

Guillaume Donizetti

Kevin Mougin

Xavier Martin

Kevin Migliore

Tristan Etesse

Mise en place d'une base de données graphe avec Neo4j et intégration dans une application simple

1. Installation de Neo4j

Mettre en place une base de données Neo4j localement (via Desktop ou Docker) et s'assurer de son bon fonctionnement.

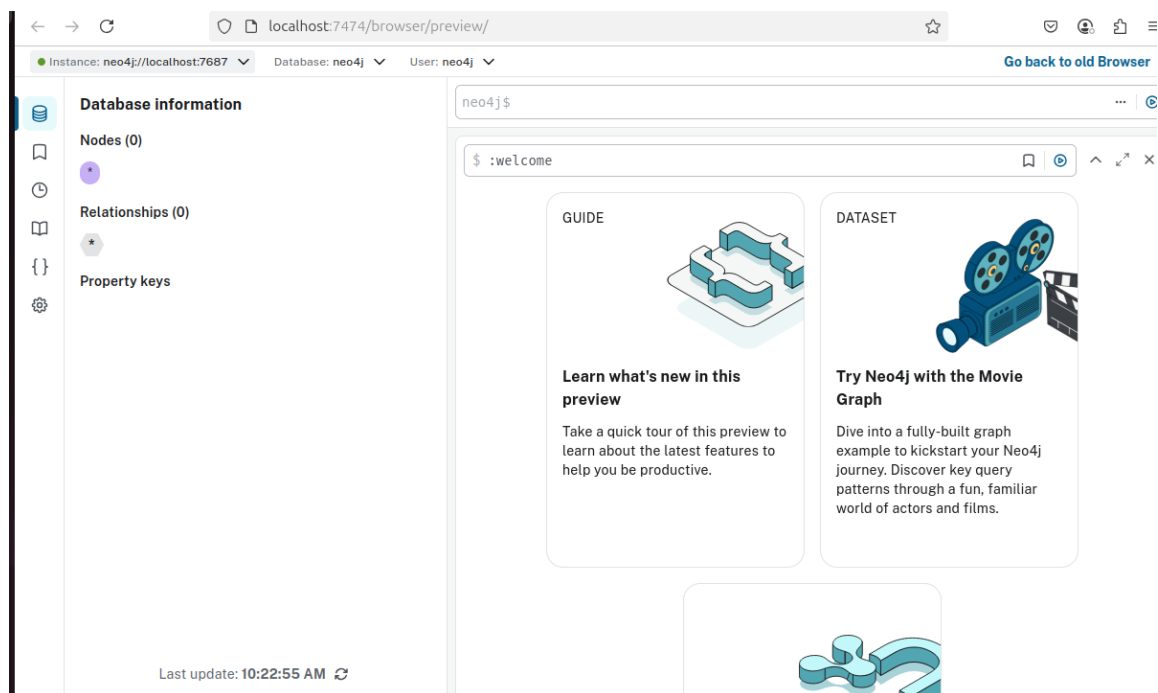
- Téléchargement de l'image Docker de Neo4j 5
- Lancer un conteneur nommé neo4j
- Ouvrir les ports nécessaires :
 - 7474 pour l'interface web
 - 7687 pour le protocole Bolt (utilisé par les drivers)

```
guillaume@guillaume-ubuntu:~$ docker run \
  --name neo4j \
  -p7474:7474 -p7687:7687 \
  -d \
  -e NEO4J_AUTH=neo4j/test \
  neo4j:5
Unable to find image 'neo4j:5' locally
5: Pulling from library/neo4j
cc41ef31545f: Pull complete
92b6a2a9835b: Pull complete
1b93abe8eaad: Pull complete
941327bbe0af: Pull complete
d0de631b0bd6: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:5216b8f7bcca893448d5b4a809d91761609fc8aea900691d59531ada76c28c62
Status: Downloaded newer image for neo4j:5
9014081f4d42c5b0dded819f3636296f8c3799ba33ccc3f7b959351044a85100
guillaume@guillaume-ubuntu:~$
```

Problème rencontré:

```
guillaume@guillaume-ubuntu:~$ docker logs neo4j
Invalid value for password. The minimum password length is 8 characters.
If Neo4j fails to start, you can:
  1) Use a stronger password.
  2) Set configuration dbms.security.auth_minimum_password_length to override the minimum password length requirement.
  3) Set environment variable NEO4J_dbms_security_auth__minimum__password__length to override the minimum password length requirement.
org.neo4j.commandline.admin.security.exception.InvalidPasswordException: A password must be at least 8 characters.
```

