Neo4j

Advanced Topics on NoSQL databases

A4 - S8

ESILV
nicolas.travers (at) devinci.fr

Chapter 1

Neo4j

Neo4j is a graph oriented database developped in Java. It allows to store highly connected data and to apply patterns on the graph.

1.1 Install

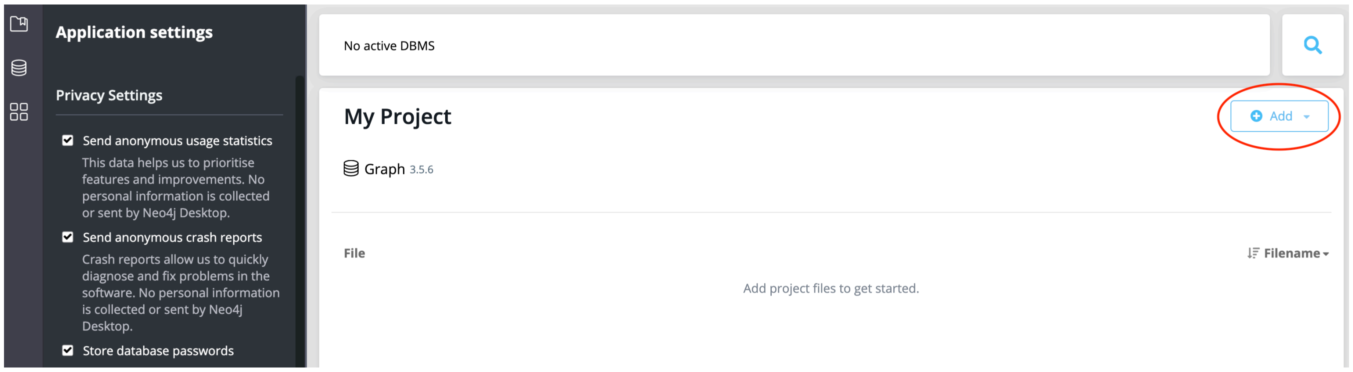
1.1.1 Neo4j Desktop

Download the `https://neo4j.com/download-neo4j-now/`, keep the activation key for the registration of the software.

Or more simply: `https://neo4j.com/download/` (but without activation key).

Once Neo4j Desktop is launched:

- Create a new project for your local DBMS and choose a folder to store data. ⚠Do not forget your password!



Application settings

Privacy Settings

- ☑ Send anonymous usage statistics
  This data helps us to prioritise features and improvements. No personal information is collected or sent by Neo4j Desktop.
- ☑ Send anonymous crash reports
  Crash reports allow us to quickly diagnose and fix problems in the software. No personal information is collected or sent by Neo4j Desktop.
- ☑ Store database passwords

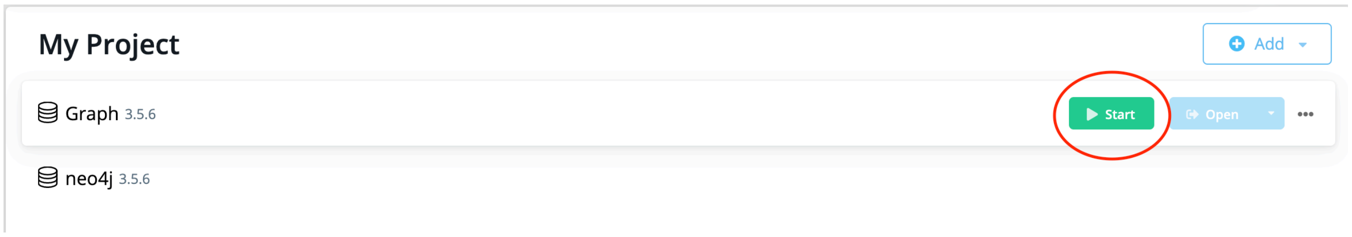
No active DBMS

My Project

Graph 3.5.6

Add project files to get started.

