Graph Databases

Activity 3 - Cypher

You will be querying three Neo4j databases, provided to you. These databases are: (1) A graph representation of the Northwind operational database, denoted **northwindhg.db**; (2) A graph representation of the MusicBrainz database, called **MusicBrainz.db**. This database contains a portion of the data in the web site of the same names, representing releases and events performed by artists, either individually or in collaborations; (3) A **rivers** database, with data from the Flanders river system, in Belgium.

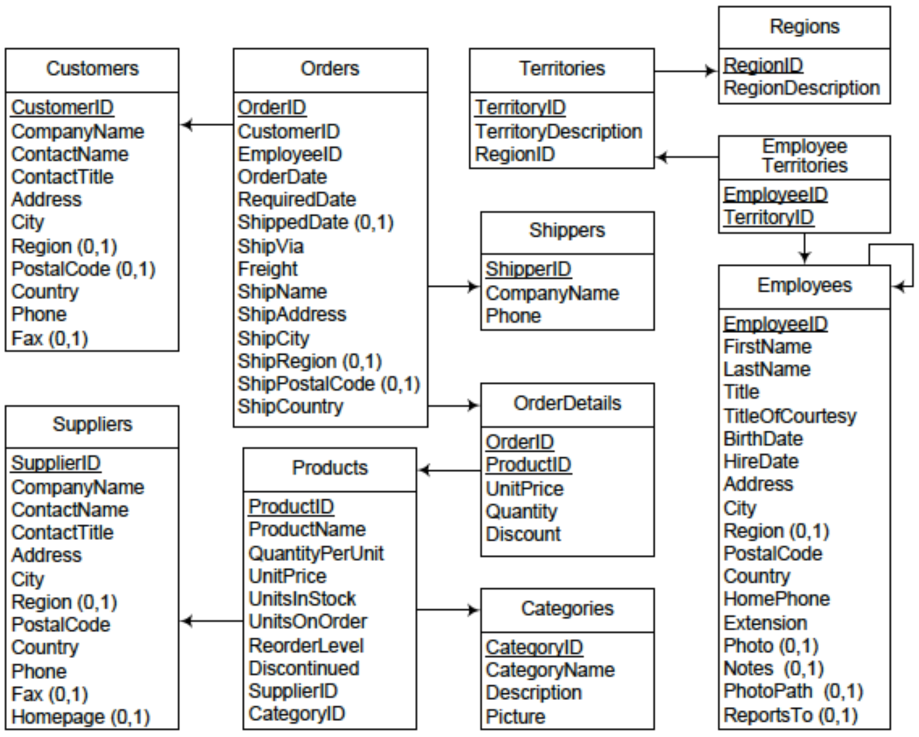
Before starting the Neo4j server, you need to choose the database you will work with. For this, you go to the **conf** folder, and edit the **neo4j.conf** file. You will find something like this:

```
#dbms.default_database=minigraphweb
#dbms.default_database=musicbrainz
#dbms.default_database=northwindhg
#dbms.default_database=northwindoltp
dbms.default_database=rivers
#dbms.default_database=neo4j
#dbms.default_database=webgraph3
#dbms.default_database=webdb
#dbms.default_database=minigraphweb
```

