Introduction to Graph Databases

Neo4j

Alejandro Vaisman
avaisman@itba.edu.ar

Neo4j – Advanced Querying

Useful Libraries

1. **APOC (Awesome Procedures on Cypher)**

- Contains many different procedures that extend the capabilities of Neo4j.
- Provides features not covered by Cypher
- Exposes functions (returning a single value) and procedures (producing a result stream) related to:
    - Extensions of Cypher with, for instance, dynamic labels or property keys and periodic commits for all operations
    - Graph refactoring (cloning nodes, changing a relationship's starting or ending node, and so on)
    - Managing collections and lists
    - Database introspection (graph schema, types of properties, and so on)
    - Import from/export to files in different formats (JSON, XML, and so on)
- To install, download the right version (this is VERY important, must be the **exact version corresponding to the Neo4j version you are using**) and copy it into the **Plugins** folder.
- In neo4j.conf unmark and write:

    dbms.security.procedures.unrestricted = apoc.*, …

    dbms.security.procedures.allowlist = apoc.*, …

Useful Libraries

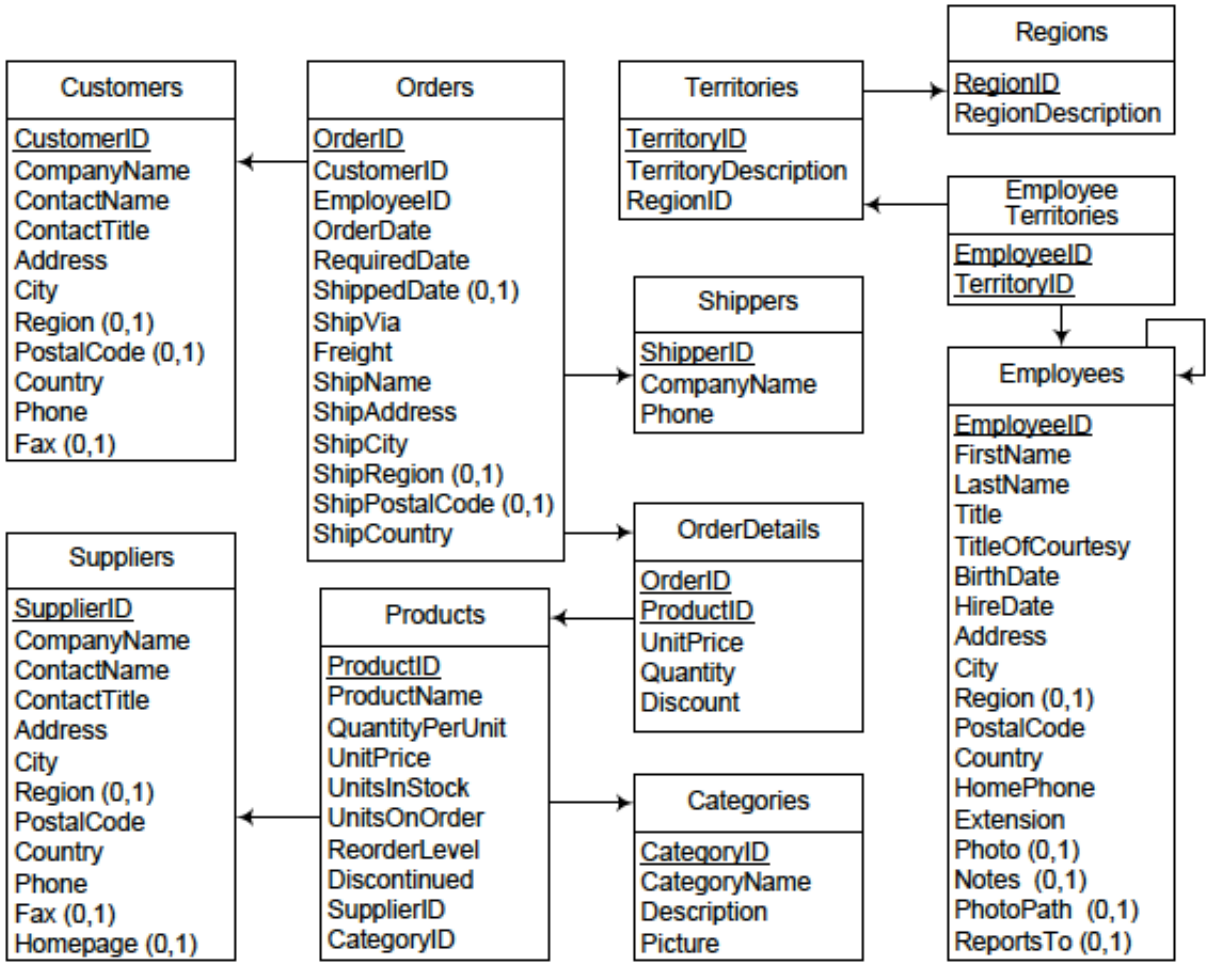
2. Graph Data Science (GDS previously graph-algo)

- Contains tools to be used in a data science project using data stored in Neo4j:
  - Path-related algorithms: Dijkstra, A*, etc.
  - Graph algorithms
    - Centrality
    - Community detection
    - Similarity
  - Machine learning (ML) models and pipelines
  - Python client: allows GDS to be called from Python, without using Cypher
- To install, download the right version and copy it into the Plugins folder.
- In neo4j.conf:

  dbms.security.procedures.unrestricted = apoc.*, gds.*, n10s.*,…..

  dbms.security.procedures.allowlist = apoc.coll.*, apoc.load.*,gds.*, apoc.*, n10s.*,….

Loading the Northwind (graph) database



Bulk-loading a Neo4j graph

1. Using the LOAD CVS statement

Note: The .CSV file **must be in the import folder** in Neo4j

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///territories.csv" AS row
CREATE (:Territory {territoryID: row.territoryid, name: row.territorydescription});
============
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///employees.csv" AS row
CREATE (:Employee{employeeID: row.employeeid,
        lastName: row.lastname,firstName: row.firstname, city:row.city,region:row.region,country:row.country});
==============

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///employeeterritories.csv" AS row
MATCH (t:Territory {territoryID: row.territoryid})
MATCH (e:Employee {employeeID: row.employeeid})
MERGE (e)-[:AssignedTo]->(t)
```

Bulk-loading a Neo4j graph

2. Using the LOAD CVS statement without USING PERIODIC COMMIT

Note: The .CSV file **must be in the import folder** in Neo4j

```
LOAD CSV WITH HEADERS FROM "file:///territories.csv" AS row
CREATE (:Territory {territoryID: row.territoryid, name: row.territorydescription});
```

```
:auto LOAD CSV WITH HEADERS FROM 'file:///territories.csv' AS row
CALL {
WITH row
CREATE (e: Territory)
SET e = {
  territoryID: row.territoryid,
  name: row.territorydescription
        }
} IN TRANSACTIONS OF 10 ROWS;
```

Loading the graph

3. Loading from a Postgres DB

- Copy database driver to the “Plugins” folder
- APOC library must also be copied in the “Plugins” folder
- **Check the right APOC version for your Neo4j version!!!**
- **Must download also apoc-5.10.0-extended.jar NOT just apoc-5.10.0-core.jar**

```
WITH "jdbc:postgresql://localhost:5434/NorthwindOLTP?user=postgres&password=postgres" as url
// NorthwindOLTP:  your database in the PostgreSQL instance
// url: to be used in the procedure call
CALL apoc.load.jdbc(url,"select * from categories") YIELD row
// row: a “row variable” just as before
RETURN row.description,row.categoryname
```

This lists the table “categories” in Neo4j.
We can use this also for loading data into Neo4j.

Loading the graph

```
WITH "jdbc:postgresql://localhost:5434/NorthwindOLTP?user=postgres&password=postgres" as url

CALL apoc.load.jdbc(url,"select * from products") YIELD row

CREATE (:Product {productID: row.productid,productName:row.productname, supplier: row.supplierid,
category:row.categoryid, qtyperunit:row.quantityperunit})

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

WITH "jdbc:postgresql://localhost:5434/NorthwindOLTP?user=postgres&password=postgres" as url