- Wait until all folders and config files are prepared,
- Start your database:

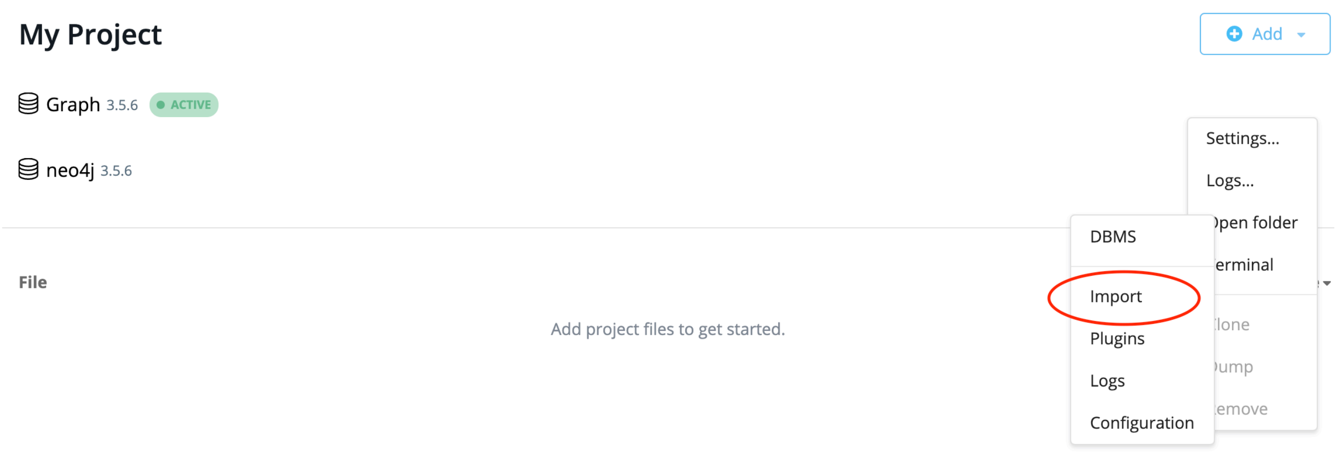


My Project

Graph 3.5.6

neo4j 3.5.6

- Import your CSV files in the "*import folder*" (see figure below).



The dataset is available online and import scripts on the practice work PDF.

1.1.2 Docker

If Neo4j Desktop does not work, as usual, you can download the official Docker image:

```
docker pull neo4j
```

The redirection ports are: 7473, 7474, 7687.
For the Neo4j browser and Bolt protocols.

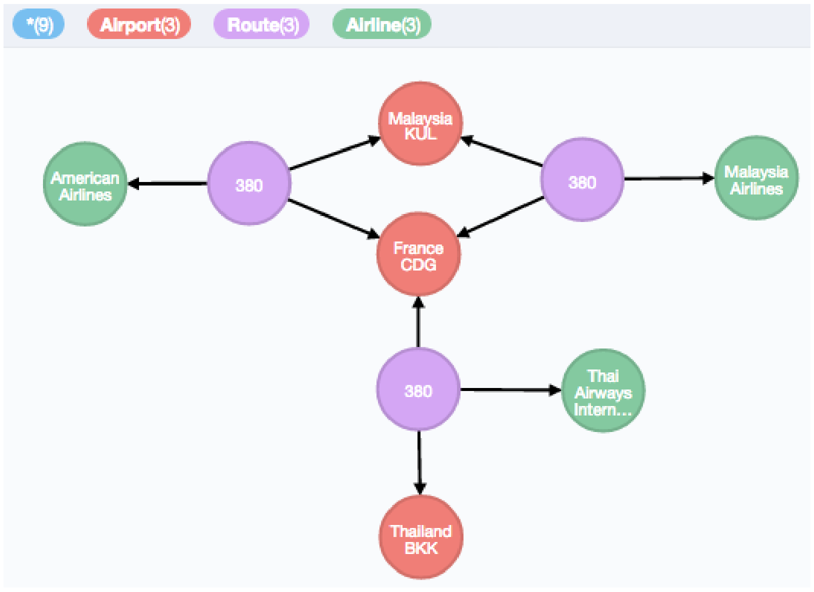
1.1.3 User Interface

You can click the "open" button to have the web browser User Interface. It is usually available on "`http://localhost:7474`"
⚠The port can be checked in the parameters of the database.