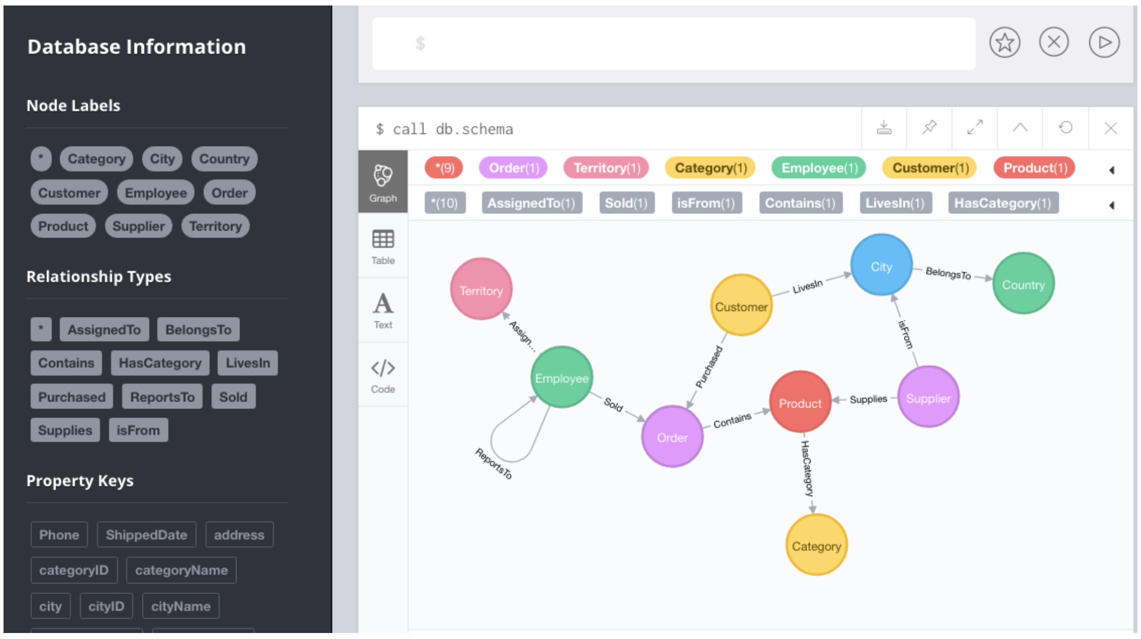
Since dbms.default_database =rivers is unmarked, to change the database to northwindhg, you mark #dbms.default_database =rivers, and unmark dbms.default_database = northwindhg. Save the changes, and quit the file. Then you run: `./bin/neo4j console` to start the Server. Then, open a browser, and type the following url: **localhost:7474.** Now you can start writing Cypher queries.

Exercise 1.

Consider the Northwind database, whose schema is:



This database has been exported to Neo4j, and you can find it at: /……../data/databases/northwindhg. The graph schema is:



```
$ call db.schema
```

Write in Cypher the following queries over the northwindhg.db database:

Query 1 - List products and their unit price.

Query 2 - List information about products 'Chocolade' & 'Pavlova'.

Query 3 - List information about products with names starting with a "C", whose unit price is greater than 50.

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

Query 5 - Total amount purchased by customer and product.

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

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

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

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

Query 10 - To whom do persons called "Robert" report to?

Query 11 - Who does not report to anybody?

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

Query 13 - Suppliers who supply beverages

Query 14 - Customer who purchases the largest amount of beverages

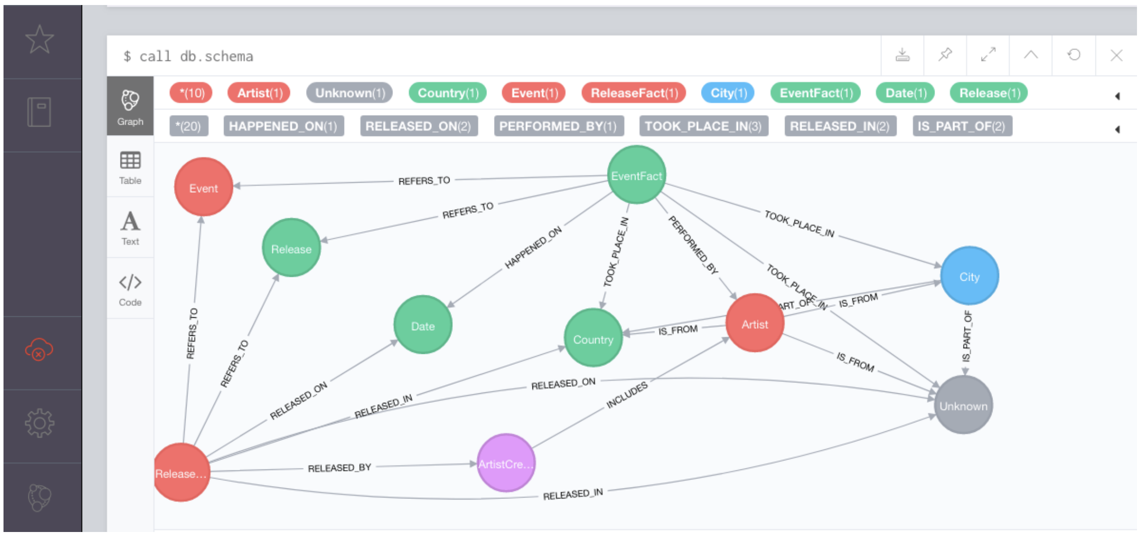
Query 15 - List the five most popular products (considering number of orders)

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

Answer: In the lecture slides

Exercise 2.

