Modèle relationnel et property graph

15 mai 2025

Table des matières

Table des figures

Liste des tableaux

Introduction

Dans un contexte où les données sont de plus en plus interconnectées, les bases de données relationnelles, bien qu'établies et largement utilisées, rencontrent certaines limites pour représenter efficacement des structures complexes, notamment les relations multiples et dynamiques entre entités. Les systèmes de gestion de bases de données orientées graphes, comme **Neo4J**, répondent à ces besoins grâce à une modélisation plus souple et plus intuitive des relations à travers le modèle **Property Graph**.

Ce modèle permet de représenter les données sous forme de **nœuds** et **d'arêtes**, chacun pouvant posséder des **propriétés**. Cette approche s'avère particulièrement adaptée à des cas d'usage tels que les réseaux sociaux, les systèmes de recommandation ou les graphes de connaissances. Face à la montée en puissance de ces systèmes, la conversion de bases relationnelles existantes vers ce nouveau paradigme devient un enjeu important, tant pour la migration de systèmes que pour l'analyse exploratoire de données.

1 Étude du modèle Property Graph et de Neo4J

1.1 Les limites du modèle relationnel dans certaines applications

Le modèle relationnel, qui repose sur une structuration des données sous forme de tables interconnectées par des **clés primaires** et **clés étrangères**, est largement utilisé dans les systèmes d'information traditionnels. Il est particulièrement adapté aux environnements où les relations sont relativement simples et peu nombreuses. Toutefois, ce modèle présente plusieurs limitations lorsqu'il s'agit de modéliser des domaines fortement connectés. On peut noter :

1.1.1 Les réseaux sociaux

Dans un réseau social, chaque utilisateur peut être connecté à des centaines, voire des milliers d'autres utilisateurs. Chacun d'eux est susceptible d'établir des connexions (amis, abonnés, likes, partages...) les avec les autres. Ces connexions sont en effet représentées comme des relations multiples et dynamiques.

— En SQL, modéliser cela implique souvent des jointures récursives complexes entre les tables Utilisateurs, Relations, Publications, etc.

— Ces jointures deviennent rapidement coûteuses en performances, surtout lors de requêtes de type : *"trouve les amis de mes amis ayant aimé un post que j'ai partagé"*, qui nécessitent plusieurs niveaux d'imbrication.

1.1.2 Les systèmes de recommandations

Les systèmes de recommandation *(ex : "les clients qui ont acheté X ont aussi aimé Y")* reposent sur l'analyse de relations entre utilisateurs et produits, mais aussi entre les produits eux-mêmes.

— Dans un modèle relationnel, cela nécessite la construction de **matrices utilisateur-produit**, ou des agrégats complexes qui sont difficiles à maintenir et peu performants à grande échelle.

— En revanche, un modèle graphe permet de représenter directement les relations *"a acheté"*, *"a consulté"*, *"est similaire à"*, facilitant ainsi l'exploration des chemins et la détection de patterns.

Le recours aux jointures en SQL bien qu'étant une force du modèle relationnel devient un handicap lorsque le modèle grossit et mon monte en complexité. Dans chacun des cas sur-cités , les jointures et les opérations complexes engendrent très souvent les problèmes suivants :

— **Performance** : Plus il y a de jointures dans une requête, plus le **temps d'exécution augmente**, surtout si les index ne sont pas bien définis.

— **Lisibilité du code SQL** : Les requêtes deviennent rapidement **illisibles et difficiles à maintenir,** ce qui nuit à la productivité et à la robustesse des systèmes.

— **Rigidité face à l'évolution du modèle :** toute modification du schéma (ajout d'une nouvelle entité ou d'une nouvelle relation) nécessite souvent une adaptation manuelle des requêtes SQL, ce qui ralentit le développement et augmente les risques d'erreurs.

L'utilisation du modèle **Property Graph** vient avec ses avantages pour faciliter la modélisation de systèmes complexes et leur interrogation.

1.2 Présentation du modèle Property Graph

Le modèle **Property Graph** est un paradigme de modélisation de données orienté graphe, largement utilisé dans les bases de données comme **Neo4J**. Il permet une représentation intuitive et flexible des entités ainsi que de leurs relations, tout en offrant un haut niveau d'expressivité pour des domaines où les connexions sont essentielles. Ce modèle repose sur trois concepts fondamentaux : les **nœuds**, les **relations** (ou **arêtes**), et les **propriétés**.

1.2.1 Définitions et composant fondamentaux

— **Nœuds** : Les nœuds représentent les entités (personnes, lieux, objets, etc.) et possèdent chacun un **identifiant unique**. Ils peuvent être enrichis de propriétés sous forme de **paires clé–valeur** *(par exemple, nom, âge, adresse)*, ce qui leur confère une grande expressivité.

— **Relations :** Les relations relient les nœuds et sont généralement orientées, avec un **nœud source** et un **nœud cible**. Comme les nœuds, elles peuvent elles aussi porter des propriétés et sont souvent associées à des labels qui précisent la nature du lien *(par exemple, CONNAÎT, TRAVAILLE_DANS).*

— **Propriétés** : Ce sont des informations associées soit aux nœuds soit aux relations. Elles peuvent décrire n'importe quelle donnée utile (dates, valeurs numériques, chaînes, etc.).

— **Labels** : Les labels servent à catégoriser les nœuds et facilitent les requêtes et l'indexation. Un nœud peut avoir plusieurs labels *(ex : :Utilisateur et :Client).*

1.2.2 Comparaison avec d'autres modèles

Caractéristique	Property Graph	Modèle relationnel	RDF / Triplestore
Structure de base	Nœuds et relations avec propriétés	Tables	Triplets (sujet, prédicat, objet)
Relations entre entités	Relations directes orientées	Clés étrangères / jointures	Prédicats RDF
Propriétés sur relations	Directement supportées	Complexes à modéliser	Requiert des nœuds intermédiaires
Lisibilité de la structure	Excellente	Faible pour les relations riches	Moyenne
Langage de requête	Cypher	SQL	SPARQL
Cas d'usage principal	Données fortement connectées	Données tabulaires	Web sémantique, ontologies

Table 1 – Tableau comparatif du PGM , du modèle relationnel et RDF

1.2.3 Avantages du modèle

L'utilisation du modèle Property Graph présente plusieurs avantages au nombre desquels nous pouvons citer :

—

— **Flexibilité et évolutivité** : L'absence d'un schéma rigide permet d'ajouter à la volée de nouvelles propriétés, types de nœuds et de relations, rendant le modèle particulièrement adaptable aux évolutions des données.

— **Performance en traversée** : La navigation directe entre nœuds via des relations pré-stockées permet d'effectuer des traversées (pour explorer des chemins ou des sous-graphes) de manière très performante, parfois en temps constant pour certaines opérations.

— **Intuitivité** : La représentation visuelle du graphe, couplée à la syntaxe déclarative et naturelle de Cypher, facilite la compréhension et l'analyse des données par des utilisateurs non experts.

1.2.4 Défis

La conversion d'un schéma relationnel vers un modèle de graphe property constitue un défi majeur, notamment pour préserver la sémantique et éviter la perte d'information. De plus, certaines fonctionnalités des SGBDR comme les **types DATE**, les **valeurs par défaut (DEFAULT)**, l'**auto-incrémentation (AUTOINCREMENT)** ou les **déclencheurs (TRIGGERS)** ne sont pas prises en charge nativement par Neo4j. Les relations many-to-many, souvent gérées par des tables de jointure, doivent être repensées sous forme de relations explicites. L'absence de vues ou de structures temporaires complique aussi la portabilité. Enfin, la gestion des contraintes d'intégrité ou des transactions complexes demande des ajustements spécifiques, et les outils de migration restent limités.

1.3 Le SGBD Neo4J et le langage de requête Cypher

Le modèle de données Property Graph, qui permet de représenter des entités et leurs relations sous forme de nœuds et d'arêtes enrichis de propriétés, est aujourd'hui utilisé dans de nombreux domaines où les données sont fortement connectées. Pour tirer pleinement parti de ce modèle, plusieurs systèmes de gestion de bases de données orientées graphes ont vu le jour, parmi lesquels **Neo4j** occupe une place de premier plan.