Chapter 2 Cypher: Pattern queries on graphs

`Cypher` is a query language for graphs and helps to define patterns which express nodes and relationships between nodes. To query your database, you can use the web interface embedded in Neo4j: `http://localhost:7474`

Our dataset contains `Aiport`, `Airline` and `Route` nodes. The `Route` connects `Airport` and `Airline` with directed relationships (*from Route to other nodes*). Every pattern queries must respect this orientations on relationships in order to produce a result. Here a sample of the graph we have in the database:



The queryable properties are:

- Node / `Airport` : id, name, IATA, country, TimeZone, city, latitude, longitude, altitude
- Node / `Airline` : id, name, country, IATA, alias, active
- Node / `Route` : equipment
- Relationship / Route -[:`from`]-> Airport
- Relationship / Route -[:`to`]-> Airport
- Relationship / Route -[:`by`]-> Airline

2.1 Simple queries

2.1.0.1 Give French Airports' name and IATA code,
Correction :

```
MATCH (a:Airport{country:"France"})
RETURN a.name, a.IATA
```

2.1.0.2 Give names and IATA codes of French Airline companies only when the IATA code exists and when it is an active Airline,
Correction :

```
MATCH (a:Airline{country:"France"})
WHERE EXISTS(a.IATA) AND a.active = "Y"
RETURN a.name, a.IATA
```

2.1.0.3 Names of French Airlines with at least one existing route,
Correction :

```
MATCH (a:Airline{country:"France"}) <-- (r:Route)
RETURN DISTINCT a.name
```

2.1.0.4 Graph[1] of routes which departure is Charles de Gaulle (CDG),
Correction :

```
MATCH (cdg:Airport{IATA:"CDG"}) <-[f:from]- (r:Route)
RETURN cdg, f, r
```

2.1.0.5 Graph of routes from CDG delivered by a A380 (equipment),
Correction :

```
MATCH (cdg:Airport{IATA:"CDG"}) <-[f:from]- (r:Route)
WHERE r.equipment CONTAINS "380"
RETURN cdg, f, r
```

2.1.0.6 Cities and Countries which are the destinations of routes from CDG delivered by an A380,
Correction :

```
MATCH (cdg:Airport{IATA:"CDG"}) <-[:from]- (r:Route) -[:to]-> (dest:Airport)
WHERE r.equipment CONTAINS "380"
RETURN dest.country, dest.city
```

2.1.0.7 From the previous result, give the corresponding airlines name,
Correction :

```
MATCH (cdg:Airport{IATA:"CDG"}) <-[:from]- (r:Route) -[:to]-> (dest:Airport),
        (r) -[:by]-> (comp)
WHERE r.equipment CONTAINS "380"
RETURN dest.country, dest.city, comp.name
```

2.1.0.8 Graph of routes from CDG to any French airport,
Correction : We can refer the total path by a variable

```
MATCH p=(:Airport{IATA:"CDG"}) <-[:from]- (:Route) --> (:Airport{country:"France"})
RETURN p
```

2.1.0.9 Graph of all routes delivered by an A380,
Correction :

```
MATCH p=(:Airport) <-- (r:Route) --> (:Airport)
WHERE r.equipment CONTAINS "380"
RETURN p
```

[1] give the corresponding nodes and the relationships

2.1.0.10 Graph of all routes coming from a French airport to British airport (United Kingdom),
Correction :

```
MATCH p=(FR:Airport{country:"France"}) <-[:from]- (r:Route) -[:to]->
        (UK:Airport{country:"United Kingdom"})
RETURN p
```

2.1.0.11 From previous result, give the distinct list of airline names,
Correction :

```
MATCH (FR:Airport{country:"France"}) <-[:from]- (r:Route) -[:to]->
        (UK:Airport{country:"United Kingdom"}),
        (r) -[:by]-> (comp)
RETURN DISTINCT comp.name
```

2.1.0.12 Idem, but only for which they are delivered by an A320.
Correction :

```
MATCH (FR:Airport{country:"France"}) <-[:from]- (r:Route) -[:to]->
        (UK:Airport{country:"United Kingdom"}),
        (r) -[:by]-> (comp)
WHERE r.equipment CONTAINS "320"
RETURN DISTINCT comp.name
```

