IMT Mines Alès
École Mines-Télécom

NO-SQL

Contributor :
Valentin POIROT
Luca DESLOT

École des Mines d'Alès
INFRES 14

23/02/2024

Laurent Bayart

1. Introduction

L'objectif principal de ce travail pratique (TP) est d'explorer et d'analyser les comportements d'achat des utilisateurs au sein d'un réseau social, en utilisant à la fois une base de données relationnelle standard (SGBDR) et une base de données NoSQL, spécifiquement Neo4j. Dans le cadre de cette étude, nous nous concentrons sur la modélisation, l'implémentation et le test en volumétrie d'un service d'analyse comportementale, en mettant en lumière les avantages, inconvénients et performances de chaque solution de stockage de données.

Pour atteindre cet objectif, nous avons développé un logiciel capable de lancer des requêtes sur les deux types de bases de données, permettant une comparaison directe des temps de réponse sans nécessiter d'intervention manuelle dans le code. Ce choix technologique offre une flexibilité et une accessibilité maximales pour les utilisateurs finaux, souhaitant tester les performances des systèmes de base de données dans des conditions réelles d'utilisation.

Ce rapport documente l'ensemble du processus, depuis la conception initiale jusqu'aux résultats finaux des tests de performance, en passant par la méthodologie de génération de données, l'architecture système, et l'analyse comparative des temps de réponse. Notre étude offre ainsi des insights précieux sur l'efficacité des bases de données relationnelles et NoSQL dans le traitement et l'analyse de données complexes issues de réseaux sociaux, ouvrant la voie à de nouvelles réflexions sur le choix des technologies de stockage de données pour des applications similaires.

2. Architecture et technologies utilisées

Ce projet s'appuie sur une architecture orientée services, construite autour de Spring Boot pour exploiter efficacement ses capacités en matière de développement d'applications et d'intégration de bases de données. Le choix de cette architecture facilite non seulement la gestion des interactions avec les bases de données SQL et NoSQL, mais garantit également une séparation claire entre la logique métier et l'interface utilisateur, permettant une maintenance et une évolution simplifiées de l'application.

Composants Clés

- Spring Boot Application (SocialNetworkAnalysisNoSqlApplication.java) : Définit le point d'entrée de l'application NEO4J et configure automatiquement Spring Boot.
- Controllers (ProductController.java, UserController.java) : Fournissent les API pour interagir avec l'application via HTTP, traitant les requêtes et retournant les réponses à l'utilisateur.
- Services (ProductService.java, UserService.java) : Encapsulent la logique métier, interagissant avec les repos pour effectuer les opérations CRUD, ainsi que les opérations spécifiques comme l'insertion de données aléatoires, la gestion des relations entre utilisateurs et produits, et l'exécution de requêtes complexes.
- Repositories (UserRepo.java, ProductRepo.java et leurs implémentations) : Abstraient l'accès aux données, utilisant Neo4j pour interroger et manipuler les données relatives aux utilisateurs et aux produits.

Flexibilité et Extensibilité

L'architecture conçue offre une grande flexibilité, permettant une évolution facile du système. Les services et les repos sont clairement définis et séparés, facilitant l'ajout de nouvelles fonctionnalités ou l'intégration de technologies supplémentaires. De plus, l'utilisation de Spring Boot et de ses annotations simplifie la configuration et l'extension du système.

3. Injection et génération des données

Génération de données

Utilisateurs

Le système génère et insère un million d'utilisateurs factices dans la base de données. Chaque utilisateur est créé avec un identifiant unique et un ensemble de propriétés de base, telles que le nom. Ce processus est facilité par la méthode insertRandomUsers dans UserService, qui utilise le UserRepo pour interagir avec la base de données.

Produits

De même, 10 000 produits sont générés avec des caractéristiques uniques (nom, description, prix) et insérés dans la base de données. Cette tâche est accomplie grâce à la méthode insertRandomProducts dans ProductService, qui appelle le ProductRepo.

Injection des données

Suivi entre utilisateurs

Pour simuler les relations de réseau social, des liens de suivi sont établis entre les utilisateurs de manière aléatoire, en respectant la contrainte qu'un utilisateur peut suivre entre 0 et 20 autres utilisateurs. Cela est géré par followUsersFromCsv dans UserService, qui lit les relations à partir d'un fichier CSV et les insère dans la base de données en utilisant Neo4j.

Achats de produits

De manière similaire, les interactions d'achat entre les utilisateurs et les produits sont générées. Chaque utilisateur peut avoir acheté entre 0 et 5 produits. Ce processus est implémenté par buyProductsFromCsv dans UserService, qui insère ces relations dans la base de données à partir d'un fichier CSV.

