Université Abou Bekr Belkaid
Tlemcen Algérie
جامعة أبي بكر بلقايد
تلمسان الجزائر

République Algérienne Démocratique et Populaire
Université Abou Bakr Belkaid– Tlemcen
Faculté des Sciences
Département d'Informatique

**Mémoire de fin d'études pour l'obtention du diplôme de Master en Informatique**

***Option : Génie Logiciel (G.L)***

# Thème

## Gestion de la jointure spatiale lors de l'exploration des graphes RDF

Réalisé par :

- Mébarka Meriem Benikhlef

Jury :

- M. Azeddine CHIKH (Président)
- M. Mohamed LEHSAINI (Examinateur)
- M. Houcine MATALLAH (Encadrant)
- M. Nadir GUERMOUDI (Co-encadrant)

***27/06/2024***

## Remerciements :

✓ Je souhaite tout d'abord exprimer ma profonde gratitude à mes mentors, M. Houcine MATALLAH, M. Nadir Guermoudi et M. Amine Mesmoudi, pour leur soutien, leurs conseils avisés et leur patience tout au long de ce projet. Leurs connaissances approfondies et leur dévouement ont été essentiels à la réussite de cet effort.

✓ Je tiens également à exprimer ma sincère gratitude à l'Université Abou Bekr Belkaid pour m'avoir fourni les ressources nécessaires et pour avoir créé un environnement propice à l'apprentissage et à la recherche.

✓ Je remercie mes collègues et camarades de classe pour leur collaboration, leurs échanges riches et leurs encouragements constants. Leurs contributions ont grandement enrichi ma perspective et ma compréhension du sujet.

✓ Je tiens également à remercier ma famille et mes amis pour leur soutien, leurs encouragements et leur compréhension indéfectibles tout au long de mon parcours académique.

✓ Enfin, je souhaite remercier tous ceux qui ont contribué, directement ou indirectement, à la réalisation de ce projet. Leur aide et leurs encouragements ont été précieux et ont grandement contribué à sa réussite.

Benikhlef Mébarka Meriem

## Dédicaces :

- ✓ Je tiens à exprimer ma profonde gratitude envers mes parents, Mon père Mohamed et Ma mère Badia qui ont été une source inestimable de soutien, d'amour et d'inspiration tout au long de ce voyage. Leur sacrifice, leurs encouragements inconditionnels et leur confiance en moi sont les fondations de ma réussite. Cette œuvre est dédiée à leur amour infini et à leur dévouement sans faille.

- ✓ À mes chers grands-parents et ma tante maternelle Wahiba, Mon grand père Si Abd Asslam et Ma grande mère Fatima, Je tiens à exprimer ma reconnaissance profonde pour votre sagesse, votre amour et vos précieux conseils. Votre patience et votre bienveillance ont été des phares guidant mes pas tout au long de ce voyage. Votre héritage de valeurs et de traditions est une source d'inspiration constante, et cette œuvre est dédiée à votre amour inébranlable et à votre soutien indéfectible.

- ✓ À mes sœurs Hadjer et Fatima Zohra, merci d'avoir partagé avec moi les hauts et les bas de cette aventure académique. Vos encouragements, votre présence et votre amitié ont rendu les moments difficiles plus lumineux et les victoires encore plus précieuses. Votre soutien indéfectible est une source précieuse de force et de motivation.

- ✓ À mes frères Mehdi Zakaria et Tarik Rached, merci pour votre soutien constantet votre encouragement. Votre présence et vos mots d'encouragement ont été d'une grande aide tout au long de ce parcours.

- ✓ À M. Nadir Guermoudi, vos conseils judicieux, votre expertise et votre patience infinie ont été essentiels pour mener à bien ce projet. Votre dévouement et votre accompagnement ont été une boussole tout au long de mon parcours académique.

- ✓ Enfin, je me remercie pour ma persévérance et ma foi en ma capacité à surmonter les défis. Que ce succès soit un rappel constant de la force et de la détermination qui sont en moi.

Benikhlef Mébarka Meriem

# Table Des Matières


**1 Introduction** **10**
- 1.1 Contexte du projet . . . 10
- 1.2 Problématique . . . 11
- 1.3 Objectif . . . 11
- 1.4 Intégration de l'équipe RDF_QDAG . . . 12

**2 Analyse de l'existant : RDF_QDAG** **14**
- 2.1 Introduction . . . 14
- 2.2 Présentation architecturale de RDF_QDAG . . . 14
- 2.3 Stratégies d'évaluation des requêtes . . . 16
  - 2.3.1 BGP-First: . . . 16
  - 2.3.2 Stratégie Spatial-First . . . 18
- 2.4 Représentation des données . . . 19
  - 2.4.1 Représentation des données RDF . . . 19
  - 2.4.2 Représentation des données spatiales avec RDF_QDAG . . . 20
- 2.5 Utilisation du WKT dans la représentation des données spatiales . . . 21
- 2.6 Fonctionnalités existantes de RDF_QDAG . . . 22
- 2.7 La syntaxe d'une requête SPARQL . . . 23
- 2.8 Le parsing des requêtes SPARQL . . . 25
  - 2.8.1 Introduction au Parsing des Requêtes : . . . 25
  - 2.8.2 Étapes du Parsing des Requêtes : . . . 25
- 2.9 Limitations actuelles . . . 26
- 2.10 Conclusion . . . 26

**3 Gestion Et Conception** **28**
- 3.1 Introduction . . . 28
- 3.2 Outils Collaboratif . . . 28
  - 3.2.1 Overleaf . . . 28
  - 3.2.2 GitHub . . . 28
  - 3.2.3 Docker . . . 29
  - 3.2.4 Teams . . . 29
  - 3.2.5 Google Meet . . . 29
  - 3.2.6 Webex . . . 30
  - 3.2.7 Draw.io . . . 30
  - 3.2.8 Visual Studio Code . . . 30
- 3.3 Modèle de développement et planning . . . 31
  - 3.3.1 Approche adoptée : Agile (Scrum) . . . 31
  - 3.3.2 Planning : . . . 32
- 3.4 Analyse et conception de l'intégration de la jointure . . . 34
- 3.5 Analyse et conception de l'intégration de la jointure spatiale . . . 37
- 3.6 Spécifications fonctionnelles et techniques . . . 40
- 3.7 l'Intégration Continue . . . 43
- 3.8 Conclusion . . . 44

**4 Réalisation** **46**
4.1 Introduction . . . 46
4.2 Configuration et Débogage du Projet RDF QDAG . . . 46
4.2.1 Installation des Outils Requis : . . . 46
4.2.2 L'utilisation de GitHub : . . . 46
4.2.3 Configuration de l'Environnement de Débogage : . . . 47
4.2.4 Exécution des Commandes pour le Débogage : . . . 47
4.3 Parsing des Requêtes et Préparation pour le Produit Cartésien . . . 47
4.4 Division des requêtes en sous-requêtes . . . 48
4.5 Gestion des Tampons de Résultats et de Remplissage dans RDF QDAG 54
4.6 Implémentation du Produit Cartésien . . . 55
4.7 Implémentaion de la jointure . . . 58
4.8 Résultats et Analyses de l'Implémentation des Stratégies d'Évaluation des Requêtes . . . 62
4.8.1 Introduction : . . . 62
4.8.2 Méthodologie : . . . 62
4.8.3 Analyse . . . 63
4.8.4 Conclusion: . . . 63
4.9 Conclusion: . . . 66

**5 Conclusion et Perspectives** **68**
5.1 Conclusion . . . 68
5.2 Perspectives : . . . 69

# Liste Des Figures


1 Présentation architecturale de RDF_QDAG . . . . . 16
2 L'exécution d'une requête selon la stratégie BGP-First . . . . . 17
3 Exécution d'une requête selon la stratégie Spatial-First . . . . . 19
4 Graph RDF VS Query . . . . . 20
5 Geometry Primitives (2D) . . . . . 21
6 Multipart Geometries (2D) . . . . . 22
7 Méthode Agile Scrum . . . . . 31
8 Diagramme de classe de la jointure . . . . . 35
9 Diagramme de classe de la jointure spatiale . . . . . 38
10 Processus métier d'une jointure . . . . . 42
11 Résultats d'une jointure entre 2 graphes des données simples . . . . . 65
12 Résultats d'une jointure entre 2 graphes pour des données spatiales . 66

# Liste Des Algorithmes


1 Exemple de requête SPARQL Simple . . . . . 23
2 Exemple de requête SPARQL Spatiale . . . . . 24
3 Requête SPARQL Avant Le Parsing . . . . . 48
4 Requête SPARQL Après Le Parsing . . . . . 48
5 Méthode Java dividerQuery(String query) . . . . . 49
6 Méthode Java extractTriplets(String qdagQuery) . . . . . 50
7 Construction Dynamique de la Clause SELECT à partir des Triplets de Graphe . . . . . 51
8 Ajout Dynamique de la Clause GROUP BY dans la Requête SPARQL 51
9 Ajout Dynamique de Filtres par rapport aux Variables du Graphe . . 52
10 Méthode Java writeAggResults(GroupesWriterBuffer bffer) . . . . . 55
11 Méthode Java writeCombinedResults() . . . . . 56
12 Méthode Java combineAndDisplayResults . . . . . 57
13 Méthode Java allIndicesWithinBounds() . . . . . 57
14 Méthode Java moveNext() . . . . . 58
15 Méthode Java extractSelectVariables() . . . . . 59
16 Méthode Java extractFilters() . . . . . 59
17 Méthode Java FilterVerification() . . . . . 60
18 Méthode Java extractValues() . . . . . 60
19 Méthode Java applyOperator() . . . . . 61
20 Méthode Java doGeometriesIntersect() . . . . . 61
21 Requête SPARQL . . . . . 64
22 graphe de requete1 . . . . . 64
23 graphe de requete2 . . . . . 64

# CHAPITRE 01 :
# INTRODUCTION

# 1 Introduction

Le Web sémantique[1] et les graphes de connaissances[2] ont ouvert de nouvelles perspectives passionnantes pour la représentation et l'exploitation de l'information. Cependant, cette évolution a présenté des défis importants, notamment en ce qui concerne la gestion efficace des données interrogées.[8]

Dans ce contexte dynamique, les systèmes RDF[3] (Resource Description Framework) et les requêtes SPARQL[4] ont gagné en importance en tant qu'outils majeurs pour représenter et interroger des données structurées selon le modèle de graphe. Le RDF est une technologie fondamentale du Web sémantique, permettant de structurer les informations en graphes de connaissances interconnectés, facilitant ainsi leur accès et leur manipulation.[7]

Notre projet de fin d'études s'inscrit dans le cadre d'une collaboration entre le laboratoire LRIT de l'université de Tlemcen (Algérie) et le laboratoire LIAS de l'université de Poitiers (France). Nous avons rejoint l'équipe de recherche qui développe le Triplestore RDF_QDAG(http://qdag.projets.univ-poitiers.fr/) pour intégrer de nouvelles techniques de traitement des données RDF, en particulier l'implémentation du produit cartésien, qui constitue la base de l'intégration des jointures, actuellement non supportée par RDF_QDAG.

## 1.1 Contexte du projet

Le concept de triplestore RDF (Resource Description Framework) et des systèmes similaires, sont principalement utilisés dans le domaine de l'informatique et de la gestion de données. Cependant, leurs applications et les avantages qu'ils apportent s'étendent à de nombreux autres domaines, comme le domaine médico-administratif[15], le domaine biomedical (en particulier de la biologie et de la physiologie)[18]

Le triplestore RDF_QDAG, développé par l'équipe de recherche du laboratoire LIAS de l'université de Poitiers, offre une solution spécialisée pour la gestion et l'interrogation de données structurées selon le modèle RDF (Resource Description Framework). En utilisant un modèle ***sujet-prédicat-objet***, il permet une représentation sémantique et une manipulation flexible des données, contrairement aux bases de données relationnelles classiques.

[1] https://www.ionos.fr/digitalguide/web-marketing/search-engine-marketing/web-semantique/

[2] https://www.ibm.com/fr-fr/topics/knowledge-graph

[3] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/

[4] https://web-semantique.developpez.com/tutoriels/jena/arq/introduction-sparql/

Dans le cadre de ce projet, nous nous concentrons sur l'intégration de la jointure spatiale dans le traitement des requêtes par RDF_QDAG. Avant d'aborder cette partie de la jointure, une étape préalable essentielle consiste à implémenter le produit cartésien.

## 1.2 Problématique

Ces dernières années, nous avons observé une évolution significative dans les pratiques de création et d'exploitation des bases de connaissances. Cette transformation est principalement due au passage du Web classique, axé sur les documents, au Web de données, permettant une interconnexion massive des données et la création de graphes de connaissances. Ces graphes apportent une sémantique contextuelle pour l'intégration, l'unification, l'analyse et le partage des données, en utilisant des technologies comme le Resource Description Framework (RDF) et le langage SPARQL.[4]

Cependant, la gestion efficace des données RDF reste un défi majeur, surtout lorsqu'il s'agit de traiter des requêtes sur des milliards de triplets RDF. Les systèmes appelés "Triplestores" ont été développés pour ce but, mais ils peinent souvent à offrir des temps de réponse acceptables et des performances garanties en raison de l'absence de schémas explicites de données.[14]

RDF_QDAG a été conçu pour surmonter ces défis en intégrant des opérateurs standards et des techniques d'optimisation, notamment pour les requêtes spatiales. Avant d'implémenter la jointure spatiale, une tâche essentielle consistait à gérer le produit cartésien, une opération de base en traitement de requêtes.

L'objectif de ce projet de fin d'études est de poursuivre cette démarche en évaluant l'utilisation d'un index de type R-Tree pour améliorer les jointures spatiales dans RDF_QDAG. Il s'agit de trouver la stratégie la plus adaptée pour optimiser les performances des requêtes spatiales, garantissant ainsi une gestion efficace des données spatiales RDF.

Avant d'implémenter la jointure spatiale, une tâche cruciale réalisée était la gestion du produit cartésien. Cette opération consiste à générer toutes les combinaisons possibles entre tous les graphes RDF, formant ainsi une base pour les opérations de jointure ultérieures. Le produit cartésien est une étape fondamentale pour comprendre les interactions possibles entre les données, bien qu'il soit coûteux en termes de ressources.

## 1.3 Objectif

L'objectif de ce projet de fin d'études est d'évaluer l'utilisation d'un index de type R-Tree pour améliorer les jointures spatiales[10] dans RDF_QDAG. Nous devons considérer des algorithmes de jointures existants et trouver la stratégie la plus adaptée pour optimiser les performances des requêtes spatiales, garantissant ainsi une gestion efficace des données spatiales.[21]

## 1.4 Intégration de l'équipe RDF_QDAG

En rejoignant l'équipe RDF_QDAG, notre rôle principal est de développer et d'intégrer la jointure [5] successive dans le triplestore RDF_QDAG. Cela implique la conception et l'implémentation d'algorithmes optimisés pour exécuter des produits cartésiens[13] itératifs entre plusieurs graphes de requêtes dans RDF[2]. Nous veillons également à ce que cette nouvelle fonctionnalité s'intègre harmonieusement avec les fonctionnalités existantes, sans compromettre les performances ni la convivialité du système.