2.2 Complex queries

2.2.0.1 To make more complex queries, we need first to create homogeneous relationships between airports in order to make ”jumps”. We will create new relationships which labels are airlines name. For this, use the following queries:

```
MATCH (FROM:Airport) <-[:from]- (r:Route) -[:to]-> (TO:Airport),
        (r) -[:by]-> (comp)
WHERE FROM <> TO
MERGE (FROM)-[:path{airline:comp.name}]-> (TO)
```

```
CREATE INDEX ON :path(airline);
```

2.2.0.2 Give the graph of paths delivered by ”Air France” between all French airports,
Correction :

```
MATCH (from:Airport{country:"France"}) -[p:path{airline:"Air France"}]->
        (to:Airport{country:"France"})
RETURN from, to, p
```

2.2.0.3 Give per destination country the number of paths delivered by ”Air France”. Sort the result decreasingly,
Correction :

```
MATCH (from:Airport) -[:path{airline:"Air France"}]-> (to:Airport)
RETURN to.country, count(*) AS NB
ORDER BY NB DESC
```

2.2.0.4 Idem, but when the path does not come from a French airport,
Correction :

```
MATCH (from:Airport) -[:path{airline:"Air France"}]-> (to)
WHERE from.country <> "France"
RETURN to.country,count(*) AS NB
ORDER BY NB DESC
```

2.2.0.5 Give paths of lengths 2 or 3 (number of relationships) from *Nantes* to *Salt Lake City*,
Correction :

```
MATCH p=(:Airport{city:"Nantes"}) -[:path*2..3]-> (:Airport{city:"Salt Lake City"})
RETURN p
```

2.2.0.6 Give the shortest path from *Nantes* to *Salt Lake City*,
Correction :

```
MATCH p=shortestpath( (FROM:Airport{city:"Nantes"}) -[:path*]->
        (TO:Airport{city:"Salt Lake City"}) )
RETURN p;
```

2.3 Hard queries

2.3.0.1 Give paths of lengths 2 or 3 from *Nantes* to *Salt Lake City*, delivered only by ”Air France”,
Correction :

```
MATCH p=(:Airport{city:"Nantes"}) -[:path*2..3]-> (:Airport{city:"Salt Lake City"})
WHERE ALL(path in relationships(p) WHERE path.airline="Air France")
RETURN p
```

2.3.0.2 All paths of length 2 from Paris only delivered by ”Air France” (without direct flights),
Correction :

```
MATCH p=(paris:Airport{city:"Paris"}) -[:path*2]-> (to:Airport)
WHERE ALL(path in relationships(p) WHERE path.airline="Air France")
        AND NOT (paris) -[:path]-> (to)
RETURN p
```

2.3.0.3 For those destinations, give per country the number of paths sorted decreasingly,
Correction :

```
MATCH p=(paris:Airport{city:"Paris"}) -[:path*2]-> (dest:Airport)
WHERE ALL(path in relationships(p) WHERE path.airline="Air France")
        AND NOT EXISTS((paris) -[:path]-> (dest))
RETURN dest.country, COUNT(*) AS NB
ORDER BY NB DESC
```

2.3.0.4 From question 2.3.2, give only those which stops at least once in the United states,
Correction :

```
MATCH p=(paris:Airport{city:"Paris"}) -[:path*2]-> (dest:Airport)
WHERE ALL(path in relationships(p) WHERE path.airline="Air France")
        AND NOT EXISTS((paris) --> (dest))
        AND ANY(n in nodes(p) WHERE n.country="United States")
RETURN p
```

2.4 Execution plan

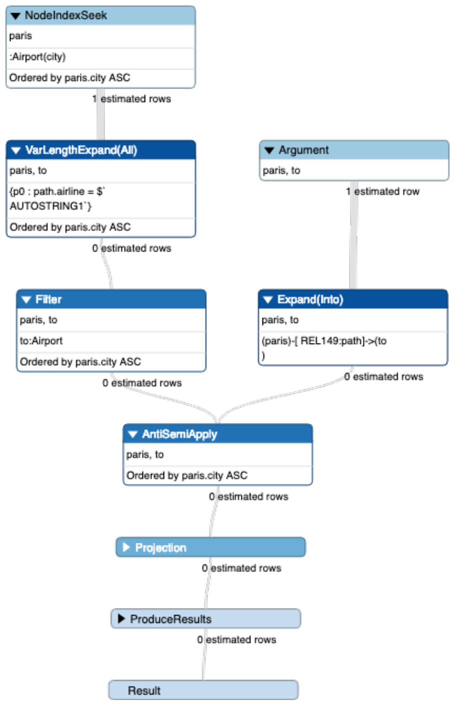
We wish to extract execution plans from each query in order to see if indexes where properly used. To achieve this, prefix your query with ”EXPLAIN”.