CALL apoc.load.jdbc(url,"select * from suppliers") YIELD row

CREATE (:Supplier {supplierID: row.supplierid, supplierName:row.companyname, city:row.city, region:row.region,
country:row.country})
```

Loading the graph

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/NWdata/city.csv" AS row
CREATE (:City {cityID:row.citykey,cityName: row.cityname});

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/NWdata/territories.csv" AS row
CREATE (:Territory {territoryID: row.territoryID, name: row.territoryDescription});

...

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/NWdata/employee-territories.csv" AS row
MATCH (territory:Territory{territoryID: row.territoryID})
MATCH (employee:Employee {employeeID: row.employeeID})
MERGE (employee)-[:AssignedTo]->(territory);

Loading the graph

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/NWdata/orders.csv" AS row
MATCH (order:Order {orderID: row.orderID})
MATCH (employee:Employee {employeeID: row.employeeID})
MERGE (employee)-[:Sold]->(order);

LOAD CSV WITH HEADERS FROM "file:/NWdata/order-details.csv" AS row
MATCH (order:Order {orderID: row.orderID})
MATCH (product:Product {productID: row.productID})
MERGE (order)-[:Contains{unitPrice:row.unitPrice,quantity:row.quantity,  discount:row.discount}]->(product);

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/NWdata/products.csv" AS row
MATCH (product:Product {productID: row.productID})
MATCH (supplier:Supplier {supplierID: row.supplierID})
MERGE (supplier)-[:Supplies]->(product);
```

Loading the graph

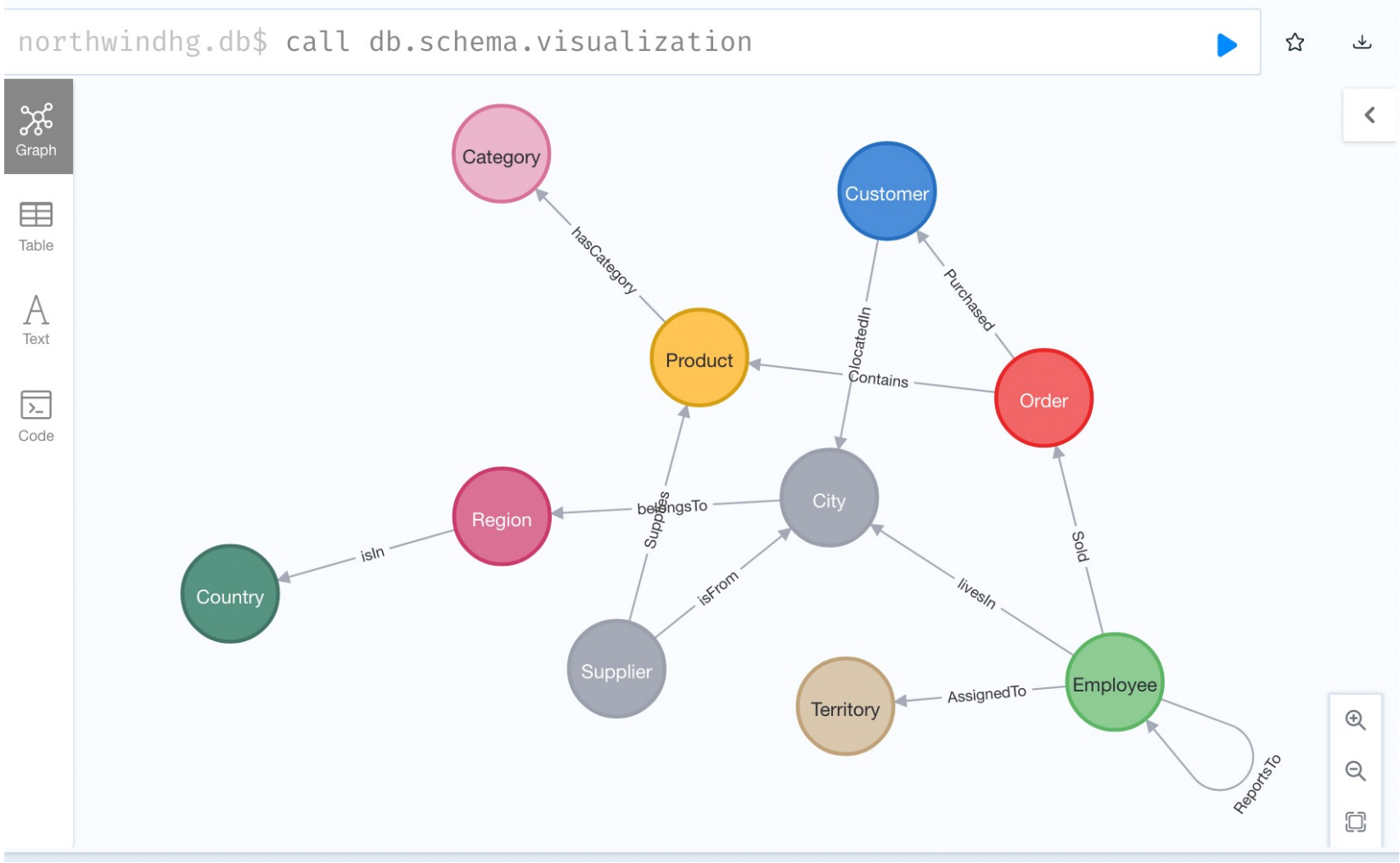
```
CALL apoc.load.jdbc('jdbc:postgresql://localhost:5433/NorthwindOLTP?user=postgres&password=postgres','select * from employees') YIELD row
MATCH (employee:Employee {employeeID: row.employeeid})
MATCH (employee1:Employee {employeeID: row.reportsto})
MERGE (employee)-[:ReportsTo]->(employee1);

--  Create a view to put together orders and order details

CREATE VIEW order1 AS (SELECT o.orderid AS orderID, o.orderdate AS  orderDate,  o.shippeddate AS shippedDate, o.shipname AS shipName, sum(quantity)  AS totqty, sum(unitprice*quantity) AS totAmount
FROM orders o, orderdetails o1
WHERE o.orderid = o1.orderid
GROUP BY o.orderid, o.orderdate, o.shippeddate, o.shipname
ORDER BY orderid  asc)

CALL apoc.load.jdbc('jdbc:postgresql://localhost:5433/NorthwindOLTP?user=postgres&password=postgres','select * from ordershg') YIELD row
CREATE (:Order {orderID: row.orderid, orderDate: row.orderdate, ShippedDate: row.shippeddate, shipName:row.shipname, totalQty:row.totqty, totalAmount:row.totamount});
```

Schema: Northwindhg database



Problem 1. Northwindhg database

- Query 1. List all product names together with their unit price

```
MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
```

- Query 2. List the nodes corresponding to products 'Chocolade' & 'Pavlova'

```
MATCH (p:Product)
WHERE p.productName IN ['Chocolade','Pavlova']
RETURN p
```

- Query 3. List all product names together with their unit Price for products with names starting with a "C", whose unit price is greater than 50

```
MATCH (p:Product)
WHERE p.productName STARTS WITH "C" AND  p.unitPrice > 50
RETURN p.productName, p.unitPrice;
```

	p.productName	p.unitPrice
1	"Carnarvon Tigers"	62.5
2	"Côte de Blaye"	263.5

Problem 1. Northwindhg database

- Query 4. Same as 3, but considering the sales unit price, not the product's price.

```
MATCH (p:Product) <- [c:Contains] - (o:Order)
WHERE p.productName STARTS WITH "C" AND  c.unitPrice > 50
RETURN distinct p.productName, p.unitPrice, c.unitPrice;
```

- Query 5. Total purchased by customer and product

```
MATCH (c:Customer)
OPTIONAL MATCH (p:Product)<-[pu:Contains]-(:Order)-[:Purchased]->(c)
RETURN c.customerName, p.productName, sum(pu.unitPrice * pu.quantity) as volume
ORDER BY p.productName desc
// 1687 records in the answer
// Check the result omitting the OPTIONAL keyword – 1685 answers
```

c.customerName	p.productName	volume
"FISSA Fabrica Inter. Salchichas S.A."	*null*	0
"Paris spécialités"	*null*	0
"Around the Horn"	"Zaanse koeken"	237.5

	c.customerName	p.productName	volume
1	"Around the Horn"	"Zaanse koeken"	237.5
2	"Berglunds snabbköp"	"Zaanse koeken"	579.5

