Étude du modèle "Property Graph" et du language de requêtes "Cypher"

Diogo Miguel OLIVEIRA PEREIRA

20 mai 2024

Résumé

Ce projet explore la mise en œuvre d'une base de donnée en graphe de propriété (property graph), visant spécifiquement à maintenir un entrepôt de données mis à jour permettant une analyse par des équipes spécialisés. En passant des structures de bases de données relationnelles traditionnelles à des modèles de property graph plus flexibles, ce projet permet l'étude des changements/modifications de données afin d'obtenir un entrepôt de données actualisé.

Introduction

Un graphe de propriété est un type de base de données orientée graphes où chaque nœud et chaque arête peuvent posséder un ensemble de propriétés sous forme de paires key :value. Les nœuds représentent des entités, tandis que les arêtes représentent les relations entre ces entités.

Ce modèle est particulièrement utile pour représenter des structures de données complexes (systèmes de recommendation, réseaux sociaux,..) ; ils offrent plusieurs avantages : ils permettent de représenter facilement des structures de données complexes et hétérogènes (*flexibilité*), optimisent les requêtes sur les relations, surpassant souvent les bases de données relationnelles pour les opérations sur les réseaux (*performance*), et la modélisation des données sous forme de graphes est souvent plus intuitive et naturelle, facilitant la compréhension et l'analyse des données (*intuitivité*). Le DQL (Data Query Language) associé à ce modèle, est nommé "Cypher". Il se distingue par sa syntaxe déclarative et expressive, permettant de formuler des requêtes complexes de manière simple et lisible. Cypher facilite la navigation et l'interrogation des relations dans un graphe, offrant des outils pour extraire des insights détaillés et exploiter pleinement les capacités des bases de données orientées graphes.

1 Modèle d'un graphe de propriété

Un **graphe de propriété** (property graph) est défini comme un multigraphe orienté étiqueté où chaque nœud (sommet) ou arête peut avoir un ensemble de paires propriété-valeur qui lui sont attachées. La structure formelle d'un graphe de propriétés est représentée par une n-uplet $G = (N, E, \rho, \lambda, \sigma)$, où :

- **Nœuds** N : Un ensemble fini de nœuds, chacun représentant une entité.
- **Arêtes** E : Un ensemble fini d'arêtes, chacune représentant une relation entre deux nœuds.
- **Fonction d'incidence** ρ : Une fonction totale qui associe chaque arête dans E à une paire de nœuds dans N, définissant la connexion entre les nœuds.
- **Fonction d'étiquetage** λ : Une fonction partielle qui attribue un ensemble d'étiquettes provenant d'un ensemble infini d'étiquettes L à chaque nœud ou arête. Ces étiquettes catégorisent les nœuds et les arêtes en différents types basés sur leurs rôles ou caractéristiques au sein du graphe.
- **Fonction de propriété** σ : Une fonction partielle qui associe à chaque nœud ou arête un ensemble fini de propriétés tirées d'un ensemble infini de noms de propriétés P, et attribue un ensemble de valeurs provenant d'un ensemble infini de valeurs atomiques V à ces propriétés.

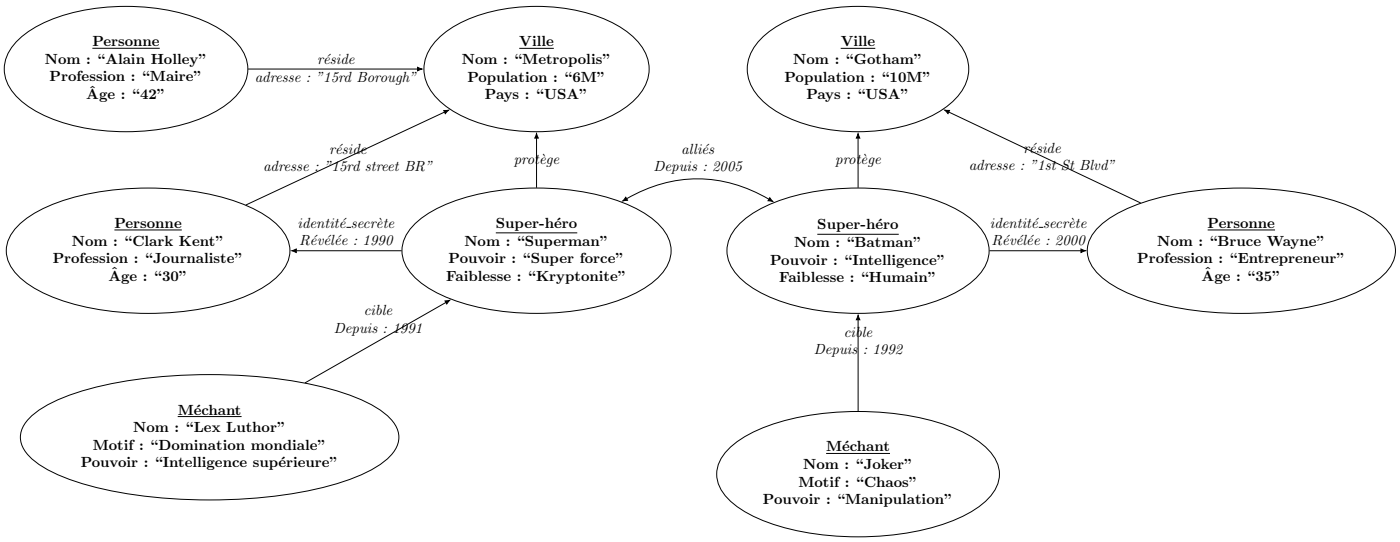


FIGURE 1 – Représentation d'un modèle de graphe de propriété sur l'information de super-héro fictif

Dans la figure ci-dessus, nous visualisons un exemple concret de modélisation dans un graphe de propriétés, illustrant les interactions complexes entre diverses entités telles que les villes, les super-héros, et d'autres personnages. Ce modèle capture non seulement les attributs individuels de chaque entité—comme les noms et les caractéristiques, tel que l'age, profession.. —mais également les relations dynamiques telles que les lieux de résidence, les identités secrètes révélées, et les alliances formées au fil du temps.