2.4.0.1 From query 2.3.2 show the corresponding execution plan,
Correction :

```
EXPLAIN
MATCH p=(paris:Airport{city:"Paris"}) -[:path*2]-> (to:Airport)
WHERE ALL(path in relationships(p) WHERE path.airline="Air France")
AND NOT (paris) -[:path]-> (to)
RETURN p
```

In the following plan, we can see:

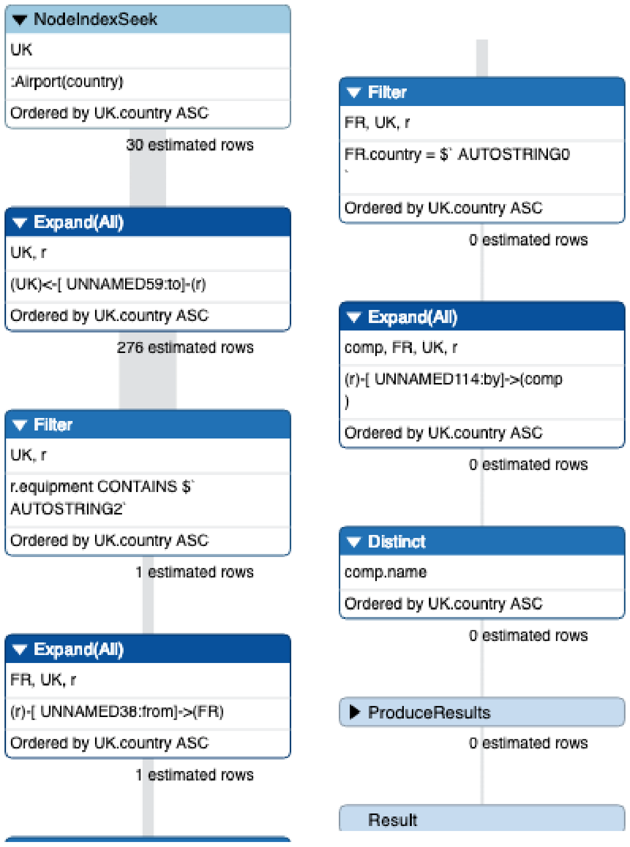
- On top of the plan a ”NodeIndexSeek” which scan the index for Airport nodes with a city label whose value is ”Paris”.
- Then, all paths with a filter on the airline ($ means the input value ”Air France”) with a given path length (here 2)
- The right hand-side of the plan gives the ”NOT” filter on direct paths from Paris. This branch ends with the ”AntiSemiApply” which corresponds to the ”NOT”.
- Then, project the output and produces the results.



2.4.0.2 In the following query, a filter is put either on nodes FR and UK which both refer to the index on countries. From which nodes the execution plan begins? When the other side of the path is dealt?

```
EXPLAIN
MATCH (FR:Airport{country:"France"}) <-[:from]- (r:Route) -[:to]->
        (UK:Airport{country:"United Kingdom"}),
        (r) -[:by]-> (comp)
WHERE r.equipment CONTAINS "320"
RETURN DISTINCT comp.name
```

Correction : We can see that **1)** we begin with the UK node, **2)** expand to the routes node, **3)** filter with "320", **4)** expand to from nodes, **5)** filter them by country name (France), **6)** get companies' name from corresponding patterns, **7)** produce distinct values of companies.



```
▼ NodeIndexSeek
UK
:Airport(country)
Ordered by UK.country ASC
                30 estimated rows

▼ Expand(All)
UK, r
(UK)<-[ UNNAMED59:to]-(r)
Ordered by UK.country ASC
                276 estimated rows

▼ Filter
UK, r
r.equipment CONTAINS $`
AUTOSTRING2`
Ordered by UK.country ASC
                1 estimated rows

▼ Expand(All)
FR, UK, r
(r)-[ UNNAMED38:from]->(FR)
Ordered by UK.country ASC
                1 estimated rows

▼ Filter
FR, UK, r
FR.country = $` AUTOSTRING0
`
Ordered by UK.country ASC
                0 estimated rows

▼ Expand(All)
comp, FR, UK, r
(r)-[ UNNAMED114:by]->(comp
)
Ordered by UK.country ASC
                0 estimated rows

▼ Distinct
comp.name
Ordered by UK.country ASC
                0 estimated rows

▶ ProduceResults
                0 estimated rows

Result
```