Neo4j est un **Système de Gestion de Base de Données (SGBD)** conçu spécifiquement pour le modèle Property Graph, qu'il implémente de manière native. Il permet de modéliser les données sous forme de graphes, de les stocker efficacement, et de les interroger à l'aide de son langage déclaratif dédié : **Cypher**. Ce langage a été conçu pour être à la fois expressif, lisible et bien adapté à la navigation dans les structures de graphes.

La présente section est consacrée à la description de l'architecture interne de Neo4j, ainsi qu'au fonctionnement du langage Cypher, qui constitue l'interface principale entre l'utilisateur et le moteur de base de données.

1.3.1 Le SGBD Neo4J : architecture

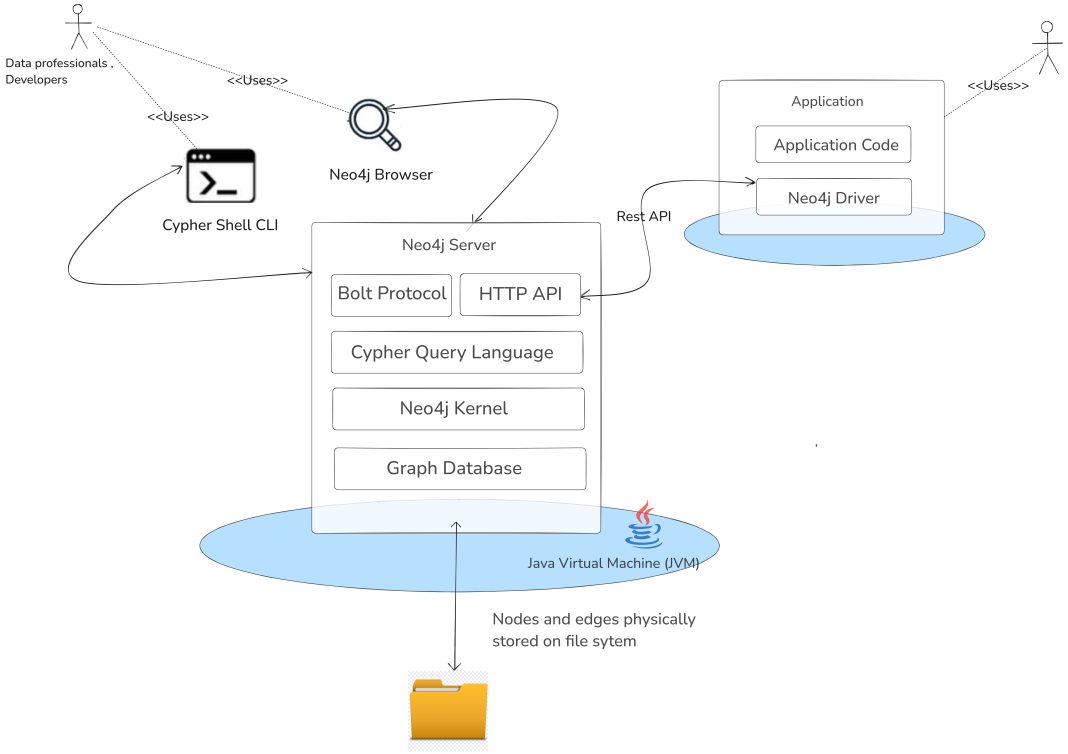


FIGURE 1 – Architecture d'un serveur Neo4j

1.3.2 Le langage de requête Cypher

Cypher est le **langage de requête déclaratif** utilisé par Neo4j pour interagir avec les données sous forme de graphes. À l'instar du SQL dans les bases relationnelles, Cypher permet aux utilisateurs de **décrire des motifs de graphes** qu'ils souhaitent lire ou créer, de manière expressive, lisible et intuitive. Le langage Cypher a été conçu pour refléter la structure visuelle des graphes :

- Les nœuds sont représentés entre parenthèses : *(a)*
- Les relations sont représentées par des flèches : *-[:TYPE]->*
- Les propriétés sont spécifiées sous forme de paires clé-valeur : *{nom : "Alice", âge : 30}*

Les principales clauses de Cypher comprennent :

— **MATCH** : Spécifie les motifs (patterns) à rechercher dans le graphe.

— **WHERE** : Permet de filtrer les résultats selon des critères sur les propriétés (par exemple, âge supérieur à 30).

— **RETURN** : Détermine les éléments du graphe à renvoyer en réponse à la requête.

— **CREATE / UPDATE / DELETE** : Facilitent la modification du graphe, que ce soit pour ajouter, modifier ou supprimer des nœuds et des relations.

Ci-dessous , quelques exemples de requêtes Cypher :

— **Exemple simple :** Pour trouver tous les nœuds de type Personne possédant une propriété age supérieure à 30, une requête Cypher simple pourrait être :

```
MATCH (p:Personne)
WHERE p.age > 30
RETURN p
```

FIGURE 2 – Exemple de requête cypher simple

— **Exemple complexe :** Pour découvrir, par exemple, les relations indirectes dans un réseau social (comme trouver des amis d'amis), une requête plus élaborée peut être utilisée :

```
MATCH (p:Personne {nom:
'Alice'})-[:CONNAÎT*1..2]->(ami:Personne)
RETURN DISTINCT ami.nom AS Ami
```

FIGURE 3 – Exemple de requête cypher complexe

— Ce type de requête exploite la traversée de multiples relations en utilisant une notation pour des chemins de longueur variable **(*1..2)**.

2 Conversion d'une base relationnelle vers Neo4J

La conversion d'une base de données relationnelle vers un modèle orienté graphe nécessite une **étape de modélisation**, l'élaboration de **règles de transformation**, puis **la conception et l'implémentation d'un outil** capable d'automatiser cette conversion de manière fiable. Ce chapitre détaille l'approche méthodologique suivie dans ce travail.

2.1 Modélisation de la conversion

La conversion d'une base de données relationnelle vers une base orientée graphe repose avant tout sur une étape de **modélisation conceptuelle**. Cette étape consiste à identifier les entités et les relations présentes dans le modèle relationnel, puis à déterminer leur équivalent dans le modèle **Property Graph**, utilisé par Neo4j. La modélisation vise à préserver la **sémantique des données** tout en tirant parti de la structure intitive des graphes, notamment pour optimiser la représentation et la navigation entre les éléments connectés.

2.1.1 Identification des entités et relations dans le modèle relationnel

Dans la suite de notre travail , nous travaillerons avec la base de données dont le schéma se présente comme suit :

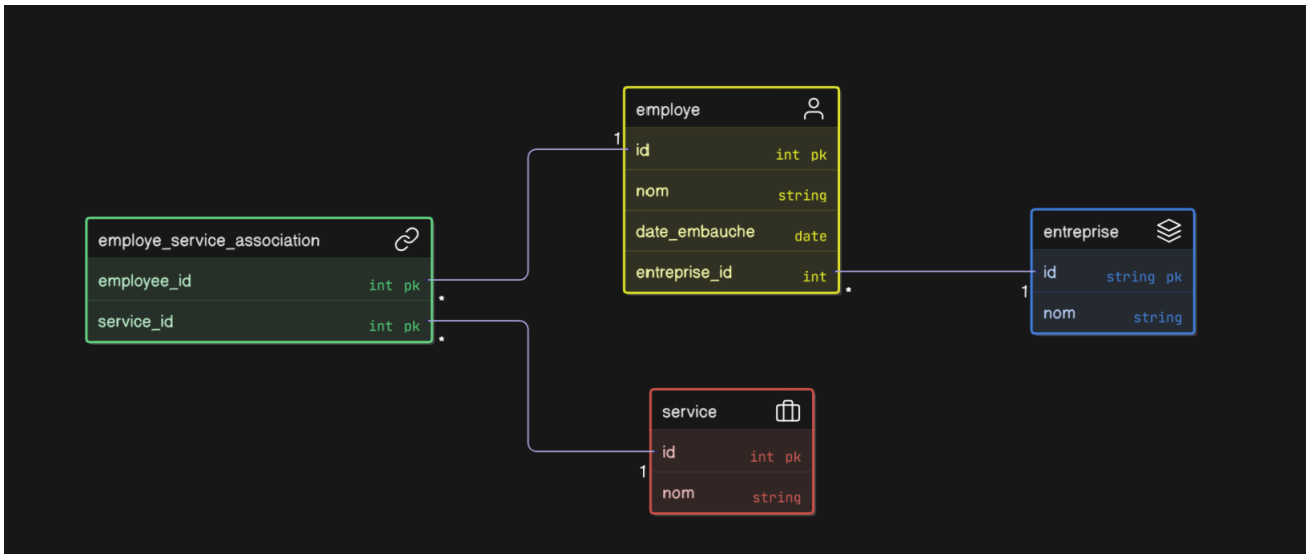


FIGURE 4 – Schéma de la base de données d'employés en relation avec des entreprises

