

BASES DE DONNÉES NoSQL : NEO4J (GRAPHES)

Philippe Helluy

CONTENTS

1. Objectifs
2. SGBD NoSQL orienté graphe: Neo4j/Cypher
3. Langage de requêtes Cypher
4. Exemple : base vétérinaire en graphe
5. Conclusion

OBJECTIFS

- Les SGBD relationnels (ex : PostgreSQL) utilisent des **tables** liées par des **clés étrangères**.
- Limites :
 - Difficulté pour représenter les **relations complexes**.
 - Coût élevé des **jointures multiples**.
- NoSQL offre des alternatives : clé-valeur, document, **graphe**.

Exemple vétérinaire (SQLite) :

client

id	nom	prenom
1	Dupont	Jean
2	Martin	Claire

animal

id	nom	espece	age
1	Rex	Chien	5
2	Miaou	Chat	3
3	Panpan	Lapin	2

maitre

animal_id	client_id
1	1
2	2
3	2

- Chaque propriétaire peut posséder plusieurs animaux.
- Ces liens sont modélisés par la table `maitre`.
- En graphes : ces relations deviennent naturelles (“nœuds reliés”).

SGBD NoSQL ORIENTÉ GRAPHE: NEO4J/ CYPHER

- Une base graphe stocke des **nœuds** (entités) et des **arêtes** (relations).
- Chaque nœud ou relation peut avoir des **propriétés** (clé/valeur).
- Permet de représenter les relations complexes de manière **naturelle** et **navigable**.

- **Nœuds (nodes)** : objets (Client, Animal)
- **Relations (edges)** : liens entre objets (POSSEDE)
- **Propriétés** : valeurs attachées (nom, espèce, âge...)

Exemple simple :

(Client: Dupont) $\text{--}[:\text{POSSEDE}]\rightarrow$ (Animal: Rex)

“SQL (Relationnel)”	“Graphe (Neo4j)”
• Tables avec clés primaires et étrangères • Schéma fixe	• Nœuds et relations étiquetés • Schéma flexible
• Jointures pour relier les tables	• Relations intégrées, navigation directe
• Exemple : PostgreSQL	• Exemple : Neo4j

- Base de données orientée graphe la plus utilisée.
- Accessible par le langage **Cypher**.
- Supporte les propriétés sur nœuds et relations.
- Implémente les principes **ACID** :
 - Atomicité
 - Cohérence
 - Isolation
 - Durabilité

- **Create** : créer des nœuds et relations
- **Read** : interroger le graphe
- **Update** : modifier les propriétés
- **Delete** : supprimer les nœuds ou relations

```
!apt-get install openjdk-17-jdk-headless -qq
!wget -O neo4j-community.tar.gz 'https://neo4j.com/artifact.php?name=neo4j-
community-5.5.0-unix.tar.gz'
!tar -xf neo4j-community.tar.gz
!neo4j-community-5.5.0/bin/neo4j-admin dbms set-initial-password yourpassword
!neo4j-community-5.5.0/bin/neo4j start
!pip install py2neo
```

```
from py2neo import Graph
graph = Graph("bolt://localhost:7687", auth=("neo4j", "yourpassword"))
graph.run("MATCH (n) DETACH DELETE n")
```

- Connexion via le protocole **Bolt**
- Py2Neo permet d'exécuter des requêtes **Cypher** directement

```
graph.run(\"CREATE (a:Person {name:'Alice'})\")  
graph.run(\"CREATE (b:Person {name:'Bob'})\")  
graph.run(\"CREATE (a) -[:FRIEND] ->(b)\")
```

Graphe obtenu :

Alice → Bob

```
CREATE (jean:Client {nom:'Dupont', prenom:'Jean'})  
CREATE (rex:Animal {nom:'Rex', espece:'Chien', age:5})  
CREATE (jean)-[:POSSEDE]->(rex)
```

Attention:

- les attributs ressemble à du JSON, mais en moins souple: pas d'imbrication possible. Pour créer des structures plus complexes, il faut créer plusieurs nœuds et les relier.
- Si un arc existe déjà entre deux nœuds, une nouvelle relation sera créée.

```
MERGE (jean:Client {nom:'Martin', prenom:'Jean'})  
MERGE (miaou:Animal {nom:'Miaou', espece:'Chat', age:3})  
MERGE (jean)-[:POSSEDE]->(miaou)
```

MERGE ne crée l'élément que s'il n'existe pas.