- Query 6. Top 10 employees, considering the number of orders sold

```
MATCH (:Order)<-[:Sold]-(e:Employee)
RETURN e.firstName,e.lastName, count(*) AS Orders
ORDER BY Orders DESC LIMIT 10
```

Problem 1. Northwindhg database

- Query 7. For each employee, build a list with the assigned territories

```
MATCH (t:Territory)<-[:AssignedTo]-(e:Employee)
RETURN e.lastName, COLLECT(t.name);
```

	e.lastName	COLLECT(t.name)
1	"Fuller"	["Westboro", "Bedford", "Georgetow", "Boston", "Cambridge", "Braintree", "Loui
2	"Buchanan"	["Providence", "Morristown", "Edison", "New York", "New York", "Mellvile", "Fairp

- Query 8. For each city, list the companies settled in that city

```
MATCH (c:City)<-[:locatedIn]-(c1:Customer)
RETURN c.cityname, COLLECT(c1.customerName);
```

Query 9. How many persons an employee reports to, either directly or transitively?

```
MATCH (report:Employee)
OPTIONAL MATCH (e)<-[rel:ReportsTo*]-(report)
RETURN report.lastName AS e1, COUNT(rel) AS reports

// What happens id we do not use OPTIONAL?  Why do we need the first MATCH clause?
```

- Query 10. To whom do persons called “Robert” report to?

```
MATCH (e:Employee)<-[:ReportsTo*]-(sub:Employee)
WHERE sub.firstName = 'Robert'
RETURN e.firstName,e.lastName,sub.lastName
```

Problem 1. Northwindhg database

- Query 11. Who does not report to anybody?

```
MATCH        (e:Employee)
WHERE NOT (e)-[:ReportsTo]->()
RETURN       e.firstName as TopBossFirst, e.lastName AS TopBossLast
```

- Query 12. Suppliers, number of categories they supply, and a list of such categories

```
MATCH  (s:Supplier)-->(:Product)-->(c:Category)
WITH     s.supplierName as Supplier, COLLECT distinct c.categoryName) as Categories
RETURN Supplier, Categories, size(Categories) AS Quantity ORDER BY  Quantity  DESC
// We cannot write collect(distinct c.categoryName) as Categories, size(categories), but we can write size(collect(distinct c.categoryName))

MATCH (s:Supplier)-->(:Product)-->(c:Category)
WITH     s.supplierName as Supplier, collect(distinct c.categoryName) as Categories, size(COLLECT(distinct c.categoryName))  as Quantity
RETURN Supplier, Quantity ORDER BY Quantity DESC
```

- Query 13. Suppliers who supply beverages

```
MATCH (c:Category)<--(:Product)<--(s:Supplier) WHERE c.categoryName = "Beverages"
RETURN DISTINCT s.supplierName as ProduceSuppliers;
```

Problem 1. Northwindhg database

- Query 14. Customer who purchases the largest amount of beverages

```
MATCH  (cust:Customer)<-[:Purchased]-(:Order) - [o:Contains] -> (p:Product), (p) - [:hasCategory] -> (c:Category{categoryName:"Beverages"})
RETURN cust.customerName as CustomerName, SUM(o.quantity) LIMIT 1
```

- Query 15. List the 5 most popular products (considering the number of orders)

```
MATCH     (c:Customer) <- [:Purchased] - (o:Order) - [o1:Contains] -> (p:Product)
RETURN    c.customerName, p.productName, count(o1) as orders
ORDER BY  orders desc  LIMIT 5
```

Problem 1. Northwindhg database

- Query 16. Products ordered by customers from the same country than their suppliers

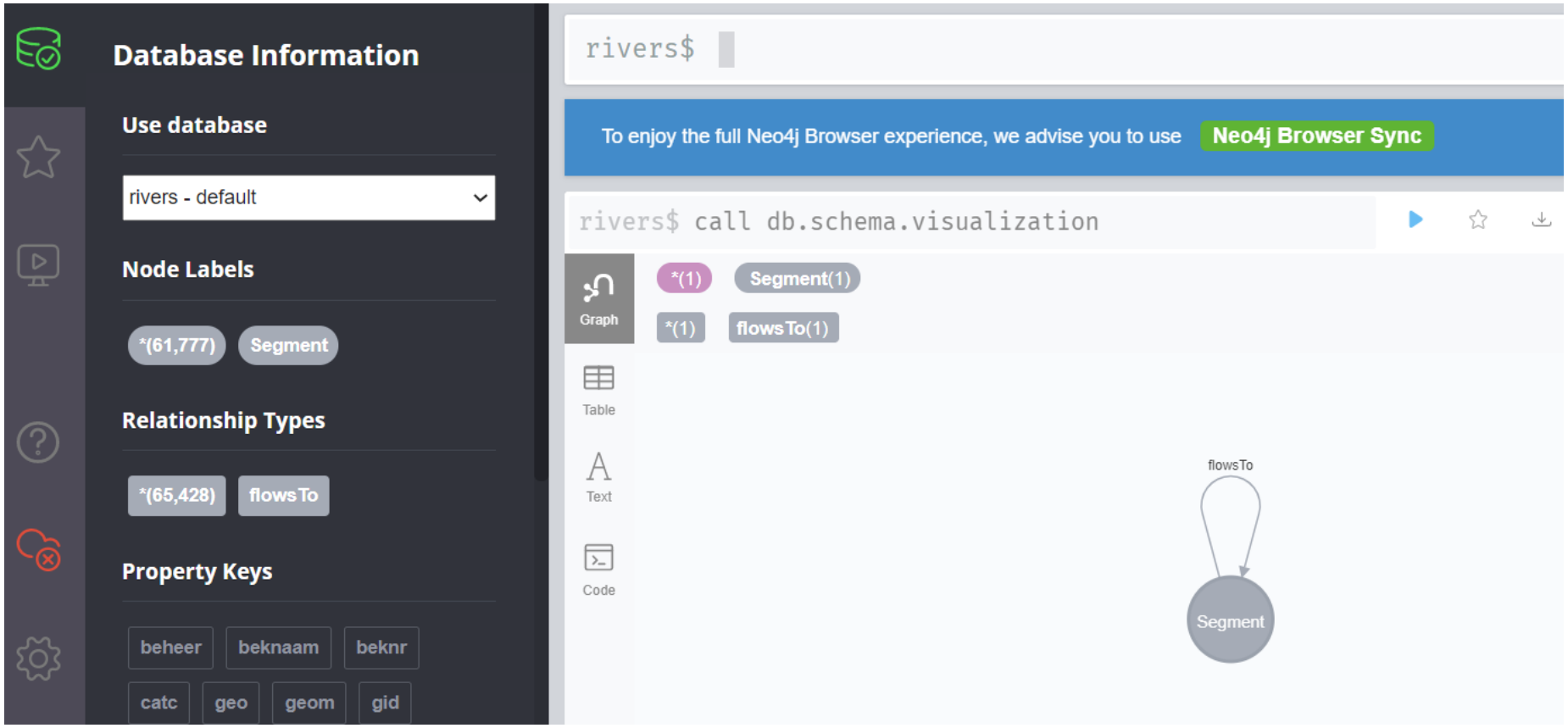
```
MATCH   (c:Customer) -[r:locatedIn]->(cy:City)-[:belongsTo]->(:Region)-[:isIn]->(co:Country)
WITH  co, c
MATCH    (s:Supplier) WHERE co.countryname = s.country
WITH  s, co, c
MATCH     (s)-[su:Supplies]-(p:Product)<-[:Contains]-(o:Order)-[:Purchased]->(c)
RETURN    c.customerName,s.supplierName,co.countryname,p.productName
```