D'après notre définition d'un graphe de propriété, on a que :

$N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$
$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$
$\lambda(n_1) = \{\text{Ville}\}, (n_1, \text{nom}) = \text{“Metropolis”}, (n_1, \text{population}) = \text{“6M”}, (n_1, \text{pays}) = \text{“USA”}$
$\lambda(n_2) = \{\text{Ville}\}, (n_2, \text{nom}) = \text{“Gotham”}, (n_2, \text{population}) = \text{“10M”}, (n_2, \text{pays}) = \text{“USA”}$
$\lambda(n_3) = \{\text{Personne}\}, (n_3, \text{nom}) = \text{“Alain Holley”}, (n_3, \text{profession}) = \text{“Maire”}, (n_3, \text{âge}) = \text{“42”}$
$\lambda(n_4) = \{\text{Personne}\}, (n_5, \text{nom}) = \text{“Clark Kent”}, (n_5, \text{profession}) = \text{“Journaliste”}, (n_5, \text{âge}) = \text{“30”}$
$\lambda(n_5) = \{\text{Personne}\}, (n_6, \text{nom}) = \text{“Bruce Wayne”}, (n_6, \text{profession}) = \text{“Entrepreneur”}, (n_6, \text{âge}) = \text{“35”}$
$\lambda(n_6) = \{\text{Super-héro}\}, (n_6, \text{nom}) = \text{“Superman”}, (n_6, \text{pouvoir}) = \text{“Super force”}, (n_6, \text{faiblesse}) = \text{“Kryptonite”}$
$\lambda(n_7) = \{\text{Super-héro}\}, (n_7, \text{nom}) = \text{“Batman”}, (n_7, \text{pouvoir}) = \text{“Intelligence”}, (n_7, \text{faiblesse}) = \text{“Humain”}$
$\lambda(n_8) = \{\text{Méchant}\}, (n_8, \text{nom}) = \text{“Joker”}, (n_8, \text{motif}) = \text{“Chaos”}, (n_8, \text{pouvoir}) = \text{“Manipulation”}$
$\lambda(n_9) = \{\text{Méchant}\}, (n_9, \text{nom}) = \text{“Lex Luthor”}, (n_9, \text{motif}) = \text{“Domination mondiale”},$
$(n_9, \text{pouvoir}) = \text{“Intelligence supérieure”}$
$\sigma(e_1) = (n_3, n_1), \lambda(e_1) = \{\text{réside}\}, (e_1, \text{adresse}) = \text{“15rd Borough”}$
$\sigma(e_2) = (n_4, n_1), \lambda(e_2) = \{\text{réside}\}, (e_2, \text{adresse}) = \text{“15rd street BR”}$
$\sigma(e_3) = (n_5, n_2), \lambda(e_3) = \{\text{réside}\}, (e_3, \text{adresse}) = \text{“1st St Blvd”}$
$\sigma(e_4) = (n_9, n_6), \lambda(e_4) = \{\text{cible}\}, (e_4, \text{depuis}) = \text{“1991”}$
$\sigma(e_5) = (n_8, n_7), \lambda(e_5) = \{\text{cible}\}, (e_5, \text{depuis}) = \text{“1992”}$
$\sigma(e_6) = (n_6, n_1), \lambda(e_6) = \{\text{protège}\}$
$\sigma(e_7) = (n_7, n_2), \lambda(e_7) = \{\text{protège}\}$
$\sigma(e_8) = (n_6, n_4), \lambda(e_8) = \{\text{identité\_secrète}\}, (e_8, \text{révélée}) = \text{“1990”}$
$\sigma(e_9) = (n_7, n_5), \lambda(e_9) = \{\text{identité\_secrète}\}, (e_9, \text{révélée}) = \text{“2000”}$
$\sigma(e_{10}) = (n_6, n_7), \lambda(e_{10}) = \{\text{allies}\}, (e_{10}, \text{depuis}) = \text{“2005”}$
$\sigma(e_{11}) = (n_7, n_6), \lambda(e_{11}) = \{\text{allies}\}, (e_{11}, \text{depuis}) = \text{“2005”}$

2 Contraintes d'intégrité

Les contraintes d'intégrité dans un modèle de graphe de propriété définissent l'ensemble des états de la base de données cohérents, les changements d'état, ou les deux. Elles comprennent la cohérence instance-schéma, les contraintes d'identité et d'intégrité référentielle, ainsi que les dépendances fonctionnelles et d'inclusion.

Ces contraintes sont cruciales pour maintenir la précision, la consistance et la validité des données au sein du graphe, en spécifiant les types de nœuds et d'arêtes, ainsi que les propriétés associées. Voici les principales contraintes appliquées dans les graphes de propriétés :

1. **Unicité des Identifiants** : Chaque nœud et arête doit avoir un identifiant unique, assurant une identification et référence distincte pour les opérations de recherche, mise à jour ou suppression.
2. **Contraintes de Typage** : Le schéma du graphe définit des types spécifiques pour les nœuds et les arêtes avec des propriétés bien définies, garantissant que les données correspondent aux attentes du modèle.
3. **Contraintes de Relation** : Les arêtes doivent respecter des règles spécifiques, par exemple, une arête de type *cible* doit lier un nœud de type *Méchant* à un nœud de type *Super-héro*, assurant des relations logiques et cohérentes.
4. **Intégrité Référentielle** : Assure que les identifiants de nœuds référencés par des arêtes existent dans le graphe, prévenant les références orphelines et garantissant la cohérence des liens.
5. **Contraintes de Valeur** : Des règles pour les valeurs de propriétés, comme des plages admissibles ou des formats spécifiques, par exemple, l'adresse ou `réside` une personne, doit être une chaîne de caractères.

Ces contraintes sont essentielles pour assurer que les données stockées dans le graphe de propriétés sont précises et exploitables pour l'analyse et les requêtes, utilisant des outils comme le langage de requête Cypher pour interroger efficacement ces structures.