Nos contributions spécifiques incluent :

- Intégration de la jointure spatiale : Conception et implémentation de l'algorithme de produit cartésien comme base initiale avant d'aborder la jointure spatiale.[9]
- Optimisation des performances : Création de techniques d'optimisation pour assurer une exécution rapide et scalable.[19]
- Tests et validation : Mise en place de tests unitaires et de performance pour garantir la robustesse et l'efficacité de l'intégration.

[5] https://fr.wikiversity.org/wiki/SPARQL_Protocol_and_RDF_Query_Language/Requtes_de_lecture

# CHAPITRE 02 :

# Analyse de l’existant :

# RDF QDAG

# 2 Analyse de l'existant : RDF_QDAG

## 2.1 Introduction

Dans ce chapitre, nous examinons en détail le triplestore RDF QDAG, en nous concentrant sur son architecture, la représentation et le stockage des données RDF, ainsi que ses fonctionnalités existantes. Nous analyserons également les limitations actuelles du système, en particulier l'absence de prise en charge des requêtes contenant une jointure.

## 2.2 Présentation architecturale de RDF_QDAG

La structure de RDF QDAG se compose de plusieurs couches, chacune comprenant plusieurs composants, comme le montre la figure 1. Cette partie du mémoire offre un aperçu de l'architecture du système et examine en détail le processus d'évaluation des requêtes.

1. **Stockage de données :**
   La couche de stockage de RDF QDAG joue un rôle essentiel dans le stockage et la récupération efficace de divers types de données, en mettant particulièrement l'accent sur les données graphiques et spatiales. Les données de RDF QDAG incluent divers types natifs tels que les chaînes, les entiers et les doubles. Pour garantir une interrogation optimale de ces données, RDF QDAG utilise principalement trois méthodes d'accès : l'index B+tree, l'index R-tree et un dictionnaire. Le B+tree agit comme le principal mécanisme d'indexation pour les données graphiques, facilitant l'indexation et la récupération efficace des données basées sur des paires clé-valeur. Grâce à l'utilisation de B+trees, la couche de stockage assure une organisation et un accès ordonné et optimisé aux données graphiques. L'index R-tree est utilisé pour indexer les données spatiales, offrant une gestion efficace des informations géographiques et géométriques, ce qui permet des requêtes spatiales rapides et précises.

   Afin de préserver la sémantique du graphe, qui dépend des relations entre ses éléments (ou prédicats dans le contexte de RDF), le graphe est divisé en fragments, en tenant compte de la connectivité entre eux. Une stratégie de partitionnement de graphes optimale vise à maximiser la connectivité entre les fragments tout en minimisant la connectivité à l'intérieur de chaque fragment. Cette approche maintient l'intégrité du graphe tout en améliorant les performances et l'efficacité des requêtes sur les données du graphe. La combinaison des arbres B+tree comme structure de stockage principale et d'une stratégie de partitionnement de graphes efficace permet à RDF QDAG de stocker et d'accéder efficacement aux données du graphe, tout en préservant sa sémantique et en facilitant les requêtes à haut rendement. Chaque fragment de graphe comprend une collection de Data Stars. Les Data Stars développent le concept de tuple dans le modèle relationnel. Formellement, nous définissons les Data Stars comme suit :

- **Data Star:** Le Data Star dans RDF QDAG consiste en l'ensemble des triplets partageant soit le même sujet, soit le même objet, organisé en fragments de graphe, distingués en Forward Data Star et Backward Data Star, en fonction de leur relation avec le nœud central.
- **Graph Fragment:** Un Fragment de Graphe dans RDF QDAG est défini comme un ensemble d'étoiles de données, soit Forward (avant) s'ils regroupent des Étoiles de Données Forward, et Backward (arrière) s'ils regroupent des Étoiles de Données Backward. Après partitionnement du graphique en fragments de graphique, chaque fragment est chargé dans un indice, utilisant un arbre B+ comme index dans le contexte de RDF - QDAG. Des techniques de compression sont également appliquées pour optimiser l'espace de stockage et réduire le nombre de pages chargées lors de l'évaluation des requêtes. En outre, pour gérer la taille potentiellement importante des fragments, les sujets et les objets sont remplacés par un ID s'ils sont du type String ou URI, nécessitant la création d'un dictionnaire pour stocker les paires <valeur, ID>. Enfin, une méthode d'accès spatial, l'index R-tree, est introduite pour faciliter la prise en charge des requêtes spatiales dans RDF QDAG.

2. **Couche de planification:**
   La couche de planification comprend principalement l'optimiseur, chargé de choisir le meilleur plan d'exécution pour une requête donnée. Ce processus se divise en deux étapes : le dénombrement du plan et l'estimation des coûts, où le plan avec le coût estimé le plus bas est sélectionné pour l'évaluation[1]. Dans le contexte de RDF QDAG, le concept d'étoile de données est introduit, semblable à un tuple dans les systèmes relationnels, et la notion de requête en étoile est proposée, regroupant les modèles triples partageant le même sujet ou objet en avant ou en arrière étoile des données. Un plan d'exécution est alors une fonction de classement appliquée à un ensemble d'étoiles de requête et de filtre, déterminant l'ordre dans lequel les mappages pour chaque étoile de requête seront identifiés et l'ordre dans lequel les unités de filtrage seront évaluées.

3. **Couche moteur:**
   La couche moteur est chargée d'évaluer la requête, mettant l'accent sur l'exécution du plan le plus avantageux fourni par l'optimiseur. L'évaluation d'une Star-Query implique d'identifier les correspondances entre les variables de l'étoile de requête et les nœuds dans le graphique de données, puis de combiner les mappages associés aux triplets individuels pour construire les correspondances pour la requête étoile. L'évaluation complète de la requête combine l'évaluation de la partie BGP et des filtres, où un plan d'exécution est considéré comme acceptable s'il couvre tous les nœuds et prédicats de la requête et si la tête de chaque étoile de requête est déjà instanciée.[20]
   La figure 1 représente la prsentation architecturale de RDF_QDAG

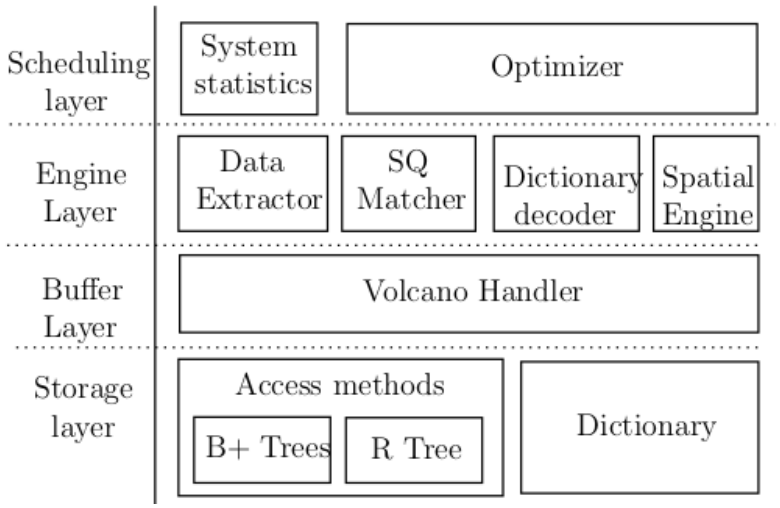



Figure 1: Présentation architecturale de RDF_QDAG

## 2.3 Stratégies d'évaluation des requêtes

Les stratégies d'évaluation des requêtes jouent un rôle essentiel dans le domaine des bases de données, en particulier dans le contexte des données spatiales représentées dans le modèle RDF (Resource Description Framework). Ces stratégies déterminent la manière dont les requêtes sont exécutées et les résultats sont obtenus. Dans cette section, nous explorerons deux stratégies d'évaluation de requêtes pour les requêtes Geo-SPARQL, mises en œuvre dans le cadre de RDF_QDAG (RDF Query Directed Acyclic Graph). Nous examinerons en détail la stratégie BGP-First ainsi que la stratégie Spatial-First, mettant en lumière leurs mécanismes et leur impact sur le traitement des requêtes géospatiales.

### 2.3.1 BGP-First:

La stratégie BGP-First (Basic Graph Pattern First) est une approche utilisée pour évaluer les requêtes Geo-SPARQL dans des bases de données RDF, en se concentrant d'abord sur la correspondance des motifs graphiques avant d'appliquer les filtres spatiaux. Voici comment cette stratégie fonctionne :

1. **Préparation de la requête:** Une requête SPARQL est préparée, utilisant GeoSPARQL pour interroger la base de données RDF. La requête inclut une clause WHERE définissant les conditions que les données doivent remplir pour être incluses dans les résultats.

2. **Exécution de la requête :** Le moteur de requête commence par rechercher les correspondances pour le modèle graphique de la requête. Il identifie les triplets RDF correspondant à chaque motif dans le graphe de requête. Par exemple, il recherche les événements culturels (?o) avec leur géométrie associée (?p) dans la base de données.

3. **Stockage des résultats intermédiaires :** Les résultats de cette recherche graphique sont stockés dans un tampon temporaire, appelé SQ-buffer. Ce

tampon contient les informations nécessaires pour évaluer les étapes suivantes de la requête.

4. **Filtrage spatial :** Une fois les correspondances graphiques trouvées, la requête applique un filtre spatial pour vérifier si les géométries des événements trouvés intersectent le polygone spécifié. Ce filtre utilise des techniques d'intersection spatiale pour comparer les géométries des événements avec la zone définie (par exemple, intersects comme exemple de filtre spatial et il existe d'autres filtres spatiaux : within, disjoint, contains, overlaps, touches, equals).

5. **Raffinement des résultats :** Après le filtrage initial, une étape de raffinement est effectuée pour éliminer les faux positifs. Cette étape implique une vérification plus précise des géométries des événements pour s'assurer que seuls les événements réellement situés dans la zone spécifiée sont inclus dans les résultats finaux.

6. **Récupération des résultats :** Une fois le filtrage et le raffinement terminés, les résultats finaux, c'est-à-dire les événements culturels se déroulant dans la zone géographique donnée, sont extraits du SQ-buffer et renvoyés en tant que résultats de la requête.

La figure 2 représente les étapes de l'execution d'une requête selon la stratégie BGP-First

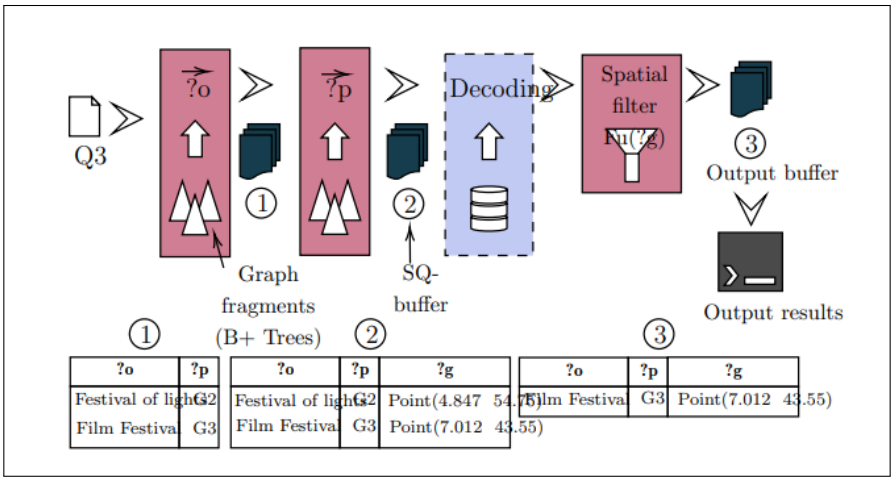



Figure 2: L'exécution d'une requête selon la stratégie BGP-First

En résumé, la stratégie BGP-First permet d'évaluer efficacement les requêtes spatiales dans un environnement RDF en traitant d'abord les correspondances graphiques, puis en appliquant des filtres spatiaux pour affiner les résultats. Cette approche optimise le processus de requête en traitant de manière séquentielle les motifs graphiques et les conditions spatiales.

### 2.3.2 Stratégie Spatial-First

La stratégie Spatial-First (Spatial d'abord) est une approche alternative pour évaluer les requêtes Geo-SPARQL dans des bases de données RDF. Contrairement à la stratégie BGP-First, cette méthode commence par le filtrage spatial avant de traiter les motifs graphiques. Voici une explication détaillée de cette stratégie :

1. **Préparation de la requête:** Une requête SPARQL est préparée, utilisant GeoSPARQL pour interroger la base de données RDF. Cette requête inclut une clause WHERE définissant les conditions que les données doivent remplir pour être incluses dans les résultats.

2. **Début par le filtre spatial:** Contrairement à BGP-First, cette stratégie commence par appliquer le filtre spatial. Par exemple, pour la requête Q1, le plan d'exécution commence par le filtre spatial, noté [Fu(?g), ←-?g, ←-?p, -→?o]. Ce filtre spatial exploite un indice spatial, tel qu'un R-tree, pour accélérer l'évaluation du filtre.

3. **Utilisation de l'indice spatial:** L'indice spatial (R-tree)2.3.2 stocke des approximations des objets géométriques sous forme de MBR (Minimum Bounding Rectangle) 2.3.2 . Lors de l'application du filtre spatial, l'indice permet de minimiser le nombre de pages à examiner, améliorant ainsi l'efficacité de l'opération. Les MBR servent de clés pour naviguer dans l'arbre et identifier les géométries pertinentes.

4. **Exploration des pages de l'indice:** Les pages de l'indice spatial sont structurées pour optimiser le stockage et la recherche. Les pages intérieures contiennent des entrées avec des MBR et des pointeurs vers d'autres pages, tandis que les pages feuilles contiennent des points ou des MBR représentant des géométries complexes. Une fois que les MBR pertinents sont identifiés, seuls ceux qui potentiellement intersectent la zone de requête sont conservés.

5. **Évaluation des motifs graphiques:** Après le filtrage spatial initial, les géométries identifiées sont utilisées pour explorer les motifs graphiques de la requête. Cela implique de rechercher les correspondances pour les triplets RDF restants dans le graphe de requête, comme les événements culturels et leurs géométries associées.

6. **Décodage et raffinement:** Une fois les motifs graphiques évalués, les résultats sont décodés pour remplacer les identifiants d'objets par leurs valeurs réelles. Les approximations MBR des géométries sont également remplacées par les vraies formes géométriques. Ensuite, une étape de raffinement est effectuée pour éliminer les faux positifs, similaire à la stratégie BGP-First.