Switch to the MusicBrainz database, doing the **same steps as in Assignment 2.** Now, the database is **musicbrainz**. The schema is:



```
$ call db.schema
```

Query 1. Compute the total number of releases per artist.

MATCH (r:ReleaseFact)-[]->(a:ArtistCredit)-[]->(a1:Artist)
RETURN a1, count(r)

Query 2. Compute the total number of releases per artist and per year.

MATCH (r:ReleaseFact)-[r1:RELEASED_BY]->(a:ArtistCredit)-[]->(a1:Artist),
(d:Date)<-[rd:RELEASED_ON]-(r)

RETURN a1.name, d.year, count(r) ORDER BY a1.name ASC,d.year ASC

Query 3. Compute the total number of events per artist.

MATCH (e:EventFact)-[r:PERFORMED_BY]->(a:Artist)
RETURN a.name, count(e)

Query 4. Compute the number of times the artist performed in each event.

MATCH (e1:Event)<-[:REFERS_TO]-(e:EventFact)-[:PERFORMED_BY]->(a:Artist)
RETURN e1.name,a.name, count(e) ORDER BY e1.name ASC, a.name ASC

Query 5. For each (event, artist, year) triple, compute the number of times the artist performed in an event on an year.

```
MATCH (e1:Event)<-[r1:REFERS_TO]-(e:EventFact)-[r:HAPPENED_ON]->(d:Date)
WITH e,d,e1
MATCH (e)-[p:PERFORMED_BY]->(a:Artist)
RETURN e1.name, a.name, d.year, count(*)
ORDER BY e1.name asc, d.year asc, a.name asc
```

Query 6. Same as Query 5, for artists in the United Kingdom and events happened after year 2006.

```
MATCH (e1:Event)<-[r1:REFERS_TO]-(e:EventFact)-[r:HAPPENED_ON]->(d:Date)
WHERE d.year > 2006
WITH e,d,e1
MATCH (e)-[p:PERFORMED_BY]->(a:Artist)-
[IS_FROM]->(c:Country{name:'United Kingdom'})
RETURN e1.name, d.year,a.name, count(*)
ORDER BY e1.name asc, d.year asc, a.name asc
```

Query 7. Compute the number of releases, per language, in the UK.

```
MATCH (rd:Release)<-[REFERS_TO]-(r:ReleaseFact)
-[r1:RELEASED_IN]->(c:Country{name:'United Kingdom'})
RETURN rd.language,count(*) as nbr
ORDER BY nbr desc
```

Query 8. Compute, for each pair of artists, the number of times they have performed together at least twice in an event.

```
MATCH (a1:Artist)<-[]-(e:EventFact)-[]->(a2:Artist)
WHERE a1.id < a2.id
WITH a1, a2, COLLECT(e) AS events WHERE SIZE(events) > 1
RETURN a1.name, a2.name, SIZE(events)
ORDER BY SIZE(events) desc
```

Query 9. Compute the triples of artists, and the number of times they have performed together in an event, if this number is at least 3.

```
MATCH (a1:Artist)<-[]-(e:EventFact)-[]->(a2:Artist)
WHERE a1.id < a2.id
WITH a1,a2,COLLECT(e) AS events WHERE SIZE(events) > 2
MATCH (a1:Artist)<-[]-(e1:EventFact)-[]->(a2:Artist)
MATCH (a3:Artist)<-[]-(e1) WHERE a2.id < a3.id
WITH a1.name as name1, a2.name as name2,a3.name as name3 ,
COUNT(e1.idEvent) as nbrTimes WHERE nbrTimes > 2
```

```
RETURN name1,name2,name3, nbrTimes ORDER BY nbrTimes DESC
```

Query 10. Compute the quadruples of artists, and the number of times they have performed together in an event, if this number is at least 3.

```
MATCH (a1:Artist)<-[]-(e:EventFact)-[]->(a2:Artist)
WHERE a1.id < a2.id
WITH a1,a2,COLLECT(e) AS events
WHERE SIZE(events) > 1
MATCH (a1:Artist)<-[]-(e1:EventFact)-[]->(a2:Artist)
MATCH (a3:Artist)<-[]-(e1) WHERE a2.id < a3.id
WITH a1, a2,a3,collect(e1) as events1 WHERE size(events1) > 1
MATCH (a1:Artist)<-[]-(e2:EventFact)-[]->(a2:Artist)
MATCH (a3:Artist)<-[]-(e2:EventFact)-[]->(a4:Artist)
WHERE a3.id < a4.id
WITH a1.name as name1, a2.name as name2, a3.name as name3,
a4.name as name4, COUNT(e2.idEvent) as nbrTimes WHERE nbrTimes > 2
RETURN name1,name2,name3,name4,nbrTimes ORDER BY nbrTimes DESC
```