2.1 Schéma d'un graphe de propriété

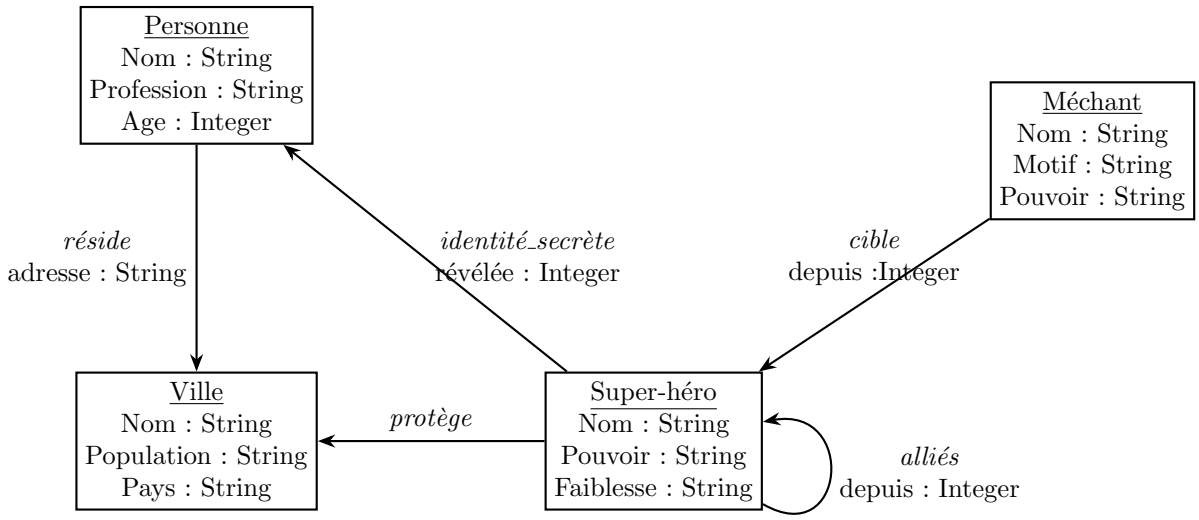


FIGURE 2 – Schéma d'un graphe de propriété

L est un ensemble infini de libellés, P est un ensemble infini de propriétés, et T est un ensemble fini de types de données (par exemple, *String*, *Integer*, etc.). Informellement, un schéma de graphe de propriétés est un schéma de métadonnées qui décrit les types de nœuds, les types de liens, et les propriétés pour de tels types (y compris les types de données correspondants).

Définition. Un schéma de graphe de propriétés est un n-uplet $S = (T_N, T_E, \beta, \delta)$ où :

1. $T_N \subset L$ est un ensemble fini de libellés représentant les types de nœuds ;
2. $T_E \subset L$ est un ensemble fini de libellés représentant les types de liens, satisfaisant que T_E et T_N sont disjoints ;
3. $\beta : (T_N \cup T_E) \times P \to T$ est une fonction partielle qui définit les propriétés pour les types de nœuds et de liens, et les types de données des valeurs correspondantes ;
4. $\delta : (T_N, T_N) \to \text{SET}(T_E)$ est une fonction partielle qui définit les types de liens autorisés entre une paire donnée de types de nœuds.

La Figure 2 présente une représentation graphique d'un schéma de graphe de propriétés pour le graphe de propriétés présenté dans la Figure 1. La description formelle de ce schéma est comme suit :

— $T_N = \{\text{Ville, Personne, Super-Héro, Méchant}\}$
— $T_E = \{\text{réside, cible, identité\_secrete, alliés, protège}\}$
— $\beta(\text{Méchant}, \text{Nom}) = \text{String}, \ldots$
— $\delta(\text{Méchant}, \text{Super-héro}) = \{\text{cible}\}$, $\delta(\text{Personne}, \text{Ville}) = \{\text{réside}\}$

3 Conception de la base de donnée

L'objectif de cette section est de concevoir une base de données relationnelle qui sera par la suite transformée en un graphe de propriétés, en détaillant son schéma pour une implémentation, en utilisant Cypher. Cette démarche se divise en plusieurs étapes :

- **Modélisation ER (Entité-Relation)** : Nous débutons par la création d'un diagramme ER pour structurer les données selon une approche relationnelle. Ce modèle comprendra les entités principales telles que les étudiants, les enseignants, les projets, et les laboratoires, avec leurs relations respectives.
- **Traduction en graphe de propriétés** : Ensuite, le diagramme ER est transformé en un graphe de propriétés. Cette transformation comprend la reconfiguration des entités en nœuds et des relations en arêtes, avec des propriétés détaillées attachées à chaque élément. Chaque entité et relation du diagramme ER sera réévaluée pour s'assurer qu'elle capture toutes les interactions nécessaires et les attributs informatifs dans le graphe.
- **Implémentation dans un SGBD adapté** : La base de données en graphe sera mise en œuvre dans un système de gestion de base de données (SGBD) qui supporte les graphes de propriétés (Neo4j). Cette étape comprend également l'écriture de requêtes en langage Cypher pour l'implémentation.

3.1 Spécifications

Cette base de donnée consiste au développement d'un système capable de suivre la sélection de projets par les étudiants. Les enseignants proposent des projets, avec une quantité spécifiée d'étudiants. Ils peuvent appartenir à des entités externes (laboratoires, par exemple) en tant que chercheur, consultant... Les laboratoires peuvent également proposer des projets, dans lesquels un enseignant est désigné en tant que "directeur". Un projet a un titre et des domaines d'expertise ; il a également des exigences qui doivent être assorties aux compétences et qualifications des étudiants. Un étudiant ne peut choisir qu'un seul projet, dans lequel il aurait un rôle (stagiaire). Un enseignant oriente les étudiants ; leur donne des retours, hebdomadairement, des conseils et des évaluations. Si le projet est destiné à plusieurs étudiants, des rôles plus spécifiés (chef d'équipe, etc.) leur sont attribués.

Notre système devrait répondre aux questions suivantes :

- Liste des projets proposés par des laboratoires externes ;
- Projets proposés par un enseignant spécifique ;
- Projet qu'un étudiant spécifique a choisi ;
- Projet(s) qu'un enseignant a ajouté ;
- Rôle d'une personne particulière (enseignant/étudiant) qui a choisi un projet proposé par un laboratoire (un enseignant peut diriger un projet proposé par une entité externe) ;
- Tous les étudiants qui ont choisi un projet particulier ;
- Tous les rôles existants pour un étudiant ;
- Tous les rôles existants pour un enseignant ;
- Liste des étudiants qu'un enseignant encadre.