Cela n'a pas marché car j'ai mis un mot de passe trop court, il faut 8 caractères minimum. Je recommence avec un mot de passe conforme, cela fonctionne :



2. Modélisation du graphe

Définir un modèle contenant au minimum trois types de nœuds : **Client**, **Commande**, et **Produit**, reliés par des relations telles que **A_EFFECTUÉ** et **CONTIENT**.
Représenter le schéma de manière claire (diagramme ou légende).

(Client) —[:A_EFFECTUÉ]—> (Commande) —[:CONTIENT]—> (Produit)

Explication :

Chaque **Client** passe une ou plusieurs **Commandes**, d'où la relation **A_EFFECTUÉ**.
Chaque **Commande** contient un ou plusieurs **Produits**, ce qu'on représente avec la relation **CONTIENT**.

Par exemple :

Alice a passé la commande CMD001, qui contient un stylo et un cahier.

Types de nœuds utilisés :

- Client : nom, identifiant
- Commande : identifiant, date
- Produit : nom, prix

Relations :

- **A_EFFECTUÉ** : entre Client et Commande
- **CONTIENT** : entre Commande et Produit

← → ↺

localhost:7474/browser/preview/

☆

🔍 ⬇️ 👤 📄 ☰

● Instance: neo4j://localhost:7687 ▾ Database: neo4j ▾ User: neo4j ▾ [Go back to old Browser](#)

🗄️

📖

🕒

📖

{ }

⚙️

Database information

Nodes (8)

Client

Commande

Produit

Relationships (6)

A_EFFECTUÉ

CONTIENT

Property keys

annee

date

id

nom

prix

titre

ⓘ Automatic updates of node and relationship counts have been disabled for performance reasons, likely due to [RBAC configuration](#). Use the reload button below to manually trigger the recounts.

Last update: 11:33:49 AM 🔄

neo4j\$

neo4j\$ MATCH (n)-[r]->(m) RETURN n, r, m

Graph

Table

RAW

	n	r	m
1	()	[:A_EFFECTUÉ]	()
2	()	[:A_EFFECTUÉ]	()
3	()	[:CONTIENT]	()
4	()	[:CONTIENT]	()
5	()	[:CONTIENT]	()

Started streaming 5 records after 44 ms and completed after 66 ms.

neo4j\$ // Création des clients CREATE (alice:Client {id: "C001", nom

✓ // Création des clients

✓ CREATE (bob:Client {id: "C002", nom: "Bob"});

3. Insertion des données

Créer un jeu de données illustrant plusieurs clients ayant passé des commandes contenant des produits variés. S'assurer que le graphe est correctement structuré et visible dans Neo4j.

Un jeu de données simple a été inséré dans Neo4j afin de représenter des interactions entre clients, commandes et produits. Deux clients sont créés : Alice et Bob. Chacun a passé une commande : CMD001 pour Alice, CMD002 pour Bob. La première commande contient un stylo et un cahier, la seconde contient une trousse.

Chaque entité est représentée par un nœud de type Client, Commande ou Produit. Les relations entre ces nœuds permettent de structurer les interactions : A_EFFECTUÉ relie un client à une commande, tandis que CONTIENT relie une commande à un ou plusieurs produits.

Les données ont été insérées avec le script suivant :

```
cypher
CopierModifier
CREATE
  (alice:Client {nom: "Alice"}),
  (bob:Client {nom: "Bob"}),

  (cmd1:Commande {nom: "CMD001", date: "2025-07-01"}),
  (cmd2:Commande {nom: "CMD002", date: "2025-07-02"}),

  (stylo:Produit {nom: "Stylo", prix: 2.5}),
  (cahier:Produit {nom: "Cahier", prix: 3.0}),
  (trousse:Produit {nom: "Trousse", prix: 5.0}),

  (alice)-[:A_EFFECTUÉ]->(cmd1),
  (bob)-[:A_EFFECTUÉ]->(cmd2),

  (cmd1)-[:CONTIENT]->(stylo),
  (cmd1)-[:CONTIENT]->(cahier),
  (cmd2)-[:CONTIENT]->(trousse);
```

Une fois les données insérées, elles peuvent être visualisées dans Neo4j à l'aide de la requête :

```
cypher
CopierModifier
MATCH (n)-[r]->(m) RETURN n, r, m
```

Cette visualisation permet de parcourir le graphe et de confirmer la bonne structure des données.