Query 11. Compute the pairs of artists that have performed together in at least two events and that have worked together in at least one release, returning the number of events and releases together.

```
MATCH (a1:Artist)<-[]-(e:EventFact)-[]->(a2:Artist)
WHERE a1.id < a2.id
WITH a1, a2, COLLECT(e) AS events
WHERE SIZE(events) > 1
WITH a1, a2, events
MATCH (a1)<-[:INCLUDES]-(ac:ArtistCredit)-[:INCLUDES]->(a2)
WITH a1, a2, ac, events
MATCH (r:ReleaseFact)-[:RELEASED_BY]->(ac)
WITH a1, a2, events, collect(r) AS releases
WHERE SIZE(releases) > 0
RETURN a1.name, a2.name, SIZE(events),
SIZE(releases) AS nbrReleases
ORDER BY nbrReleases DESC
```

Query 12. Compute the number of artists who released a record and performed in at least an event, and the year(s) this happened.

```
MATCH (a:Artist)<-[PERFORMED_BY]-(e:EventFact)-[:HAPPENED_ON]->(d:Date)
WITH distinct a,d.year as year
MATCH (r:ReleaseFact)-[r1:RELEASED_BY]->(a2:ArtistCredit)-[]->(a),
(r)-[r2:RELEASED_ON]->(d1:Date) WHERE d1.year=year
RETURN DISTINCT a.name, year
```

Exercise 3.

We will query the Flanders river system depicted in Figure 1. The schema and properties are shown in Figures 2 to 4. Segments are represented as nodes, with label :Segment (and their corresponding properties), and the relation between the nodes is called :flowsTo, defined as follows: there is a relation :flowsTo from node A to node B if the water flows to segment B from segment A. This is stored in the **rivers database.**

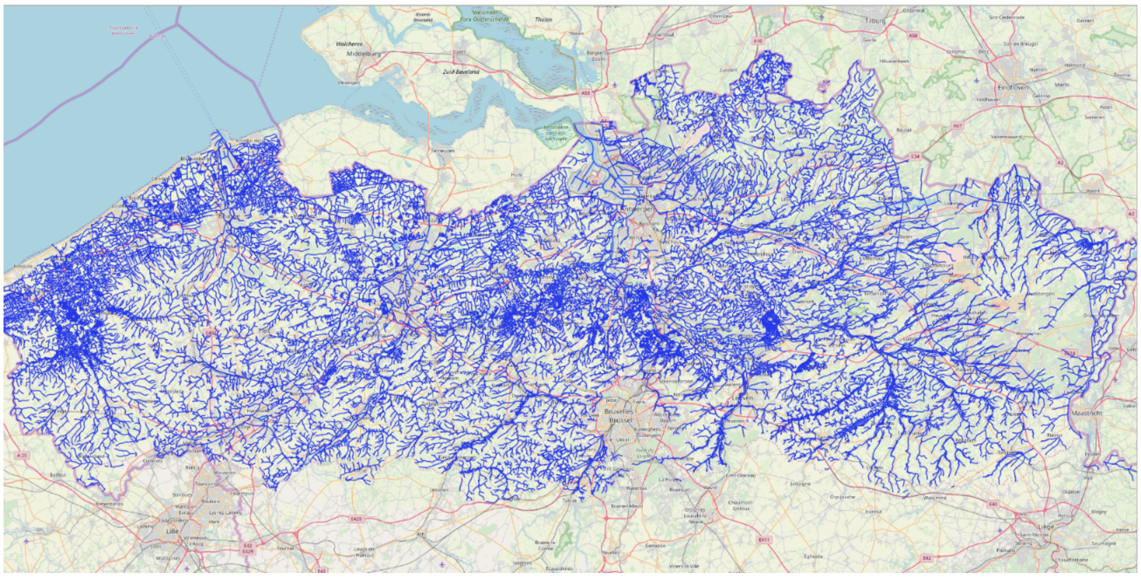
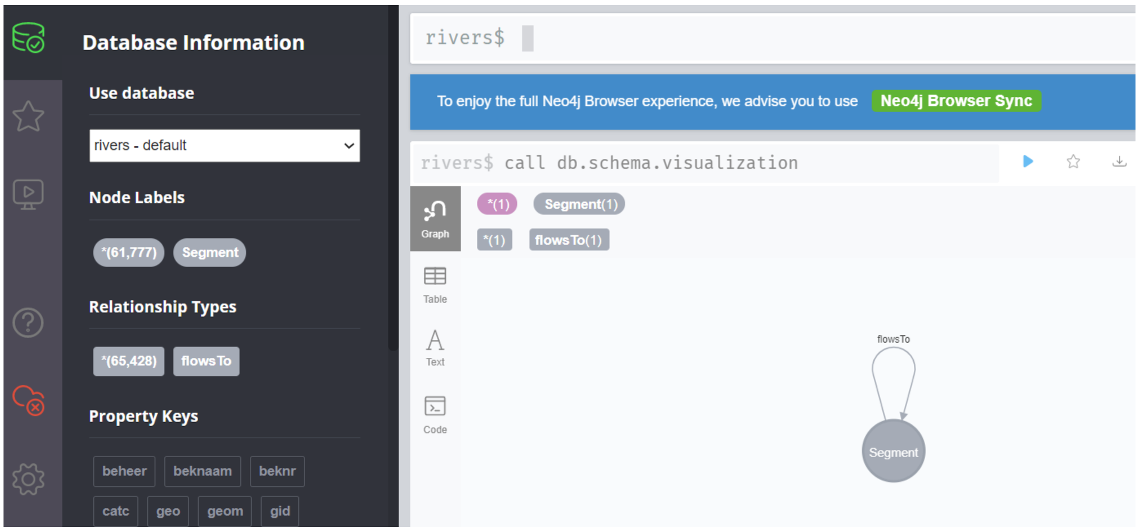