3.2 Modélisation ER (Entité-Relation)

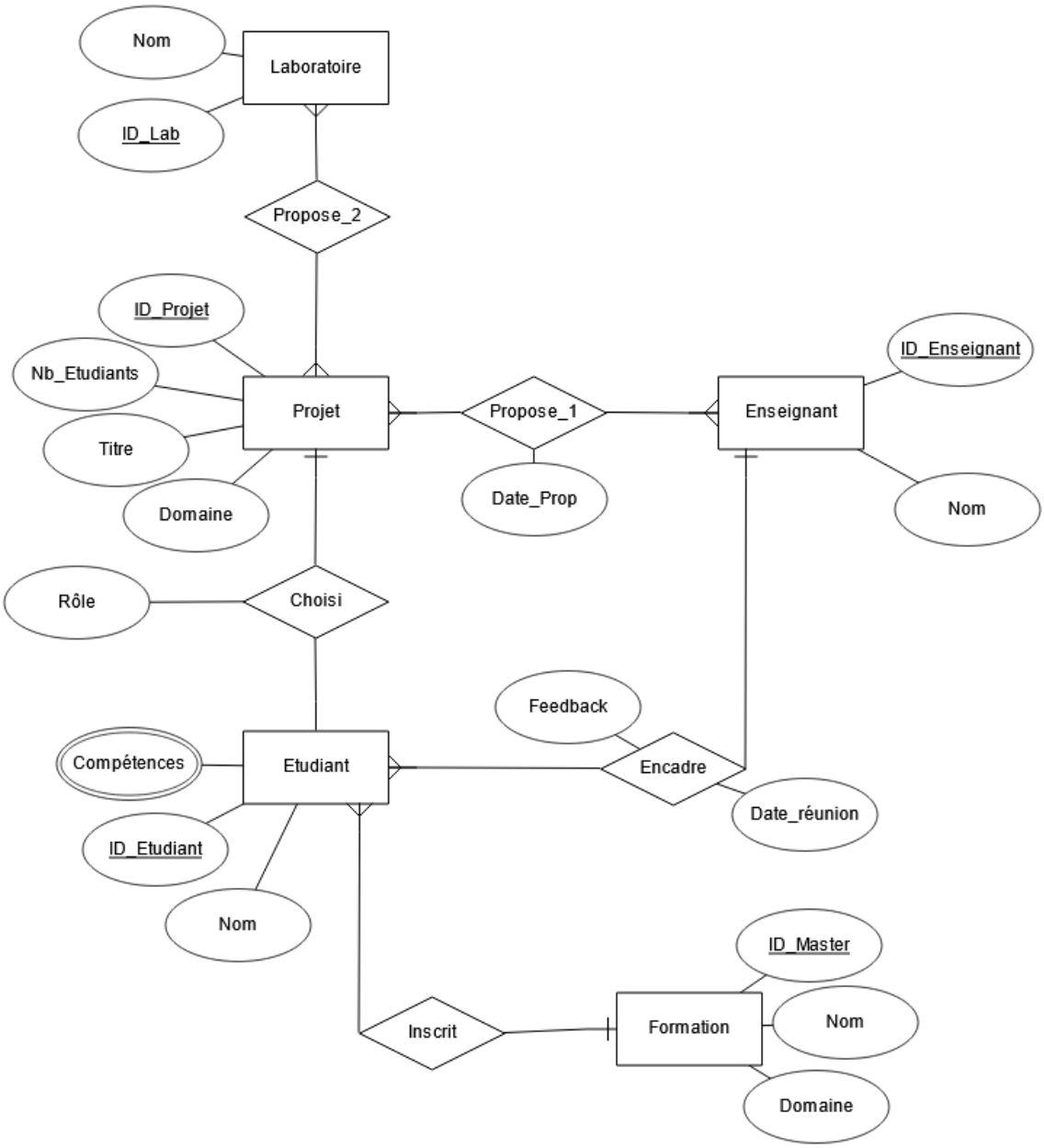


FIGURE 3 – Modèle ER (Entité-Relation) selon les spécifications

Le schéma ci-dessus illustre un modèle Entité-Relation (ER) conçu pour gérer et suivre les projets académiques dans une institution éducative. Il décrit les entités majeures telles que les laboratoires, les projets, les enseignants, les étudiants, et les formations, ainsi que leurs interrelations

3.3 Traduction en graphe de propriétés

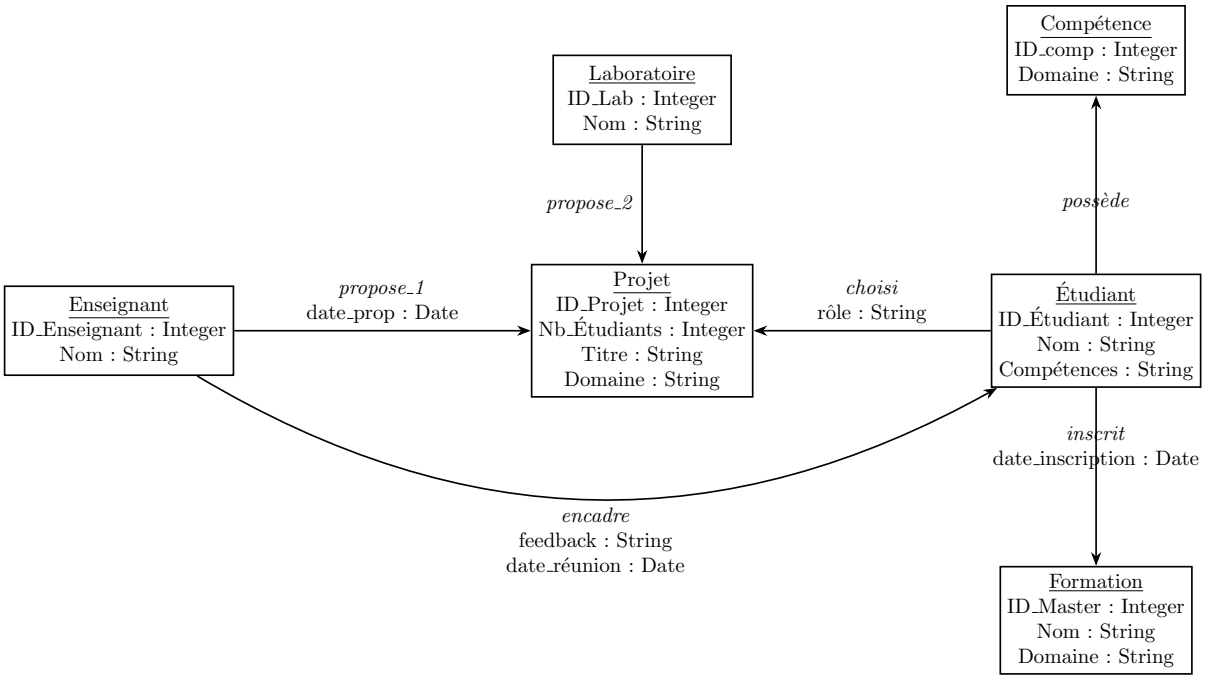


FIGURE 4 – Schéma du graphe de propriété d'après le modèle ER

De façon similaire à la description formelle de la figure 2, on a :

— $T_N = \{\text{Enseignant, Projet, Laboratoire, Etudiant, Formation, Compétence}\}$
— $T_E = \{\text{propose\_1, propose\_2, , choisi, inscrit, encadre, possède}\}$
— $\beta(\text{Projet}, \text{Titre}) = \text{String}, \ldots$
— $\delta(\text{Enseignant}, \text{Projet}) = \{\text{propose\_1}\}$, $\delta(\text{Enseignant}, \text{Etudiant}) = \{\text{encadre}\}$

La transformation du modèle Entité-Relation vers un modèle de graphe de propriétés implique plusieurs étapes pour adapter les structures et les relations du modèle traditionnel aux spécificités du graphe.

Chaque **entité** du modèle relationnel, telle que Laboratoire, Enseignant, Étudiant et Projet, est convertie en un **nœud** dans le modèle de graphe. Chaque nœud conserve les attributs de l'entité originale pour préserver les données et leur contexte. Par exemple, le nœud *Laboratoire* conserve les attributs tels que l'ID et le Nom, permettant ainsi de maintenir une continuité des informations.