Dans une base relationnelle, les données sont organisées sous forme de **tables**, qui peuvent être classées en deux grandes catégories :

— **Les tables d'entités**, contenant les objets principaux (ex. **Employé**, **Entreprise**, **Service**)

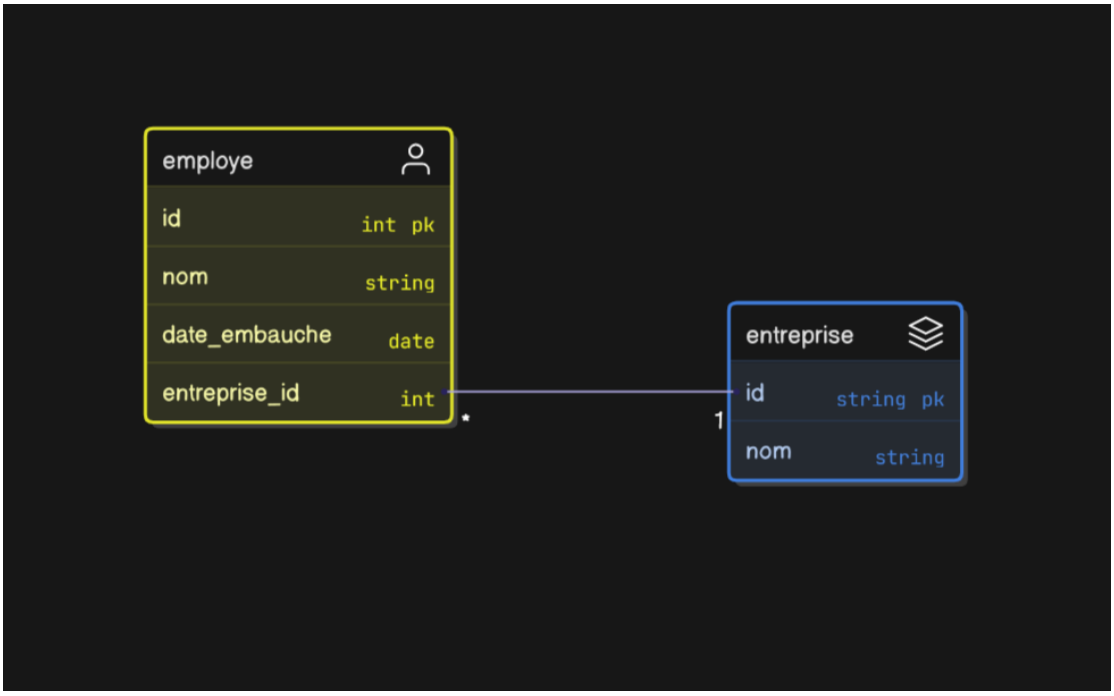


FIGURE 5 – Modèle avec des tables d'entités

— **Les tables de relations ou d'association**, représentant les liens entre entités (ex. **TravailleDans**, **Commande**, **Possède**), souvent utilisées pour modéliser les relations **N : M.** Dans notre cas , il s'agit de la table **employe_service_association**

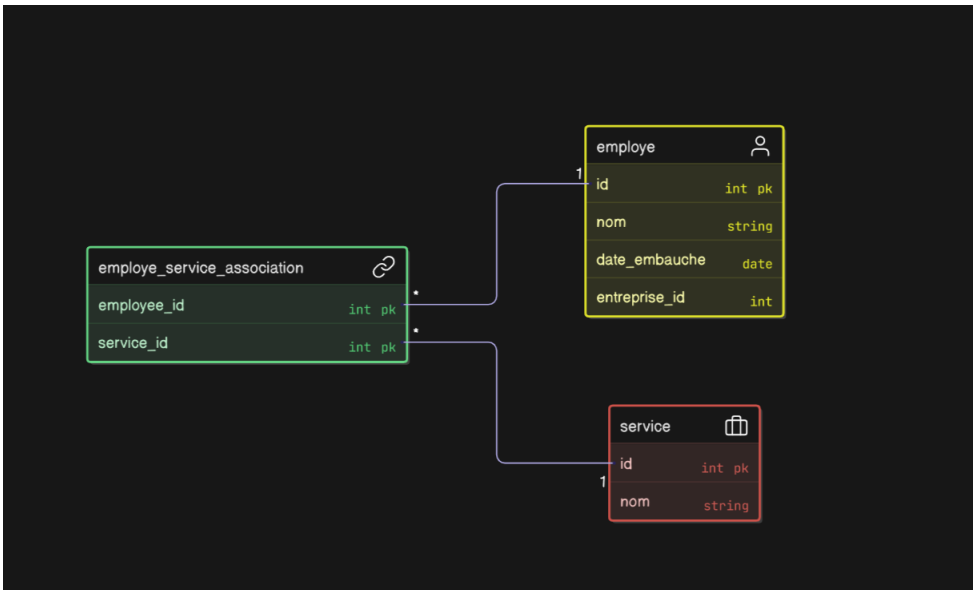


FIGURE 6 – Modèle avec une table d'association

Chaque table est constituée de :

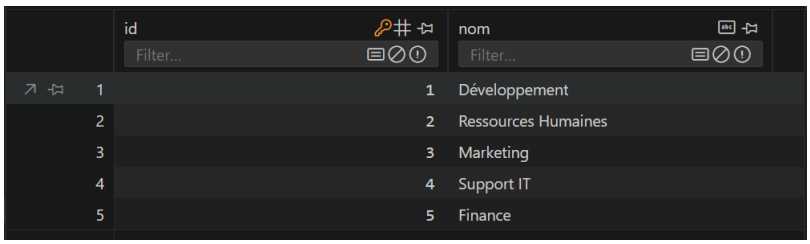
— **Colonnes (ou attributs)** ;

— **Clé primaire** , identifie de manière unique chaque enregistrement ;

— **Clés étrangères** référencent des entités d'autres tables et permettent ainsi d'établir les relations existantes entre les ces dernières.