Figure 1



Database Information

Use database

rivers - default

Node Labels

*(61,777) Segment

Relationship Types

*(65,428) flowsTo

Property Keys

beheer beknaam beknr catc geo geom gid

```
rivers$
```

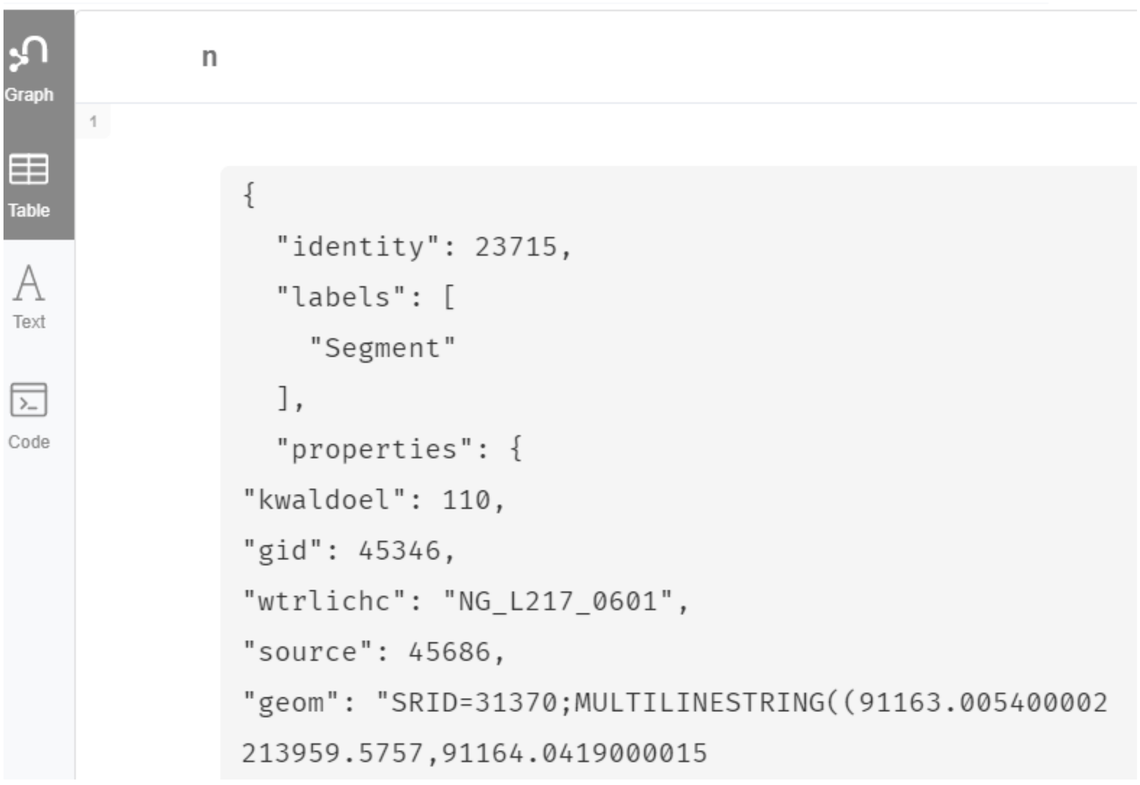
To enjoy the full Neo4j Browser experience, we advise you to use Neo4j Browser Sync

```
rivers$ call db.schema.visualization
```