Les **clés étrangères et les relations** du modèle relationnel sont transformées en **arêtes** dans le modèle de graphe. Ces arêtes, telles que *Propose_2* entre un *Laboratoire* et un *Projet*, montrent des connexions directes et sont enrichies de propriétés descriptives qui clarifient le type de relation, par exemple, la date de proposition dans l'arête *Propose_1*.

Les relations complexes qui nécessiteraient des jointures multiples dans un modèle relationnel sont représentées plus simplement dans les graphes. Ainsi, la relation *Encadre* entre un *Enseignant* et un *Étudiant* inclut directement des propriétés telles que le *Feedback* et la *Date_réunion*, facilitant l'accès et l'analyse des interactions directes sans requêtes complexes.

Le modèle de graphe permet une gestion flexible de l'héritage entre entités, ce qui est illustré par la possibilité d'ajouter des étiquettes multiples aux nœuds pour refléter différentes classes ou rôles d'une entité dans des contextes variés. Cette flexibilité est démontrée par la manière dont un *Étudiant* peut simultanément être lié à un *Projet* et une *Formation*, agissant dans des rôles différents qui pourraient être représentés par des sous-classes dans un modèle relationnel.

3.4 Implémentation

Cypher est un langage de requête spécialement conçu pour les bases de données graphes utilisant le modèle de graphes de propriété. Il permet d'exprimer de manière intuitive et expressive des requêtes complexes sur des structures de données graphiques.

Fonctionnalités Principales

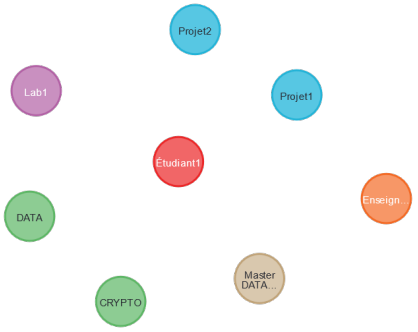
Cypher offre plusieurs fonctionnalités clés pour la manipulation et l'interrogation de graphes :

- **Requêtes sur les chemins** : Optimisé pour extraire des chemins à travers des relations complexes.
- **Syntaxe déclarative** : Utilise des clauses comme `MATCH`, `WHERE`, et `RETURN` pour simplifier la formulation des requêtes.
- **Opérations d'agrégation** : Supporte des fonctions telles que `COUNT`, `SUM`, `AVG`, permettant des analyses quantitatives complexes.

Pour mettre en œuvre le modèle de base de données décrit précédemment dans Neo4j, nous utilisons le langage de requête Cypher. Ci-dessous, nous présentons les commandes pour créer les nœuds et les relations entre eux, ainsi que des explications pour chaque portion de code.

Création des Nœuds

Les entités du modèle ER sont transformées en nœuds dans le graphe avec des attributs spécifiques :




```
CREATE (lab1:Laboratoire {ID_Lab: 1, Nom: "Lab1"})
CREATE (ens1:Enseignant {ID_Enseignant: 1, Nom: "Enseignant1"})
CREATE (projet1:Projet {ID_Projet: 1, Nb_Etudiants: 3, Titre: "Projet1", Domaine: "DATA"})
CREATE (projet2:Projet {ID_Projet: 2, Nb_Etudiants: 1, Titre: "Projet2", Domaine: "ALGO"})
CREATE (etu1:Etudiant {ID_Étudiant: 1, Nom: "Étudiant1", Compétences: "DATA"})
CREATE (forma2:Formation {ID_Master: 2, Nom: "Master DATASCALE", Domaine: "DATA"})
CREATE (com1:Competence {ID_comp: 1, Domaine: "DATA"})
CREATE (com2:Competence {ID_comp: 2, Domaine: "CRYPTO"})
```

FIGURE 5 – Code Cypher correspondant à la création des noeuds et visualisation du graphe

Création des Relations

```
MATCH (lab:Laboratoire), (projet:Projet)
WHERE lab.ID_Lab = 1 AND projet.ID_Projet = 1
CREATE (lab)-[:PROPOSE_2]->(projet)

MATCH (ens:Enseignant), (projet:Projet)
WHERE ens.ID_Enseignant = 1 AND projet.ID_Projet = 2
CREATE (ens)-[:PROPOSE_1 {Date_Prop: date('2024-05-01')}]->(projet)

MATCH (etu:Etudiant), (proj:Projet)
WHERE etu.ID_Étudiant = 1 AND proj.ID_Projet = 1
CREATE (etu)-[:CHOISI {Rôle: "Chef de projet"}]->(proj)

MATCH (etu:Etudiant), (comp1:Competence), (comp2:Competence)
WHERE etu.ID_Étudiant = 1 AND comp1.ID_comp = 1 AND comp2.ID_comp = 2
CREATE (etu)-[:POSSEDE]->(comp1),
       (etu)-[:POSSEDE]->(comp2)

MATCH (etu:Etudiant), (forma:Formation)
WHERE etu.ID_Étudiant = 1 AND forma.ID_Master = 2
CREATE (etu)-[:INSCRIT {Date_Inscription: date('2024-04-01')}]->(forma)

MATCH (ens:Enseignant), (etu:Etudiant)
WHERE ens.ID_Enseignant = 1 AND etu.ID_Étudiant = 1
CREATE (ens)-[:ENCADRE {Feedback: "Continuez ainsi", Date_réunion: date('2024-04-25')}]->(etu)
```

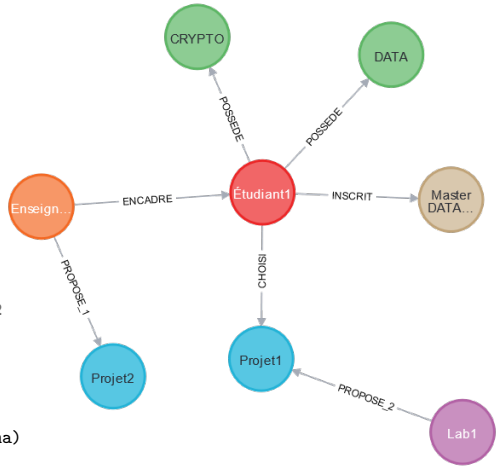


FIGURE 6 – Code Cypher correspondant à la création des relations et graphe correspondant

Les relations entre les nœuds sont établies, en fonction des connexions définies dans le modèle ER ; chaque commande `CREATE` crée un nœud dans la base de données Neo4j avec des attributs spécifiés comme le nom et l'ID. Les relations entre les nœuds sont ensuite créées en utilisant `MATCH` pour identifier les nœuds spécifiques basés sur leurs attributs, suivis de `CREATE` pour établir les liens avec les propriétés relationnelles appropriées comme les dates et les rôles.