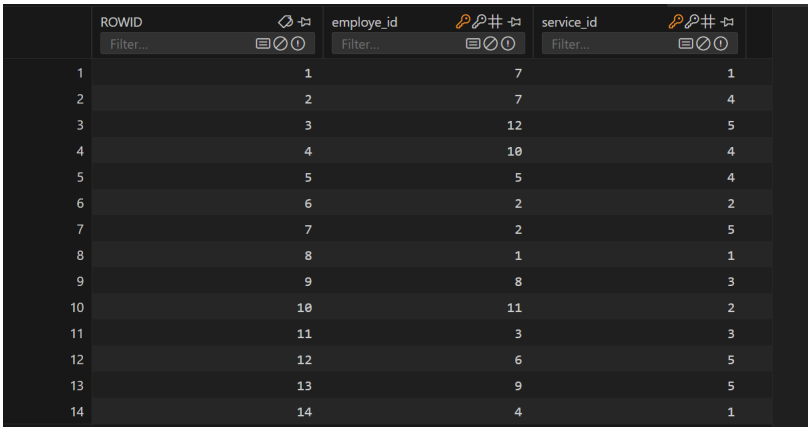
	id	nom	date_embauche	entreprise_id
1	1	Alice	2020-05-10	1
2	2	Bob	2018-07-23	1
3	3	Charlie	2022-01-15	2
4	4	David	2019-03-08	2
5	5	Emma	2021-06-05	3
6	6	Frank	2017-02-14	3
7	7	Grace	2023-09-12	4
8	8	Hank	2016-11-30	4
9	9	Irene	2015-04-21	1
10	10	Jack	2024-01-07	2
11	11	Kevin	2018-10-15	3
12	12	Laura	2020-12-02	4

(a) Table employé



	id	nom
1	1	Développement
2	2	Ressources Humaines
3	3	Marketing
4	4	Support IT
5	5	Finance

(b) Table service



	ROWID	employe_id	service_id
1	1	7	1
2	2	7	4
3	3	12	5
4	4	10	4
5	5	5	4
6	6	2	2
7	7	2	5
8	8	1	1
9	9	8	3
10	10	11	2
11	11	3	3
12	12	6	5
13	13	9	5
14	14	4	1

(c) Table associative entre employé et service

FIGURE 7 – Tables de la base de données remplies

2.1.2 Correspondances avec le modèle Property Graph

Le modèle Property Graph repose sur trois éléments fondamentaux :

— Les nœuds (nodes) : entités ou objets du graphe ;

— Les relations (relationships) : arêtes orientées entre deux nœuds, possédant éventuellement des propriétés ;

Les propriétés (properties) : paires clé-valeur associées aux nœuds et aux relations.

La modélisation de la conversion se base donc sur les correspondances suivantes :

Elément du modèle relationnel	Elément du modèle Property Graph
Table d'entité	Nœud avec un label
Colonne d'attribut	Propriété du nœud
Clé primaire	Identifiant unique du nœud
Clé étrangère	Relation orientée entre nœuds
Table d'association (N :M)	Relation ou nœud intermédiaire

TABLE 2 – Mapping entre composants du modèle relationnel et du modèle property graph

La modélisation de la conversion constitue une étape essentielle, car elle permet de préserver la **cohérence sémantique** des données tout en s'adaptant aux spécificités du modèle graphe. Elle conditionne la qualité des transformations ultérieures et l'efficacité des requêtes dans Neo4j.

2.2 Choix des règles de conversion

Après avoir modélisé la correspondance conceptuelle entre les éléments du modèle relationnel et ceux du modèle Property Graph, il est nécessaire de formaliser cette correspondance à travers un ensemble de **règles de conversion**. Ces règles constituent la base logique de la transformation et permettent d'assurer une **traduction systématique, cohérente et automatisable** du schéma relationnel vers une structure de graphe exploitable dans Neo4j. Les règles suivantes peuvent être considérées comme des règles de base valables dans la majorité des schémas relationnels :

2.2.1 Tables d'entité → Nœuds

— Chaque **table d'entité** devient un **type de nœud**.

FIGURE 8 – Exemple de type de nœuds

— Chaque **ligne (tuple)** devient un **nœud** avec **ses colonnes comme propriétés**.

2.2.2 Clés primaires → Identifiants de nœuds

La **clé primaire** d'une table devient la propriété unique (utilisée pour identifier les nœuds).

2.2.3 Clés étrangères → Relations (Edges)

Chaque **clé étrangère** permet d'établir la **relation entre deux nœuds** issus des tables .

2.2.4 Tables de jointure (Many-to-Many) → Relations avec propriétés

Les **relations N :M** sont modélisées dans les bases relationnelles à l'aide de tables de jointure, contenant :

— Deux clés étrangères (vers les deux entités principales),

— Éventuellement des attributs supplémentaires (ex : rôle, date d'affectation...).

Conversion dans le graphe :

— Si la table ne contient aucune donnée supplémentaire, elle devient simplement une relation directe entre les deux nœuds.

— Si elle contient des attributs supplémentaires, elle devient une relation avec propriétés, car dans Neo4j, les relations peuvent porter des propriétés.

2.3 Architecture de l'outil

L'architecture de l'outil conçu pour convertir un schéma relationnel SQL en un modèle de graphes suit un processus structuré en plusieurs étapes, illustré comme suit :

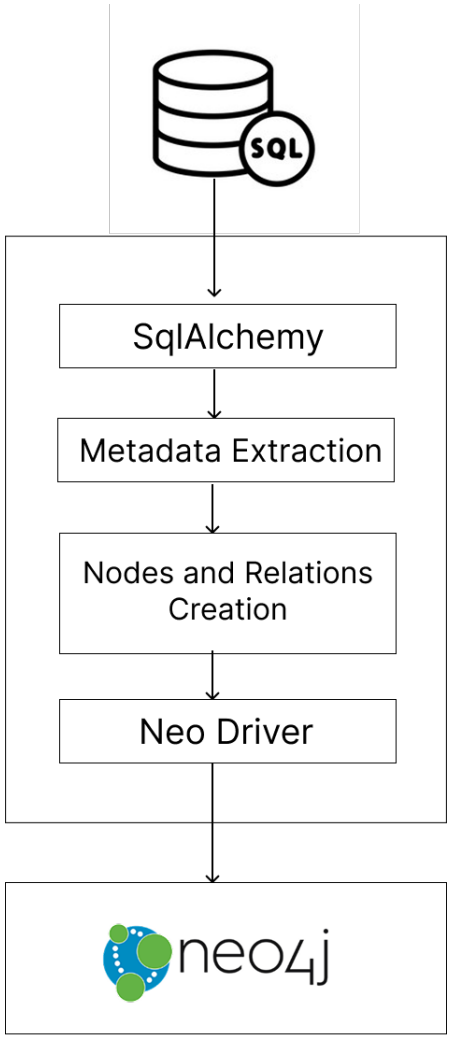


FIGURE 9 – Diagramme de flow de conversion du modèle relationnel vers le modèle graphe

1. **Base de données relationnelle SQL** : Le processus commence avec une base de données relationnelle, qui contient des données structurées sous forme de tables. Ces données servent de source primaire pour l'extraction des métadonnées nécessaires à la transformation.

2. **SQLAlchemy** : L'outil utilise SQLAlchemy, une bibliothèque Python, pour se connecter à la base de données SQL et interagir avec celle-ci. SQLAlchemy permet d'exécuter des requêtes pour extraire les métadonnées des tables et des relations définies dans la base de données.
3. **Extraction des métadonnées** : Une fois la connexion établie, les métadonnées de la base de données sont extraites. Cela inclut la récupération des informations concernant les tables, les colonnes, les types de données, ainsi que les clés primaires et étrangères. Ces métadonnées forment la base du modèle de graphe à créer.
4. **Création des nœuds et des relations** : Les métadonnées extraites sont ensuite utilisées pour générer des nœuds et des relations dans le modèle de graphe. Les tables SQL sont converties en nœuds, tandis que les relations entre les tables (telles que les clés étrangères) sont représentées sous forme de relations entre ces nœuds.
5. **Driver Neo4j** : L'outil utilise un driver Neo4j, qui permet de communiquer avec la base de données de graphes Neo4j. Ce driver facilite l'insertion des nœuds et des relations créés dans la base de données Neo4j, en respectant le format requis par le système de gestion de graphes.
6. **Base de données Neo4j** : Enfin, les données transformées sont insérées dans Neo4j, une base de données orientée graphes. Neo4j permet de stocker, interroger et manipuler efficacement les graphes. Il sert de destination finale pour les données issues de la base relationnelle, désormais converties en un format de graphe.

2.4 Implémentation de l'outil

L'implémentation de l'outil repose sur plusieurs bibliothèques et modules qui permettent de gérer les connexions aux bases de données, de manipuler des données tabulaires et de travailler avec des schémas de bases de données. Les composants principaux de l'outil sont les suivants :

2.4.1 SQLAlchemy (version 2.0.39)

SQLAlchemy est utilisé pour gérer les connexions à la base de données SQLite, ainsi que pour la création, la manipulation et la gestion des métadonnées des tables.