7. **Récupération des résultats finaux:** Les résultats finaux, après filtrage et raffinement, sont récupérés et renvoyés en tant que résultats de la requête.

La figure 3 représente les étapes de l'execution d'une requête selon la stratégie Spatiale First

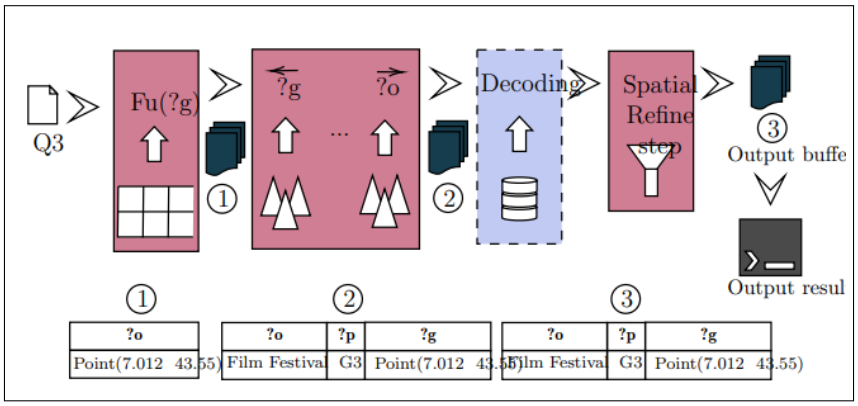



Figure 3: Exécution d'une requête selon la stratégie Spatial-First

En résumé, la stratégie Spatial-First tire parti des indices spatiaux pour accélérer l'évaluation des filtres spatiaux avant de traiter les motifs graphiques. Cette approche permet de réduire le nombre de données à examiner lors des étapes ultérieures, améliorant ainsi l'efficacité globale de l'évaluation des requêtes Geo-SPARQL dans un environnement RDF. [20]

*Definition 2.3.2.1 : (R-Tree)* Un R-tree[6] est une structure de données arborescente utilisée pour indexer des objets géométriques tels que des points, des lignes, des polygones et d'autres formes spatiales. Il est particulièrement efficace pour les requêtes spatiales, comme les recherches d'intersection ou de contiguïté, en réduisant le nombre de comparaisons nécessaires pour trouver les objets qui répondent à une condition géographique donnée.

*Definition 2.3.2.2 : (MBR)* Le MBR[7] (Minimum Bounding Rectangle) est le plus petit rectangle aligné sur les axes qui contient complètement un objet géométrique. Les MBR sont utilisés dans les R-trees pour simplifier et accélérer les tests d'intersection entre formes géométriques complexes.

## 2.4 Représentation des données

### 2.4.1 Représentation des données RDF

Les graphes RDF sont traditionnellement représentés comme des ensembles de triplets (sujet, prédicat, objet)[16], mais cette approche peut entraîner la perte d'informations contextuelles importantes. En traitant les triplets séparément, on néglige les relations entre triplets voisins, ce qui limite les mécanismes d'optimisation.

[6] https://en.wikipedia.org/wiki/R-tree
[7] https://en.wikipedia.org/wiki/Minimum_bounding_rectangle

Pour résoudre ce problème, l'équipe RDF_QDAG a introduit l'idée d'étoiles de données, où les triplets sont regroupés par sujet ou objet. Les étoiles de données peuvent être sortantes (en fonction des arcs sortants du nœud) ou entrantes (en fonction des arcs entrants). Cette structure a permis une meilleure identification du type implicite de nœuds et un regroupement des triplets pertinentes. Les ensembles de fonctionnalités, définis comme des ensembles de prédicats associés à un sujet ou un objet, permettaient de regrouper les étoiles de données partageant les mêmes prédicats en parties de graphique. Cela a facilité la recherche des données en organisant physiquement les triplets en parties indexées sous forme de grille.

Contrairement aux approches basées sur des ontologies explicites, l'équipe RDF_-QDAG a utilisé un type implicite défini par les prédicats associés à l'entité. Cette organisation a amélioré l'évaluation des requêtes en regroupant les données selon leurs caractéristiques contextuelles.[20]
La figure 4 suivante représente la différence entre un graph RDF et une étoile de requête.

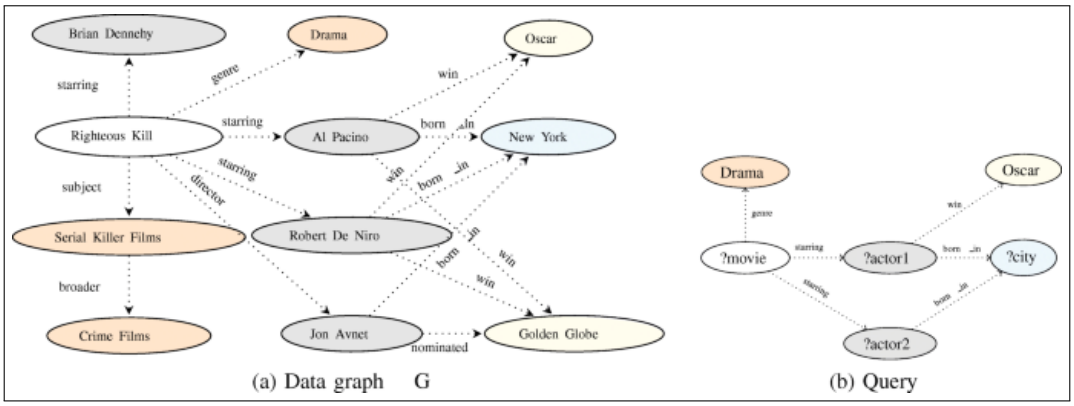



Figure 4: Graph RDF VS Query

### 2.4.2 Représentation des données spatiales avec RDF_QDAG

La façon dont RDF QDAG représente les données spatiales offre des fonctionnalités très avancées pour manipuler les informations géographiques, notamment celles décrivant la localisation, la forme ou les caractéristiques des objets dans l'espace. Pour cela, RDF QDAG utilise des ontologies géospatiales spéciales qui définissent des concepts tels que les coordonnées géographiques ou les frontières des lieux.

Dans RDF QDAG, chaque lieu ou région est représenté en tant que "ressource" avec des détails spécifiques. Par exemple, une ville peut être représentée par un point sur une carte, accompagné d'informations telles que son nom, ses coordonnées GPS et sa catégorie (ville, pays, monument, etc.).[8]

[8] `https://enterpriseintegrationlab.github.io/icity/Spatial/doc/index-en.html`

Ce système nous permet de connecter toutes ces informations spatiales ensemble, comme dans un réseau. Ça veut dire qu'on peut poser des questions complexes sur les relations entre les différents endroits. Et avec RDF_QDAG, poser ces questions et analyser les réponses devient plus simple. Ça ouvre plein de nouvelles possibilités, surtout dans des domaines comme la géographie, l'urbanisme et la navigation.[5]

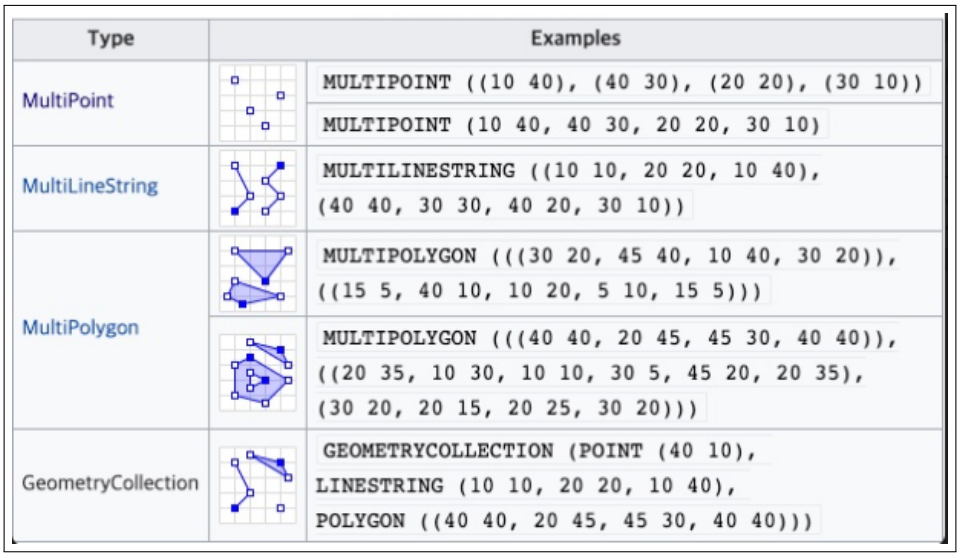
## 2.5 Utilisation du WKT dans la représentation des données spatiales

Le Well-Known Text (WKT) est un format standard pour représenter les géométries géospatiales de manière textuelle et lisible. Il permet de décrire des entités géométriques.

Les figures 5 et 6 illustrent les différents types de données géométriques que RDF QDAG peut gérer, comme les points, les lignes et les polygones, ainsi que des collections de ces types. Les deux figures montrent comment ces différents types de données géométriques sont représentés et stockés. Ces représentations permettent à RDF QDAG de gérer efficacement des requêtes spatiales complexes en utilisant des structures telles que les index R-trees pour les données spatiales.

| Type | Examples | |
|---|---|---|
| Point |  | `POINT (30 10)` |
| LineString |  | `LINESTRING (30 10, 10 30, 40 40)` |
| Polygon |  | `POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))` |
| |  | `POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))` |

Figure 5: Geometry Primitives (2D)



| Type | Examples |
|---|---|
| MultiPoint | MULTIPOINT ((10 40), (40 30), (20 20), (30 10)) |
| | MULTIPOINT (10 40, 40 30, 20 20, 30 10) |
| MultiLineString | MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10)) |
| MultiPolygon | MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5))) |
| | MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20))) |
| GeometryCollection | GEOMETRYCOLLECTION (POINT (40 10), LINESTRING (10 10, 20 20, 10 40), POLYGON ((40 40, 20 45, 45 30, 40 40))) |

Figure 6: Multipart Geometries (2D)

Les coordonnées dans WKT sont exprimées en paires (x, y), où x représente la longitude (coordonnée horizontale) et y représente la latitude (coordonnée verticale). Dans le cas des polygones, la première et la dernière coordonnée doivent être identiques pour indiquer que la forme est fermée.[9]

## 2.6 Fonctionnalités existantes de RDF_QDAG

RDF_QDAG offre une gamme complète de fonctionnalités pour interroger et manipuler les données RDF :

- **Requêtes SPARQL :** Prise en charge de la norme SPARQL avec des fonctionnalités de base spécifiques à RDF QDAG pour l'interrogation des données RDF.L'exécution des requêtes SPARQL dans RDF_QDAG se fait principalement via des commandes dans le terminal, souvent à partir d'un environnement de développement intégré comme Visual Studio Code (VSCode). Par exemple, une commande Docker comme :

  sudo docker run -v "/(*pwd*)/../*data* : /*data*" − *v*"/(pwd)/conf:/app/conf" -p 5005:5005 p_qdag:1.0.2 -db /data/loaded/watdiv100k -q /data/queries/watdiv/dolap/T3.in -sh

  est utilisée pour exécuter une requête SPARQL contenue dans le fichier T3.in. Telle que :

  - **/data/loaded/watdiv100k :** Ç'est le chemin de la base de données watdiv100K[10]
  - **/data/queries/watdiv/dolap/T3.in :** Ç'est le chemin de la requête SPARQL

[9] https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry
[10] https://dgraux.github.io/supervision/Sejdiu_PhD_2020.pdf

- **Filtrage et agrégation et tri :** Fonctions avancées pour filtrer et agréger et trier les résultats des requêtes.

## 2.7 La syntaxe d'une requête SPARQL

Les requêtes SPARQL sont utilisées pour interroger des données stockées sous forme de graphes RDF (Resource Description Framework). Une requête SPARQL simple permet de rechercher et d'extraire des informations en définissant des conditions sémantiques basées sur des triplets (sujet, prédicat, objet). En revanche, les requêtes SPARQL spatiales étendent cette fonctionnalité pour interroger des données géospatiales. Elles intègrent des opérations de filtrage et de comparaison basées sur des relations spatiales, telles que l'intersection, la proximité et l'inclusion. Ces extensions permettent de définir des critères spatiaux dans les requêtes afin de trouver des entités géospatiales qui répondent à des conditions spécifiques, combinant ainsi des critères spatiaux et sémantiques pour une analyse géospatiale plus approfondie.

Voici un exemple d'une requête SPARQL simple (Listing1) et spatiale (Listing2):

```
1 SELECT ?v2 ?v3 ?v4 ?v5 WHERE
2 {
3     ?v0 <http://schema.org/isbn> ?v2 .
4     ?v0 <http://schema.org/editor> ?v3 .
5     ?v1 <http://purl.org/goodrelations/price> ?v4 .
6     ?v1 <http://purl.org/goodrelations/validFrom> ?v5 .
7 }
```

Listing 1: Exemple de requête SPARQL Simple

**Explication de la syntaxe de cette requête :**

- **SELECT ?v0 ?v2 ?v3 :** Cette clause indique que nous souhaitons sélectionner les valeurs des variables ?v0, ?v2, et ?v3 dans les résultats de la requête.
- **WHERE ... :** Cette clause spécifie les motifs de triplets à rechercher dans le graphe RDF.
- **?v0 <http://www.w3.org/1999/02/22-rdf-syntaxnstype> <http://db.uwaterloo.ca/galuc/wsdbm/ProductCategory7>** Ce triplet recherche les ressources ?v0 de type ProductCategory7.
- **?v0 <http://schema.org/description> ?v2 :** Ce triplet indique que la ressource ?v0 a une description liée à la variable ?v2.
- **?v0 <http://schema.org/keywords> ?v3 :** Ce triplet indique que la ressource ?v0 a des mots-clés liés à la variable ?v3.
- **?v0 <http://schema.org/language> <http://db.uwaterloo.ca/ galuc/wsdbm/Language0> :** Ce triplet indique que la ressource ?v0 utilise la langue Language0.

```
SELECT ?g WHERE
{
  ?o <http://linkedgeodata.org/ontology/boat> ?b .
  ?o <http://geovocab.org/geometry#geometry> ?p .
  ?p <http://www.opengis.net/ont/geosparql#asWKT> ?g .
}
FILTER ?g INTERSECTS "POLYGON((-100 20, -80 20, -80 40,
   -100 40, -100 20))";
```

Listing 2: Exemple de requête SPARQL Spatiale

- **SELECT '?g' :** Cela indique que nous souhaitons sélectionner la valeur de la variable ?g dans les résultats de la requête. Cette variable représentera les géométries WKT des résultats.
- **'WHERE ... ' :** Cette clause spécifie les motifs de triplets à rechercher. Dans ce cas, nous avons trois triplets imbriqués.
- **'?o <http://linkedgeodata.org/ontology/boat> ?b' :** Ce triplet recherche des triplets où une ressource '?o' est liée à la propriété '<http://linkedgeodata.org/ontology/boat>' avec une valeur '?b'. Cela permet de filtrer les ressources qui sont liées à la classe ou à l'objet "boat" dans l'ontologie Linkedgeodata.
- **'?o <http://geovocab.org/geometry#geometry> ?p' :** Ce triplet recherche des triplets où la même ressource '?o' est liée à la propriété '<http://geovocab.org/geometry#geometry>' avec une valeur '?p'. Cela permet de filtrer les ressources qui ont une géométrie associée.
- **'?p <http://www.opengis.net/ont/geosparql#asWKT> ?g' :** Ce triplet recherche des triplets où la variable '?p' est liée à la propriété '<http://www.opengis.net/ont/geosparql#asWKT>' avec une valeur '?g'. Cela permet de récupérer la géométrie WKT correspondante à la ressource.
- **'FILTER ?g INTERSECTS "POLYGON((-100 20, -80 20, -80 40, -100 40, -100 20))"' :** Cette clause de filtre permet de spécifier une condition spatiale en utilisant l'opération 'INTERSECTS'. Elle filtre les résultats pour ne sélectionner que les géométries '?g' qui ont une intersection avec le polygone donné (défini en WKT).