OR

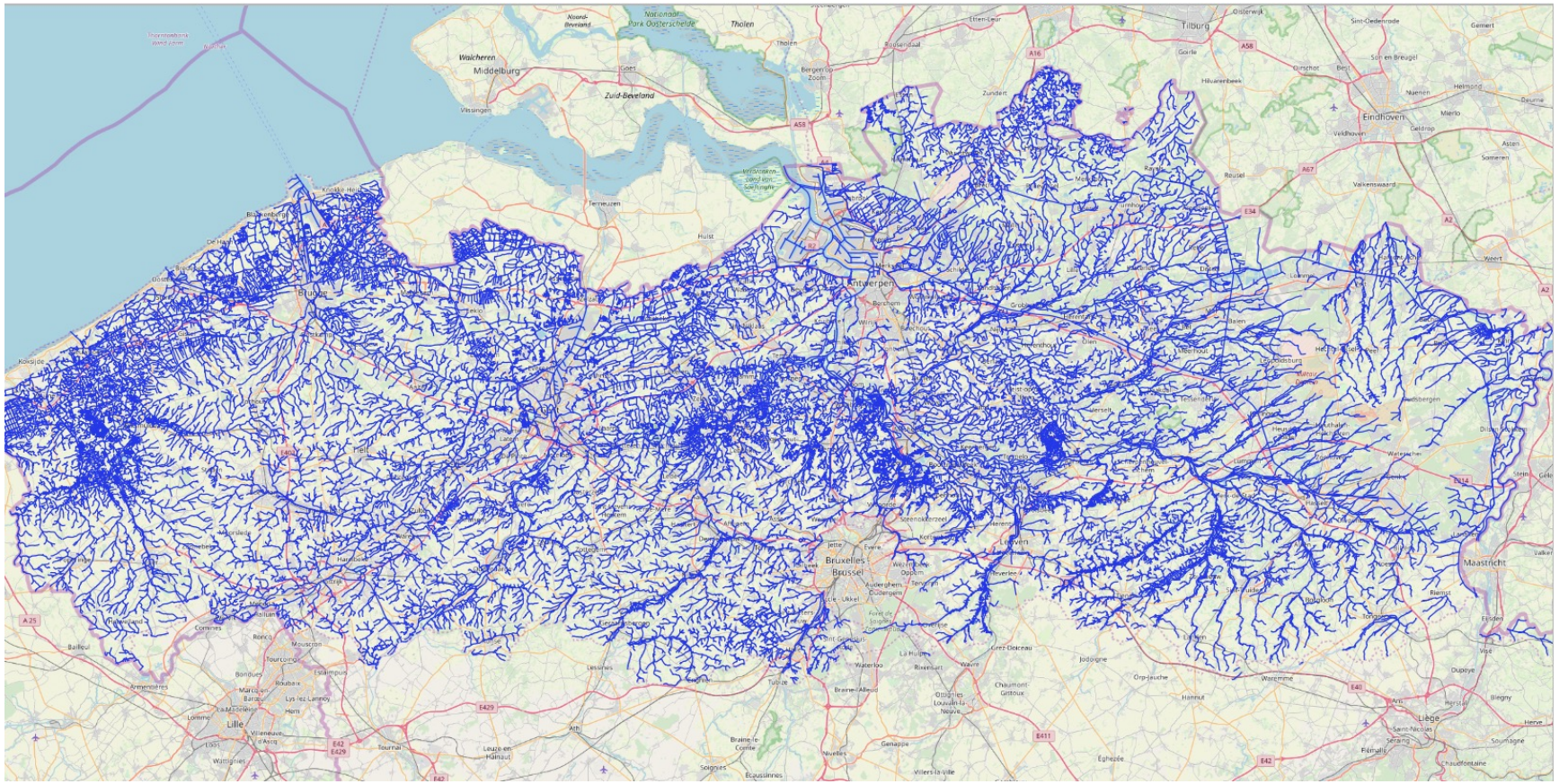
```
MATCH    (c:Customer) -[r:locatedIn]->(cy:City)-[:belongsTo]->(:Region)-[:isIn]->(co:Country)
WITH       co, c
MATCH   (s:Supplier)-[su:Supplies]-(p:Product) <- [:Contains]-(o:Order) - [:Purchased] - >(c)
WHERE   co.countryname = s.country
RETURN  c.customerName,s.supplierName,co.countryname,p.productName

// Check that we obtain the same result
```