Une fois la base de données modélisée et créée à l'aide du modèle de graphes de propriétés, nous nous tournons vers son exploitation pratique au sein de notre pipeline de données.

4 Intégration et gestion des données en temps réels

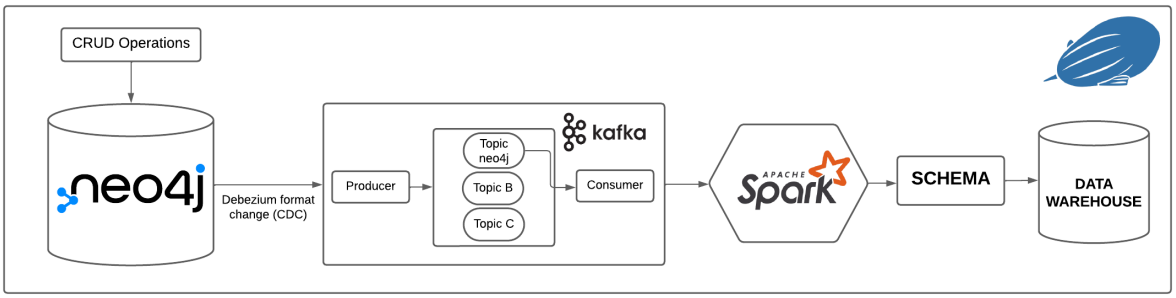


FIGURE 7 – Pipeline de données

L'intégration de Neo4j avec Apache Kafka permet de transformer les entrepôts de données traditionnels en entrepôt de données réactifs. En tirant parti des capacités des flux Neo4j, des avancées significatives dans la gestion et le streaming des données peuvent être réalisées, facilitant le développement d'une plateforme analytique en temps réel.

Fonctionnement et Architecture de Apache Kafka

Apache Kafka est une plateforme de streaming distribuée conçue pour gérer efficacement le traitement de grandes quantités de données en temps réel. Son architecture robuste permet une intégration facile avec divers systèmes de bases de données, y compris Neo4j, facilitant ainsi la capture des changements de données et leur distribution.

Composants principaux

- **Producers :** Les producteurs envoient des données à Kafka. Dans le contexte de l'intégration avec Neo4j, le module CDC de Neo4j agit comme un producteur en envoyant des événements de changement de données à Kafka.
- **Consumers :** Les consommateurs lisent ces données depuis Kafka. Ils peuvent être des applications d'analyse, des bases de données ou d'autres systèmes qui nécessitent un traitement des données en temps réel.
- **Brokers :** Kafka utilise un ensemble de serveurs appelés brokers pour gérer et stocker les flux de données. Les données sont partitionnées et répliquées à travers les brokers pour assurer la disponibilité et la résilience.
- **Topics :** Les données dans Kafka sont organisées en sujets. Chaque sujet peut être subdivisé en partitions pour permettre une distribution et un parallélisme plus larges des données. (Par défaut, les évenements seront envoyés dans le topic neo4j).

4.1 Capture des Changements de Données (CDC)

La Capture des Changements de Données (CDC) dans les flux Neo4j est une fonctionnalité essentielle qui permet la capture et le streaming des modifications de données directement dans les sujets Kafka. Ce processus assure que tous les changements dans la base de données sont immédiatement disponibles pour le traitement en aval, l'analyse ou l'intégration avec d'autres systèmes de données.

4.1.1 Format Debezium

Le module CDC de Neo4j transforme chaque transaction en éléments atomiques de la transaction qui sont ensuite diffusés.

Par exemple, la création de deux nœuds et d'une relation entre eux sera transformée en trois événements distincts (deux pour les nœuds, un pour la relation, voir figures 8, 9 et 10).

Chaque événement contient des métadonnées sur la transaction et une section de payload décrivant l'état des données avant et après la transaction.

```
{
  "meta": { },
  "payload": {
    "before": { },
    "after": { }
  }
}
```

4.1.2 Transformation en événements

Chaque modification de la base de données est capturée et transformée en événements distincts, qui sont ensuite diffusés via Kafka. Voici un exemple de transformation d'une transaction en plusieurs événements Debezium :

```
CREATE (etu:Etudiant{ID_Etudiant:2, Nom:"Diogo"})-[:CHOISI {Rôle: "Chef de
↪  Projet"}]->(proj:Projet{ID_Projet: 3, Nb_Etudiants: 1, Titre:"Property Graphs", Domaine: "DATA"})
```

On créer un étudiant nommé Diogo qui est le chef de projet pour un projet intitulé "Property Graphs" dans le domaine "DATA". Ces événements sont ensuite publiés sur le topic Kafka configuré (par défaut, topic "neo4j").

La transaction ci-dessus sera transformée en trois événements :


```
{
  "meta": {
    "timestamp":
    ↪  1615247714000,
    "username": "neo4j",
    "tx_id": 2,
    "tx_event_id": 0,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname":
      ↪  "neo4j.mycompany.com"
    }
  },
  "payload": {
    "id": "2001",
    "type": "node",
    "after": {
      "labels": ["Etudiant"],
      "properties": {
        "ID_Etudiant": 2,
        "Nom": "Diogo"
      }
    }
  }
}
```


FIGURE 8 – Création du noeud Etudiant


```
{
  "meta": {
    "timestamp":
    ↪  1615247714000,
    "username": "neo4j",
    "tx_id": 2,
    "tx_event_id": 1,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname":
      ↪  "neo4j.mycompany.com"
    }
  },
  "payload": {
    "id": "2002",
    "type": "node",
    "after": {
      "labels": ["Projet"],
      "properties": {
        "ID_Projet": 3,
        "Nb_Etudiants": 1,
        "Titre": "Property
        ↪  Graphs",
        "Domaine": "DATA"
      }
    }
  }
}
```


FIGURE 9 – Création du noeud Projet


```
{
  "meta": {
    "timestamp": 1615247714000,
    "username": "neo4j",
    "tx_id": 2,
    "tx_event_id": 2,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname":
      ↪  "neo4j.mycompany.com"
    }
  },
  "payload": {
    "id": "2003",
    "type": "relationship",
    "label": "CHOISI",
    "start": {
      "labels": ["Etudiant"],
      "id": "2001"
    },
    "end": {
      "labels": ["Projet"],
      "id": "2002"
    },
    "after": {
      "properties": {
        "Rôle": "Chef de Projet"
      }
    }
  }
}
```


FIGURE 10 – Création de la relation Etudiant/Projet

4.2 Traitement des données

Les **événements** doivent être ensuite consommés afin d'alimenter notre entrepôt de données ; pour cela, on utilisera Apache Spark, permettant d'intégrer une pipeline basée sur une architecture "schema-on-write", offrant une structure plus *flexible*.