← → ↻

localhost:7474/browser/

☆

🔍 ⬇️ 👤 📄 ☰

neo4j\$ MATCH (n)-[r]→(m) RETURN n, r, m

Graph

Table

Text

Code

The graph displays the results of the Cypher query. It features 7 nodes and 5 directed relationships. The nodes are: two orange nodes labeled '2025-07-...', one purple node labeled 'Bob', one blue node labeled '5.0', one blue node labeled '2.5', one blue node labeled '3.0', and one purple node labeled 'Alice'. The relationships are: 'A_EFFECTUE' (orange to purple), 'CONTIENT' (orange to blue), 'CONTIENT' (blue to blue), 'CONTIENT' (blue to orange), and 'A_EFFECTUE' (purple to orange).

Overview

Node labels

* (7) Client (2) Commande (2) Produit (3)

Relationship types

* (5) A_EFFECTUE (2) CONTIENT (3)

Displaying 7 nodes, 5 relationships.

🔍

🔍

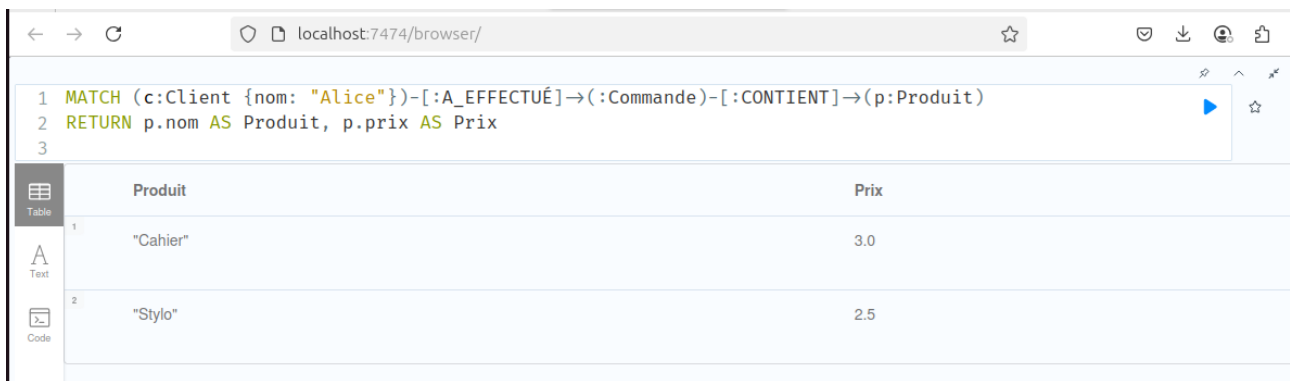
🔍

4. Requêtage

Écrire un ensemble de requêtes permettant de :

- Trouver tous les produits achetés par un client
- Identifier les clients ayant acheté un produit donné
- Dresser la liste des commandes contenant un produit spécifique
- Explorer des suggestions de produits basées sur des comportements similaires

Trouver tous les produits achetés par un client :



The screenshot shows a web browser at localhost:7474/browser/. The Cypher query editor contains the following query:

```
1 MATCH (c:Client {nom: "Alice"})-[:A_EFFECTUÉ]→(:Commande)-[:CONTIENT]→(p:Produit)
2 RETURN p.nom AS Produit, p.prix AS Prix
3
```

The results are displayed in a table with two columns: "Produit" and "Prix".

	Produit	Prix
1	"Cahier"	3.0
2	"Stylo"	2.5

Cette requête permet de retrouver tous les produits liés aux commandes effectuées par un client précis (ici, Alice).