## 2.8 Le parsing des requêtes SPARQL

Le parsing des requêtes SPARQL est une étape essentielle dans le traitement des données RDF, particulièrement lorsqu'il s'agit d'intégrer des fonctionnalités avancées comme le produit cartésien successif. Voici une explication détaillée du processus de parsing des requêtes dans RDF_QDAG.

### 2.8.1 Introduction au Parsing des Requêtes :

Le parsing des requêtes SPARQL consiste à analyser et transformer la requête initiale en une structure de données intermédiaire manipulable par le système RDF QDAG. Cette étape décompose la requête en éléments fondamentaux et prépare les données pour les étapes ultérieures de traitement et d'optimisation.

### 2.8.2 Étapes du Parsing des Requêtes :

1. Analyse Lexicale :

   - La première étape du parsing est l'analyse lexicale, où la requête SPARQL est divisée en tokens. Ces tokens représentent les éléments de base de la requête, tels que les mots-clés (SELECT, WHERE), les variables (?v0, ?v2), les URI (<http://schema.org/description>), ainsi que les opérateurs spatiaux et les valeurs géospatiales pour les requêtes spatiales.

2. Analyse Syntaxique :

   - Cette étape vérifie que les morceaux de la requête s'assemblent correctement selon les règles de la grammaire SPARQL. Pour les requêtes spatiales, elle garantit la cohérence et la bonne position des opérations géospatiales.
   - Une fois qu'on a compris la structure de la requête, on construit un arbre syntaxique, qui est un peu comme une carte pour naviguer à travers les différentes parties de la requête. Dans le cas des requêtes spatiales, cet arbre prend en compte les opérations géospatiales pour représenter les relations entre les éléments géographiques.

3. Optimisation :

   - Cette phase vérifie la cohérence des opérations spatiales avec les données géospatiales et peut optimiser la requête en réorganisant les filtres ou les agrégations pour améliorer l'efficacité du traitement.

En résumé, que ce soit pour des requêtes simples ou spatiales, le parsing des requêtes SPARQL dans RDF QDAG est comparable à un jeu de construction où les pièces sont assemblées pour obtenir une compréhension claire de ce que l'utilisateur demande, en tenant compte des spécificités des données géospatiales lorsque nécessaire.

## 2.9 Limitations actuelles

Malgré ses performances et ses nombreuses fonctionnalités, RDF QDAG présente plusieurs limitations :

- **Absence de produit cartésien successif :** Cette fonctionnalité manquante empêche RDF_QDAG de générer toutes les combinaisons possibles entre plusieurs graphes RDF de manière itérative, limitant ainsi sa capacité à effectuer des analyses complexes nécessitant une combinaison exhaustive de différents ensembles de données.

- **Absence de jointure :**RDF_QDAG ne supporte pas nativement les opérations de jointure entre différents graphes RDF, ce qui restreint sa capacité à combiner et interroger efficacement des données provenant de diverses sources RDF.

- **Absence de jointure spatiale :**La gestion des données géospatiales est cruciale pour de nombreuses applications, mais RDF QDAG ne prend pas en charge les opérations de jointure spatiale, limitant ainsi les capacités d'analyse spatiale et l'intégration de données géographiques complexes.

Ces opérations, particulièrement coûteuses en termes de ressources, sont essentielles pour de nombreux cas d'utilisation avancés. Leur intégration permettrait au prototype de recherche RDF QDAG et aux chercheurs travaillant dessus de comparer leurs travaux avec d'autres triplestores et recherches similaires.

## 2.10 Conclusion

En conclusion, l'analyse de RDF QDAG met en lumière ses capacités avancées en matière de gestion et d'interrogation des données RDF, notamment grâce à des stratégies telles que "BGP-First" et l'utilisation des structures comme les R-trees et les MBR pour optimiser les requêtes spatiales. Cependant, malgré ses nombreuses fonctionnalités innovantes, RDF QDAG présente certaines limitations notables, telles que l'absence de produit cartésien successif et de jointure. Ces fonctionnalités manquantes restreignent son efficacité dans les analyses complexes nécessitant une intégration exhaustive des données provenant de multiples sources RDF. L'amélioration de ces aspects pourrait significativement étendre les capacités analytiques et d'application de RDF QDAG, ouvrant ainsi la voie à une exploitation plus complète et efficiente des données spatiales et RDF.

# CHAPITRE 03 :
# Gestion Et Conception

# 3 Gestion Et Conception

## 3.1 Introduction

Dans ce chapitre, nous allons examiner la manière utilisée pour gérer notre projet, notamment la planification des tâches, le modèle de développement, et les différents outils collaboratifs utilisés. Par ailleurs, ce chapitre explore également la conception de l'intégration de la jointure spatiale dans le traitement des requêtes par RDF_QDAG. L'objectif est de détailler les spécifications, le modèle de conception et les étapes de traitement nécessaires pour permettre une exécution efficace des jointures spatiales dans un environnement RDF. En combinant ces deux aspects, nous fournissons une vue d'ensemble de la gestion et de la conception technique de notre projet, assurant ainsi une compréhension complète des processus et des méthodologies employées.

## 3.2 Outils Collaboratif

### 3.2.1 Overleaf

Overleaf est un éditeur LaTeX en ligne et gratuit utilisé pour la rédaction d'articles et de rapports scientifiques. Nous avons choisi d'utiliser la plateforme Overleaf pour les raisons suivantes :
Collaboration en temps réel : Possibilité de rédiger de manière collaborative, permettant à plusieurs personnes de travailler simultanément sur un même document. Flexibilité structurelle : Adaptabilité de la structure du rapport selon les besoins, offrant une personnalisation aisée. Visualisation en PDF : Possibilité de prévisualiser le rapport au format PDF pour une meilleure appréhension visuelle.


### 3.2.2 GitHub

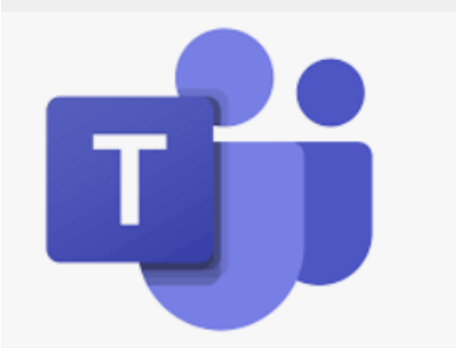
GitHub est une plateforme permettant aux développeurs de travailler sur un même projet de manière collaborative et de stocker leurs projets avec leurs différentes versions. Nous avons utilisé cette plateforme pour faciliter le travail en équipe et assurer la gestion efficace des différentes étapes d'évolution de notre projet.

### 3.2.3 Docker

Docker est une plateforme logicielle permettant de créer, déployer et exécuter des applications de manière isolée dans des conteneurs. Ces conteneurs offrent un environnement léger et portable, garantissant une cohérence d'exécution indépendamment du système d'exploitation sous-jacent. Docker vise à simplifier le déploiement et la gestion des applications, en résolvant les défis et les complexités liés à la mise en place d'un environnement d'exécution cohérent pour les applications, que ce soit en développement, en test ou en production.[11]


### 3.2.4 Teams

Microsoft Teams est une plateforme de communication audio et vidéo lancée par Microsoft en novembre 2016, principalement destinée aux entreprises et aux établissements éducatifs. Cette plateforme offre une interface graphique conviviale ainsi que diverses fonctionnalités telles que le partage d'écran, l'enregistrement des sessions et le transfert/sauvegarde de fichiers. Nous avons utilisé cet outil pour la tenue de nos réunions de projet.



### 3.2.5 Google Meet

Google Meet est un service de visioconférence développé par Google. Il offre la possibilité d'organiser des réunions en ligne avec une qualité vidéo et audio élevée. Google Meet permet également le partage d'écran et la collaboration en temps réel sur des documents. Nous avons utilisé Google Meet pour des réunions de travail à distance.

### 3.2.6 Webex


Webex est une plateforme de visioconférence développée par Cisco. Elle offre des fonctionnalités avancées pour les réunions en ligne, telles que le partage d'écran, la discussion en groupe et l'enregistrement de réunions. Webex permet également d'organiser des événements en ligne de grande envergure. Nous avons utilisé Webex pour des réunions et des présentations en ligne dans le cadre de notre projet.

### 3.2.7 Draw.io

Draw.io est un outil en ligne de création de diagrammes qui permet de concevoir facilement des schémas, des organigrammes, des diagrammes de flux, des plans de réseau, et bien plus encore. Utilisé couramment dans la gestion de projet, il facilite la visualisation et la planification des différentes phases. Son interface intuitive et ses nombreuses fonctionnalités permettent de collaborer en temps réel avec les membres de l'équipe, de personnaliser les diagrammes selon les besoins spécifiques, et d'intégrer ces visuels directement dans les documents de projet. Draw.io prend en charge l'importation et l'exportation dans divers formats, assurant une compatibilité et une flexibilité optimales pour la documentation et la communication des processus.

### 3.2.8 Visual Studio Code

Visual Studio Code (VS Code) est un éditeur de code source développé par Microsoft. Il est gratuit, open source et multiplateforme, ce qui signifie qu'il fonctionne sur Windows, macOS et Linux. VS Code est apprécié pour sa légèreté, sa rapidité et sa flexibilité, tout en offrant des fonctionnalités puissantes qui le rendent adapté à une large gamme de tâches de développement.[12]

## 3.3 Modèle de développement et planning

### 3.3.1 Approche adoptée : Agile (Scrum)

Pour la gestion de l'intégration du produit cartésien successif dans RDF_QDAG, nous avons adopté la méthodologie Agile, et plus spécifiquement le cadre de travail Scrum[17]. Cette approche nous permet de rester flexibles, de répondre rapidement aux changements et de garantir une livraison continue de valeur tout au long du projet.[3]

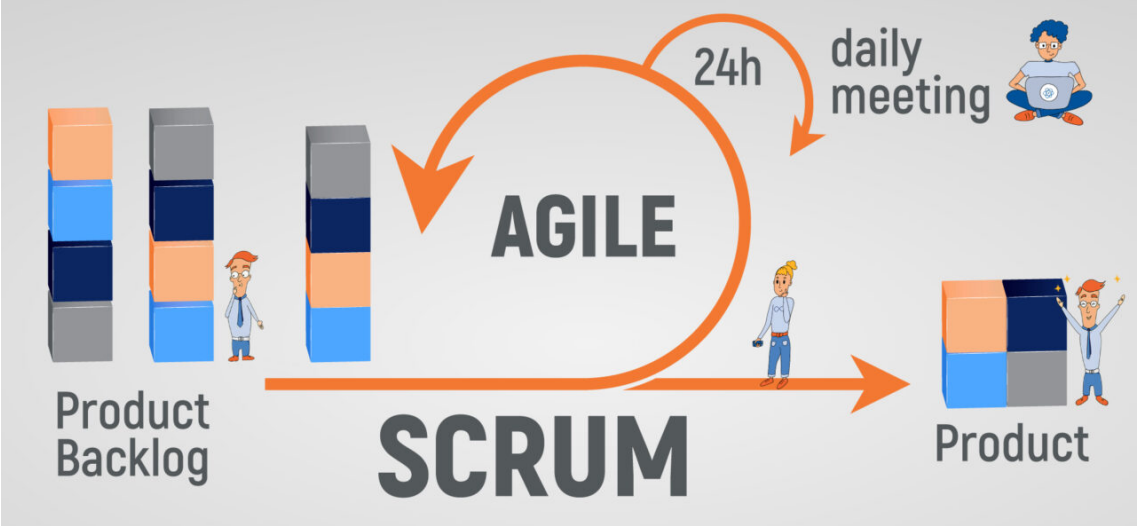



Figure 7: Méthode Agile Scrum

**Principes de la méthodologie Agile :**

- **Flexibilité et adaptabilité:** La capacité de s'adapter aux exigences changeantes et aux retours des parties prenantes.[6]
- **Livraison continue:** Livraison régulière de fonctionnalités fonctionnelles et incrémentielles.
- **Collaboration étroite:** Collaboration constante entre les membres de l'équipe et les parties prenantes. Amélioration continue : Évaluation et amélioration continues des processus et des pratiques de travail.

**Cadre Scrum :**
Le cadre Scrum se compose de plusieurs cérémonies clés pour gérer et organiser le travail de l'équipe : Sprint Planning, Daily Stand-up, Sprint Review, et Sprint Retrospective.[17]

- **Sprint Planning:** Réunion de planification au début de chaque sprint pour déterminer les éléments du backlog de produit à inclure dans le sprint en cours.
- **Daily Stand-up:** Réunion quotidienne de 15 minutes où chaque membre de l'équipe partage ce qu'il a fait, ce qu'il prévoit de faire et les obstacles rencontrés.

- **Sprint Review:** À la fin de chaque sprint, présentation des incréments de produit aux parties prenantes pour obtenir des retours.
- **Sprint Retrospective:** Réunion à la fin de chaque sprint pour discuter des aspects positifs et des domaines à améliorer dans le processus de travail.