*(1) Segment(1)

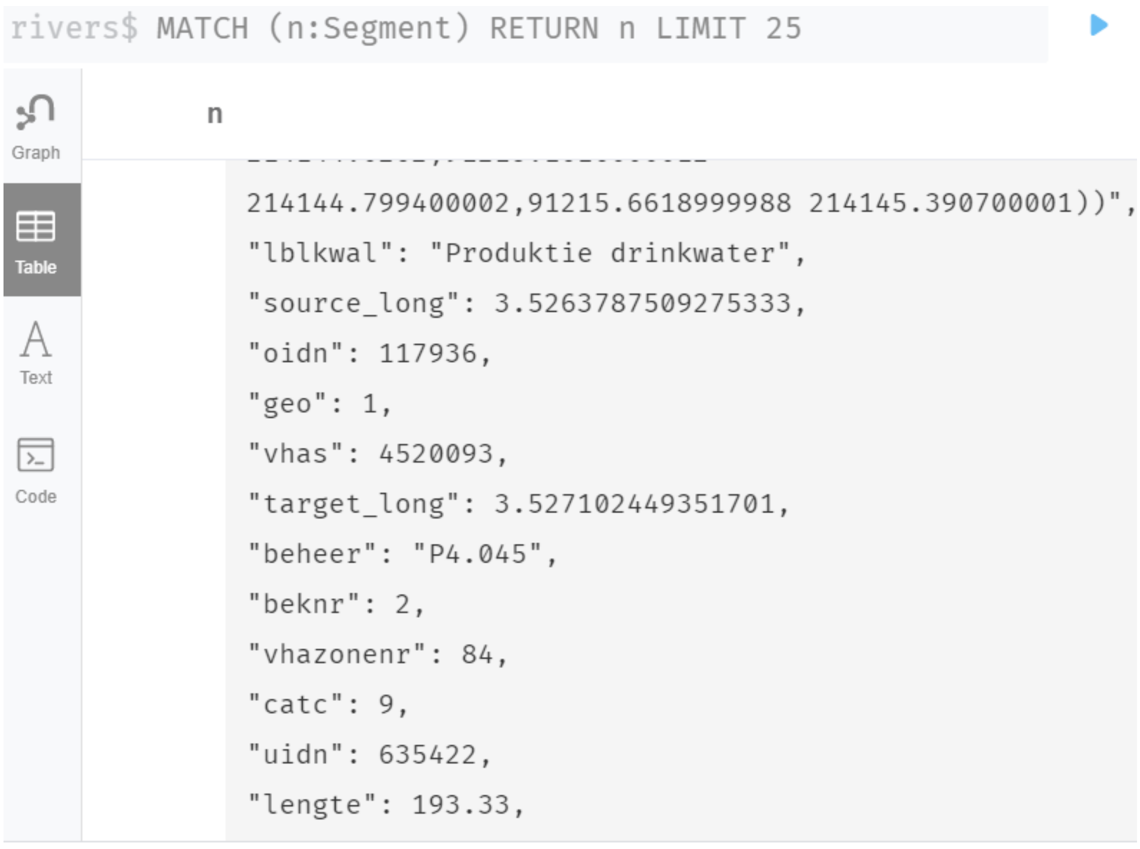
*(1) flowsTo(1)