Identifier les clients ayant acheté un produit donné :



The screenshot shows a web browser at localhost:7474/browser/. The Cypher query editor contains the following code:

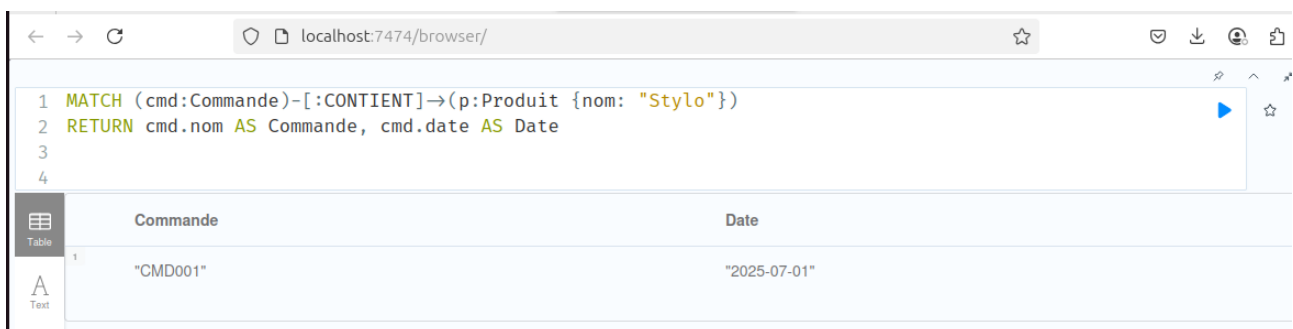
```
1 MATCH (c:Client)-[:A_EFFECTUÉ]→(:Commande)-[:CONTIENT]→(p:Produit {nom: "Trousse"})
2 RETURN DISTINCT c.nom AS Client
3
4
```

The result is displayed in a table view with the following data:

Client
"Bob"

Elle permet de remonter depuis un produit (par exemple, une trousse) vers les clients qui l'ont acheté.

Dresser la liste des commandes contenant un produit spécifique :



The screenshot shows a web browser at localhost:7474/browser/. The Cypher query editor contains the following code:

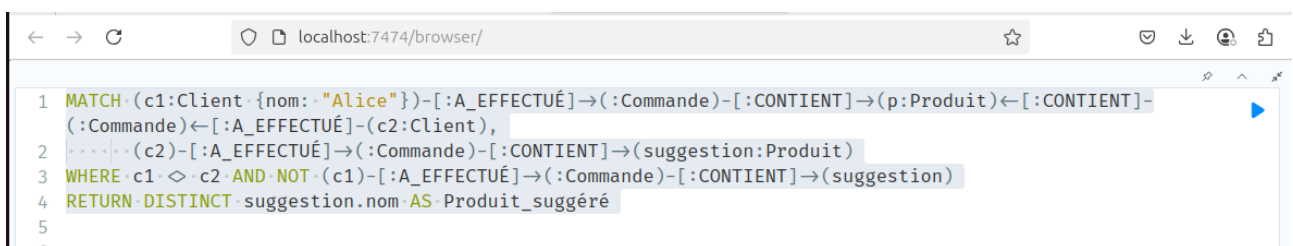
```
1 MATCH (cmd:Commande)-[:CONTIENT]→(p:Produit {nom: "Stylo"})
2 RETURN cmd.nom AS Commande, cmd.date AS Date
3
4
```

The result is displayed in a table view with the following data:

Commande	Date
"CMD001"	"2025-07-01"

Cela permet d'identifier toutes les commandes dans lesquelles un produit donné apparaît, ici pour le stylo.

Explorer des suggestions de produits basées sur des comportements similaires



The screenshot shows a web browser at localhost:7474/browser/. The Cypher query editor contains the following code:

```
1 MATCH (c1:Client {nom: "Alice"})-[:A_EFFECTUÉ]→(:Commande)-[:CONTIENT]→(p:Produit)←[:CONTIENT]→
  (:Commande)←[:A_EFFECTUÉ]-(c2:Client),
2 ...-[:A_EFFECTUÉ]→(:Commande)-[:CONTIENT]→(suggestion:Produit)
3 WHERE c1 <-> c2 AND NOT (c1)-[:A_EFFECTUÉ]→(:Commande)-[:CONTIENT]→(suggestion)
4 RETURN DISTINCT suggestion.nom AS Produit_suggéré
5
6
```