Problem 2 – Rivers



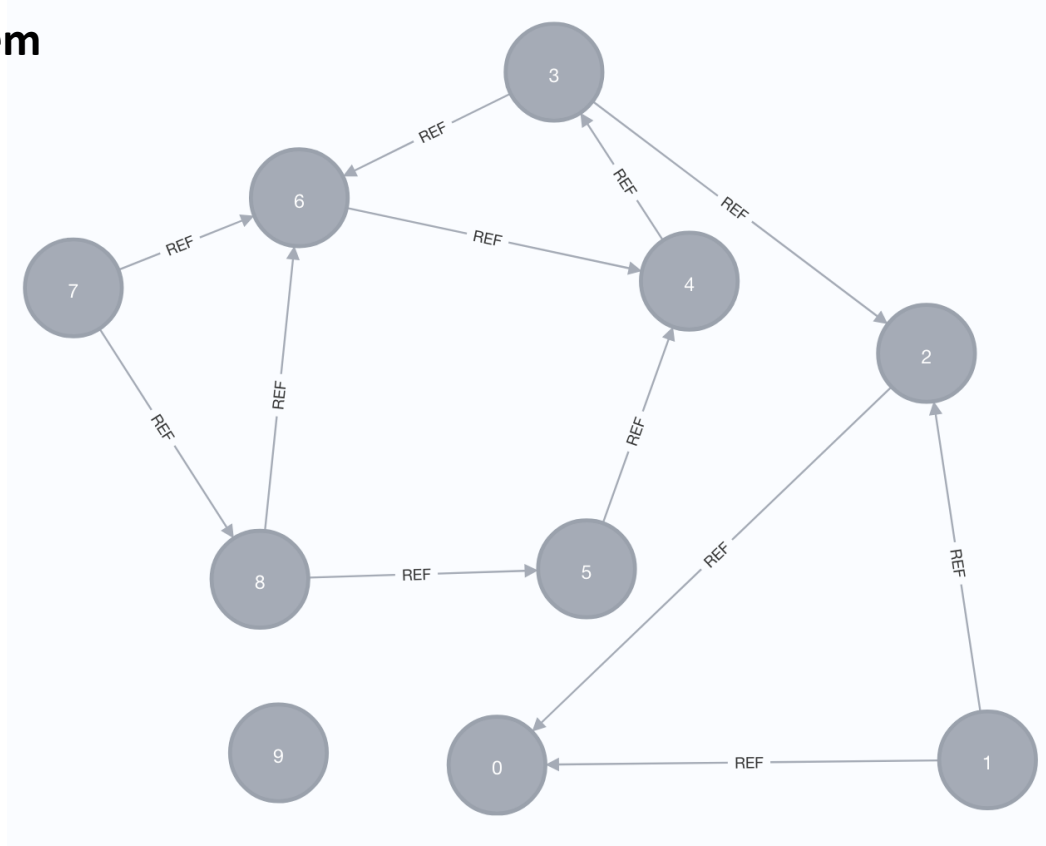
Problem 2 – Rivers



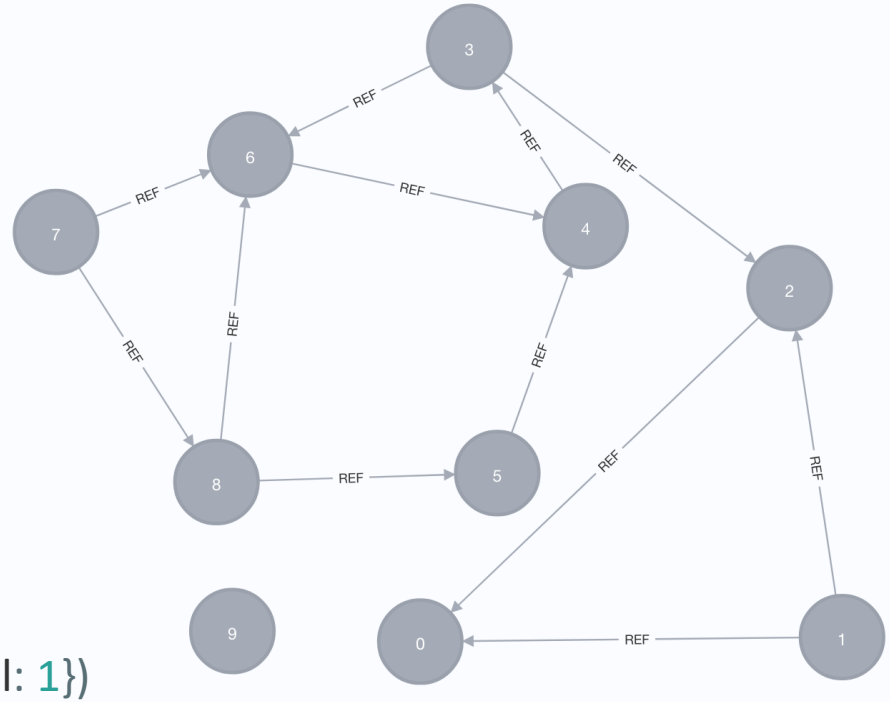
Problem 2 – Rivers

Before we start, let us analyze a smaller problem

Consider the graph “miniwebgraph”.



Problem 2 – Rivers



Spanning Tree. All nodes reachable from a given node

Consider the graph “miniwebgraph”.

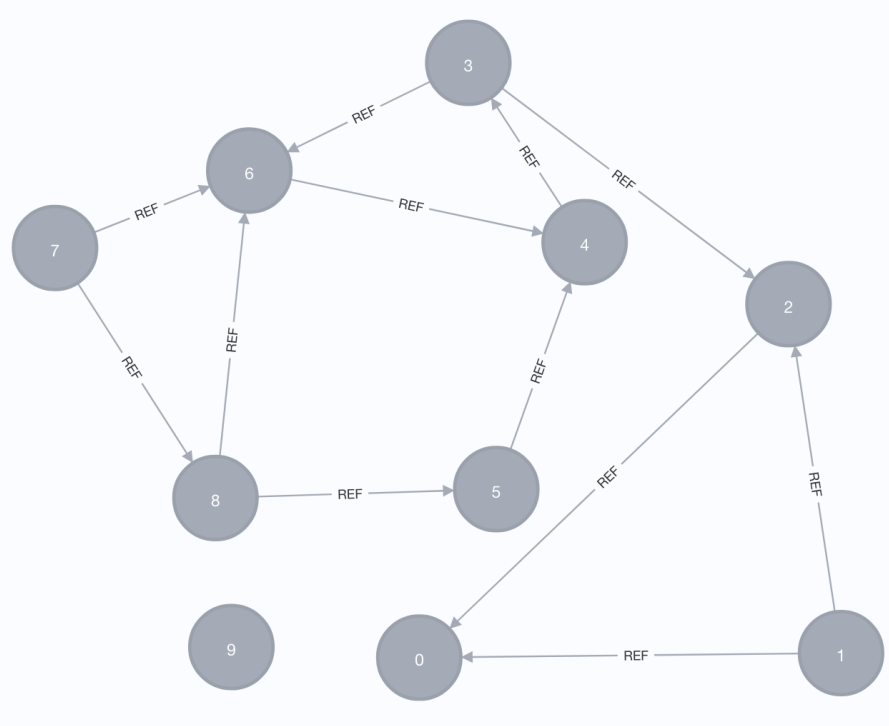
```
MATCH (n:URL {name:7})
CALL    apoc.path.spanningTree(n,{relationshipFilter:"REF>", minLevel: 1})
          YIELD path AS pp
RETURN [p in NODES(pp)|p.name]
```

This query returns all nodes reachable from node 7.

Note: [7, 6, 4] is in the answer, but NOT [7, 8, 5, 4] because Node 4 has already been reached

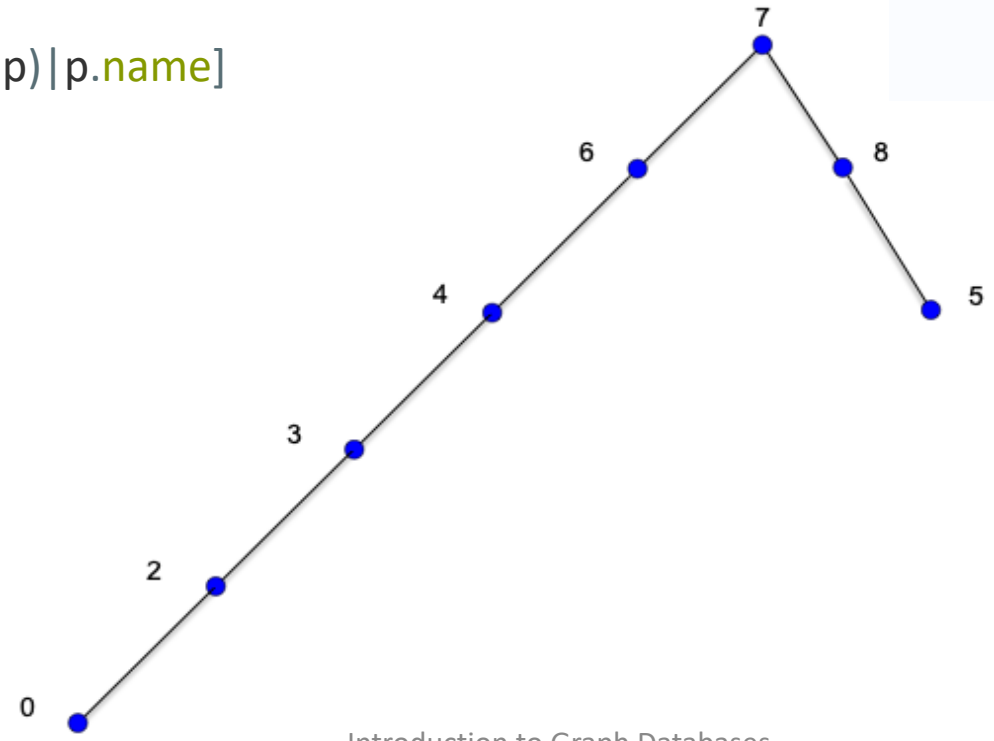
[7, 6]
[7, 8, 5]
[7, 6, 4]
[7, 6, 4, 3]
[7, 6, 4, 3, 2]
[7, 6, 4, 3, 2, 0]

Problem 2 – Rivers



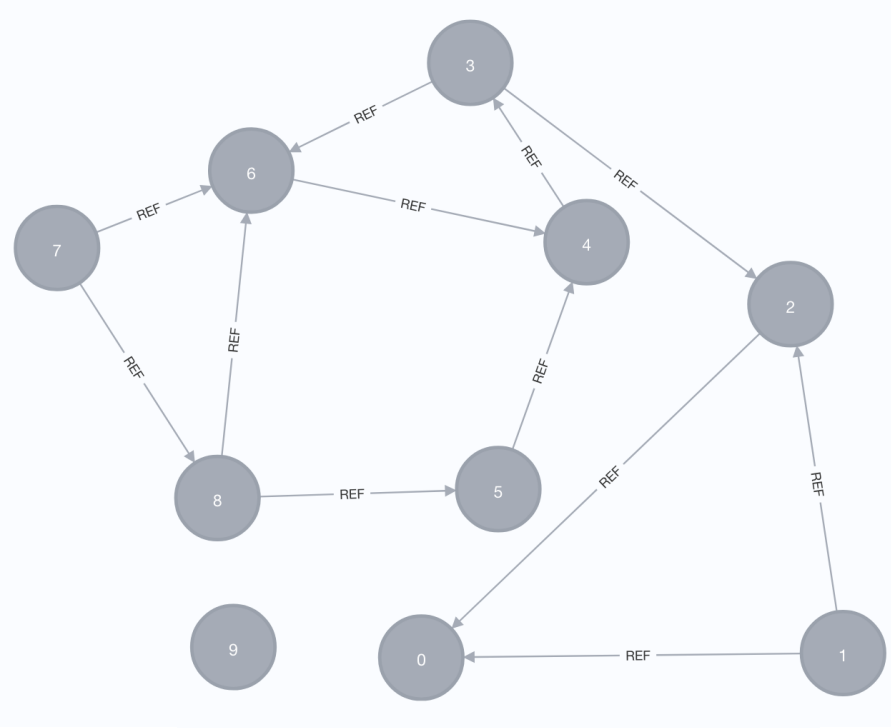
Spanning Tree for Node 7

```
MATCH (n:URL {name:7})
CALL apoc.path.spanningTree(n,{relationshipFilter:"REF>", minLevel:
        YIELD path AS pp
RETURN [p in NODES(pp)|p.name]
```