4. Développement du logiciel

Conception de l'API

L'API est structurée autour de deux contrôleurs principaux, ProductController et UserController, qui exposent divers endpoints pour interagir avec les données des utilisateurs et des produits. Ces contrôleurs gèrent des opérations telles que la création de produits, l'assignation d'achats à des utilisateurs, l'analyse de l'influence des utilisateurs sur les achats dans leur réseau, et bien plus.

Swagger

Pour faciliter les interactions avec l'API, le système utilise Swagger, une suite d'outils open-source pour concevoir, construire, documenter et utiliser des services web RESTful. Swagger offre une interface utilisateur web intuitive qui permet aux développeurs et aux utilisateurs finaux d'explorer, de tester et de documenter les API de manière interactive.

L'interface utilisateur de Swagger offre une expérience interactive pour explorer les différents endpoints de l'API, comprenant les méthodes HTTP, les paramètres attendus, et les formats de réponse.

Lien vers les contrats d'api :

Contrat d'API NoSQL : http://localhost:8080/swagger-ui/index.html#/

Contrat d'API SQL : http://localhost:8081/swagger-ui/index.html#/

Fonctionnalités de l'API

Les endpoints exposés par l'API permettent une variété d'analyses et d'opérations, y compris mais non limité à :

- Gestion des utilisateurs et des produits : Création et gestion de l'information des utilisateurs et des produits.
- Analyse d'influence : Obtention de la liste et du nombre de produits commandés par les cercles de followers d'un utilisateur.

- Popularité des produits : Détermination du nombre de personnes ayant commandé un produit spécifique dans un cercle de followers.

- Importation de données : Support pour l'importation de données utilisateur et produit à partir de fichiers CSV pour faciliter les tests en volumétrie.

5. Requêtes d'Analyse

Les requêtes d'analyse sont au cœur de notre système, permettant d'extraire des insights significatifs sur les comportements d'achat au sein du réseau social. Utilisant Neo4j, ces requêtes exploitent la puissance des bases de données graphiques pour analyser les relations complexes entre les utilisateurs et leurs interactions d'achat.

Requêtes Neo4j Principales

Voici les requêtes Neo4j spécifiques implémentées pour analyser les comportements d'achat et l'influence au sein du réseau social, utilisant les capacités graphiques de Neo4j pour explorer les relations complexes entre utilisateurs et produits :

Obtenir la liste et le nombre de produits commandés par les cercles de followers d'un individu (niveau 1, ..., niveau n) :

```
MATCH (influencer:User) <-[:FOLLOWERS*1..n]-(follower:User)-[:BOUGHT]-
>(product:Product)
WHERE influencer.name = $name
WITH product, COLLECT(DISTINCT follower) AS buyers
RETURN id(product) AS id, product.name AS name, product.description AS description,
product.price AS price, buyers AS boughtBy
```

Cette requête analyse les cercles d'influence d'un utilisateur spécifique, identifiant combien de produits ont été achetés par ses followers et à différents niveaux de ses relations de suivi.

Même requête mais avec spécification d'un produit particulier:

```
MATCH (product:Product {name: %s})<-[:BOUGHT]-(buyer:User)<-[:FOLLOWERS*1..%d]-
(follower:User)-[:BOUGHT]->(product)
RETURN COUNT(DISTINCT follower)
```

Cette version filtre l'analyse à un produit spécifique, permettant d'évaluer l'influence d'un utilisateur sur les ventes de ce produit au sein de son réseau.

Les deux dernières requêtes se trouvent dans le fichier **ProductRepoImpl.java.**

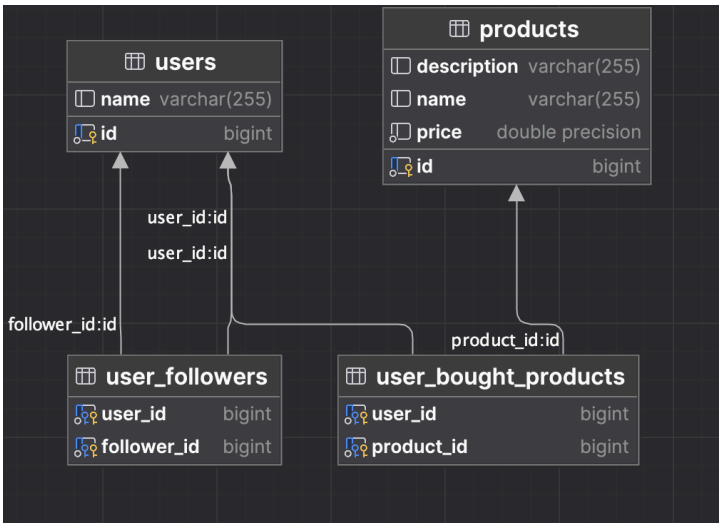
Pour une référence de produit donné, obtenir le nombre de personnes l'ayant commandé dans un cercle de followers "orienté" de niveau n:

```
MATCH (influencer:User) <-[:FOLLOWERS*1..4]-(follower:User)-[:BOUGHT]-
>(product:Product)
WHERE influencer.name = $userName AND product.name = $productName
WITH COUNT(DISTINCT follower) AS buyersCount
RETURN buyersCount
```