### 3.3.2 Planning :

- Sprint 1 : **Compréhension de RDF et SPARQL**
  (1er Février - 15 Février) Pendant cette période, nous avons exploré RDF (Resource Description Framework) et le langage SPARQL en utilisant des exemples sur Virtuoso. Nous avons également appris à utiliser Docker et GitHub pour configurer l'environnement de travail, y compris le débogage du projet de triplestore RDF QDAG.
  À la fin de ce sprint, une compréhension approfondie du RDF et du langage SPARQL est attendue, avec la capacité de charger des bases et d'exécuter des requêtes sur Virtuoso. De plus, l'environnement de développement avec Docker et GitHub doit être configuré et prêt à être utilisé pour le projet de triplestore RDF QDAG, y compris le processus de débogage.
- Sprint 2: **Division des Requêtes**
  (16 Février - 15 Mars) Nous avons travaillé sur la compréhension de la logique et des objectifs du projet. Cela a commencé par la division de la requête (après le parsing) en sous-requêtes. Nous avons conçu et implémenté la classe principale QueryDivider, en ajoutant des tests unitaires pour vérifier son bon fonctionnement.
  À la fin de ce sprint, la classe principale QueryDivider doit être implémentée, permettant la division des requêtes en sous-requêtes après le parsing. Des tests unitaires doivent être en place pour vérifier le bon fonctionnement de cette classe. Une compréhension approfondie de la logique et des objectifs du projet doit être acquise, et une documentation claire doit être disponible.
- Sprint 3: **Conception et Implémentation du Produit Cartésien**
  (16 Mars - 15 Avril) Nous avons élaboré la conception et l'architecture nécessaires pour appliquer le produit cartésien entre les résultats des sous-requêtes. Nous avons ensuite implémenté la classe JoinConsoleWriter et créé les relations nécessaires avec d'autres classes du système, telles que QueryUtils. Nous avons également ajouté des tests unitaires pour assurer la qualité du code.
  À la fin de ce sprint, la conception et l'architecture pour l'application du produit cartésien entre les résultats des sous-requêtes doivent être complètes. La classe JoinConsoleWriter doit être implémentée et intégrée avec QueryUtils. Des tests unitaires doivent être disponibles pour garantir la qualité du code, et la documentation de cette phase doit être complète.

- Sprint 4: **La Jointure**
  (16 Avril - 15 Mai) Durant ce sprint, nous avons analysé et conçu l'intégration des jointures. Nous avons implémenté les méthodes FilterVerification et applyOperator, ainsi que d'autres méthodes dans la classe JoinConsoleWriter. Nous avons également développé une classe Filters pour récupérer les valeurs et l'opérateur de la jointure, en ajoutant des tests unitaires pour garantir leur bon fonctionnement.
  À la fin de ce sprint, les méthodes FilterVerification et applyOperator doivent être implémentées dans la classe JoinConsoleWriter, ainsi que la classe Filters pour récupérer les valeurs et l'opérateur de la jointure. Tous ces éléments doivent être testés unitairement pour assurer leur bon fonctionnement. Une documentation détaillant l'analyse et la conception des jointures doit être disponible.

- Sprint 5: **La Jointure Spatiale**
  (16 Mai - 15 Juin) Après avoir validé le fonctionnement des jointures sur des données simples, nous avons travaillé avec une autre base de données, LockThing-WaySorted, pour les données spatiales. Nous avons analysé et conçu l'intégration des jointures spatiales, en ajoutant les modifications nécessaires, telles que la méthode doGeometries dans chaque classe de filtres spatiaux. Des tests suffisants ont été réalisés pour confirmer l'intégration des jointures spatiales dans le triplestore RDF QDAG.
  À la fin de ce sprint, les méthodes FilterVerification et applyOperator doivent être implémentées dans la classe JoinConsoleWriter, ainsi que la classe Filters pour récupérer les valeurs et l'opérateur de la jointure. Tous ces éléments doivent être testés unitairement pour assurer leur bon fonctionnement. Une documentation détaillant l'analyse et la conception des jointures doit être disponible.

## 3.4 Analyse et conception de l'intégration de la jointure

Dans la conception d'une jointure spatiale pour un projet Java orienté objet (POO), les principes de lisibilité, de maintenabilité et de flexibilité du code revêtent une grande importance. Ce processus implique l'encapsulation des données et des fonctionnalités au sein de classes et d'objets, essentielle pour structurer le code de manière cohérente.

Notre tâche consistait à implémenter une jointure spatiale en utilisant des classes concrètes avec des méthodes publiques, privées et même statiques. Cette méthode alternative, bien que différente, offre plusieurs avantages significatifs.

En définissant des classes concrètes, nous avons pu encapsuler précisément les comportements et les données nécessaires à la réalisation de la jointure spatiale. Les méthodes publiques offrent une interface contrôlée pour interagir avec ces classes, facilitant ainsi l'utilisation par d'autres parties du système. Les méthodes privées, quant à elles, assurent que certaines opérations internes restent cachées et protégées, préservant l'intégrité des données et des traitements.

L'utilisation de méthodes statiques nous a permis de regrouper des fonctionnalités utilitaires ou globales directement liées à la jointure spatiale, sans nécessiter l'instanciation d'objets spécifiques. Cela simplifie l'accès à certaines fonctions communes et améliore l'efficacité globale de notre implémentation.

Grâce à cette approche, nous avons développé une solution flexible et maintenable pour la jointure spatiale dans le triplestore RDF QDAG. Bien que nous n'ayons pas utilisé d'abstractions au sens strict (classes et méthodes abstraites), nous avons respecté les principes de la POO en organisant le code de manière modulaire et en séparant clairement les responsabilités. Cela facilite non seulement le développement initial, mais aussi la maintenance et l'évolution future du code.

En résumé, l'utilisation de classes concrètes et de méthodes publiques, privées et statiques nous a permis de créer une implémentation robuste et claire de la jointure spatiale. Cette approche garantit une bonne encapsulation des données et des comportements, tout en restant fidèle aux principes fondamentaux de la programmation orientée objet. La figure 7 montre le diagramme de classe de l'intégration de la jointure dans RDF_QDAG.

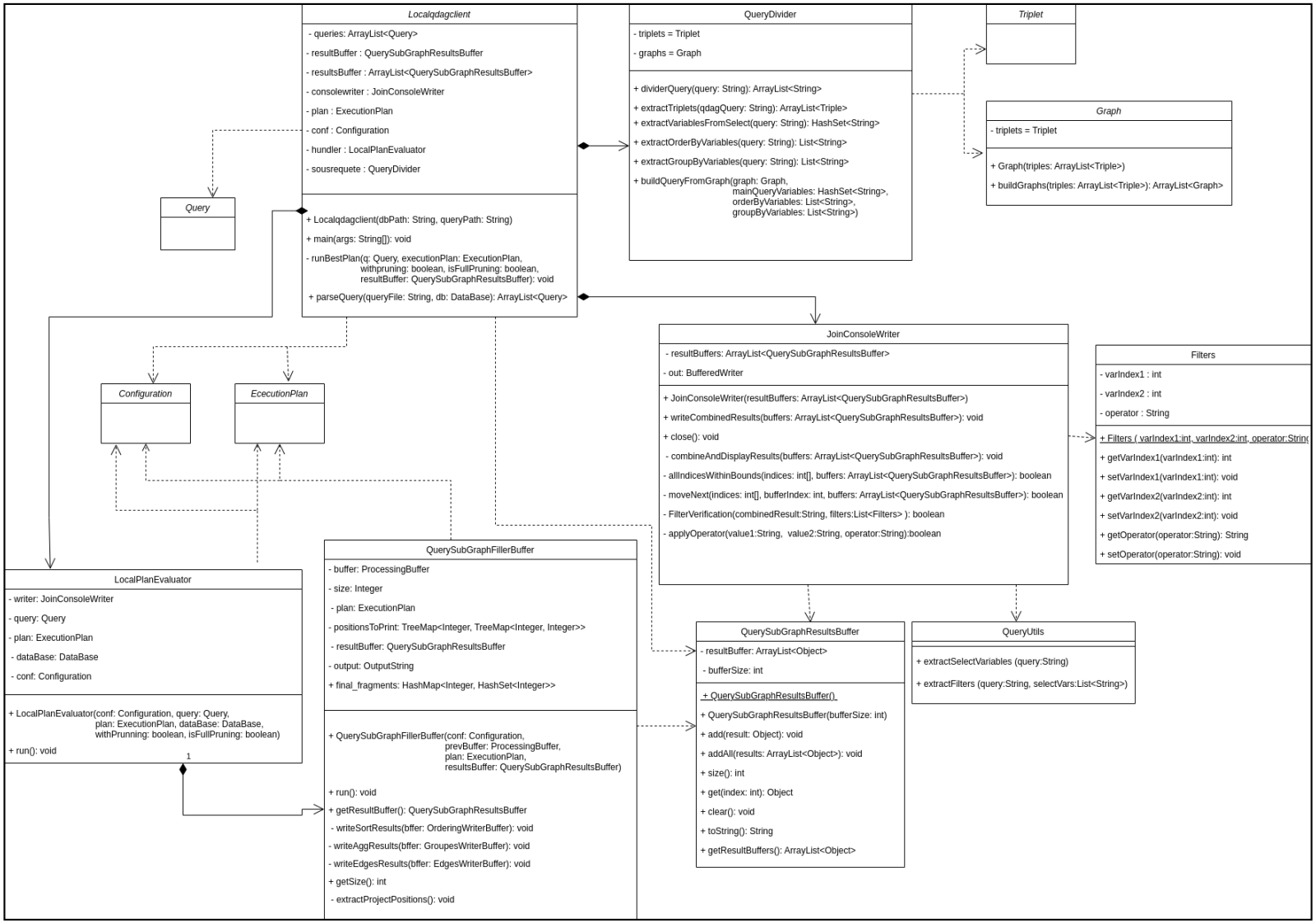



Figure 8: Diagramme de classe de la jointure

Voici une description révisée et clarifiée de chaque classe de ce diagramme de classe :

- La classe Localqdagclient a travers la méthode main() prend en entrée une requête SPARQL principale à partir d'un fichier (.in) ainsi que le chemin vers la base de données chargée.
  Elle effectue le parsing de cette requête SPARQL en utilisant la méthode parseQuery.

- La classe Triplet est essentielle pour modéliser les données RDF et permet diverses opérations sur ces triples.

- La classe Graph crée des graphes de requêtes à partir des triplets formatés précédemment.

- La classe QueryDivider divise la requête principale, après le parsing, en sous-requêtes.
  Chaque sous-requête contient un seul graphe dans la clause WHERE ainsi que les informations supplémentaires présentes dans la requête principale.

- La classe LocalPlanEvaluator crée un plan spécifique pour chaque sous-requête.

- La classe ExecutionPlan exécute la sous-requête selon le plan créé par LocalPlanEvaluator.

- La classe QuerySubGraphFillerBuffer remplit les données de chaque sous-requête dans un ProcessingBuffer.

- La classe QuerySubGraphResultBuffer stocke les résultats de chaque ProcessingBuffer dans un resultBuffer.

- La classe QueryUtils extrait les variables de la clause SELECT et les filtres dans le cas où la requête principale contient une jointure.

- La classe Filters encapsule les détails d'un filtre, y compris les valeurs des variables impliquées et l'opérateur à utiliser pour la jointure.

- La classe JoinConsoleWriter prend les résultats des resultBuffer et applique la combinaison de ces résultats (produit cartésien) ainsi que les filtres dans le cas d'une jointure, puis affiche les résultats sur la console.

## 3.5 Analyse et conception de l'intégration de la jointure spatiale

Dans notre projet, nous avons implémenté des méthodes pour effectuer des opérations de jointure spatiale. En particulier, nous avons ajouté des méthodes pour vérifier si une géométrie contient une autre, si elles se chevauchent, se touchent, etc. Ces méthodes utilisent des filtres géométriques pour évaluer les relations spatiales entre des objets géométriques représentés au format WKT (Well-Known Text).

La figure 8 ci-dessous montre le diagramme de classe de l'intégration de la jointure spatiale dans RDF_QDAG.

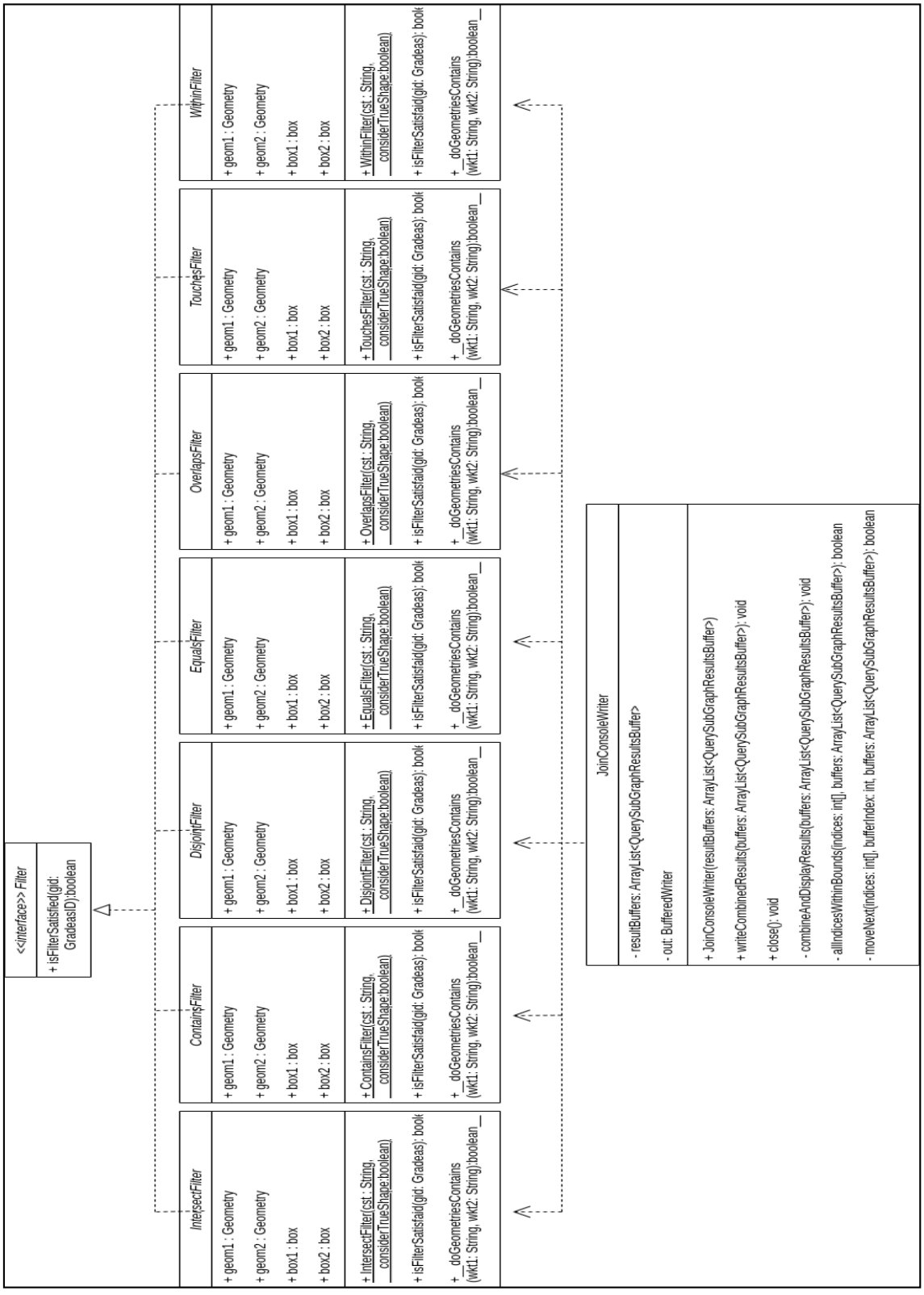



Figure 9: Diagramme de classe de la jointure spatiale

Le diagramme représente un ensemble de filtres utilisés pour des opérations géométriques ainsi qu'une classe de gestion des résultats de ces opérations. Les filtres sont basés sur différents prédicats géométriques (comme l'intersection, la contenance, etc.). La classe JoinConsoleWriter gère la sortie de ces filtres.