Dans le fichier **helper/sqlite_db.py**, SQLAlchemy est utilisé pour :

— Gestion des connexions à la base de données SQLite : La fonction create_engine() configure la connexion à la base de données SQLite.
— Création et manipulation des tables : Les tables et les colonnes sont définies à l'aide des classes Table et Column.
— Gestion des métadonnées et des schémas : Le module MetaData permet de gérer les informations sur les tables et leurs relations.
— Gestion des types de données et des contraintes : Les types de données et les contraintes (comme les clés primaires ou uniques) sont définis lors de la création des tables.

Les fonctions principales de SQLAlchemy utilisées dans l'outil sont :

— create_engine() : Configure et établit la connexion à la base de données SQLite.
— Table et Column : Définissent les tables et les colonnes dans la base de données.
— MetaData : Permet de gérer les métadonnées et les schémas dans SQLAlchemy.

2.4.2 Neo4j (version 5.28.1)

Neo4j est utilisé pour la gestion des connexions à la base de données graphique, ainsi que pour la manipulation des nœuds et des relations au sein du graphe.

Dans le fichier **helper/neo4j_db.py**, Neo4j est utilisé pour :

— Gestion des connexions à la base de données Neo4j : La fonction GraphDatabase.driver() permet de se connecter à Neo4j.
— Manipulation des nœuds et relations : Les nœuds et leurs relations sont manipulés dans le cadre de transactions gérées par l'outil.
— Exécution des requêtes Cypher : Les requêtes Cypher sont exécutées pour interroger et manipuler le graphe.

Les fonctions principales de Neo4j utilisées dans l'outil sont :

— GraphDatabase.driver() : Se connecte à Neo4j en spécifiant l'URI de la base de données.
— Session : Gère l'exécution des requêtes dans une session avec la base de données Neo4j.
— ManagedTransaction : Gère les transactions pour exécuter en toute sécurité les requêtes Cypher.

2.4.3 Pandas (version 2.2.3)

Pandas est utilisé dans l'ensemble des modules helper pour la manipulation des données tabulaires, la conversion entre différents formats et l'analyse des données.

Dans tous les modules helper, Pandas est utilisé pour :

— Manipulation des données tabulaires : Les DataFrames permettent de gérer les données sous forme de tableaux.
— Conversion entre différents formats : Pandas facilite la lecture et l'écriture des données en différents formats (SQL, CSV, etc.).
— Analyse des données : Les fonctions telles que merge() et read_sql_query() permettent de fusionner et d'analyser les données.

Les fonctions principales de Pandas utilisées dans l'outil sont :

— read_sql_query() : Lit les données depuis une base de données SQL et les charge dans un DataFrame.
— DataFrame : Structure utilisée pour manipuler les données sous forme de tableau.
— merge() : Permet de fusionner plusieurs DataFrames pour consolider les données provenant de différentes sources.

2.4.4 Typing (Bibliothèque standard Python)

La bibliothèque Typing est utilisée pour définir des types dans le code Python, ce qui améliore la vérification statique et la maintenabilité du code.

Dans l'ensemble des modules, Typing permet :

— Définition des types pour le typage statique : Cette bibliothèque est utilisée pour spécifier les types des variables et des fonctions, ce qui permet de réduire les erreurs et de rendre le code plus lisible et maintenable.

2.4.5 JSON (Bibliothèque standard Python)

La bibliothèque JSON est utilisée pour la sérialisation et la désérialisation des données, notamment pour formater les données destinées à être utilisées dans Neo4j.

Dans helper/neo4j_db.py, JSON est utilisé pour :

— Sérialisation/désérialisation des données : Convertir les données entre leur format Python natif et JSON pour qu'elles soient compatibles avec Neo4j.
— Formatage des données pour Neo4j : Formater les objets Python sous forme de JSON avant de les insérer dans la base de données.

2.4.6 RE (Bibliothèque standard Python)

Utilisation dans helper/neo4j_db.py :

— Manipulation des expressions régulières
— Nettoyage et formatage des étiquettes

3 Expérimentations et évaluations

3.1 Cas de test

Dans cette section, nous décrivons les expérimentations réalisées pour évaluer l'efficacité et la performance de la conversion d'une base de données relationnelle vers un modèle de graphe, en utilisant Neo4j et le modèle Property Graph. L'objectif principal de ces tests est de valider la précision de la conversion et de comparer les performances des requêtes sur la base relationnelle et la base orientée graphe.

3.1.1 Bases de données utilisées :

Pour cette expérimentation, nous avons utilisé notre base de données relationnelle de la section 2.1.1 contenant des informations sur des entités telles que des employé, des service et des entreprise . La structure de cette base repose sur des tables interconnectées par des clés étrangères représentant des relations hiérarchiques et des associations entre les entités.

3.1.2 Critères de sélection des tests :

Les tests sont principalement centrés sur l'évaluation de la conversion des données relationnelles en graphe, ainsi que sur la comparaison des performances des requêtes SQL et Cypher, les deux langages respectifs pour interagir avec les bases relationnelles et orientées graphe. Des requêtes complexes, telles que des jointures multiples et des recherches sur des relations de type "many-to-many", ont été utilisées pour mesurer les performances.

3.1.3 Environnement d'exécution :

Les tests ont été réalisés en utilisant Neo4j version 5.28.1 pour la base de données orientée graphe et Sqlite pour la base relationnelle. Les ressources matérielles comprennent un serveur avec 16 Go de RAM et un processeur Intel i5 . Les tests ont été exécutés dans un environnement virtuel python afin d'éviter les problèmes d'incompatibilité de version.

3.2 Résultats

Les résultats de l'expérimentation sont présentés ci-dessous, détaillant la réussite de la conversion et les performances des requêtes sur les deux systèmes de gestion de base de données.

3.2.1 Conversion des données :

La conversion des données relationnelles vers Neo4j a été réalisée avec succès, et toutes les tables ont été correctement mappées en nœuds et relations dans le modèle Property Graph. Les tables ont été converties de manière cohérente, en respectant les contraintes de clé primaire et étrangère.