4.2.1 Schema-On-Read vs Schema-On-Write

Le choix entre "schema-on-read" et "schema-on-write" est crucial pour la conception de notre pipeline de données. Le "schema-on-write", utilisé dans les entrepôts de données, nécessite la définition d'un schéma avant l'écriture des données. Cela permet une **validation stricte** et optimise les requêtes, mais peut être **rigide et coûteux** en maintenance lors de changements de données.

À l'inverse, le "schema-on-read" définit le schéma des données à la lecture, offrant ainsi une *grande flexibilité*. Les données peuvent être stockées sous forme brute et interprétées selon le contexte de la requête. Apache Spark, grâce à ses capacités de traitement en streaming et son intégration avec le module CDC, facilite l'implémentation d'un entrepôt de données en temps réel utilisant le "schema-on-read", rendant le système plus adaptable aux évolutions des données.

4.2.2 Simulation d'un changement (CREATE - operation)

La librairie **APOC** (Awesome Procedures On Cypher) est une extension de Neo4j offrant une large gamme de procédures utiles pour la manipulation et l'analyse de graphes. APOC simplifie grandement de nombreuses tâches courantes, telles que le traitement de chaînes, les transformations de données, et la gestion de transactions. Dans notre contexte, APOC est particulièrement utile pour simuler l'ajout de projets par des enseignants.

Pour illustrer ces capacités, considérons un exemple où un enseignant ajoute un nouveau projet.

Création périodique de projet et association

```
CALL apoc.periodic.repeat(
  'create-ens-and-proj',
  '
  MERGE (e:Enseignant {ID_Enseignant: 1, Nom: "Ens-T"})

  WITH e, {id: apoc.create.uuid(), nbEtudiants: 0, titre: "Projet-" + apoc.text.random(5),
  ↪  domaine: "N/A"} AS project

  CREATE (proj:Projet {ID_Projet: project.id, Titre: project.titre, Nb_Etudiants:
  ↪  project.nbEtudiants, Domaine: project.domaine})

  WITH e, proj
  CALL apoc.create.relationship(e, "PROPOSE_1", {date_prop: date()}, proj) YIELD rel

  RETURN e, proj, rel;
  ',
  10
)
```

Dans ce script, nous utilisons la procédure 'apoc.create.uuid()' pour générer des identifiants uniques pour chaque projet. Les projets sont créés et associés à un enseignant spécifié via la relation 'PROPOSE_1'. Cette opération est déclenchée toutes les 10 secondes pour ajouter un nouveau projet, et grâce au CDC NEO4j, l'événement sera automatiquement capturé et transmis à Kafka.

Nous allons également créer les index suivants, permettant, par la suite, la manipulation et l'analyse des données présentes :

```
CREATE INDEX ON :Enseignant(ID_Enseignant);
CREATE INDEX ON :Projet(ID_Projet);
```

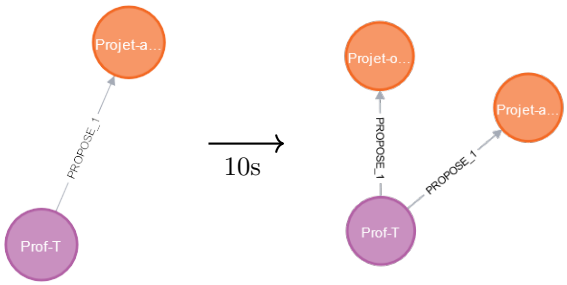


FIGURE 11 – Deux executions de la procédure 'create-ens-and-proj'

4.2.3 Récupération des données

Une fois la procédure exécutée, nous allons maintenant utiliser Apache Spark afin de consommer les données publiées dans le topic `neo4j`. Pour cela, nous allons suivre les étapes suivantes :

1. Configuration de Apache Spark pour la lecture des événements Kafka
2. Définition du schéma des données de CDC
3. Transformation des données brutes Kafka en JSON structuré
4. Extraction des événements de création de nœuds "Projet"
5. Écriture des données structurées dans le système de fichiers

⋆ ⋆ ⋆

1 - Lecture des évenements

La lecture des événements à partir de Kafka permet de capturer en temps réel les modifications survenues dans la base de données Neo4j. Apache Spark (en language Scala) est utilisé pour lire ces événements, les transformer en un format structuré et les enregistrer dans un système de fichiers.

La configuration d'Apache Spark pour la lecture des événements Kafka implique la définition de la connexion à Kafka et la spécification du topic `neo4j` à partir duquel les données seront lues.

Apache Spark API permet de traiter les données en flux de manière continue et en temps réel.

Elle utilise un *modèle de traitement incrémental* où les données sont traitées en micro-lots, comme illustré dans la figure 12. Chaque micro-lot capture les données disponibles jusqu'à un point de temps donné, exécute des requêtes sur ces données et génère des résultats incrémentaux qui sont ensuite écrits dans un système de stockage cible.

Ce modèle assure une **faible latence et une consistance** des données, ce qui est crucial pour les applications nécessitant un traitement en temps réel.

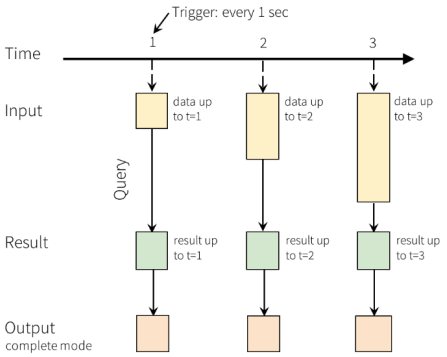


FIGURE 12 – Exécution incrémentale dans Apache Spark.