- Les requêtes Neo4j sont basées sur le **filtrage par motifs** (*pattern matching*) :

```
MATCH (a:Person) - [:FRIEND] -> (b:Person)  
RETURN a.name, b.name
```

- (a:Label) définit un nœud avec étiquette Label
- [r:TYPE] définit une relation de type TYPE

```
MATCH (c:Client)-[:POSSEDE]->(a:Animal)  
RETURN c.nom, a.nom, a.espece
```

Résultat possible :

Client	Animal	Espèce
Dupont	Rex	Chien
Martin	Miaou	Chat

```
MATCH (c:Client)-[:POSSEDE]->(a:Animal)
RETURN a.espece AS espece, COUNT(a) AS nb_animaux
```

Résultat :

Espèce	nb_animaux
Chien	1
Chat	1
Lapin	1

LANGAGE DE REQUÊTES CYPHER

- Cypher est le langage de requête pour Neo4j.
- Il va probablement être retenu comme standard pour les bases de données graphe.
- Il permet de créer, lire, mettre à jour et supprimer des nœuds et des relations.
- Voici quelques informations supplémentaires pour illustrer les possibilités du langage.

Créer un nœud avec propriétés

```
CREATE (p:Person {name: 'Alice', age: 30})
```

Créer une relation entre deux nœuds existants

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})  
CREATE (a)-[:FRIEND]->(b)
```

Trouver tous les nœuds d'un type (label) donné

```
MATCH (p:Person)  
RETURN p
```

Trouver des nœuds avec une condition sur une propriété

```
MATCH (p:Person)  
WHERE p.age > 25  
RETURN p.name, p.age
```

Trouver des relations spécifiques entre nœuds

```
MATCH (a:Person) - [r:FRIEND] -> (b:Person)
RETURN a.name, b.name, type(r)
```

Supprimer une relation

```
MATCH (a:Person) - [r:FRIEND] -> (b:Person)
WHERE a.name = 'Alice' AND b.name = 'Bob'
DELETE r
```

Supprimer un nœud et ses relations

```
MATCH (p:Person {name:'Alice'})
DETACH DELETE p
```

Mettre à jour une propriété d'un nœud

```
MATCH (p:Person {name: 'Alice'})  
SET p.age = 31  
RETURN p
```

Trouver les amis d'amis

```
MATCH (p:Person)-[:FRIEND]->()-[:FRIEND]->(foaf)  
WHERE p.name = 'Alice'  
RETURN foaf.name
```

Compter le nombre de relations par type

```
MATCH ()-[r]->()
```

```
RETURN type(r) AS relationType, count(*) AS count
```

EXEMPLE : BASE VÉTÉRINAIRE EN GRAPHE

Relations à créer :

`(Client) -[:POSSEDE]→ (Animal)`

Création en Cypher :

```
CREATE (dupont:Client {nom:'Dupont', prenom:'Jean'})  
CREATE (rex:Animal {nom:'Rex', espece:'Chien', age:5})  
CREATE (dupont) -[:POSSEDE] -> (rex)
```

```
CREATE (martin:Client {nom:'Martin', prenom:'Claire'})
CREATE (miaou:Animal {nom:'Miaou', espece:'Chat', age:3})
CREATE (panpan:Animal {nom:'Panpan', espece:'Lapin', age:2})
CREATE (martin)-[:POSSEDE]->(miaou)
CREATE (martin)-[:POSSEDE]->(panpan)
```

Trouver tous les animaux et leurs propriétaires :

```
MATCH (c:Client)-[:POSSEDE]->(a:Animal)  
RETURN c.nom, c.prenom, a.nom, a.espece, a.age
```

```
query = \"\"\"
MATCH (a:Person)-[r:FRIEND]->(b:Person)
RETURN a.name AS from, b.name AS to
\"\"\"
edges = graph.run(query).data()
import networkx as nx
import matplotlib.pyplot as plt
G = nx.DiGraph()
for edge in edges:
    G.add_edge(edge['from'], edge['to'])
nx.draw(G, with_labels=True, node_color='lightblue', node_size=2000)
plt.show()
```

CONCLUSION

- Cypher est un langage basé sur le *pattern matching* pour manipuler des graphes dans Neo4j.
- Les exemples ci-dessus couvrent les opérations CRUD, le filtrage, la navigation dans le graphe et les agrégations.
- Ces requêtes sont utiles les pour commencer à travailler avec Neo4j, mais il existe de nombreuses autres fonctionnalités à explorer.

- Requêtes intuitives sur les relations
- Très rapide pour des graphes riches en connexions
- Schéma flexible et évolutif
- Visualisation native des graphes

- Moins adapté aux opérations analytiques massives
- Consommation mémoire importante pour gros graphes