[7, 6]
[7, 8, 5]
[7, 6, 4]
[7, 6, 4, 3]
[7, 6, 4, 3, 2]
[7, 6, 4, 3, 2, 0]

Problem 2 – Rivers



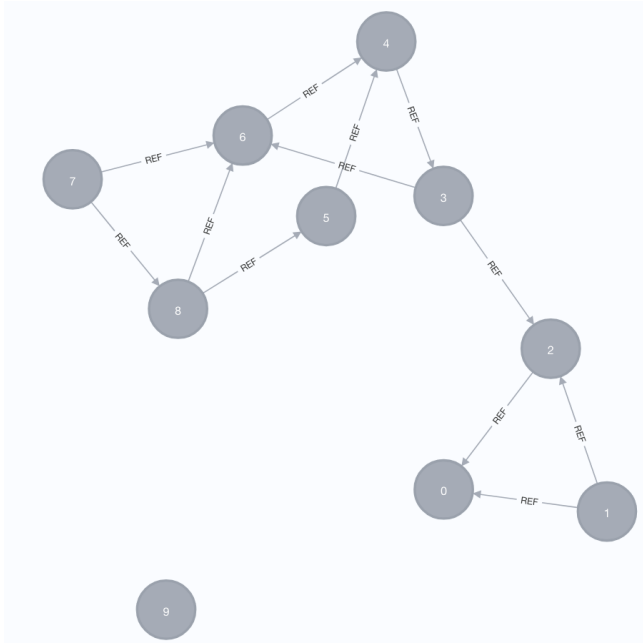
Spanning Tree

- Consider now the query:
- All nodes **directly reachable from the nodes reachable** from node

```
MATCH (n:URL {name:7})
//Nodes reachable from Node 7:
CALL apoc.path.spanningTree(n,{relationshipFilter:"REF>", minLevel: 1})
YIELD   path AS pp
UNWIND NODES(pp) as p
//Nodes directly reachable from Node 7:
MATCH (p)-[:REF]->(r:URL)
RETURN p.name, COLLECT(distinct r.name )
```

	p.name	COLLECT(distinct r.name)
1	7	[8, 6]
2	8	[6, 5]
3	6	[4]
4	5	[4]
5	4	[3]
6	3	[6, 2]
7	2	[0]

Problem 2 – Rivers



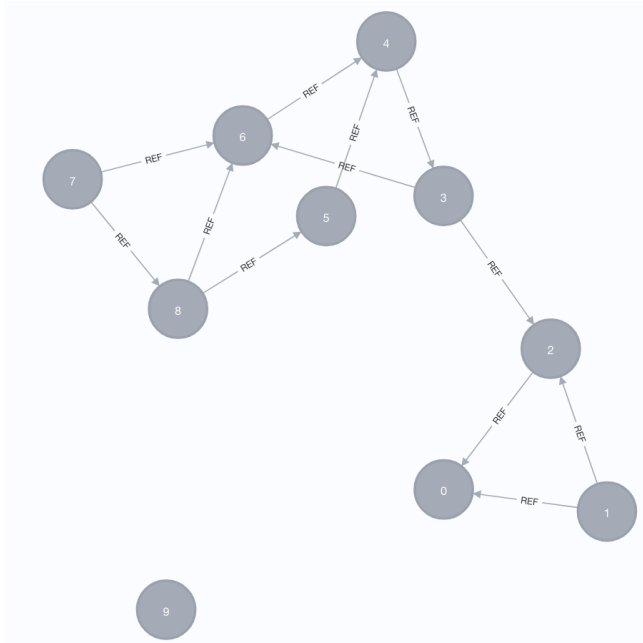
Spanning Tree

- Consider now the query:
- All nodes **directly reachable from the nodes reachable** from node 7 **such that there is a split at such nodes**

```
MATCH (n:URL {name:7})
CALL    apoc.path.spanningTree(n,{relationshipFilter:"REF>", minLevel: 1})
          YIELD  path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:REF]->(r:URL)
WITH p, count(DISTINCT r) as co WHERE co > 1
RETURN p.name
```

	p.name
1	7
2	8
3	3

Problem 2 – Rivers



All Possible Paths Computation

- Consider now the query:
- All the paths starting at node 7

```
MATCH  (n:URL {name:7})
CALL     apoc.path.expandConfig(n, {relationshipFilter:"REF>", minLevel: 1})
           YIELD path AS pp
RETURN [p in NODES(pp)|p.name]
```

Note: Now the query returns [7, 6, 4] AND [7, 8, 5, 4]

Problem: computational cost

	[p in NODES(pp)|p.name]
4	[7, 8, 5]
5	[7, 6, 4]
6	[7, 8, 6, 4]
7	[7, 8, 5, 4]
8	[7, 6, 4, 3]

Problem 2 – Rivers

Query 5. Find the segments with the maximum number of incoming segments.

```
MATCH (n:Segment)
OPTIONAL MATCH (src:Segment)-[:flowsTo]->(n)
WITH n, COUNT(distinct src) as indegree
WITH COLLECT ([n, indegree]) as tuples, MAX(indegree) as max
RETURN [t in tuples WHERE t[1] = max |t[0].vhas], max
```

Let's start now with the Rivers graph database

Problem 2 – Rivers

Query 6. Find the number of splits in the downstream path of segment 6020612

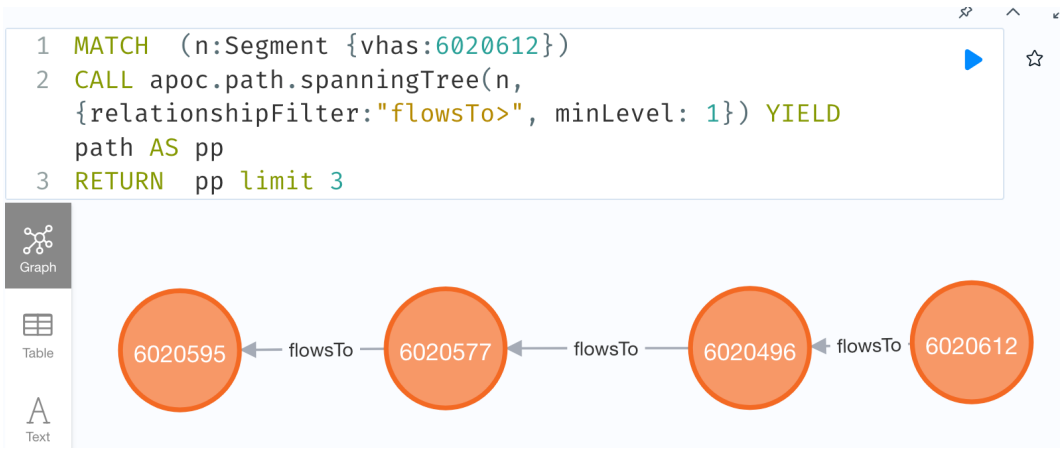
```
MATCH (n:Segment {vhas:6020612})
CALL    apoc.path.spanningTree(n,{relationshipFilter:"flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH    p, count(DISTINCT r) as co WHERE co > 1
RETURN count(p)
```

Let us analyze this query.