- L'interface Filter définit la méthode isFilterSatisfied qui doit être implémenté par toutes les classes souhaitant agir comme des filtres dans le système. Elle ne contient pas de logique d'implémentation directe mais spécifie que toute classe qui implémente Filter doit fournir une méthode isFilterSatisfied qui prend en entré un objet de type GradeasID3.5 retourne un booléen pour indiquer si le filtre est satisfait ou non.

- La classe IntersectFilter implémente l'interface Filter Elle est conçue pour effectuer des filtrages spatiaux basés sur l'intersection géométrique entre deux formes géométriques représentées par des objets Geometry. Voici un résumé de son fonctionnement principal :

  Constructeurs : La classe propose deux constructeurs pour initialiser le filtre en utilisant des représentations géométriques (WKT) et en spécifiant si l'intersection doit être vérifiée avec les formes réelles ou simplement avec les boîtes englobantes.

  Méthodes publiques :

  .. doGeometriesIntersect : Vérifie si deux géométries données s'intersectent en utilisant leur représentation en WKT.
  .. isFilterSatisfied : Implémente la méthode de l'interface Filter. Elle utilise une stratégie de "filtrage et de raffinement" pour déterminer si une géométrie spécifiée par un GradeasID 3.5satisfait le filtre d'intersection avec la géométrie déjà fournie à la construction.

- La classe JoinConsoleWriter est responsable de l'écriture des résultats combinés des sous-requêtes dans la console. Voici un résumé de son fonctionnement :

  Constructeur : Initialise la classe avec des buffers des résultats un pour chaque sous-requête et la requête originale, et configure un BufferedWriter pour écrire dans la sortie standard.
  Méthodes publiques :
  .. writeCombinedResults : Combine et affiche les résultats des buffers de requêtes, en utilisant les filtres spécifiés dans la requête originale.
  Méthodes privées :
  .. combineAndDisplayResults : Gère la combinaison des résultats des buffers en utilisant des indices et vérifie les résultats par rapport aux filtres spécifiés.
  .. FilterVerification : Vérifie si un résultat combiné satisfait tous les filtres spécifiés dans la requête en utilisant des opérateurs tels que >, <, ==, INTERSECTS, etc.

.. applyOperator : contient un switch pour l'affectation du type de filtre défini dans la requête.

Chaque classe remplit un rôle distinct dans le système, contribuant à la gestion des filtres spatiaux et à la sortie des résultats de requêtes combinées.

*Definition 2.3.2.2 : (GradeasID)* GradeasID est une classe qui encapsule des données géospatiales importantes pour la logique de filtrage spatiale implémentée dans les calsses des filtres spatiaux. Elle est utilisée pour récupérer des informations telles que la boîte englobante (Bbox) et les données géographiques réelles (Geometry). Son rôle principal est de fournir une abstraction pour manipuler et vérifier des données géospatiales à l'intérieur de la logique de notre système RDF_QDAG.

## 3.6 Spécifications fonctionnelles et techniques

Les spécifications suivantes sont définies pour l'intégration de la jointure spatiale dans RDF_QDAG :

1. **Exigences pour l'intégration de la jointure spatiale :**
   - **Fonctionnalités nécessaires ::** RDF_QDAG doit permettre le calcul de jointures spatiales entre les résultats combinés de différentes sous-requêtes générées après le parsing et la division de la requête principale.
   - **Compatibilité avec les requêtes SPARQL :** Le système doit être compatible avec les diverses clauses SPARQL telles que ORDER BY, GROUP BY, et FILTER, tout en prenant en charge les opérations spatiales comme INTERSECTS, WITHIN, etc.
   - **Optimisation de l'efficacité :** Le calcul des jointures spatiales doit être optimisé pour minimiser le temps de traitement et l'utilisation des ressources, en utilisant des techniques telles que l'indexation spatiale et la sélection d'algorithmes efficaces.
   - **Extensibilité :** La solution doit être conçue pour permettre l'ajout de nouvelles fonctionnalités sans modification majeure de l'architecture existante, assurant ainsi une flexibilité et une évolutivité continues.
2. **Processus de traitement des requêtes:**
   Le processus de traitement des requêtes par RDF_QDAG inclut plusieurs étapes cruciales pour l'intégration de la jointure spatiale :
   - **Parsing de la requête SPARQL :** Analyse de la requête principale pour identifier les opérations de jointure spatiale et les sous-requêtes associées.
   - **Division de la requête principale en sous-requêtes :** Séparation de la requête initiale en sous-requêtes individuelles basées sur les opérateurs de jointure spatiale.
   - **Calcul du produit cartésien entre les sous-requêtes :** Étape où les résultats des sous-requêtes sont combinés à l'aide de l'opération de produit cartésien.

- **Intégration de la jointure spatiale :** Application des opérations spatiales spécifiées (INTERSECTS, WITHIN, DISJOINT, OVERLAPS, CONTAINS, TOUCHES, EQUALS) pour filtrer les résultats du produit cartésien en fonction des critères spatiaux définis dans la requête.

Le diagramme de processus métier suivant illustre le flux de traitement des requêtes avec l'intégration de la jointure spatiale dans RDF_QDAG : Ce diagramme visuel aide à comprendre comment les composants interagissent pendant le traitement des requêtes, en mettant en évidence les transitions entre les différentes phases du processus.

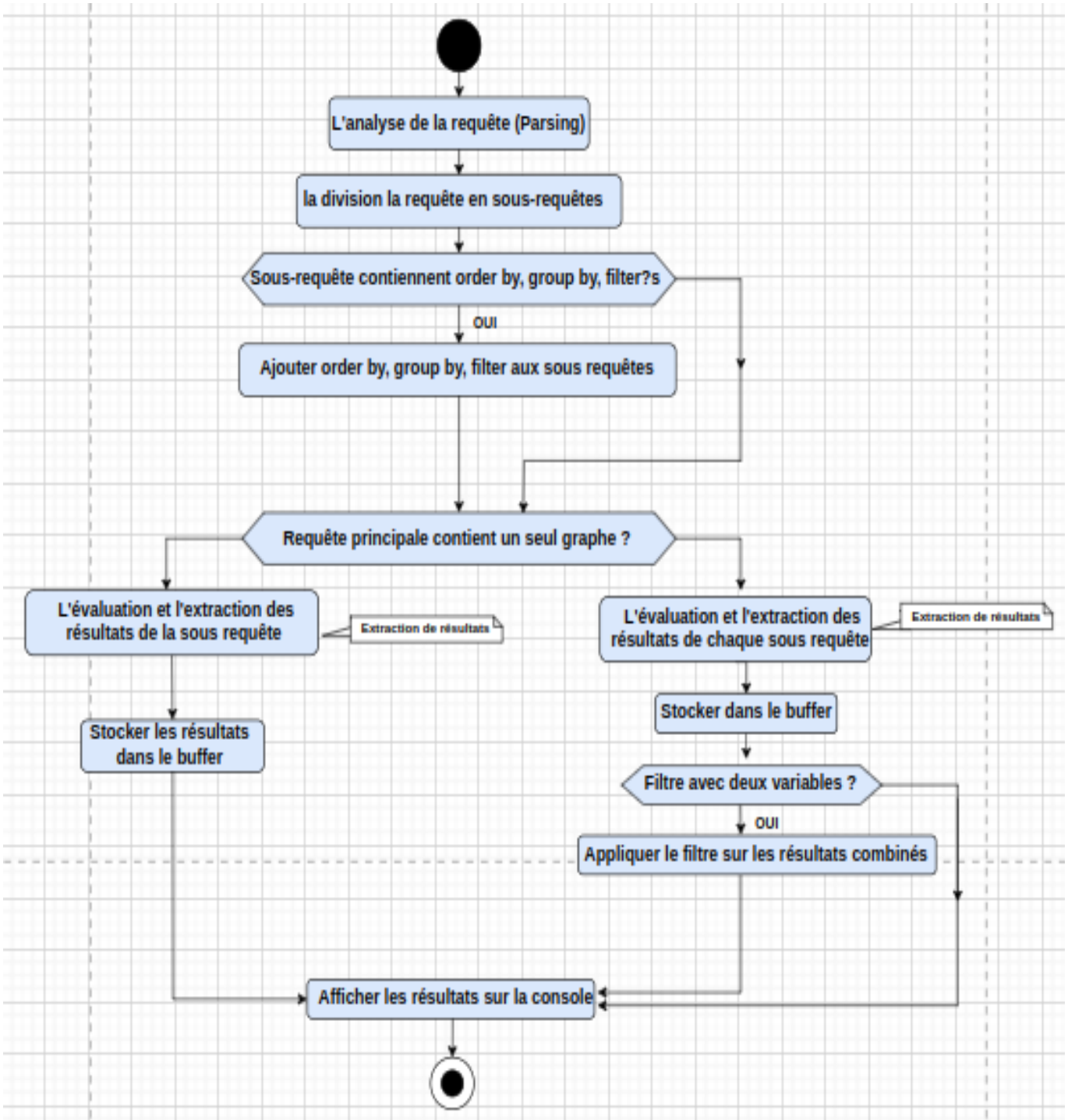



Figure 10: Processus métier d'une jointure

Explication du commentaire **Extraction de résultats :**

.. Si la requête principale contient un seul graphe :
. La sous-requête est évaluée par la classe LocalPlanEvaluator.
. Les résultats sont extraits par la classe ExecutionPlan.
. Ces résultats sont ensuite stockés dans un buffer.

.. Si la requête principale contient plusieurs graphes :
. Chaque sous-requête est évaluée séparément par la classe LocalPlanEvaluator.
. Si un filtre impliquant deux variables est présent, il est appliqué sur les résultats combinés des sous-requêtes.
. Les résultats de chaque sous-requête sont extraits par la classe ExecutionPlan.
. Les résultats extraits sont stockés dans le buffer.

## 3.7 l'Intégration Continue

Dans le cadre de l'intégration continue, les tests doivent être ajoutés pour s'exécuter automatiquement chaque fois qu'un développeur effectue un "push" vers la branche principale (main). Ces tests sont intégrés dans le script d'intégration continue utilisant GitHub Actions comme plateforme d'automatisation.
**Description de Notre Travail :**

(a) **Création de Tests Automatisés :**Dans le cadre de l'intégration continue, les tests doivent être ajoutés pour s'exécuter automatiquement chaque fois qu'un développeur effectue un "push" vers la branche principale (master). Ces tests sont intégrés dans le script d'intégration continue utilisant GitHub Actions comme plateforme d'automatisation.

(b) **Intégration des Tests dans le Script CI :** Les tests sont intégrés dans le script d'intégration continue en configurant un workflow GitHub Actions. Ce workflow est déclenché automatiquement à chaque "push" effectué sur la branche principale. Il compile le code, exécute les tests et vérifie les résultats.

(c) **Automatisation des Tests :** Grâce à l'utilisation de GitHub Actions, chaque modification apportée au code source déclenche automatiquement l'exécution des tests. Si un test échoue, le développeur responsable de la modification est immédiatement notifié, ce qui permet une correction rapide des erreurs.

(d) **Amélioration de la Qualité du Code :** En ajoutant ces tests, la qualité du code est améliorée. Les tests permettent de détecter rapidement les erreurs et de s'assurer que les nouvelles fonctionnalités n'affectent pas le reste du système

(e) **Cycle de Développement Accéléré :** L'intégration des tests automatisés a également contribué à accélérer le cycle de développement. Cela permet aux développeurs de fusionner leurs modifications plus rapidement et en toute confiance, réduisant ainsi les risques de conflits et d'incompatibilités.

## 3.8 Conclusion

Ce chapitre a détaillé la conception et la gestion des filtres géométriques dans le système RDF QDAG. Les filtres, basés sur des prédicats géométriques tels que l'intersection et la contenance etc..., permettent d'effectuer des opérations spatiales précises. L'interface Filter standardise la manière dont ces filtres doivent être implémentés, assurant ainsi une cohérence dans l'ensemble du système.

La classe IntersectFilter, par exemple, montre comment les filtres peuvent être spécialisés pour des tâches spécifiques comme la vérification des intersections géométriques. D'autre part, la classe JoinConsoleWriter illustre comment les résultats des opérations de filtrage peuvent être combinés et présentés de manière cohérente.

En intégrant ces composants, le système est capable de traiter des requêtes complexes, en combinant et en filtrant les résultats de sous-requêtes spatiales. La prise en charge des requêtes SPARQL et l'optimisation de l'efficacité à travers des techniques avancées garantissent une performance robuste et scalable.

Enfin, l'intégration continue avec des tests automatisés assure que les modifications du code sont rapidement vérifiées et validées, améliorant ainsi la qualité et la fiabilité du système. L'approche modulaire et extensible permet d'ajouter de nouvelles fonctionnalités sans perturber l'architecture existante, favorisant une évolutivité continue.

En résumé, ce chapitre a mis en lumière l'importance d'une conception bien pensée et d'une gestion efficace des composants pour une réalisation performante et évolutive.

# CHAPITRE 04 :

# RÉALISATION

# 4 Réalisation

## 4.1 Introduction

Ce chapitre se concentre sur la mise en œuvre concrète de l'intégration des jointures en général et les jointures spatiales, en particulier au sein du TripleStore RDF_-QDAG, y compris les jointures spatiales. Nous allons décrire en détail comment nous avons intégré cette fonctionnalité, en mettant en lumière les aspects techniques et les défis rencontrés, notamment en ce qui concerne la gestion des données géospatiales.

Pour effectuer des jointures, nous devons d'abord réaliser le produit cartésien, qui génère toutes les combinaisons possibles entre les résultats des graphes de requête. Cela constitue la base des opérations de jointure ultérieures. Cependant, cette étape est très coûteuse en termes de calcul, surtout lorsqu'il s'agit de milliards de triplets.

Pour gérer efficacement ces grandes quantités de données, nous utilisons des buffers. Dans notre projet, nous avons fixé une limite de buffer à 1000 éléments.