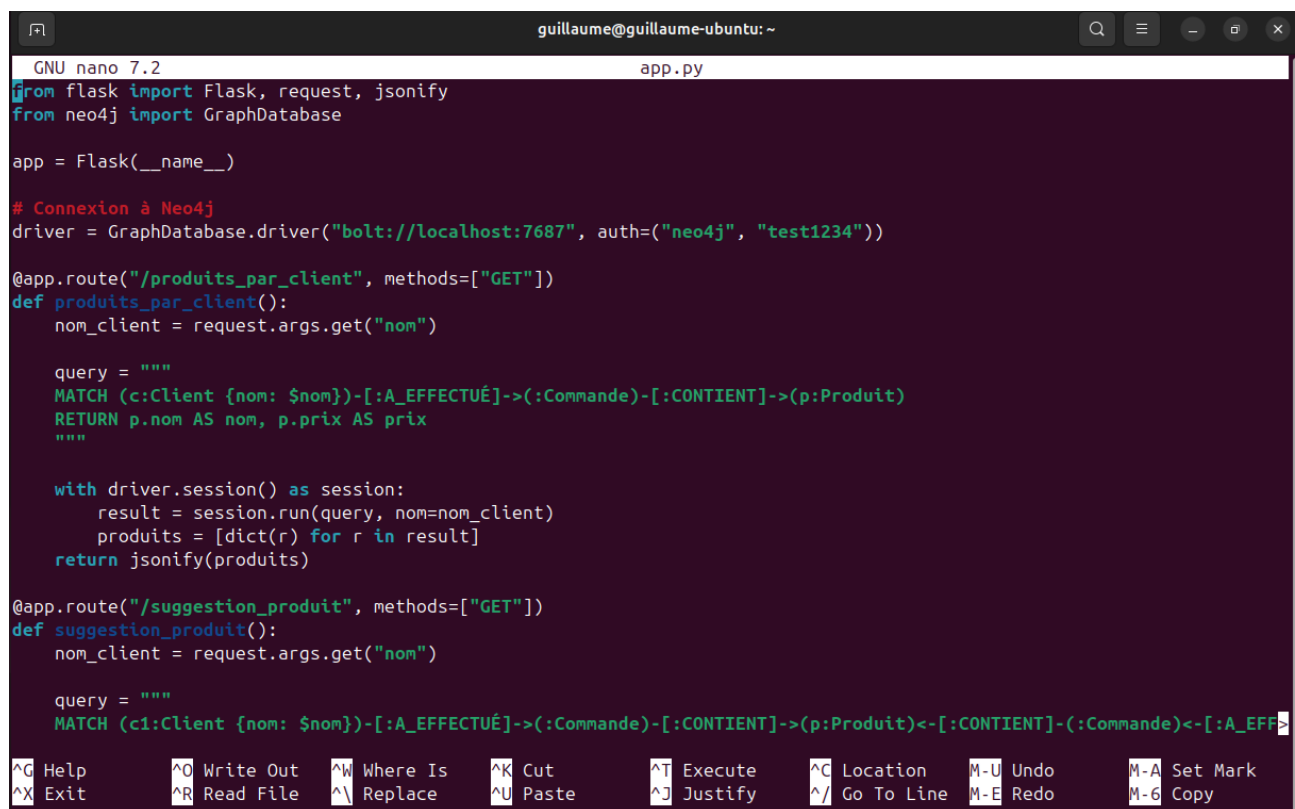
Cette requête cherche des clients ayant acheté les mêmes produits qu'un client donné (Alice) et propose d'autres produits qu'ils ont achetés, mais que le client d'origine n'a pas encore achetés. Cela permet de mettre en place une logique de recommandation de produits.

5. Intégration dans une application simple

Développer une API REST ou une interface web permettant d'interagir avec la base de données. L'application devra permettre de lancer certaines requêtes dynamiquement, comme :

- **Obtenir les produits achetés par un client donné**
- **Obtenir une suggestion de produit pour un utilisateur**

