percent twenty"]

LOAD CSV WITH HEADERS FROM "file:///C:/coursera/data/test.csv" AS line

MERGE (n:MyNode {Name:line.Source})

MERGE (m:MyNode {Name:line.Target})

MERGE (n) -[:TO {dist:line.distance}]-> (m)

//For mac OSX use something like the following

//[NOTE: replace any spaces in your path with %20, "percent twenty"]

LOAD CSV WITH HEADERS FROM "file:///coursera/data/test.csv" AS line

MERGE (n:MyNode {Name:line.Source})

MERGE (m:MyNode {Name:line.Target})

MERGE (n) -[:TO {dist:line.distance}]-> (m)

//Script to import global terrorist data

LOAD CSV WITH HEADERS FROM "file:///Users/jsale/sdsc/coursera/data/
terrorist_data_subset.csv" AS row

MERGE (c:Country {Name:row.Country})

MERGE (a:Actor {Name: row.ActorName, Aliases: row.Aliases, Type: row.ActorType})

```
MERGE (o:Organization {Name: row.AffiliationTo})

MERGE (a)-[:AFFILIATED_TO {Start: row.AffiliationStartDate, End: row.AffiliationEndDate}]->(o)

MERGE(c)<-[:IS_FROM]-(a);
```

Basic Graph Operations with CYPHER

//Counting the number of nodes

match (n:MyNode)

return count(n)

//Counting the number of edges

match (n:MyNode)-[r]->()

return count(r)

//Finding leaf nodes:

match (n:MyNode)-[r:TO]->(m)

where not ((m)-->())

return m

//Finding root nodes:

match (m)-[r:TO]->(n:MyNode)

where not (()-->(m))

return m

//Finding triangles:

match (a)-[:TO]->(b)-[:TO]->(c)-[:TO]->(a)

return distinct a, b, c

//Finding 2nd neighbors of D:

match (a)-[:TO*..2]-(b)

where a.Name='D'

return distinct a, b

//Finding the types of a node:

match (n)

where n.Name = 'Afghanistan'

return labels(n)

//Finding the label of an edge:

match (n {Name: 'Afghanistan'})<-[r]-()

return distinct type(r)

//Finding all properties of a node:

match (n:Actor)

return * limit 20

//Finding loops:

match (n)-[r]->(n)

return n, r limit 10

//Finding multigraphs:

match (n)-[r1]->(m), (n)-[r2]-(m)

where r1 <> r2

return n, r1, r2, m limit 10

//Finding the induced subgraph given a set of nodes:

match (n)-[r:TO]-(m)

where n.Name in ['A', 'B', 'C', 'D', 'E'] and m.Name in ['A', 'B', 'C', 'D', 'E']

```
return n, r, m
```

Path Analytics with CYPHER

//Viewing the graph

match (n:MyNode)-[r]->(m)

return n, r, m

//Finding paths between specific nodes*:

match p=(a)-[:TO*]-(c)

where a.Name='H' and c.Name='P'

return p limit 1

*Your results might not be the same as the video hands-on demo. If not, try the following query and it should return the shortest path between nodes H and P:

match p=(a)-[:TO*]-(c) where a.Name='H' and c.Name='P' return p order by length(p) asc limit 1

//Finding the length between specific nodes:

match p=(a)-[:TO*]-(c)

where a.Name='H' and c.Name='P'

return length(p) limit 1

//Finding a shortest path between specific nodes:

match p=shortestPath((a)-[:TO*]-(c))

where a.Name='A' and c.Name='P'

return p, length(p) limit 1

//All Shortest Paths:

```
MATCH p = allShortestPaths(source-[r:TO*]-destination)
WHERE source.Name='A' AND destination.Name = 'P'
RETURN EXTRACT(n IN NODES(p)| n.Name) AS Paths
```

//All Shortest Paths with Path Conditions:

```
MATCH p = allShortestPaths(source-[r:TO*]->destination)
WHERE source.Name='A' AND destination.Name = 'P' AND LENGTH(NODES(p)) > 5
RETURN EXTRACT(n IN NODES(p)| n.Name) AS Paths,length(p)
```