Problem 2 – Rivers

```
MATCH (n:Segment {vhas:6020612})
CALL     apoc.path.spanningTree(n,{relationshipFilter:"flowsTo>", minLevel: 1}) YIELD path AS pp
RETURN pp
```

pp is a set of paths



The figure shows three paths, of lengths 1, 2 and 3

Problem 2 – Rivers

```
MATCH   (n:Segment {vhas:6020612})
CALL.       apoc.path.spanningTree(n,{relationshipFilter:"flowsTo>", minLevel: 1}) YIELD path AS pp
RETURN  [p in NODES(pp)|p.vhas] limit 3
```

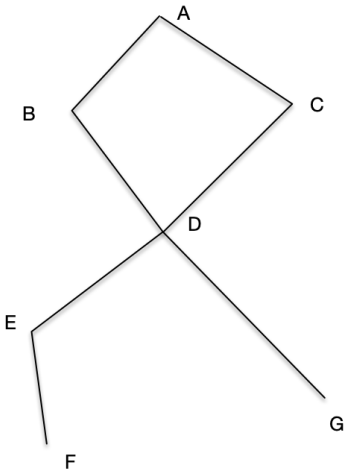
	[p in NODES(pp)|p.vhas]
1	[6020612, 6020496]
2	[6020612, 6020496, 6020577]
3	[6020612, 6020496, 6020577, 6020595]

The figure shows the identifier of the nodes in the three paths, of lengths 1, 2 and 3

Problem 2 – Rivers

Query 8. Determine if there is a loop in the downstream path of segment 6031518.

```
MATCH (n:Segment {vhas:6031518})
CALL    apoc.path.spanningTree(n, {relationshipFilter: "flowsTo>", minLevel: 1}) YIELD path AS pp
WITH    [p in NODES(pp) | p] as nodelist
UNWIND nodelist as p
CALL     apoc.path.expandConfig (p, {relationshipFilter:"flowsTo>", minLevel: 1, terminatorNodes:[p],
          whitelistNodes:nodelist})  YIELD path as loop
RETURN count(loop) >0 as loops
```



- When the same node can be reached following different paths, and we want all possible paths, spanningTree is not enough
- spanningTree returns all reachable nodes from A: A, B, C, D, E, F, G, to filter out nodes that will not be used
- Once it finds a path, it does not check another one, it would find A, B, D or A, C, D, not both

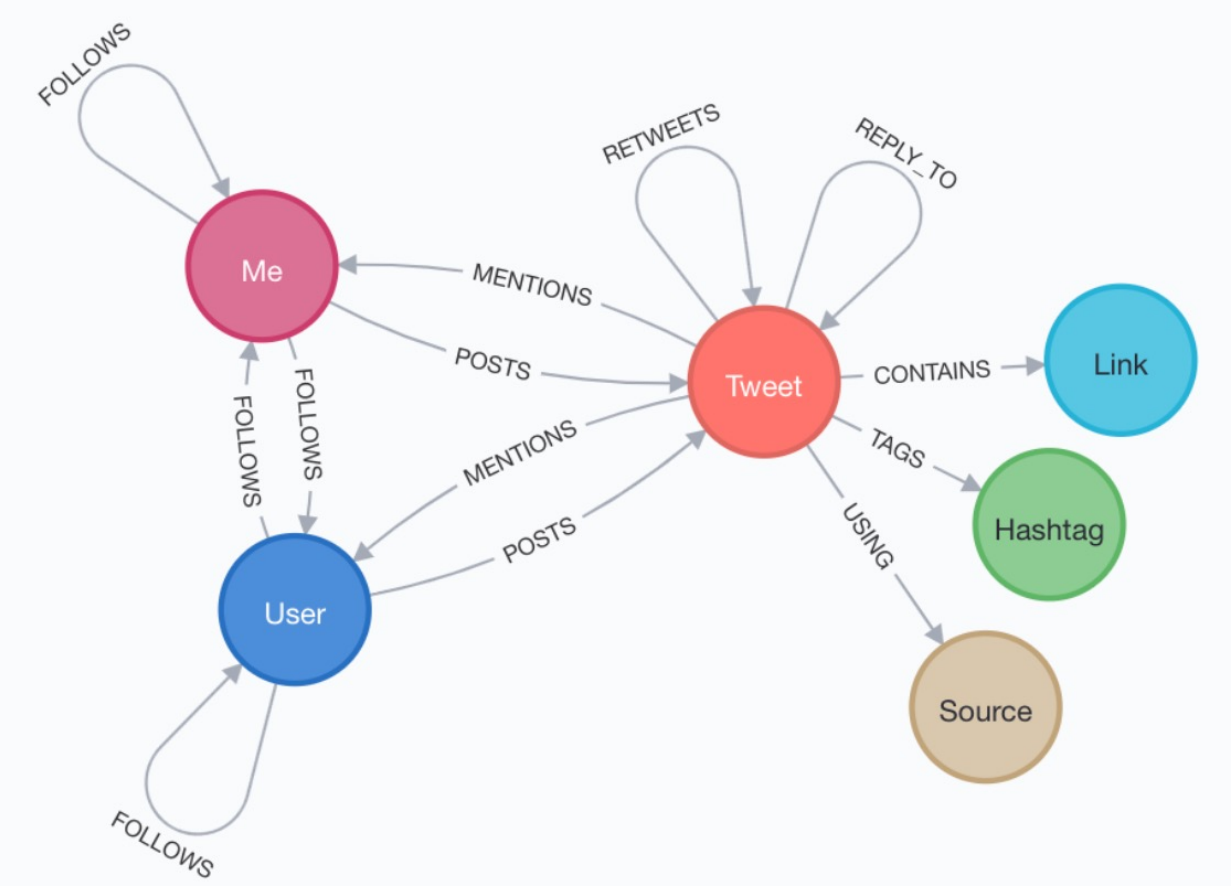
Problem 2 – Rivers

Query 11. Find all segments reachable from the segment closest to Antwerpen's Groenplaats

```
CALL apoc.spatial.geocodeOnce('Groenplaats Antwerpen Flanders Belgium')
      YIELD location as ini
MATCH (n:Segment)
WITH n, ini, point.distance(point({longitude:n.source_long,
                                   latitude:n.source_lat}),
                                   point(ini)) as d
WITH n, d order by d asc limit 1
CALL apoc.path.spanningTree(n,{relationshipFilter:"flowsTo>", minLevel: 1})
       YIELD path as pp
UNWIND NODES(pp) as p
RETURN p.vhas;
```

	p.vhas
1	6033894
2	6033902
3	6033894
4	6033902
5	7051909

Problem 3 – Twitter



Note that in this case, there is a node of type User that also has the label "Me"

Problem 3 – Classic queries - Twitter

1. Who do I mention in Twitter?

```
MATCH (u:Me:User) - [p:POSTS] -> (t:Tweet) - [:MENTIONS] -> (m:User)
WITH u, p, t, m, COUNT(m.screen_name) AS count ORDER BY  count DESC
RETURN    u, p, t, m
```

2. Detailed list and count of my mentions

```
MATCH  (u:User:Me)- [:POSTS] -> (t:Tweet)-[:MENTIONS] -> (m:User)
RETURN  m.screen_name AS screen_name, COUNT(m.screen_name) AS count
ORDER BY  count DESC
```

Note that u:Me implies that it is referring to myself (in this case, ‘Neo4j’). It is like asking u.name = ‘Neo4j’

Problem 3 – Twitter

3. Who are my most influential followers?

```
MATCH  (follower:User) - [:FOLLOWS] -> (u:User:Me) // we could simply write (u:Me)
RETURN follower.screen_name AS user, follower.followers AS followers
ORDER BY  followers DESC
LIMIT 10
```

4. Tags most used by me

```
MATCH (h:Hashtag) < - [:TAGS] - (t:Tweet) < - [:POSTS]-(u:User:Me)
WITH h, COUNT(h) AS Hashtags
ORDER BY Hashtags DESC
LIMIT 10
RETURN h.name, Hashtags
```

Problem 3 – Twitter

5. At what rate do people I follow also follow me back?

```
MATCH (me:User:Me) - [:FOLLOWS]->(f)
WITH  me, f,  count{(f) - [:FOLLOWS] -> (me)} as doesFollowBack  // doesFollowBack is either 0 or 1
RETURN  SUM(doesFollowBack) / toFloat(COUNT(f))  AS followBackRate
```