Le dataframe suivant `kafkaStreamingDF` récupère les données en flux du topic correspondant au module CDC de NEO4j :

```
val kafkaStreamingDF = (spark
    .readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "broker:9093")
    .option("subscribe", "neo4j")
    .load())
```

Pour cela, on spécifie le serveur Kafka (`kafka.bootstrap.servers` sur le port `9093`) et le topic (`subscribe`) à partir duquel les données doivent être lues. (par féfaut, neo4j)

2 - Définition du schéma

Le schéma CDC structuré définit la structure des événements de changement de données capturés par Neo4j et publiés dans Kafka.
Il comprend des métadonnées telles que le timestamp, le nom d'utilisateur, l'opération effectuée, et la source de l'événement. Le payload inclut les détails de l'entité modifiée, comme les identifiants, les types de nœuds et les propriétés avant et après la modification. (voir figure 10)

```
val cdcSchema = new StructType()
  .add("meta", new StructType()
    .add("timestamp", LongType)
    .add("username", StringType)
    .add("operation", StringType)
    .add("source", MapType(StringType,
    ↪  StringType)))
  .add("payload", new StructType()
    .add("id", StringType)
    .add("type", StringType)
    .add("label", StringType)
    .add("start", MapType(StringType, StringType))
    .add("end", MapType(StringType, StringType))
    .add("before", new StructType()
      .add("labels", ArrayType(StringType))
      .add("properties", MapType(StringType,
      ↪  StringType)))
    .add("after", new StructType()
      .add("labels", ArrayType(StringType))
      .add("properties", MapType(StringType,
      ↪  StringType))))
```

3 - Transformation en JSON

Une fois le schéma défini, on converte les données brutes de Kafka en JSON, structuré selon le schéma `cdcSchema`.

```
val cdcDataFrame = (kafkaStreamingDF
    .selectExpr("CAST(value AS STRING) AS VALUE")
    .select(from_json('VALUE, cdcSchema) as 'JSON))
```

4 - Extraction des événements de création de nœuds

Pour extraire les événements spécifiques à la création de nœuds avec le label "Projet" à partir des données de changement capturées par Neo4j et publiées dans Kafka, nous appliquons un filtrage sur le DataFrame.

```
val projectEventsDF = cdcDataFrame
  .where("json.payload.type = 'node' AND array_contains(json.payload.after.labels, 'Projet')")
  .selectExpr(
    "json.payload.id AS neo_id",
    "CAST(json.meta.timestamp / 1000 AS Timestamp) AS timestamp",
    "json.meta.source.hostname AS host",
    "json.meta.operation AS operation",
    "json.payload.after.labels AS labels",
    "explode(json.payload.after.properties) AS (key, value)"
  )
```

Nous utilisons la méthode `where` pour filtrer les événements de type nœud et vérifier que ces nœuds portent le label "Projet". Ensuite, avec `selectExpr`, nous sélectionnons les propriétés pertinentes des projets : l'identifiant du nœud (`neo_id`), le timestamp de la transaction converti en format lisible (`timestamp`), le nom de l'hôte source (`host`), l'opération effectuée (`operation`), les labels associés au nœud (`labels`), et enfin nous utilisons `explode` pour extraire les propriétés du projet sous forme de paires key :value.

5 - Écriture des données structurées dans le système de fichiers

Pour écrire les données structurées dans un système de fichiers en utilisant Apache Spark, nous configurons un flux de données pour la sortie. Le code suivant montre comment cette opération est effectuée en Scala.

```scala
val query_writeStream = projectEventsDF
    .writeStream
    .outputMode("append")
    .format("json")
    .option("checkpointLocation", "/zeppelin/static_json/data_warehouse/checkpoint")
    .option("path", "/zeppelin/static_json/data_warehouse")
    .start()

query_writeStream.awaitTermination()
```

Le code JSON suivant correspond aux événements capturés par le module CDC de Neo4j, transformés et structurés par Apache Spark, puis écrits dans le système de fichiers


```
{"neo_id":"1","timestamp":"2024-05-19T16:43:27.463Z","host":"neo4j","operation":"created",
↪  "labels":["Projet"],
"key":"ID_Projet","value":"1213572c-9c8e-4006-8127-0b206968517e"}
{"neo_id":"1","timestamp":"2024-05-19T16:43:27.463Z","host":"neo4j","operation":"created",
↪  "labels":["Projet"],
"key":"Titre","value":"Projet-Is6SR"}
{"neo_id":"1","timestamp":"2024-05-19T16:43:27.463Z","host":"neo4j","operation":"created",
↪  "labels":["Projet"],
"key":"Nb_Etudiants","value":"0"}
{"neo_id":"1","timestamp":"2024-05-19T16:43:27.463Z","host":"neo4j","operation":"created",
↪  "labels":["Projet"],
"key":"Domaine","value":"N/A"}
```

Conclusion

Ce projet a exploré l'implémentation d'une base de données en **graphe de propriété** pour maintenir un entrepôt de données constamment mis à jour. En passant des structures traditionnelles de bases de données relationnelles aux modèles de **property graph** plus flexibles, plusieurs avantages ont été mis en évidence.

Les **graphes de propriété** offrent une représentation plus intuitive et naturelle des structures de données complexes. Grâce à leur capacité à optimiser les **requêtes sur les relations** et à surpasser souvent les bases de données relationnelles en termes de **performance** pour les opérations sur les réseaux, les graphes de propriété facilitent grandement l'analyse et la compréhension des données interconnectées.

Enfin, l'intégration de **Neo4j** avec **Apache Kafka** pour transformer les entrepôts de données traditionnels en **entrepôts de données réactifs** a été examinée. Cette intégration permet la capture des changements de données en temps réel et leur distribution, renforçant ainsi la capacité d'analyse en temps réel et la réactivité des systèmes de gestion de données.

En somme, l'étude et la mise en œuvre d'une base de données en **graphe de propriété** en utilisant **Cypher** ont prouvé être une solution efficace et performante pour gérer des données complexes et interconnectées, tout en permettant une analyse et une gestion des données en temps réel grâce à l'intégration avec des technologies modernes (Kafka, Spark API).

Références

[1] Neo4j, *Guide : Import Relational Data and ETL*, https://neo4j.com/docs/getting-started/appendix/tutorials/guide-import-relational-and-etl/

[2] Apache Spark, *Spark Streaming*, https://spark.apache.org/streaming/

[3] Medium, *How to Leverage Neo4j Streams and Build a Just-in-Time Data Warehouse*, https://medium.com/free-code-camp/how-to-leverage-neo4j-streams-and-build-a-just-in-time-data-warehouse-64adf290f093

[4] Angles, *The Property Graph Database Model*, Semantic Scholar, https://www.semanticscholar.org/paper/The-Property-Graph-Database-Model-Angles/91d6e8ba5dd90b02fe3bd870b19da13a6167af53

[5] Neo4j, *Getting Started with Neo4j*, https://neo4j.com/docs/getting-started/

[6] Bonifati et al., *Graph Data Management : Fundamental Concepts and Algorithms*, https://perso.liris.cnrs.fr/angela.bonifati/pubs/book-Bonifati-et-al-18.pdf

[7] Apache Kafka, *Kafka Streams Architecture*, https://kafka.apache.org/10/documentation/streams/architecture

[8] Superruzafa, *Visual Scala Reference*, https://superruzafa.github.io/visual-scala-reference/

[9] Neo4j, *Change Data Capture (CDC)*, https://neo4j.com/docs/cdc/current/

[10] Neo4j, *Cypher Cheat Sheet : AuraDB Enterprise*, https://neo4j.com/docs/cypher-cheat-sheet/5/auradb-enterprise/