Figure 2. Schema



```
{
  "identity": 23715,
  "labels": [
    "Segment"
  ],
  "properties": {
"kwaldoel": 110,
"gid": 45346,
"wtrlichc": "NG_L217_0601",
"source": 45686,
"geom": "SRID=31370;MULTILINESTRING((91163.005400002
213959.5757,91164.0419000015
```

Figure 3. Properties



```
rivers$ MATCH (n:Segment) RETURN n LIMIT 25
```

```
214144.799400002,91215.6618999988 214145.390700001))",
"lblkwal": "Produktie drinkwater",
"source_long": 3.5263787509275333,
"oidn": 117936,
"geo": 1,
"vhas": 4520093,
"target_long": 3.527102449351701,
"beheer": "P4.045",
"beknr": 2,
"vhazonenr": 84,
"catc": 9,
"uidn": 635422,
"lengte": 193.33,
```

Figure 4. Properties

Property vhas is the segment iD.
Property lengte is the length of the river
Property geom is the geometry of the segment
Property catc is the category of the segment

Query 1. Compute the average segment length.

```
MATCH (n:Segment)
RETURN  avg(n.lengte) AS avglength
```

Query 2. Compute the average segment length by segment category

```
MATCH (n:Segment)
RETURN  n.catc as category, avg(n.lengte)
AS avglength order by category asc
```

Query 3. Find all segments that have a length within a 10% margin of the length of segment with ID 6020612.

```
MATCH (n:Segment {vhas:6020612})
WITH n.lengte as length
MATCH (m:Segment)
WHERE m.lengte < length*1.1 and m.lengte > length*0.9
RETURN m.vhas, m.lengte;
```

Query 4. For each segment find the number of incoming and outgoing segments.

```
MATCH (src:Segment)-[:flowsTo]->(n:Segment)-[:flowsTo]->(target:Segment)
RETURN n.vhas as nodenbr, COUNT(DISTINCT src) as segIn,
COUNT (DISTINCT target) as segOut
```

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]
```

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)
```

Here, the **spanningTree** function from the APOC library is used. This function computes all simple paths that can be reached starting from a node in the graph, using breath-first search by default. This is done visiting nodes only once. The **relationshipfilter** is “flowsTo >"}, indicating that the path must traverse only this relation, in downstream direction. The function can be parametrised in many ways, for example, indicating the minimum and maximum levels in the path (here, the latter is omitted). A collection of paths is returned (pp), which is then flattened as a table with UNWIND statement. All reachable nodes are obtained. For each node in this table, it is tested if this node has more than one outgoing segments. If this is the case, there is a split. The node with vhas:6020612 is chosen for the test because it is one of the farthest from the sea, thus its flow downstream is one of the longest ones.

Query 7. Find the number of in-flowing segments 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
WITH  [p in NODES(pp) | p.vhas] as ids
UNWIND ids as id
WITH collect(DISTINCT id) as ids
MATCH (s:Segment)-[:flowsTo]->(p)
WHERE NOT s.vhas in ids AND p.vhas <> 6020612
    AND p.vhas in ids
RETURN count(DISTINCT s) as inflows
```

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
```

Query 9. Find the length, the # of segments, and the IDs of the segments, of the longest branch of upstream flow starting from a given segment.

```
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n,{relationshipFilter:"<flowsTo", minLevel: 1}) YIELD path AS pp
WITH reduce(longi= tofloat(0), n IN nodes(pp)| longi+ tofloat(n.lengte)) AS blength,
Length(pp) as alength, [p in NODES(pp) |p.vhas] AS nodelist
WITH blength, alength, nodelist[size(nodelist)-1] as id
WITH id, max(blength) as ml,  collect([id,blength,alength]) as coll
WITH id, ml, [p in coll WHERE p[0]= id  AND p[1]=ml|p[2]] AS lhops
UNWIND lhops as hops
RETURN id,ml,hops order by id desc;
```

Query 10. How many paths exist between two given segments X and Y?

```
MATCH (n:Segment {vhas:6020612}),(m:Segment {vhas: 7036554})
 CALL apoc.path.expandConfig(n,{relationshipFilter:"<flowsTo", minLevel: 1,
terminatorNodes:[m]}) yield path as pp
 RETURN count(pp) as paths
```

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,distance(point({longitude:n.source_long, latitude:n.source_lat}),
point({longitude:ini.longitude, latitude:ini.latitude}) ) 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;
 ist)
```