Création de l'API



```
guillaume@guillaume-ubuntu: ~  
GNU nano 7.2 app.py  
from flask import Flask, request, jsonify  
from neo4j import GraphDatabase  
  
app = Flask(__name__)  
  
# Connexion à Neo4j  
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "test1234"))  
  
@app.route("/produits_par_client", methods=["GET"])  
def produits_par_client():  
    nom_client = request.args.get("nom")  
  
    query = """  
    MATCH (c:Client {nom: $nom})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)  
    RETURN p.nom AS nom, p.prix AS prix  
    """  
  
    with driver.session() as session:  
        result = session.run(query, nom=nom_client)  
        produits = [dict(r) for r in result]  
    return jsonify(produits)  
  
@app.route("/suggestion_produit", methods=["GET"])  
def suggestion_produit():  
    nom_client = request.args.get("nom")  
  
    query = """  
    MATCH (c1:Client {nom: $nom})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)<-[:CONTIENT]-(:Commande)<-[:A_EFFECTUÉ]>
```

```

guillaume@guillaume-ubuntu: ~
(monenv) guillaume@guillaume-ubuntu:~$ nano app.py
(monenv) guillaume@guillaume-ubuntu:~$ python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 276-632-015
127.0.0.1 - - [07/Jul/2025 12:22:12] "GET /suggestion_produit?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:22:17] "GET /produits_par_client?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:23:18] "GET /suggestion_produit?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:23:19] "GET /suggestion_produit?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:23:20] "GET /suggestion_produit?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:23:22] "GET /produits_par_client?nom=Alice HTTP/1.1" 200 -
127.0.0.1 - - [07/Jul/2025 12:23:27] "GET /suggestion_produit?nom=Alice HTTP/1.1"

```

Obtenir les produits achetés par un client donné :

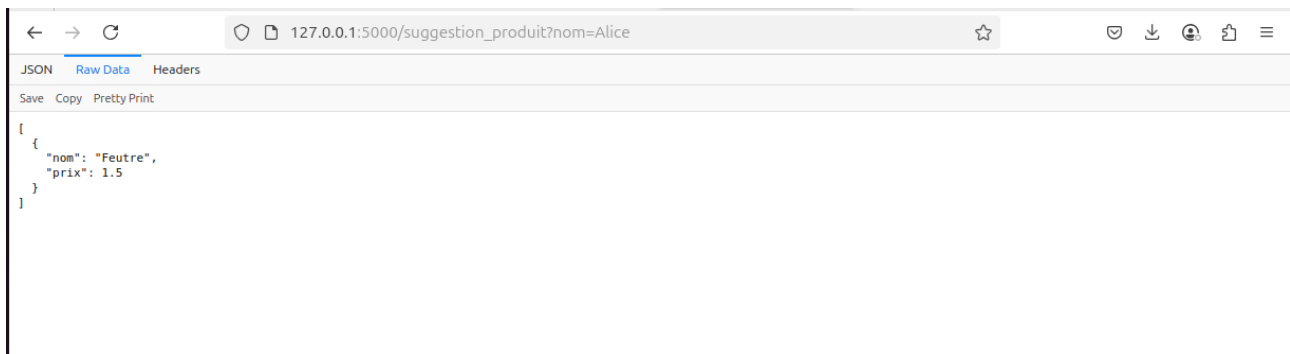
127.0.0.1:5000/produits_par_client?nom=Alice

```

JSON  Raw Data  Headers
Save Copy Collapse All Expand All Filter JSON
0:
  nom: "Carnet"
  prix: 4
1:
  nom: "Carnet"
  prix: 4
2:
  nom: "Carnet"
  prix: 4
3:
  nom: "Cahier"
  prix: 3
4:
  nom: "Stylo"
  prix: 2.5

```

- **Obtenir une suggestion de produit pour un utilisateur**



6. Livraison finale

Le rendu final devra inclure :

- Une représentation visuelle du graphe
- Le fichier décrivant l'insertion des données
- Un jeu de requêtes avec une explication pour chacune
- Le code source de l'application avec un guide de lancement
- Un court rapport résumant le travail réalisé (2 pages maximum)

Ce projet vise à modéliser et exploiter les interactions entre des clients, leurs commandes et les produits achetés. En utilisant Neo4j, une base orientée graphe, il est possible de visualiser les liens entre les entités, d'extraire des informations complexes, et de mettre en place une API simple pour rendre ces données accessibles de manière dynamique.

Modélisation du graphe

Le graphe repose sur trois types de nœuds :

- Client : une personne qui passe une commande
- Commande : un achat réalisé à une date donnée
- Produit : un article acheté dans une commande

Les relations utilisées sont :

- A_EFFECTUÉ : lie un client à une commande
- CONTIENT : relie une commande à un ou plusieurs produits

Structure du graphe :

SCSS
CopierModifier
(Client) -[:A_EFFECTUÉ]-> (Commande) -[:CONTIENT]-> (Produit)

Par exemple, si Alice a commandé un carnet et un stylo, cela sera représenté comme :

[illegible]