Cette requête se concentre sur un produit spécifique pour identifier sa popularité et son caractère "viral" au sein de différents niveaux du réseau social, montrant comment les recommandations influencent les achats.

Cette requête se trouve dans le fichier **UserRepoImpl.java**.

Requêtes SQL Principales



Le schéma SQL présenté illustre les tables et les relations pour un système de gestion de base de données relationnelle conçu pour une plateforme de réseau social avec une fonctionnalité d'achat intégrée. La table users conserve les informations de base des utilisateurs, avec des champs pour le name et un identifiant unique id. La relation d'abonnement est modélisée dans la table user_followers, où user_id et follower_id déterminent qui suit qui. Cela prend en compte les utilisateurs qui peuvent s'abonner à eux-mêmes et vise à éviter les doublons dans les requêtes. Les informations sur les

produits sont stockées dans la table products, qui contient le name, la description, et le price de chaque produit, ainsi qu'un id unique. La table user_bought_products établit une association entre les users et les products qu'ils ont achetés, indiquant quels utilisateurs ont acheté quels produits. Ce schéma permet de réaliser des analyses de comportement d'achat au sein du réseau social, telles que déterminer l'influence des utilisateurs sur les achats de leurs abonnés, identifier les produits « viraux », et autres analyses pertinentes pour le projet.

Obtenir la liste et le nombre de produits commandés par les cercles de followers d'un individu (niveau 1, ..., niveau n) :

```sql
-- Recursive SQL to fetch products bought by a user's circle of followers up to a certain depth
-- Assuming 'userName' is a placeholder for the actual user name and 'depth' for the level of depth

WITH RECURSIVE follower_tree AS (
    SELECT user_id, follower_id, 1 AS level
    FROM user_followers
    WHERE follower_id = (SELECT id FROM users WHERE name = :userName)
    UNION ALL
    SELECT uf.user_id, uf.follower_id, ft.level + 1
    FROM user_followers uf
    INNER JOIN follower_tree ft ON uf.follower_id = ft.user_id
    WHERE ft.level < :depth
)
SELECT p.id, p.name, p.description, p.price, ARRAY_AGG(DISTINCT ft.user_id) AS boughtBy
FROM products p
JOIN user_bought_products ubp ON p.id = ubp.product_id
JOIN follower_tree ft ON ubp.user_id = ft.user_id
GROUP BY p.id, p.name, p.description, p.price;

-- Example usage of the above query
-- Let's say we have a user named 'JohnDoe' and we want to find products bought by his circle of followers up to a depth of 3
-- You would replace :userName with 'JohnDoe' and :depth with 3 when running the query
```

Même requête mais avec spécification d'un produit particulier:

```sql
-- Recursive SQL to find the count of distinct users who bought a product
-- by the product name and up to a certain depth in their followers circle
-- Assuming 'productName' is a placeholder for the actual product name and 'depth' for the level of depth

WITH RECURSIVE follower_tree AS (
    SELECT id AS user_id, 0 AS level
    FROM users
    WHERE id IN (SELECT user_id FROM user_bought_products WHERE product_id = (SELECT id FROM products WHERE name = :productName))
    UNION ALL
    SELECT uf.follower_id, ft.level + 1
    FROM user_followers uf
    INNER JOIN follower_tree ft ON uf.user_id = ft.user_id
    WHERE ft.level < :depth
)
SELECT COUNT(DISTINCT user_id)
FROM follower_tree WHERE level > 0; -- Excludes level 0 which is the user who made the initial purchase

-- Example usage of the above query
-- If we want to find the count of users who have bought a product called 'GadgetX'
-- and the depth of followers' circle is 3, you would replace :productName with 'GadgetX' and :depth with 3
```