Pour les jointures spatiales, nous devons prendre en compte des considérations supplémentaires. La jointure spatiale implique l'intégration de données géospatiales en fonction de leurs relations spatiales, telles que intersects, touches, within etc... Cela nécessite une attention particulière aux index spatiaux et aux algorithmes d'optimisation pour gérer efficacement les calculs géométriques complexes.

## 4.2 Configuration et Débogage du Projet RDF QDAG

Pour préparer et tester le projet RDF QDAG, nous avons suivi plusieurs étapes. Voici un aperçu de la procédure :

### 4.2.1 Installation des Outils Requis :

- Nous avons installé Docker[11] pour gérer notre projet et ses dépendances.
- VSCode[12] a été utilisé comme éditeur de code principal pour développer et tester le projet.
- Les extensions Java nécessaires ont été ajoutées à VSCode.

### 4.2.2 L'utilisation de GitHub :

GitHub est une plateforme de développement collaboratif qui utilise Git, un système de contrôle de version distribué. Pour travailler sur notre projet, nous avons cloné le dépôt depuis GitHub sur notre ordinateur à travers une simple commande sur le terminale. Cela nous permet de disposer localement des fichiers et de l'historique du projet. Lorsque nous apportons des modifications, nous les enregistrons localement avec des commits. Pour partager nos modifications avec l'équipe, nous utilisons git push pour envoyer nos commits vers GitHub. De même, pour obtenir les dernières modifications apportées par d'autres membres de l'équipe, nous utilisons git pull pour récupérer les dernières versions du dépôt distant sur notre ordinateur.

[11] `https://www.docker.com`
[12] `https://code.visualstudio.com`

Cette approche facilite la collaboration, la gestion des versions et la synchronisation du travail entre tous les membres de l'équipe

### 4.2.3 Configuration de l'Environnement de Débogage :

- Nous avons vérifié que les extensions Java nécessaires étaient installées dans VSCode.
- Un fichier de configuration a été créé pour le débogage depuis VSCode du projet Java dans Docker.

### 4.2.4 Exécution des Commandes pour le Débogage :

Nous avons exécuté quelques commandes simples pour préparer l'environnement de développement :

- Construction de l'image Docker pour le projet.
- Démarrage d'un conteneur Docker pour les configurations initiales.
- Chargement des données RDF dans le conteneur.
- Démarrage du conteneur Docker avec le débogage activé.

**Remarque**

- RDF_QDAG utilise également Python et C++ pour diverses parties du projet, en particulier pour l'indexation des données. Nous devons être prêts à travailler avec ces langages selon les besoins du projet.

## 4.3 Parsing des Requêtes et Préparation pour le Produit Cartésien

Comme première étape nous avons d'abord travaillé sur le parsing des requêtes SPARQL. Le parsing est une étape cruciale qui permet de transformer la requête principale en une forme exploitable pour des traitements ultérieurs.

Lorsqu'une requête SPARQL est soumise, RDF_QDAG effectue un parsing pour analyser et convertir la syntaxe de la requête en une structure interne. Cette structure interne facilite la manipulation et l'optimisation des différentes composantes de la requête. Après cette étape de parsing, la requête principale est divisée en plusieurs sous-requêtes, chacune contenant un seul graphe RDF dans la clause WHERE.

Pour effectuer cette division, nous avons implémenté la classe QueryDivider. Cette classe prend en entrée la requête principale analysée, et la segmente en sous-requêtes indépendantes. Chaque sous-requête est alors traitée individuellement, ce qui permet de simplifier le processus de calcul et d'améliorer les performances de traitement.

Cette étape de parsing et de division des requêtes est essentielle pour préparer le terrain à l'intégration du produit cartésien, car elle permet de gérer les requêtes complexes en les décomposant en morceaux plus petits et plus gérables. Voici un aperçu des étapes impliquées :

1. **Parsing de la Requête Principale :** Transformation de la requête SPARQL en une structure interne compréhensible par RDF_QDAG.
2. **Division en Sous-Requêtes :** Utilisation de la classe QueryDivider pour segmenter la requête principale en plusieurs sous-requêtes, chacune contenant un seul graphe RDF dans la clause WHERE.
3. **Préparation pour le Produit Cartésien :** Chaque sous-requête est ensuite prête à être combinée avec les autres via le produit cartésien.

**Exemple:**

```
SELECT ?v2 ?v3 ?v4 WHERE
{
    ?v1 <http://schema.org/isbn> ?v2 .
    ?v1 <http://schema.org/editor> ?v3 .
    ?v0 <http://schema.org/isbn> ?v4 .
    ?v0 <http://schema.org/editor> ?v5 .
}
ORDER BY ?v2 ?v4;
```

Listing 3: Requête SPARQL Avant Le Parsing

```
select ?v2,?v3,?v4;?v1 61 ?v2;?v1 83 ?v3;?v0 61 ?v4;?v0 83 ?v5;order by ?v2 ?v4;
```

Listing 4: Requête SPARQL Après Le Parsing

En somme, le parsing et la division des requêtes sont des étapes préalables et nécessaires avant de pouvoir implémenter efficacement le produit cartésien, assurant ainsi une gestion plus fine et performante des requêtes complexes dans RDF_QDAG.

## 4.4 Division des requêtes en sous-requêtes

Dans cette étape de l'implémentation, nous nous concentrons sur la façon dont nous divisons les requêtes principales en sous-requêtes à l'aide de la classe QueryDivider. Après avoir analysé initialement les requêtes, chaque sous-requête est isolée, avec un seul graphe de requête dans sa clause WHERE. L'objectif ici est de préparer les données pour la prochaine étape, où nous exécuterons le produit cartésien entre ces sous-requêtes.

Au moment de la division, dans le cas où nous avons un ORDER BY ou un GROUP BY, chaque sous-requête se voit attribuer un ORDER BY ou un GROUP BY selon les variables concernées. Par exemple, si nous avons deux variables provenant de deux graphes différents, chaque sous-requête prendra un ORDER BY ou un GROUP BY en fonction de sa variable spécifique. De même, pour le filtrage,

si la requête principale contient un filtrage d'un seul graphe, celle-ci sera appliquée uniquement à la sous-requête suffisante.

1. **Méthode dividerQuery(String query) :**

   - Cette méthode prend en entrée une requête SPARQL principale après le parsing et retourne une liste de sous-requêtes.
   - Elle commence par extraire les triplets de la requête principale en appelant la méthode extractTriplets(query).
   - Ensuite, elle construit des graphes à partir de ces triplets en utilisant la méthode Graph.buildGraphs(triplets).
   - Pour chaque graphe construit, la méthode génère une sous-requête à l'aide de la méthode buildQueryFromGraph(graph, ...).
   - Les sous-requêtes sont stockées dans une liste et renvoyées à la fin de la méthode.

```
public static ArrayList<String> dividerQuery(String query) {

    triplets = extractTriplets(query);
    System.out.println("Triplets extraits : " + triplets);

    graphs = Graph.buildGraphs(triplets);
    System.out.println("Graphes construits nnn: " + graphs);

    ArrayList<String> sousRequetes = new ArrayList<>();
    HashSet<String> selectQueryVariables = extractVariablesFromSelect(query);
    List<String> orderByVariables = extractOrderByVariables(query);
    List<String> groupByVariables = extractGroupByVariables(query);
    List<String> filterConditions = extractFilterConditions(query);
    String filterConditionsValue = extractFilterConditionsValue(query);

    for (Graph graph : graphs) {
        String subQuery = buildQueryFromGraph(graph, selectQueryVariables,
    orderByVariables,
        groupByVariables, filterConditions, filterConditionsValue);
        if (subQuery != null) {
           sousRequetes.add(subQuery);
        }
    }

   return sousRequetes;
}
```

Listing 5: Méthode Java dividerQuery(String query)

2. **Méthode extractTriplets(String qdagQuery) :**

   - Cette méthode extrait les triplets d'une requête SPARQL donnée après le parsing.
   - Elle découpe la requête en parties séparées par des points-virgules.
   - Ensuite, elle utilise une expression régulière pour identifier les triplets dans chaque partie de la requête.

- Les triplets extraits sont stockés dans une liste et renvoyés à la fin de la méthode.

```
public static ArrayList<Triple> extractTriplets(String qdagQuery) {

         ArrayList<Triple> triples = new ArrayList<>();
         String[] parts = qdagQuery.split(";");
         String patternString = "(\\?[a-zA-Z0-9]+|<[^>]+>)\\s+(\\?[a-zA-Z0
    -9]+|<[^>]+>|\\d+)\\s+(\\?[a-zA-Z0-9]+|<[^>]+>)";


        Pattern pattern = Pattern.compile(patternString);


        for (String part : parts) {
            Matcher matcher = pattern.matcher(part.trim());
            while (matcher.find()) {
                String subject = matcher.group(1);
                String predicate = matcher.group(2);
                String object = matcher.group(3);
                triples.add(new Triple(subject, predicate, object));
            }
        }
        return triples;
    }
```

Listing 6: Méthode Java extractTriplets(String qdagQuery)

3. **Méthode buildQueryFromGraph() :**

- La méthode buildQueryFromGraph permet de construire dynamiquement une requête à partir d'un graphe et de divers critères de filtrage. Elle prend en paramètre le graphe (Graph), un ensemble de variables principales pour la clause select pour la requête, des listes de variables pour l'ordre de tri (orderByVariables) et le regroupement (groupByVariables), ainsi que plusieurs listes de conditions de filtrage spécifiques.
- Elle commence par initialiser un constructeur de chaîne (queryBuilder) et des ensembles (addedVariables et graphVariables) pour gérer les variables ajoutées et les variables du graphe respectivement. Ensuite, elle parcourt les triples du graphe pour ajouter les variables principales de la clause select au début de la requête.

```
        StringBuilder queryBuilder = new StringBuilder();
        HashSet<String> addedVariables = new HashSet<>();
        HashSet<String> graphVariables = new HashSet<>();


        for (Triple triple : graph.getTriples()) {
            graphVariables.add(triple.getSubject());
            graphVariables.add(triple.getObject());
        }
        queryBuilder.append("select ");
        for (String variable : mainQueryVariables) {
            if (graphVariables.contains(variable)) {
                queryBuilder.append(variable).append(",");
                addedVariables.add(variable);
            }
        }

        if (queryBuilder.toString().equals("select ")) {
            return null;
        }

        queryBuilder.deleteCharAt(queryBuilder.length() - 1);
        queryBuilder.append(";");



        for (Triple triple : graph.getTriples()) {
            queryBuilder.append(triple.toString()).append(";");
        }
```

Listing 7: Construction Dynamique de la Clause SELECT à partir des Triplets de Graphe

- La méthode continue en ajoutant les triples du graphe à la requête, puis vérifie la présence des variables pour l'ordre de tri et le regroupement. Elle construit également des clauses de filtrage basées sur divers types de conditions comme les filtres de correspondance, d'égalité, de comparaison, d'intersection, de relation spatiale, etc.

```
          boolean groupByVariablesPresent = false;
        for (String groupByVariable : groupByVariables) {
            if (graphVariables.contains(groupByVariable)) {
                groupByVariablesPresent = true;
                break;
            }
        }

        if (groupByVariablesPresent) {
            queryBuilder.append("group by ");
            for (String groupByVariable : groupByVariables) {
                if (graphVariables.contains(groupByVariable)) {
                    queryBuilder.append(groupByVariable).append(" ");
                }
            }
            queryBuilder.deleteCharAt(queryBuilder.length() - 1);
            queryBuilder.append(";");
        }
```

Listing 8: Ajout Dynamique de la Clause GROUP BY dans la Requête SPARQL

- Chaque type de filtre est traité séparément, en vérifiant d'abord si la variable concernée est présente dans le graphe avant de construire la clause de filtrage correspondante. Enfin, la méthode retourne la requête construite sous forme de chaîne de caractères.

```
1       for (String intersectFilter : intersectFilters) {
2            String[] parts = intersectFilter.split(" INTERSECTS ");
3            String variable = parts[0];
4            String value = parts[1];
5            if (graphVariables.contains(variable)) {
6                queryBuilder.append("filter ").append(variable).append("
    INTERSECTS ").append(value).append(";");
7            }
8        }
```

Listing 9: Ajout Dynamique de Filtres par rapport aux Variables du Graphe

- Il est important de savoir que pour simplifier la description, les parties de code répétées concernant les différents types de filtres (intersection, within, contains, overlaps, touches, equals, disjoint) ont été omises pour des raisons de concision. Chaque bloc de filtre suit un schéma similaire où les conditions sont analysées, les variables sont extraites et les clauses sont ajoutées à la requête si les variables sont présentes dans le graphe.

- Cette méthode est cruciale pour générer dynamiquement des sous requêtes en fonction des données et des conditions spécifiées, facilitant ainsi le traitement et l'interrogation flexibles de données RDF dans un environnement de projet.

Dans la classe QueryDivider, plusieurs méthodes utilitaires sont implémentées pour faciliter le processus de division des requêtes. Ces méthodes sont conçues pour extraire des informations spécifiques de la requête principale, telles que les variables, les conditions de filtrage, les ordres de tri, etc. Chaque méthode joue un rôle essentiel dans la construction des sous-requêtes à partir de la requête principale, contribuant ainsi à la fonctionnalité globale de la classe.

- extractVariablesFromSelect(String query) : Cette méthode extrait les variables de la clause SELECT de la requête principale. Elle parcourt la requête pour trouver la partie SELECT et divise ensuite cette partie en variables individuelles, qui sont ensuite ajoutées à un ensemble de variables.
- extractFilterConditions(String query) : Cette méthode extrait les conditions de filtrage de type regex de la requête principale. Elle recherche des motifs spécifiques dans la requête pour identifier les conditions de filtrage regex et les stocke dans une liste.
- extractOrderByVariables(String query) : Cette méthode extrait les variables utilisées dans la clause ORDER BY de la requête principale. Elle recherche la partie ORDER BY de la requête et extrait les variables qui y sont utilisées, puis les retourne dans une liste.

- extractGroupByVariables(String query) : Cette méthode extrait les variables utilisées dans la clause GROUP BY de la requête principale. Elle recherche la partie GROUP BY de la requête et extrait les variables qui y sont utilisées, puis les retourne dans une liste.

Les autres méthodes, telles que extractEqualityFilterConditions, extractComparisonFilterConditions, extractMatchFilterConditions, extractINTERSECTFilter etc., suivent une approche similaire en extrayant des informations spécifiques de la requête principale en fonction des motifs correspondants. Chacune de ces méthodes contribue à l'extraction des conditions de filtrage spécifiques qui sont utilisées pour construire les sous-requêtes.

4. **Exemples de requêtes après le parsing et la division:** Dans cette section, nous examinons des exemples de requêtes après leur analyse syntaxique. Nous explorerons cinq cas différents, chaqu'un comportant deux étapes distinctes : avant la division et après la division.

