Neo4J dans la pratique (TP2)

1. Préalable Neo4J

1.1 Version Neo4J autre que celle déposée sur moodle

Avant de démarrer le serveur, il faut modifier le fichier neo4j.conf (dans répertoire conf) et ajouter deux instructions (si vous ne travaillez pas avec la version Neo4J déposée sur moodle).

```
# ajout pour import/export RDF
dbms.unmanaged_extension_classes=semantics.extension=/rdf

# ajout pour autoriser export dans un fichier
apoc.export.file.enabled=true
```

Listing 1 – ajouts dans neo4j.conf

Il faut également disposer les plugins adaptés pour les procédures APOC et le plugin NeoSemantics. Pour la version 3.5.21, il s'agit des archives :

```
apoc-3.5.0.14-all.jar
neosemantics-3.5.0.4.jar
```

Listing 2 – archives utiles

1.2 Mise en route

Vous démarrerez ensuite le serveur :

```
... ./bin/neo4j start
```

Listing 3 – ordre de mise en route du serveur

2. Enrichissement du modèle

Les derniers maires successifs de Montpellier sont à ajouter au modèle

Créer les objets Personne en relation avec la commune de Montpellier (déjà présente dans la base) suivants

```
MATCH (c:Commune {name:'MONTPELLIER'})
CREATE (gf:Personne {nom:"FRECHE",prenom:"GEORGES"}) <-[ap1:ADMINISTREE_PAR
    {date_debut:1997, date_fin:2004}]- (c),
```

```
(hm:Personne {nom:"MANDROUX",prenom:"HELENE"})<-[ap2:ADMINISTREE_PAR {date_debut:2004,
    date_fin:2014}]-(c), (ps:Personne
    {nom:"SAUREL",prenom:"PHILIPPE"})<-[ap3:ADMINISTREE_PAR {date_debut:2014,
    date_fin:2020}]-(c), (md:Personne
    {nom:"DELAFOSSE",prenom:"MICKAEL"})<-[ap4:ADMINISTREE_PAR {date_debut:2020,
    date_fin:2026}]-(c)
return *
```

Listing 4 – Les maires de Montpellier

2.1 Exercices après enrichissement

1. définir l'ordre Cypher qui permet d'ajouter un label Maire aux objets de label Personne, qui sont associés à un objet de label Commune, au travers d'une relation de type ADMINISTREE_PAR

```
MATCH (p:Personne) <-[:ADMINISTREE_PAR]- (c:Commune) SET p:Maire
```

Listing 5 – Correction

2. lister l'ensemble des procédures rendues disponibles grâce à l'ajout d'archives Java dans le répertoire plugins. Lister aussi l'ensemble des fonctions.

```
CALL dbms.procedures()
CALL dbms.functions()
CALL db.schema()
```

Listing 6 – Correction

3. utiliser une de ces procédures du paquetage d'extension général (préfixe apoc) pour exporter une partie des objets du graphe au format json (apoc.export.json.query). Le fichier d'export est automatiquement sauvegardé dans le répertoire **import**. Vous renverrez l'identifiant, les labels et le nom de la commune, ainsi que le non de son département et de sa région.
   Quels sont les autres formats disponibles ?

```
call apoc.export.json.query("MATCH (a:Commune) -[w1:WITHIN]-> (d:Departement)
    -[w2:WITHIN]-> (r:Region)
Return id(a), labels(a), a.name, d.name, r.name", "test.json", {} )

les formats csv ou graphml (xml schema pour structures graphes) par exemple
les appels seront de type :
 call apoc.export.csv.query ....
 ou encore
call apoc.export.graphml.all("test.xml", {} ) pour sauvegarder un fichier au
    format graphml et pouvoir ensuite l'ouvrir avec un autre editeur de graphe
    (gephi, cytoscape)
```

Listing 7 – Correction

2.1.1 Utilisation du plugin Neosemantics

Neosemantics permet d'exploiter Neo4J à la manière d'un triplestore. Nous en explorons quelques fonctionnalités élémentaires.