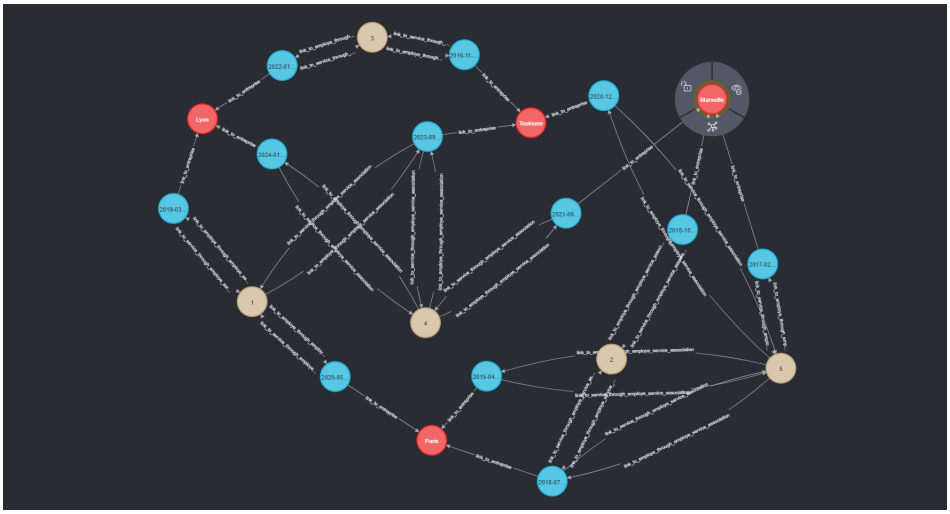
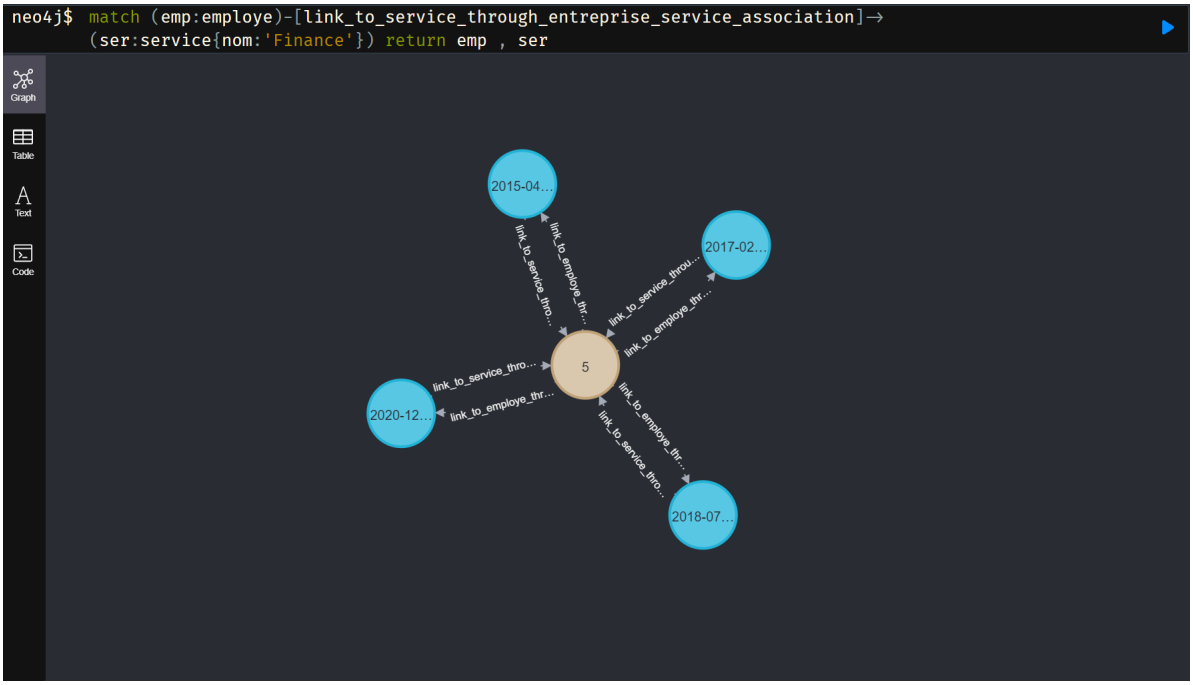
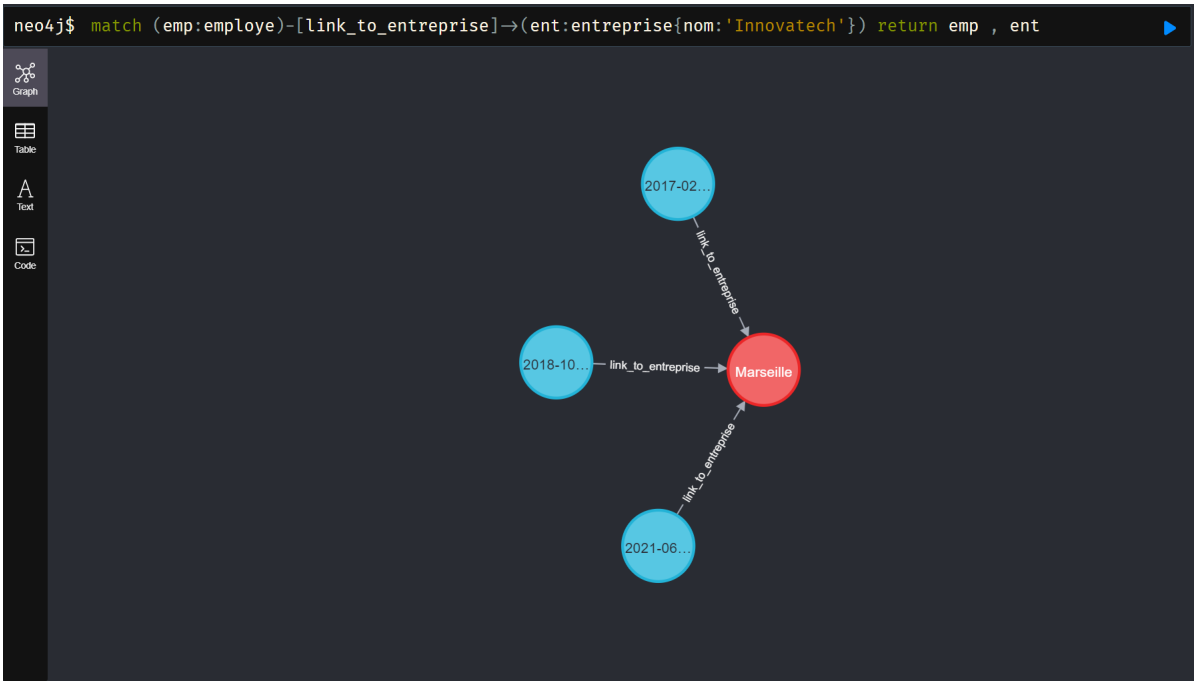