Données insérées dans Neo4j

Les données insérées sont les suivantes :

```
cypher
CopierModifier
// Création des clients
CREATE (alice:Client {nom: "Alice"});
CREATE (bob:Client {nom: "Bob"});

// Création des produits
CREATE (carnet:Produit {nom: "Carnet", prix: 4.0});
CREATE (feutre:Produit {nom: "Feutre", prix: 1.5});

// Création des commandes
CREATE (cmd1:Commande {nom: "CMD001", date: "2025-07-01"});
CREATE (cmd2:Commande {nom: "CMD002", date: "2025-07-02"});

// Liens entre clients et commandes
CREATE (alice)-[:A_EFFECTUÉ]->(cmd1);
CREATE (bob)-[:A_EFFECTUÉ]->(cmd2);

// Liens entre commandes et produits
CREATE (cmd1)-[:CONTIENT]->(carnet);
CREATE (cmd2)-[:CONTIENT]->(carnet);
CREATE (cmd2)-[:CONTIENT]->(feutre);
```

Cela permet de représenter le cas suivant :

- Alice et Bob ont tous les deux acheté un "Carnet"
- Bob a également acheté un "Feutre"

Requêtes Cypher utilisées

Produits achetés par un client :

```
cypher
CopierModifier
MATCH (c:Client {nom: "Alice"})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)
RETURN p.nom, p.prix
```

Commandes contenant un produit donné :

```
cypher
CopierModifier
MATCH (cmd:Commande)-[:CONTIENT]->(p:Produit {nom: "Carnet"})
RETURN cmd.nom, cmd.date
```

Clients ayant acheté un produit :

```
cypher
CopierModifier
MATCH (c:Client)-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit {nom: "Carnet"})
RETURN DISTINCT c.nom
```

Suggestion de produit pour un utilisateur (exemple : Alice) :

```

cypher
CopierModifier
MATCH (c1:Client {nom: "Alice"})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)<-[:CONTIENT]-(:Commande)<-[:A_EFFECTUÉ]-(c2:Client),
      (c2)-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(suggestion:Produit)
WHERE c1 <> c2 AND NOT (c1)-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(suggestion)
RETURN DISTINCT suggestion.nom, suggestion.prix

```

Application web en Python (Flask)

Une API REST a été développée en Python. Elle expose deux endpoints :

- /produits_par_client?nom=... : retourne les produits achetés par un client
- /suggestion_produit?nom=... : retourne une suggestion de produit

Fichier app.py :

```

python
CopierModifier
from flask import Flask, request, jsonify
from neo4j import GraphDatabase

app = Flask(__name__)

driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "test1234"))

@app.route("/produits_par_client", methods=["GET"])
def produits_par_client():
    nom_client = request.args.get("nom")
    query = """
    MATCH (c:Client {nom: $nom})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)
    RETURN p.nom AS nom, p.prix AS prix
    """
    with driver.session() as session:
        result = session.run(query, nom=nom_client)
        produits = [dict(r) for r in result]
    return jsonify(produits)

@app.route("/suggestion_produit", methods=["GET"])
def suggestion_produit():
    nom_client = request.args.get("nom")
    query = """
    MATCH (c1:Client {nom: $nom})-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(p:Produit)<-[:CONTIENT]-(:Commande)<-[:A_EFFECTUÉ]-(c2:Client),
          (c2)-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(suggestion:Produit)
    WHERE c1 <> c2 AND NOT (c1)-[:A_EFFECTUÉ]->(:Commande)-[:CONTIENT]->(suggestion)
    RETURN DISTINCT suggestion.nom AS nom, suggestion.prix AS prix
    """
    with driver.session() as session:
        result = session.run(query, nom=nom_client)
        suggestions = [dict(r) for r in result]
    return jsonify(suggestions)

if __name__ == "__main__":
    app.run(debug=True)

```

Résultats et test

Après exécution, on peut interroger l'API comme suit :

- http://127.0.0.1:5000/produits_par_client?nom=Alice renvoie les produits achetés
- http://127.0.0.1:5000/suggestion_produit?nom=Alice peut renvoyer "Feutre", que Bob a acheté mais pas Alice

Conclusion

Ce projet montre comment un graphe simple peut modéliser des relations entre clients, commandes et produits de façon lisible et exploitable. L'intégration d'une API légère permet de rendre ces données consultables dynamiquement. Le tout est extensible à volonté (avis, stocks, préférences...), ce qui en fait une base solide pour des applications réelles.