1. vous renverrez le modèle de connaissances de la base au format RDF. Vous pouvez dessiner sous forme de graphe une partie du résultat. Que remarquez vous comme différence avec ce que vous savez contenir le modèle ?

```
absence de toutes les proprietes concretes/litterales (attributs) des noeuds comme
    des relations du graphe neo4j. Seules des proprietes rdfs:label sont ajoutees
    et pointent vers des valeurs litterales

:GET /rdf/onto
```

Listing 8 – Correction

2. Vous renverrez au format RDF, la description du noeud correspondant à la commune de MONTPELLIER à partir de son identifiant interne. Vous pouvez dessiner sous forme de graphe une partie du résultat. Que remarquez vous par rapport à la question précédente ?

```
les proprietes litterales sont retrouvees mais parfois sans type de donnees

MATCH (m:Commune {name:'MONTPELLIER'}) return ID(m) pour avoir l'ID
:GET /rdf/describe/id/37
```

Listing 9 – Correction

3. Renvoyez le résultat de la requête suivante au format RDF

```
MATCH (c:Commune {name:'MONTPELLIER'}) RETURN c
```

Listing 10 – Requête

Est ce que ce résultat est équivalent au résultat de la description du noeud de la commune de MONTPELLIER ?

```
il faut passer par un ordre POST, requete structuree JSON, ici juste les
    proprietes litterales
:POST /rdf/cypher { "cypher":"MATCH (c:Commune {name:'MONTPELLIER'}) RETURN c" ,
    "format" : "N3"}
```

Listing 11 – Correction

4. Renvoyez les informations sur MONTPELLIER et ses différents maires au format RDF

```
:POST /rdf/cypher { "cypher":"MATCH path = (c:Commune
    {name:'MONTPELLIER'})-[ap1:ADMINISTREE_PAR]-> (p:Personne) RETURN path " ,
    "format" : "N3"}

POST /rdf/cypher { "cypher":"MATCH (c:Commune {name:'MONTPELLIER'})
    -[ADMINISTREEPAR]-> (m:Maire) RETURN *" , "format" : "N3"}
```

Listing 12 – Correction

5. Enrichissez l'ordre précédent pour renvoyer le plus d'informations possibles sur MONTPELLIER

```
:POST /rdf/cypher { "cypher":"MATCH path = (c:Commune
    {name:'MONTPELLIER'})-[n:NEARBY]- (co:Commune) RETURN path " , "format" : "N3"}
:POST /rdf/cypher { "cypher":"MATCH path = (r:Region) <-[w2:WITHIN]-
    (d:Departement) <-[w1:WITHIN]- (c:Commune
    {name:'MONTPELLIER'})-[ap1:ADMINISTREE_PAR]-> (p:Personne), (c) -[:NEARBY]-
    (c1:Commune) RETURN path " , "format" : "N3"}
:POST /rdf/cypher { "cypher":"MATCH (r:Region) <-[w2:WITHIN]- (d:Departement)
    <-[w1:WITHIN]- (c:Commune {name:'MONTPELLIER'})-[ap1:ADMINISTREE_PAR]->
    (p:Personne), (c) -[:NEARBY]- (c1:Commune) RETURN * " , "format" : "N3"}
```

```
tout n'est possible : la requete suivante donne un resultat aberrant
:POST /rdf/cypher { "cypher":"MATCH path = (r:Region) <-[w2:WITHIN]-
    (d:Departement) <-[w1:WITHIN]- (c:Commune
    {name:'MONTPELLIER'})-[ap1:ADMINISTREE_PAR]-> (p:Personne), (c) -[:NEARBY]-
    (c1:Commune) RETURN path " , "format" : "N3"}
```

Listing 13 – Correction

6. Comment faire pour renvoyer les informations qui correspondent aux propriétés valuées de la relation ADMINISTREE_PAR au format RDF ?

```
l'information est juste ignoree par neosemantics, du coup il faut creer un noeud
    intermediaire ici municipalite
:POST /rdf/cypher { "cypher":"MATCH (c:Commune {name:'MONTPELLIER'})
    -[ap1:ADMINISTREE_PAR]-> (p:Personne) MERGE (c)-[:gouvernance]->
    (m:Municipalite {dateDeb:ap1.date_debut, dateFin:ap1.date_fin})
    <-[:dirige]-(p) RETURN c, m, p" , "format" : "N3"}

:POST /rdf/cypher { "cypher":"MATCH path = (c:Commune {name:'MONTPELLIER'})
    -[ap1:ADMINISTREE_PAR]-> (p:Personne) MERGE (c)-[g:gouvernance]->
    (m:Municipalite {dateDeb:ap1.date_debut, dateFin:ap1.date_fin})
    <-[d:dirige]-(p) RETURN path" , "format" : "N3"}
```

Listing 14 – Correction

3. Import de données au format RDF

Il est possible d'importer au sein d'une base de données Neo4J, des triplets provenant de points d'accès SPARQL. Ici une requête SPARQL exploitant une des procédures du plugin neosemantics (semantics.importRDF) vous est donnée. Cette requête de type CONSTRUCT exploite le point d'accès SPARQL de Wikidata pour retourner différentes informations sur des communes (ici le concept City est emprunté au vocabulaire schema.org). Auparavant, il faudra créer l'index sur tous les objets "Resource" du graphe (au sens RDF) :

```
CREATE INDEX ON :Resource(uri)
```