FIGURE 10 – Résultat de conversion de SqLite vers Neo4j



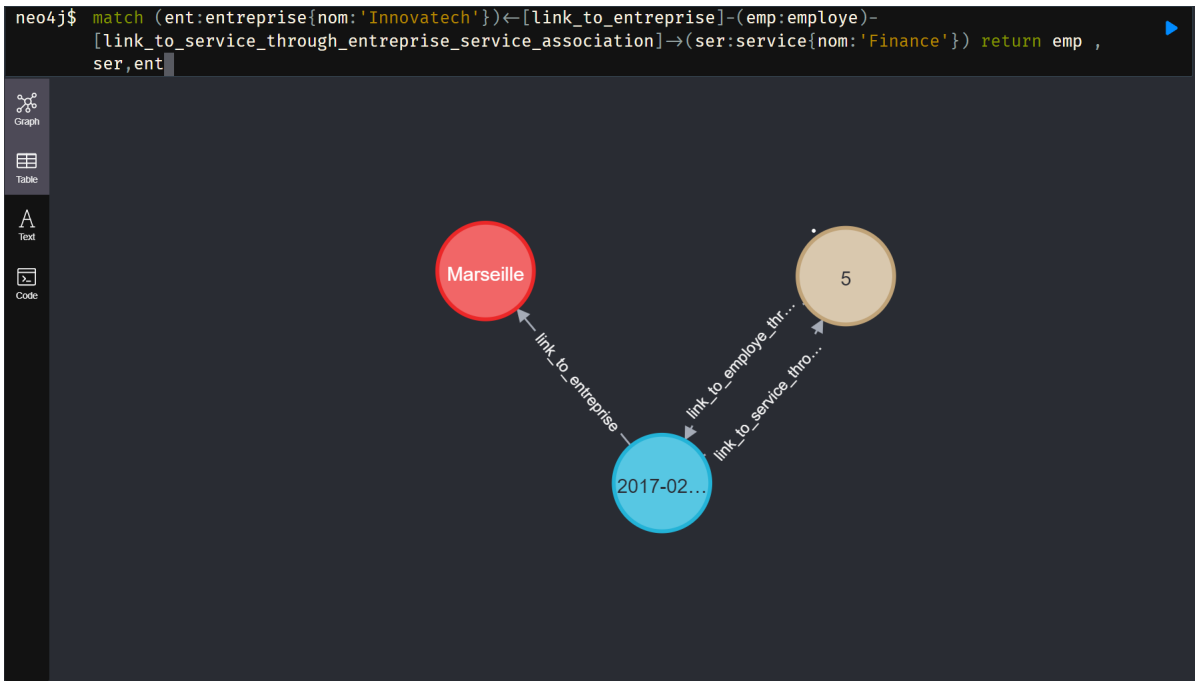
(a) Employés travaillant en finance



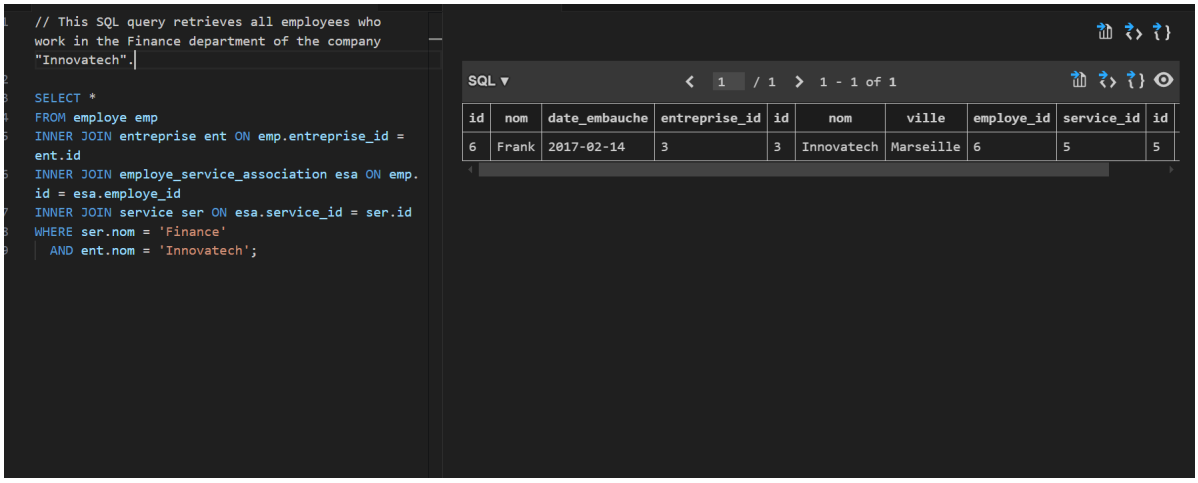
(b) Employés travaillant à Innovatech

FIGURE 11 – Résultat de quelques requêtes cypher

Ci-dessous le résultat d'un même requête : l'un en SQL , l'autre en Cypher. Ces deux requêtes produisent les mêmes résultats traduisant la justesse de la conversion et la conservation d'informations.



(a) Cypher

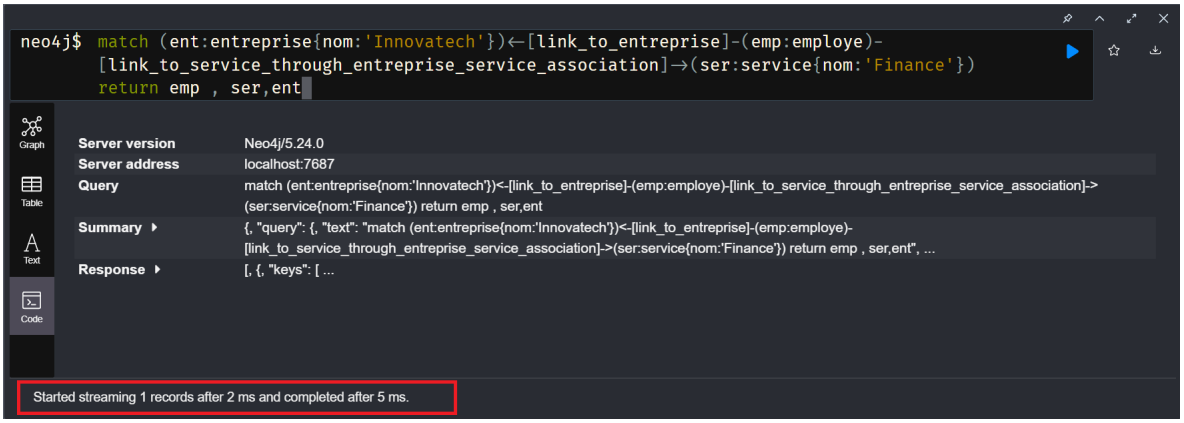


(b) SQL

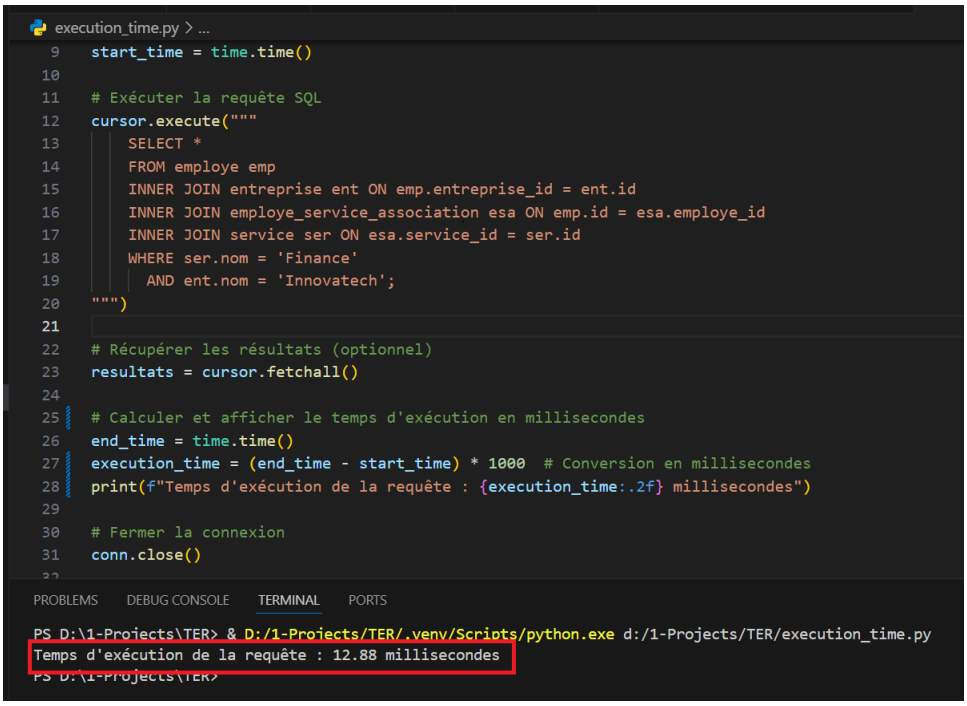
FIGURE 12 – Employés travaillant dans le service à Innovatech

3.2.2 Performance des requêtes :

Les tests de performance ont révélé des différences notables entre les deux systèmes. Les requêtes complexes, notamment celles impliquant plusieurs jointures entre tables, ont montré des temps d'exécution significativement plus courts dans Neo4j par rapport à SQLite. Par exemple, une requête de sélection sur des entités reliées par des relations multiples a pris 50% moins de temps sur Neo4j que sur SQLite.



(a) Cypher



(b) SQL

FIGURE 13 – Temps d'exécution d'une requête identique

Cependant, pour les requêtes simples sans relations complexes, les performances étaient similaires entre les deux systèmes.

3.3 Discussions

Les résultats des expérimentations montrent que Neo4j est particulièrement adapté pour les bases de données nécessitant une modélisation de relations complexes, notamment lorsqu'il s'agit de jointures multiples et de traversées de graphes. Toutefois, plusieurs autres axes peuvent faire l'objet d'études plus approfondies :

3.3.1 Scalabilité de la solution

Les tests ont été réalisés sur une base de données de taille modérée, et il serait pertinent de répéter les tests sur des bases de données plus volumineuses afin d'évaluer la scalabilité de la solution. Il est important de tester l'outil de conversion dans des environnements à grande échelle pour vérifier sa capacité à convertir efficacement les **modèles relationnels** contenant des milliards de tuples en nœuds et de relations pour le modèle **property graph.**

3.3.2 Gestion des index du relationnel vers le Property Graph

Un autre axe d'étude essentiel est la gestion des index lors de la conversion d'une base relationnelle vers un modèle Property Graph. En effet, dans les bases de données relationnelles, les index sont utilisés pour accélérer les opérations de recherche, de sélection et de jointure. Cependant, la manière dont ces index sont gérés et traduits dans le contexte d'un graphe nécessite une attention particulière. La gestion des index dans Neo4j, notamment pour les propriétés des nœuds et des relations, peut avoir un impact significatif sur les performances des requêtes. Il serait pertinent d'étudier comment gérer la conversion des index du modèle relationnel vers un environnement graphe, tout en évitant au maximum la perte d'informations et et ainsi avoir sous d'informations toujours cohérentes.