Pour une référence de produit donné, obtenir le nombre de personnes l'ayant commandé dans un cercle de followers "orienté" de niveau n:

```sql
-- Recursive SQL to get the count of products bought by a user's followers up to a certain depth
-- Assuming 'userId' and 'productId' are placeholders for the actual user ID and product ID

WITH RECURSIVE follower_tree AS (
    SELECT follower_id, user_id, 1 AS level
    FROM user_followers
    WHERE user_id = :userId -- Use the user ID passed to the method
    UNION ALL
    SELECT uf.follower_id, ft.user_id, ft.level + 1
    FROM user_followers uf
    INNER JOIN follower_tree ft ON uf.user_id = ft.follower_id
    WHERE ft.level < 4 -- Limit to 4 levels of followers
)
SELECT COUNT(*)
FROM products p
INNER JOIN user_bought_products ubp ON p.id = ubp.product_id
INNER JOIN follower_tree ft ON ubp.user_id = ft.follower_id
WHERE p.id = :productId; -- Use the product ID passed to the method

-- Example usage of the above query
-- To get the count of a product with ID '123' bought by the followers of a user with ID '456' up to 4 levels deep,
-- you would replace :userId with '456' and :productId with '123'
```

Choix d'Import CSV pour l'Optimisation

Les imports CSV sont utilisés pour une insertion efficace et en masse des données, notamment pour les relations complexes telles que les suivis entre utilisateurs et les achats de produits. Cette méthode permet :

Une réduction significative du temps d'exécution pour peupler la base de données, crucial lors du traitement de grandes quantités de données.

Une gestion optimisée de la charge sur la base de données, en regroupant les opérations d'insertion pour minimiser l'impact sur les performances.

Une facilité d'importation des relations et des données, en adaptant le format CSV aux structures attendues par Neo4j, simplifiant ainsi le processus d'initialisation et de mise à jour des données.

Les fichiers d’imports sont placés dans à la racine sous neo4j/import/buyPairs.csv pour les associations d’achats et sous neo4j/import/followPairs.csv pour les associations de followers.

6. Résultats et Comparaison des Performances

	Neo4J	SQL
Création utilisateurs	7214 ms	1465 ms
Création des produits	119 ms	42 ms
Création de suivi des utilisateurs	46440 ms	27271 ms
Création des liens d'achats	47471 ms	11823 ms
getBoughtProductCountCircle	145 ms	2745 ms
getBoughtProductCircle (depth = 3)	151 ms	2068
getBoughtProductCircle (depth = 4)	154 ms	2609 ms
findBoughtProductByProductAndDepth (depth = 5)	62 ms	850 ms

La comparaison des performances entre les systèmes de gestion de bases de données relationnelles (SGBDR) et NoSQL pour le projet de réseau social montre des différences significatives dans les temps de réponse aux requêtes spécifiques. Pour les opérations de création (utilisateurs, produits, suivis d'utilisateurs, et liens d'achats), le SGBDR est généralement plus rapide que NoSQL, bien que la différence soit moindre pour la création des produits. Cela peut être attribué à la nature optimisée des SGBDR pour les opérations transactionnelles simples et leur maturité dans le domaine.

En revanche, pour les requêtes d'analyse de comportement d'achat, qui nécessitent des opérations plus complexes telles que des traversées récursives et des jointures, NoSQL, et en particulier Neo4j, présente des performances nettement supérieures. Par exemple, pour obtenir le compte des produits achetés dans le cercle d'un utilisateur (getBoughtProductCountCircle), la base NoSQL affiche un temps de réponse bien inférieur à celui du SGBDR.

Ces résultats soulignent le rôle influent de l'utilisateur au sein du réseau pour déclencher des achats, permettant de cerner les produits qui sont devenus "viraux" au sein des cercles de followers. De manière notable, les temps de réponse pour les requêtes impliquant des niveaux de profondeur différents (getBoughtProductCircle avec depth=3 et depth=4) restent relativement constants dans NoSQL, ce qui démontre une meilleure performance pour les analyses complexes et récursives typiques des graphes sociaux.

La base NoSQL se distingue particulièrement pour les requêtes nécessitant une analyse en profondeur du réseau d'influence d'un utilisateur, où l'aspect "graphe" de la base de données facilite et accélère les traversées à plusieurs niveaux, par opposition aux bases de données relationnelles qui montrent des temps d'exécution croissants avec la profondeur de la requête.

En conclusion, bien que les SGBDR soient plus performants pour les opérations de création de données simples, les bases NoSQL, grâce à leur structure flexible et leur efficacité dans la gestion des relations complexes, sont mieux adaptées pour les analyses comportementales dans les réseaux sociaux. Ces dernières offrent des temps de réponse nettement inférieurs pour les requêtes d'analyse, ce qui est crucial pour les fonctionnalités d'analyse en temps réel et la détection de tendances dans un environnement de réseau social.