Listing 15 – Requête CYPHER/SPARQL

```
WITH ' PREFIX sch: <http://schema.org/>
CONSTRUCT{ ?item a sch:City;
        sch:address ?inseeCode;
        sch:name ?itemLabel ;
        sch:geoTouches ?otherItem .
     ?otherItem a sch:City;
     sch:name ?otheritemLabel ;
     sch:address ?otherinseeCode . }
WHERE { ?item wdt:P374 ?inseeCode .
   ?item wdt:P47 ?otherItem .
   ?otherItem wdt:P374 ?otherinseeCode .
      ?item rdfs:label ?itemLabel .
       filter(lang(?itemLabel) = "fr") .
      ?otherItem rdfs:label ?otheritemLabel .
       filter(lang(?otheritemLabel) = "fr") .
    FILTER regex(?inseeCode, "^34") .
      } limit 400 ' AS sparql CALL semantics.importRDF(
```

```
  "https://query.wikidata.org/sparql?query=" +
     apoc.text.urlencode(sparql),"JSON-LD",
    { headerParams: { Accept: "application/ld+json"} })
YIELD terminationStatus, triplesLoaded,namespaces,extraInfo
RETURN terminationStatus, triplesLoaded,namespaces,extraInfo
```

Listing 16 – Requête CYPHER/SPARQL

Une fois cet import terminé, vous répondrez à une série de questions.

3.1 Questions d'appropriation

1. Expliquer en langage naturel ce que renvoie la requête SPARQL construite. Esquissez un graphe rapide sur deux cités (villes) du résultat de l'import dans Neo4J.
2. Les nœuds importés correspondent à des villes et villages de l'Hérault. Vous ferez en sorte de les lier au nœud correspondant au département de l'Hérault via la relation WITHIN

```
MATCH (ci:sch__City)
MATCH (n:Departement {id:'34'})
CREATE (ci) -[:WITHIN]-> (n)
```

Listing 17 – Correction

3. Vous supprimerez les nœuds de type "Commune" du graphe

```
MATCH (co:Commune) DETACH DELETE co
```

Listing 18 – Correction

4. Renvoyez le nombre de communes limitrophes de la commune de Montpellier

```
MATCH (m:sch__City {sch__name:'Montpellier'})-[:sch__geoTouches]-(x)
RETURN count(x) as limitrophes
```

Listing 19 – Correction

3.2 Questions sur les chemins

1. renvoyez un des plus courts chemins entre Montpellier et Beaulieu

```
MATCH p=shortestpath((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) ) RETURN p
```

Listing 20 – Correction

2. renvoyez tous les plus courts chemins entre Montpellier et Beaulieu

```
MATCH p=allshortestpaths((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) ) RETURN p
```

Listing 21 – Correction

3. renvoyez le nom des cités traversées par un des plus courts chemins entre Montpellier et Beaulieu

```
MATCH p=shortestpath((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) )
RETURN extract(n in nodes(p)|n.sch__name) as noeudsDuChemin
```

Listing 22 – Correction

4. renvoyez le nom et l'adresse des cités traversées par un des plus courts chemins entre Montpellier et Beaulieu

```
match p=shortestpath((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) )
RETURN extract(n IN nodes(p) | {name: n.sch__name, codeInsee: n.sch__address})

match p=shortestpath((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) )
UNWIND nodes(p) as n
RETURN n.sch__name, n.sch__address
```

Listing 23 – Correction

5. renvoyez le nombre des plus courts chemins entre Montpellier et Beaulieu

```
MATCH p=allshortestpaths((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) ) RETURN count(p)
```

Listing 24 – Correction

renvoyez tous les chemins entre Montpellier et Beaulieu (très coûteuse). Que faire pour réduire la complexité ?

```
MATCH p=((m:sch__City {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City
    {sch__name:'Beaulieu'}) )
RETURN p
 create index on :sch__City(sch__name)
 ne pas renvoyer tous les chemins

 ensuite MATCH p=((m:sch__City
     {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City {sch__name:'Beaulieu'})
     ) RETURN p limit 5
 trs rapide
```

Listing 25 – Correction

retourner un des plus courts chemins qui ne passe pas par Clapiers ?

```
match p=shortestpath((m:sch__City
    {sch__name:'Montpellier'})-[:sch__geoTouches*]-(g:sch__City {sch__name:'Beaulieu'}))
where not ('Clapiers' in (extract (n in nodes(p)| n.sch__name))) return p
```

Listing 26 – Correction