Chapter 3
Bonus/Hard queries/New features

3.1 GDS - Graph Data Science

You can use advanced graph algorithms with Neo4j, for that, you must use the GDS library[1]. Types of algorithms:

- Community detection (Louvain, LabelProp) - graph clustering
- Centrality (PageRank, Betweenness, etc.)
- Similarity (Jaccard, topology)
- Link prediction
- Pathfinding
- node embedding

You first need to create a Cypher Projection (temporary grap):

```
CALL gds.graph.create("tmpGraphName", "inputNodes", {links properties}, {nodes properties})
```

And then, apply a library, here Louvain, on that temporary graph:

```
CALL gds.louvain.stream('tmpGraphName')
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds
ORDER BY name ASC
```

3.2 Add an external library

It is possible to add a Java library in Neo4j which can be called in Cypher.

3.2.1 Import library

To import a package/library in Neo4j, it must be put in the "`$NEO4J_FOLDER/plugins/`" folder.[2].

Then, you need to configure your Neo4j server in order to integrate the Java Class to be called. The config file is available here: `$NEO4J_FOLDER/conf/neo4j.conf`.

Add the following line in "neo4j.conf":

```
dbms.unmanaged_extension_classes=XXXX.extension=/YYYY
```

For which XXXX is the plugin's name and YYYY the running class' name in manifest of the plugin.

3.2.2 Call the library

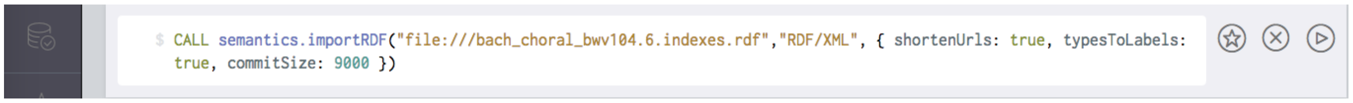
You can call the library thanks to this command line: `CALL XXXX.<function>(<params>)`.

Here is an example of the RDF reading package.

You can also find:

[1] https://neo4j.com/product/graph-data-science-library/

[2] linux: /var/lib/neoj4, windows: C:
Program Files
Neo4j



```
$ CALL semantics.importRDF("file:///bach_choral_bwv104.6.indexes.rdf","RDF/XML", { shortenUrls: true, typesToLabels:
true, commitSize: 9000 })
```

- APOC: JSON and graph procedures
- Graph algorithms: `https://github.com/neo4j-contrib/neo4j-graph-algorithms/releases`
- NeoSemantics: `https://github.com/jbarrasa/neosemantics`

3.3 Change graph renderer

You can render your graph with your own colors and labels. To achieve this, you need to define a CSS stylesheet, extended with 'grass' information. Use the command line ":`style`" to apply it. Here is an example:

```
:style node {diameter: 50px;color: #A5ABB6;border-color: #9AA1AC;border-width:
↪ 2px;text-color-internal: #FFFFFF;font-size: 10px;} relationship {color:
↪ #A5ABB6;shaft-width: 1px;font-size: 8px;padding: 3px;text-color-external:
↪ #000000;text-color-internal: #FFFFFF;caption: '<type>';} node.Airport {color:
↪ #FF756E;border-color: #E06760;text-color-internal: #FFFFFF;caption: '{country}
↪ {IATA}';} node.Airline {color: #DE9BF9;border-color: #BF85D6;text-color-internal:
↪ #FFFFFF;caption: '{name}';} node.Route {color: #FB95AF;border-color:
↪ #E0849B;text-color-internal: #FFFFFF;caption: '{equipment}';}
```

⚠The order between node styles are important since each of them overloads previous styles.
You can edit:

- `diameter`: nodes size,
- `color/border-color/text-color-internal`: colors,
- `caption`: Nodes/relationships name. Take properties with "<type>" or "{ATTRIBUTE}".