Rapport d'Analyse des Performances - SGBDR vs NoSQL

Url du depot https://github.com/Tchoupitoo/nosql
Membres du projet: Emerick Biron et Louis Rocchi

1. Introduction

Ce rapport présente une comparaison des performances entre un SGBDR (PostgreSQL) et une base NoSQL (Neo4j) dans le cadre
de l'analyse du comportement d'achat des utilisateurs d'un réseau social. L'objectif est d'évaluer les avantages et
inconvénients de chaque solution, en termes de temps d'insertion et de requêtes analytiques.

---

2. Modélisation des Données

2.1. Modèle Relationnel (PostgreSQL)

Le schéma relationnel utilisé pour PostgreSQL est structuré comme suit :

```
utilisateurs (id VARCHAR(36) PRIMARY KEY, nom VARCHAR(255))
followers (utilisateur_id VARCHAR(36), follower_id VARCHAR(36), PRIMARY KEY (utilisateur_id,
follower_id))
produits (id VARCHAR(36) PRIMARY KEY, nom VARCHAR(255), prix DECIMAL)
achats (utilisateur_id VARCHAR(36), produit_id VARCHAR(36), date_achat TIMESTAMP, PRIMARY
KEY (utilisateur_id, produit_id))
```

Relations :

- Un utilisateur peut suivre plusieurs utilisateurs (**followers**).
- Un utilisateur peut acheter plusieurs produits (**achats**).

2.2. Modèle NoSQL (Neo4j)

Le modèle de données pour Neo4j utilise une représentation orientée graphe :

- `(Utilisateur)-[:FOLLOWS]->(Utilisateur)`
- `(Utilisateur)-[:ACHAT {date: date_achat}]->(Produit)`

Contraintes définies :

```
CREATE CONSTRAINT IF NOT EXISTS FOR (u:Utilisateur) REQUIRE u.id IS UNIQUE;
CREATE CONSTRAINT IF NOT EXISTS FOR (p:Produit) REQUIRE p.id IS UNIQUE;
```

---

3. Spécifications et Conception du Logiciel

3.1. Technologies Utilisées

- **Backend :** Python 3.12 avec Flask
- **Base de données :** PostgreSQL 16 et Neo4j Community
- **Conteneurisation :** Docker & Docker Compose
- **Interface utilisateur :** HTML, CSS (Bootstrap), JavaScript (Chart.js pour la visualisation des performances)

3.2. Architecture du Logiciel

Le logiciel suit une **architecture en couches** avec une **DAL (Data Access Layer)** pour séparer l'interface utilisateur des accès aux bases de données.

```
├── docker-compose.yml
├── Dockerfile
├── app/
│   ├── app.py  # Serveur Flask
│   ├── utils.py  # Gestion du timing et de l'historique
│   ├── db/
│   │   ├── base_db.py  # Interface commune pour les bases de données
│   │   ├── postgres_db.py  # Implémentation pour PostgreSQL
│   │   ├── neo4j_db.py  # Implémentation pour Neo4j
│   ├── templates/
│   │   ├── index.html  # Interface utilisateur
│   ├── static/
│   │   ├── css/style.css
│   │   ├── js/script.js
```

---

4. Requêtes Implémentées

4.1. Insertion des Données

L'insertion des données est réalisée avec des scripts aléatoires pour :

- **Créer des utilisateurs et leurs relations de follow** (0 à 20 followers directs)
- **Créer des produits** (10 000 références)
- **Générer des achats** (0 à 5 achats par utilisateur)

4.2. Requêtes d'Analyse

a) Nombre d'achats des followers d'un utilisateur à un niveau donné

PostgreSQL :

```
WITH RECURSIVE follower_hierarchy AS (SELECT follower_id, utilisateur_id, 1 AS level
                                      FROM followers
                                      WHERE utilisateur_id = %s
```

```
UNION ALL
SELECT f.follower_id, f.utilisateur_id, fh.level + 1
FROM followers f
         INNER JOIN follower_hierarchy fh ON f.utilisateur_id = fh.follower_id
WHERE fh.level < %s
    )
SELECT p.id AS product_id, p.nom AS product_name, COUNT(*) AS nb_achats
FROM follower_hierarchy fh
         JOIN achats a ON fh.follower_id = a.utilisateur_id
         JOIN produits p ON a.produit_id = p.id
GROUP BY p.id, p.nom
ORDER BY nb_achats DESC;
```