//Diameter of the graph:

```
match (n:MyNode), (m:MyNode)
where n <> m
with n, m
match p=shortestPath((n)-[*]->(m))
return n.Name, m.Name, length(p)
order by length(p) desc limit 1
```

//Extracting and computing with node and properties:

```
match p=(a)-[:TO*]-(c)
where a.Name='H' and c.Name='P'
return extract(n in nodes(p)|n.Name) as Nodes, length(p) as pathLength,
reduce(s=0, e in relationships(p)| s + toInt(e.dist)) as pathDist limit 1
```

//Dijkstra's algorithm for a specific target node:

```
MATCH (from: MyNode {Name:'A'}), (to: MyNode {Name:'P'}),
path = shortestPath((from)-[:TO*]->(to))
WITH REDUCE(dist = 0, rel in rels(path) | dist + toInt(rel.dist)) AS distance, path
```

```
RETURN path, distance
```

//Dijkstra's algorithm SSSP:

```
MATCH (from: MyNode {Name:'A'}), (to: MyNode),

path = shortestPath((from)-[:TO*]->(to))

WITH REDUCE(dist = 0, rel in rels(path) | dist + toInt(rel.dist)) AS distance, path, from, to

RETURN from, to, path, distance order by distance desc
```

//Graph not containing a selected node:

```
match (n)-[r:TO]->(m)

where n.Name <> 'D' and m.Name <> 'D'

return n, r, m
```

//Shortest path over a Graph not containing a selected node:

```
match p=shortestPath((a {Name: 'A'})-[:TO*]-(b {Name: 'P'}))

where not('D' in (extract(n in nodes(p)|n.Name)))

return p, length(p)
```

//Graph not containing the immediate neighborhood of a specified node:

```
match (d {Name:'D'})-[:TO]-(b)

with collect(distinct b.Name) as neighbors

match (n)-[r:TO]->(m)

where

not (n.Name in (neighbors+'D'))

and

not (m.Name in (neighbors+'D'))

return n, r, m
```

```
;

match (d {Name:'D'})-[:TO]-(b)-[:TO]->(leaf)

where not((leaf)-->())

return (leaf)

;

match (d {Name:'D'})-[:TO]-(b)<-[:TO]-(root)

where not((root)<--())

return (root)
```

//Graph not containing a selected neighborhood:

```
match (a {Name: 'F'})-[:TO*..2]-(b)

with collect(distinct b.Name) as MyList

match (n)-[r:TO]->(m)

where not(n.Name in MyList) and not (m.Name in MyList)

return distinct n, r, m
```

Connectivity Analytics with CYPHER

//Viewing the graph

match (n:MyNode)-[r]->(m)

return n, r, m

// Find the outdegree of all nodes

match (n:MyNode)-[r]->()

return n.Name as Node, count(r) as Outdegree

order by Outdegree

union

match (a:MyNode)-[r]->(leaf)

where not((leaf)-->())

return leaf.Name as Node, 0 as Outdegree

// Find the indegree of all nodes

match (n:MyNode)<-[r]-()

return n.Name as Node, count(r) as Indegree

order by Indegree

union

match (a:MyNode)<-[r]-(root)

where not((root)<--())

return root.Name as Node, 0 as Indegree

// Find the degree of all nodes

match (n:MyNode)-[r]-()

return n.Name, count(distinct r) as degree

order by degree

// Find degree histogram of the graph

match (n:MyNode)-[r]-()

with n as nodes, count(distinct r) as degree

return degree, count(nodes) order by degree asc

//Save the degree of the node as a new node property

match (n:MyNode)-[r]-()

with n, count(distinct r) as degree

set n.deg = degree

return n.Name, n.deg

// Construct the Adjacency Matrix of the graph

match (n:MyNode), (m:MyNode)

return n.Name, m.Name,

case

when (n)-->(m) then 1

else 0

end as value

// Construct the Normalized Laplacian Matrix of the graph

match (n:MyNode), (m:MyNode)

return n.Name, m.Name,

case

when n.Name = m.Name then 1

```
when (n)-->(m) then -1/(sqrt(toInt(n.deg))*sqrt(toInt(m.deg)))

else 0

end as value
```