3.3.3 L'aspect sémantique de la génération des relations entre entités à partir de modèles LLM (Large Language Models)

L'usage des modèles de langage de grande taille (LLM), tels que GPT ou BERT, pour la génération sémantique des relations entre entités représente une avenue intéressante pour l'enrichissement des graphes. En effet, notre outil de transformation ne couvre pas encore cet aspect sémantique des relations pouvant exister entre différentes entités. Par exemple, un modèle LLM pourrait être utilisé pour analyser la relation pouvant exister entre une entité **Employé** et une entité **Entreprise.** La relation résultante de ces deux entités pourrait-être : **[TravailleDans**].

Conclusion

L'étude du modèle Property Graph et du système de gestion de base de données Neo4j a permis de mettre en lumière les avantages notables de cette approche par rapport aux systèmes relationnels traditionnels. Le modèle Property Graph offre une flexibilité et une performance accrues, en particulier dans le cadre de l'analyse de données fortement connectées. Contrairement au modèle relationnel, qui souffre de limitations liées à la gestion de relations complexes et multiples, Neo4j permet une représentation plus intuitive et efficiente des données, particulièrement pour les applications telles que les réseaux sociaux, les systèmes de recommandation ou les graphes de connaissances.

La conversion d'une base de données relationnelle vers un modèle orienté graphe s'est avérée réussie grâce à l'outil développé. Cette transformation a permis de maintenir la cohérence des données tout en tirant parti de la structure des graphes pour améliorer les performances des requêtes. Les expérimentations ont montré que Neo4j est particulièrement efficace pour les requêtes impliquant des jointures complexes, où il surpasse les bases de données relationnelles en termes de temps d'exécution. Cependant, des défis demeurent, notamment en ce qui concerne la gestion des index et l'optimisation de la scalabilité de la solution pour des bases de données volumineuses.

De plus, l'aspect sémantique de la génération des relations entre entités, en utilisant des modèles de langage de grande taille comme GPT ou BERT, offre une avenue prometteuse pour enrichir les graphes avec des informations plus contextuelles et automatisées. Cette approche pourrait ouvrir de nouvelles possibilités pour l'analyse sémantique des données et améliorer la qualité des transformations.

En conclusion, bien que la conversion vers le modèle Property Graph offre des bénéfices indéniables en termes de flexibilité et de performance, il reste des améliorations à apporter, notamment sur la gestion des index et l'intégration de l'analyse sémantique pour une meilleure exploitation des données. Les recherches futures devraient se concentrer sur la scalabilité de la solution et sur l'amélioration de l'outil de conversion, en intégrant des technologies plus avancées pour automatiser et enrichir les relations entre les entités.

Par ailleurs, une expérimentation complémentaire visant la conversion inverse, de Neo4j vers une base relationnelle (SQLite), a été conduite et documentée dans l'Annexe A. Elle démontre la faisabilité d'un retour vers un modèle tabulaire (relationnel).

Annexe A : Transformation du modèle graphe vers le relationnel

1. Introduction

   Cette annexe présente le processus de conversion d'une base de données orientée graphe (modélisée avec Neo4j) vers une base relationnelle . Il s'agit d'une démarche inverse à celle détaillée dans le corps du rapport (relationnel vers Neo4j), visant à vérifier la réversibilité du modèle et à proposer une structure exploitable pour des systèmes SQL classiques.

2. Architecture du processus

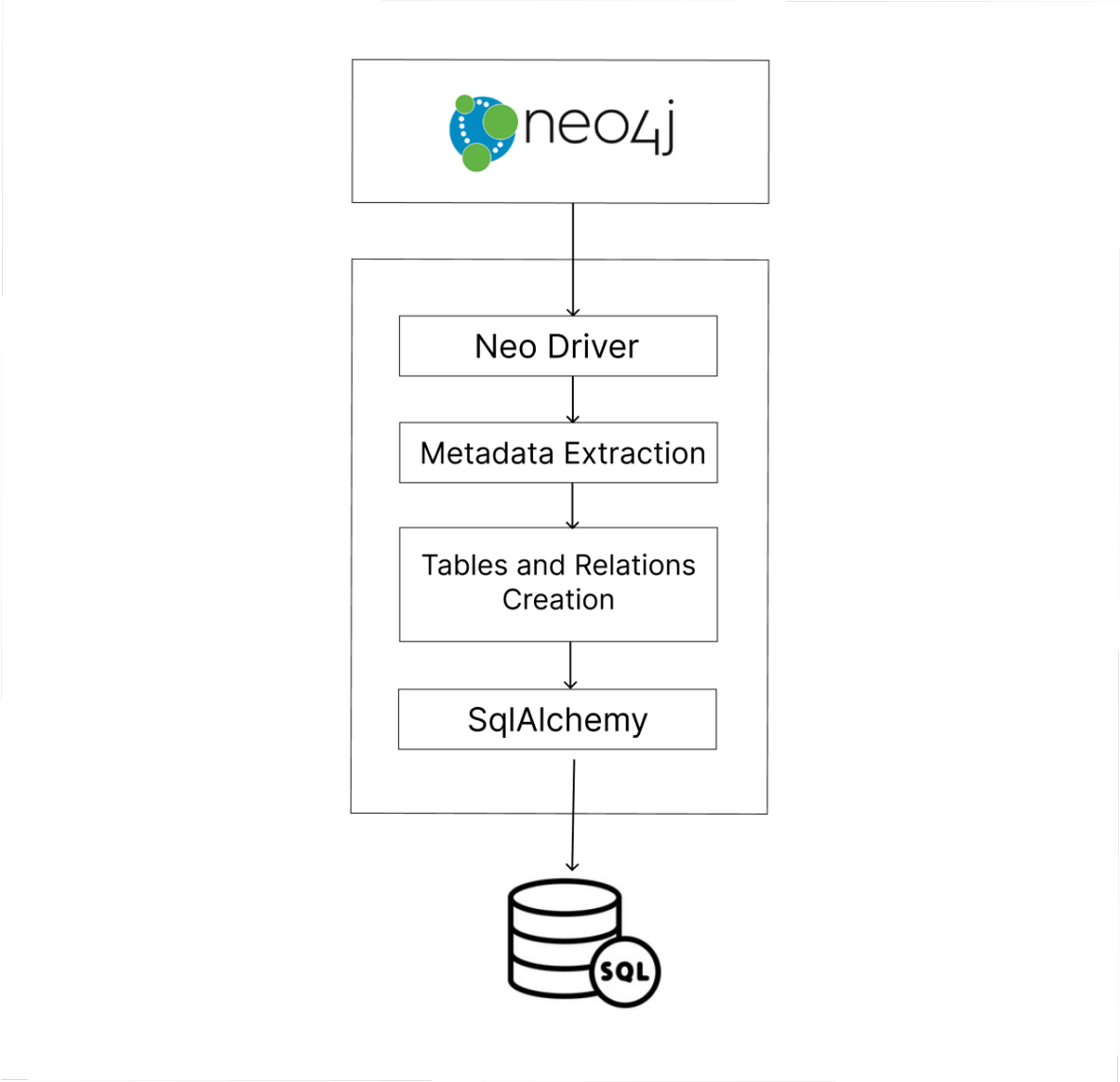


FIGURE 14 – Diagramme de flow de conversion du modèle graphe vers le relationnel

Le processus de conversion est entièrement automatisé et suit les étapes suivantes :

(a) **Connexion à Neo4j et extraction** de tous les labels (types de nœuds).

(b) **Création des tables SQLite** à partir des propriétés des nœuds (chaque label devient une table).

(c) **Détection des relations (edges)** entre nœuds et classification :

— relations **1-N** → clé étrangère,

— **relations N-N** → table d'association avec clés étrangères et propriétés si présent.

(d) **Application des contraintes** : clés primaires, not null, unique (détection dynamique).

(e) **Insertion des données** (nœuds puis relations).

3. Conclusion

Cette conversion inverse confirme que le modèle Property Graph peut être projeté vers une structure relationnelle exploitable. Bien que certaines spécificités de Neo4j (propriétés sur les relations, multétiquetage) n'aient pas d'équivalent direct en SQL, la transformation permet une exploitation cohérente des données.