Neo4j :

```
MATCH (follower)-[:FOLLOWS*1..{max_level}]->(u:Utilisateur {id: "{user_id}"})
MATCH (follower)-[:ACHAT]->(p:Produit)
RETURN p.id AS product_id, p.nom AS product_name, COUNT(*) AS nb_achats
ORDER BY nb_achats DESC
```

b) Nombre de personnes ayant acheté un produit donné dans un cercle de followers

PostgreSQL :

```
WITH RECURSIVE follower_circle AS (SELECT a.utilisateur_id, 1 AS level
                                   FROM achats a
                                   WHERE a.produit_id = %s
UNION ALL
SELECT f.follower_id, fc.level + 1 AS level
FROM followers f
       JOIN follower_circle fc ON f.utilisateur_id = fc.utilisateur_id
WHERE fc.level < %s
  )
SELECT p.id AS product_id, p.nom AS product_name, COUNT(DISTINCT a.utilisateur_id) AS
num_buyers
FROM produits p
       JOIN achats a ON p.id = a.produit_id
       JOIN follower_circle fc ON a.utilisateur_id = fc.utilisateur_id
WHERE p.id = %s
GROUP BY p.id, p.nom
ORDER BY num_buyers DESC;
```

Neo4j :

```
MATCH (follower)-[:FOLLOWS*1..{max_level}]->(u:Utilisateur)
MATCH (follower)-[:ACHAT]->(p:Produit {id: "{product_id}"})
RETURN p.id AS product_id, p.nom AS product_name, COUNT(DISTINCT follower) AS num_buyers
ORDER BY num_buyers DESC
```

---

5. Résultats et Analyse des Performances

5.1. Temps d'Insertion

Base de données	Opération	Nb entités	Temps (ms)
PostgreSQL	insert_users	1 000 000	2 093 197
PostgreSQL	insert_produits	10 000	3 872
PostgreSQL	insert_achats	10 000	518 947
Neo4j	insert_users	1 000 000	20 697 237
Neo4j	insert_produits	10 000	16 712
Neo4j	insert_achats	10 000	5 352 241

➤ **Conclusion :** PostgreSQL est **nettement plus rapide** que Neo4j pour l'insertion des utilisateurs et des achats.

5.2. Temps d'Exécution des Requêtes

Nombre d'achats des followers d'un utilisateur (nb_achats_produits)

Base de données	Profondeur	Temps (ms)
Neo4j	3	1343
Neo4j	4	3199
Neo4j	5	2785
PostgreSQL	3	2078
PostgreSQL	4	1639
PostgreSQL	5	3165

➤ **Analyse :** PostgreSQL est plus rapide pour **profondeur 3**, mais Neo4j **devient plus performant** pour des niveaux plus profonds.

Nombre d'achats d'un produit spécifique (nb_achats_produit_unique)

Base de données	Profondeur	Temps (ms)
Neo4j	3	138.863
Neo4j	4	189.104
Neo4j	5	283.976

Base de données	Profondeur	Temps (ms)
PostgreSQL	3	136.856
PostgreSQL	4	161.953
PostgreSQL	5	436.042

➤ **Analyse :** PostgreSQL et Neo4j sont **très proches** en performances pour **profondeur 3 et 4**, mais PostgreSQL **est moins performant à profondeur 5**.

---

6. Conclusion

- **PostgreSQL est supérieur pour l'insertion et les requêtes globales**, grâce à ses index optimisés et ses performances en jointure.
- **Neo4j est plus performant pour l'analyse des relations profondes**, ce qui en fait un excellent choix pour l'exploration du réseau social et le suivi des tendances d'achat en profondeur.
- **Les requêtes ciblant un produit spécifique (nb_achats_produit_unique)** montrent des performances **comparables** entre PostgreSQL et Neo4j, sauf pour des niveaux très profonds où Neo4j est légèrement meilleur.