6. Who tweets about me, but I do not follow?

```
MATCH  (ou:User) - [:POSTS] -> (t:Tweet) - [mt:MENTIONS] -> (me:User:Me)
WITH DISTINCT ou, me
WHERE (ou) - [:FOLLOWS] -> (me) AND NOT (me) - [:FOLLOWS] -> (ou)
RETURN  ou.screen_name
```

Problem 3 – Twitter

7. What links do I retweet, and how often are they favorited?

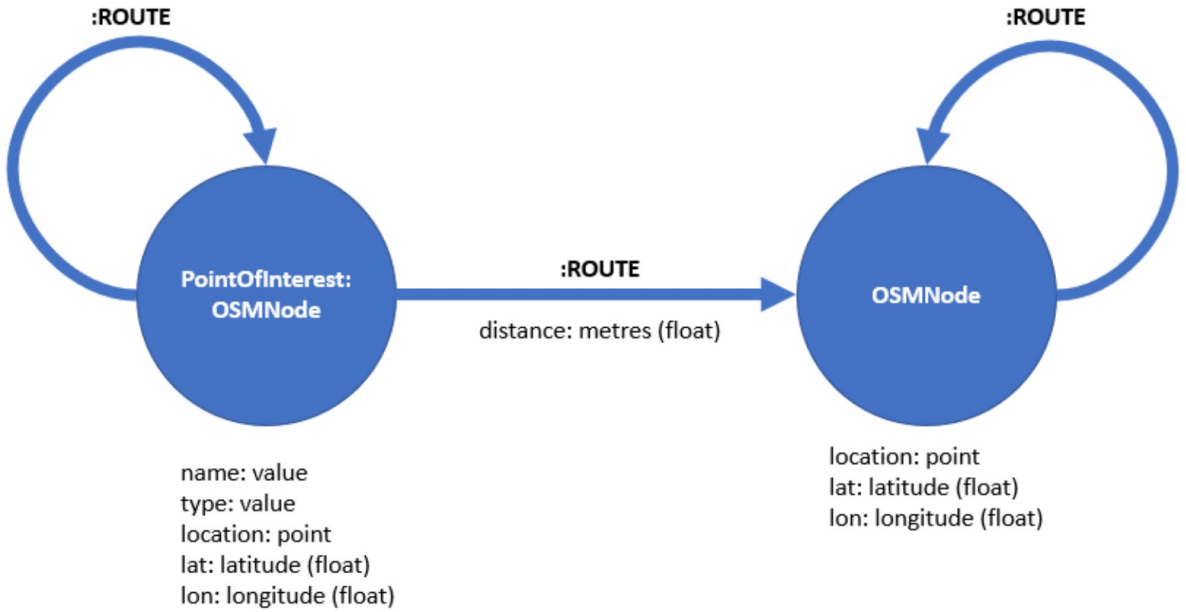
```
MATCH  (:User:Me) - [:POSTS] -> (t:Tweet) - [:RETWEETS] -> (rt) - [:CONTAINS] -> (link:Link)
RETURN  t.id_str AS tweet, link.url AS url, rt.favorites AS favorites
ORDER BY favorites DESC
```

8. Users that tweet some of my hashtags?

```
MATCH (me:User:Me)-[:POSTS]->(tweet:Tweet)-[:TAGS]->(ht)
MATCH (ht) <- [:TAGS] - (tweet2:Tweet) <- [:POSTS] - (sugg:User)
WHERE sugg <> me AND NOT (tweet2) - [:RETWEETS] -> (tweet)
RETURN sugg.name, COLLECT(distinct(ht.name)) as tags
```

Problem 4 – Spatial queries - OSM

- Route and tagged Points of Interest for Central Park, based on OpenStreetMap
- We take the role of a virtual tourist
- Plugin to extract data: https://github.com/neo4j-contrib/osm



Problem 4 – OSM

- In Cypher, APOC allows us to geocode an address and for example, compute a distance. Only points are currently supported. For example:

```
CALL apoc.spatial.geocodeOnce('Parque Rivadavia, Buenos Aires, Argentina') YIELD location as ini
CALL apoc.spatial.geocodeOnce('Teatro Colón, Buenos Aires, Argentina') YIELD location as ini1
WITH point({srid:4326, x:ini.longitude,y:ini.latitude}) AS p1, point({srid:4326, x:ini1.longitude,y:ini1.latitude}) AS p2
RETURN p1.x, p1.y, p2.x, p2.y, point.distance(p1,p2)
```

p1.x	p1.y	p2.x	p2.y	point.distance(p
-58.433456818857294	-34.61783475	-58.38318689897575	-34.601085499999996	4968.854541927

Problem 4 – OSM

```
CALL apoc.spatial.geocodeOnce('Sigmund Freud Museum, Vienna, Austria') YIELD location as ini
CALL apoc.spatial.geocodeOnce('State Opera, Vienna, Austria') YIELD location as ini1
WITH point(ini) AS p1, point(ini1) as p2
RETURN p1.x, p1.y, p2.x, p2.y, point.distance(p1,p2)
```

```
CALL apoc.spatial.geocodeOnce('Facultad de Ingeniería, Montevideo, Uruguay') YIELD location as ini
CALL apoc.spatial.geocodeOnce('Aeropuerto, Montevideo, Uruguay') YIELD location as ini1
WITH point(ini) AS p1, point(ini1) as p2
RETURN p1.x, p1.y, p2.x, p2.y, point.distance(p1,p2)
```

p1.x	p1.y	p2.x	p2.y	point.distance(p1,p2)
-56.1906122	-34.903842	-56.2645261917794	-34.78817835	14538.935620647617

Problem 4 – OSM

1. Find a PoI of type clock, and the PoIs 100 m around it

```
MATCH (p:PointOfInterest {type:'clock'}) RETURN p.name
```

```
MATCH (p1:PointOfInterest {type:'clock'}), (p2:PointOfInterest)
WHERE p1<>p2 AND point.distance(p1.location, p2.location) < 100
RETURN p2.name
```

Problem 4 – OSM

2. How far apart are the zoo school and the clock as a straight line (as the crow flies)?

```
MATCH (p1:PointOfInterest {type:'clock'}), (p2:PointOfInterest {name:'Zoo School'}) RETURN
point.distance(p1.location,p2.location)
```

3. What is the actual walking distance?

```
MATCH path=shortestpath((p1:PointOfInterest {type:'clock'})-[:ROUTE*]-(p2:PointOfInterest {name:'Zoo School'}))
WITH relationships(path) AS rels
//extract all the relationships in the path as an array
UNWIND rels AS rel
RETURN sum(rel.distance)
```

Problem 4 – OSM

4. Locate which cafe type:'cafe' is closest to a bicycle rental place type:'bicycle rental'. What's the name of the cafe?

```
MATCH path = shortestPath((p1:PointOfInterest {type:'cafe'})-[:ROUTE*]-(p2:PointOfInterest {type:'bicycle rental'}))
WITH p1, p2, relationships(path) AS rels
UNWIND rels AS rel   //unwind the array of relationships
RETURN p1.name, p2.name, sum(rel.distance) AS dist ORDER BY dist
```

Problem 4 – OSM

5. Compare the outputs of shortestPath() against weighted shortest path with the Dijkstra APOC function

```
MATCH path = (p1:PointOfInterest {type:'cafe'}),(p2:PointOfInterest {type:'bicycle rental'})
CALL apoc.algo.dijkstra(p1, p2, 'ROUTE', 'distance') YIELD weight AS dist
RETURN p1.name, p2.name, dist ORDER BY dist
```

- The shortestPath() Cypher function returns the first shortest path by #of relationship hops it finds between two specified points.
- The apoc.algo.dijkstra() APOC function returns the shortest weighted path, based on a specified property on relationships between two specified points, regardless the number of hops between them.
- Thus, we can see that the shortest path traversing the minimum number of nodes may not be the shortest distance path considering the actual trajectory.