   - **Requête simple :**
     Avant :

```
select ?v2,?v3,?v4,?v5;?v0 61 ?v2;?v0 83 ?v3;?v1 53 ?v4;?v1 15 ?v5;
```

     Après :

```
select ?v2,?v3;?v1 61 ?v2;?v1 83 ?v3;
select ?v4,?v5;?v0 53 ?v4;?v0 15 ?v5;
```

   - **Requête Avec Une Clause ORDER BY :**
     Avant :

```
select ?v2,?v3,?v4,?v5;?v0 61 ?v2;?v0 83 ?v3;?v1 53 ?v4;?v1 15 ?v5;order
     by ?v2 ?v4;
```

     Après :

```
select ?v2,?v3;?v0 61 ?v2;?v0 83 ?v3;order by ?v2;
select ?v4,?v5;?v1 53 ?v4;?v1 15 ?v5;order by ?v4;
```

   - **Requête Avec Une Clause GROUP BY :**
     Avant :

```
select ?v2,?v3,?v4,?v5;?v0 61 ?v2;?v0 83 ?v3;?v1 53 ?v4;?v1 15 ?v5;group
     by ?v2 ?v4;
```

     Après :

```
select ?v2,?v3;?v0 61 ?v2;?v0 83 ?v3;group by ?v2;
select ?v4,?v5;?v1 53 ?v4;?v1 15 ?v5;group by ?v4;
```

- **Requête Avec Une Clause FILTER D'un Seul Graphe :**
  Avant :

```
select ?v2,?v3,?v4,?v5;?v0 61 ?v2;?v0 83 ?v3;?v1 53 ?v4;?v1 15 ?v5;
    filter(regex(str(?v2),"264"));
```

```
select ?v0,?v1,?v2,?v3,;?v0 61 ?v2;?v0 83 ?v3;?v1 42 ?v4;?v1 42 <http:/
    db.uwaterloo.ca/~galuc/wsdbm/Product10>;filter ?v4 matches "2013-10"
    ;
```

```
select ?v0,?v1,?v2,?v3,;?v0 61 ?v2;?v0 83 ?v3;?v1 42 ?v4;?v1 42 <http:/
    db.uwaterloo.ca/~galuc/wsdbm/Product10>;filter (?v4 > 1234);
```

  Après :

```
select ?v2,?v3;?v0 61 ?v2;?v0 83 ?v3;filter(regex(str(?v2),"264");
select ?v4,?v5;?v1 53 ?v4;?v1 15 ?v5;
```

```
select ?v0,?v2,?v3;?v0 61 ?v2;?v0 83 ?v3;
select ?v1;?v1 42 ?v4;?v1 42 <http:/db.uwaterloo.ca/~galuc/wsdbm/
    Product10>;filter ?v4 matches "2013-10";
```

```
select ?v0,?v2,?v3;?v0 61 ?v2;?v0 83 ?v3;
select ?v1;?v1 42 ?v4;?v1 42 <http:/db.uwaterloo.ca/~galuc/wsdbm/
    Product10>;filter (?v4 > 1234);
```

- **Requête Avec Une Clause FILTERS Avec Deux Variables De Deux Graphes Différents :**
  Dans ce cas, le filtre est appliqué après l'obtention des résultats du produit cartésien appliqué au graphes et c'est ça ce qu'on appelle la jointure.

## 4.5 Gestion des Tampons de Résultats et de Remplissage dans RDF QDAG

Dans cette partie, nous allons explorer en détail la gestion des tampons de résultats intermédiaires et des tampons de remplissage dans RDF_QDAG. Nous examinerons deux classes clés : QuerySubGraphResultsBuffer et QuerySubGraphFillerBuffer, en expliquant leurs rôles, attributs et méthodes principales. Ces classes sont essentielles pour la manipulation et le traitement efficace des données lors de l'exécution des requêtes.

1. **QuerySubGraphResultsBuffer :** La classe QuerySubGraphResultsBuffer sert de tampon pour stocker les résultats intermédiaires de sous-graphiques lors de l'exécution d'une requête. Elle gère un tableau dynamique d'objets avec une capacité de tampon configurable. Ses principales fonctionnalités incluent l'ajout de résultats, la vérification si le tampon est plein, et la récupération de la taille ou des éléments du tampon. Cette classe est essentielle pour accumuler les résultats avant de les traiter plus avant.

   - **Attributs Principaux :**
     - resultBuffer : ArrayList pour stocker les résultats.
     - bufferSize : Taille maximale du tampon.

- **Méthodes Clés :**
  - add : Ajoute un seul résultat au tampon.
  - addAll : Ajoute une liste de résultats au tampon.
  - isFull : Vérifie si le tampon a atteint sa capacité maximale.
  - get et size : Accès aux éléments et à la taille du tampon.
  - clear : Vide le tampon.

2. **QuerySubGraphFillerBuffer :** La classe QuerySubGraphFillerBuffer remplit le tampon QuerySubGraphResultsBuffer avec des résultats obtenus à partir de différents types de tampons de traitement (ProcessingBuffer). Cette classe hérite de la classe Operator et s'intègre dans un plan d'exécution pour traiter et transférer des résultats de sous-graphes selon les types de tampons (agrégation, arêtes hyper, et résultats triés).

   - **Attributs Principaux :**
     - buffer : Tampon de traitement contenant les résultats à transférer.
     - plan : Plan d'exécution pour guider le traitement.
     - resultBuffer : Tampon de résultats à remplir.
     - positionsToPrint : Positions des projets pour l'affichage des résultats.
     - output : Sortie formatée des résultats.

   - **Méthodes Clés :**
     - run : Exécute le transfert des résultats selon le type de tampon.
     - writeSortResults, writeAggResults, writeEdgesResults : Méthodes pour écrire les résultats triés, agrégés, et les arêtes hyper.
     - extractProjectPositions : Extrait les positions des projets pour le traitement.

```
1  private void writeAggResults(GroupesWriterBuffer bffer) {
2        for (Pair<CompositeKey, ArrayList> result : bffer) {
3            resultBuffer.add(result.getT() + " " + result.getU());
4        }
5        size += buffer.getSize();
6        buffer.clear();
7    }
```

Listing 10: Méthode Java writeAggResults(GroupesWriterBuffer bffer)

## 4.6 Implémentation du Produit Cartésien

Dans notre projet il est crucial d'avoir des outils efficaces pour traiter les résultats intermédiaires des sous-graphes. Pour cela, nous avons mis en place plusieurs mécanismes qui permettent de combiner et afficher ces résultats de manière optimale.

Nous avons développés une classe principale pour accomplir cette tâche c'est la classe JoinConsoleWriter, qui joue un rôle unique et essentiel dans le traitement des résultats des sous-graphes.

Voyons maintenant en détail comment cette classe fonctionne pour fournir un traitement efficace de la combinaison des résultats des sous-graphes.

1. **JoinConsoleWriter :** La classe JoinConsoleWriter combine et affiche les résultats des tampons de sous-graphes en réalisant un produit cartésien. Elle prend en entrée une liste de tampons de résultats, les combine en générant toutes les combinaisons possibles, et les écrit sur la console. Cette classe permet de visualiser les résultats combinés de manière efficace.

   - **Attributs Principaux :**
     - resultBuffers : Liste des tampons de résultats à combiner.
     - out : Writer pour écrire les résultats combinés sur la console.

   - **Méthodes Clés :**
     - writeCombinedResults : Cette méthode vérifie si la liste des tampons de résultats est vide. Si c'est le cas, elle affiche un message indiquant qu'il n'y a aucun résultat à afficher. Sinon, elle appelle une autre méthode pour combiner et afficher les résultats contenus dans les tampons de sous-graphes.

```
1    private static List<String>  public void writeCombinedResults(
      ArrayList<QuerySubGraphResultsBuffer> buffers) {
2        if (buffers.size() == 0) {
3            try {
4                out.write("Aucun resultat a afficher.\n");
5                out.flush();
6            } catch (IOException e) {
7                e.printStackTrace();
8            }
9            return;
10       }
11       combineAndDisplayResults(buffers);
12   }
```

Listing 11: Méthode Java writeCombinedResults()

     - combineAndDisplayResults : Cette méthode combine les résultats des tampons de sous-graphes en générant toutes les combinaisons possibles et les affiche sur la console. Elle utilise un mécanisme d'itération pour parcourir les résultats de chaque tampon, les concatène dans une chaîne de caractères représentant une combinaison, puis écrit cette combinaison sur la console. Une fois toutes les combinaisons générées et affichées, elle arrête son exécution.

```java
private void combineAndDisplayResults(ArrayList<QuerySubGraphResultsBuffer> buffers) {
    boolean firstIndexMovedFromZero = false;
    int[] indices = new int[buffers.size()];
    int buffersLength = buffers.size();
    int resultNumber = 0;


    while (allIndicesWithinBounds(indices, buffers)) {
        StringBuilder currentCombination = new StringBuilder();

        for (int i = 0; i < buffers.size(); i++) {
            if (indices[i] < buffers.get(i).size()) {
                currentCombination.append(buffers.get(i).get(indices[i])).append(" ");
            }
        }

        String combinedResult = currentCombination.toString().trim();

            resultNumber++;
            try {
                out.write(combinedResult + "\n");
                out.flush();
            } catch (IOException e) {
                e.printStackTrace();
            }
        moveNext(indices, buffersLength - 1, buffers);

        if (indices[0] == 0 && firstIndexMovedFromZero) {
            System.out.println("Nb resultats: " + resultNumber);
            return;
        }

        if (indices[0] != 0 && !firstIndexMovedFromZero)
    firstIndexMovedFromZero = true;
    }
}
```

Listing 12: Méthode Java combineAndDisplayResults

– allIndicesWithinBounds : Cette méthode vérifie si tous les indices des tampons de résultats sont dans les limites valides. Elle parcourt chaque tampon et vérifie si l'indice associé à ce tampon est compris entre 0 et la taille du tampon. Si un indice est en dehors de ces limites, la méthode renvoie faux, sinon elle renvoie vrai.

```java
private boolean allIndicesWithinBounds(int[] indices, ArrayList<QuerySubGraphResultsBuffer> buffers) {
    for (int i = 0; i < buffers.size(); i++) {
        if (indices[i] < 0 || indices[i] >= buffers.get(i).size()) {
            return false;
        }
    }
    return true;
}
```

Listing 13: Méthode Java allIndicesWithinBounds()

– moveNext : Cette méthode déplace l'indice associé à un tampon de résultats vers le résultat suivant. Si l'indice dépasse la taille du tam-

pon, il revient à zéro et la méthode est récursivement appelée pour déplacer l'indice du tampon précédent. La méthode renvoie vrai si un déplacement a été effectué avec succès, sinon elle renvoie faux.

```
1  private boolean moveNext(int[] indices, int bufferIndex, ArrayList
     <QuerySubGraphResultsBuffer> buffers) {
2      if (bufferIndex < 0) return false;
3      indices[bufferIndex]++;
4      if (indices[bufferIndex] >= buffers.get(bufferIndex).size())
       {
5          indices[bufferIndex] = 0;
6          moveNext(indices, bufferIndex - 1, buffers);
7      }
8      return true;
9  }
```

Listing 14: Méthode Java moveNext()

## 4.7 Implémentaion de la jointure

Après avoir étudié en profondeur comment nous avons intégré la prise en compte du produit cartésien au sein du tripeStore RDF_QDAG, nous abordons maintenant une étape cruciale: l'implémentation de la jointure. La jointure est une opération fondamentale dans le traitement des données, permettant de fusionner des enregistrements provenant de différentes sources de données selon certaines conditions. Dans le cadre de notre système, nous étendons les fonctionnalités de la jointure pour prendre en charge les opérations spatiales, facilitant ainsi la manipulation efficace des données spatiales.

1. **Classe Filter :** Cette classe représente un filtre, qui est constitué de deux indices de variables et d'un opérateur. Ces informations seront utilisées pour filtrer les résultats de la jointure en fonction de certaines conditions.

2. **Classe QueryUtils :** La classe QueryUtils est une classe utilitaire destinée à extraire et manipuler les composants de requêtes, tels que les variables de sélection et les filtres. Elle fournit des méthodes statiques pour analyser les requêtes et en extraire des informations structurées, facilitant ainsi le traitement et la vérification des requêtes dans le cadre de l'exécution de jointures et de filtres.

**Méthodes clés:**

- extractSelectVariables(String query) : Cette méthode extrait les variables sélectionnées dans la clause SELECT d'une requête. Ces variables seront utilisées pour associer les filtres aux indices correspondants dans les résultats de la jointure.

```
    private static List<String> extractSelectVariables(String query) {
        List<String> selectVars = new ArrayList<>();
        String selectClause = query.split("select ")[1].split(";")[0];
        String[] variables = selectClause.split(",");
        for (String var : variables) {
            selectVars.add(var.trim());
        }
        System.out.println("selectVars"+selectVars);
        return selectVars;
    }
```

Listing 15: Méthode Java extractSelectVariables()

- extractFilters(String query, List<String> selectVars) : Cette méthode extrait les filtres de la requête. Elle analyse la clause FILTERS pour obtenir les variables et les opérateurs spécifiés dans les filtres, puis les associe aux indices correspondants dans la liste des variables sélectionnées.

```
  private static List<Filter> extractFilters(String query, List<String>
    selectVars) {
        List<Filter> filters = new ArrayList<>();
        if (query.contains("filter ")) {
            String filterClause = query.split("filter ")[1].split(";")
    [0];
            String[] filterParts = filterClause.split(" ");
            String var1 = filterParts[0];
            String operator = filterParts[1];
            String var2 = filterParts[2];

            int varIndex1 = selectVars.indexOf(var1);
            int varIndex2 = selectVars.indexOf(var2);

            if (varIndex1 != -1 && varIndex2 != -1) {
                filters.add(new Filter(varIndex1, varIndex2, operator));
            } else {
                System.err.println("Erreur : Les variables du filtre ne
    sont pas trouvees dans la clause SELECT.");
            }
        }
        return filters;
    }
```

Listing 16: Méthode Java extractFilters()

3. **Méthodes de JoinConsoleWriter :**

- FilterVerification(String combinedResult, List<Filter> filters) : Cette méthode vérifie si un résultat combiné satisfait les conditions spécifiées par les filtres. Elle extrait les valeurs du résultat combiné, puis applique les opérateurs de filtrage sur ces valeurs pour vérifier si elles répondent aux conditions.

```
1     private boolean FilterVerification(String combinedResult, List<
      Filters> filters) {
2         String[] values = extractValues(combinedResult);
3 
4         for (Filters filter : filters) {
5             int varIndex1 = filter.getVarIndex1();
6             int varIndex2 = filter.getVarIndex2();
7             String operator = filter.getOperator();
8 
9             if (varIndex1 >= values.length || varIndex2 >= values.length
      ) {
10                 System.err.println("Indice de variable hors limites:
      varIndex1 = " + varIndex1 + ", varIndex2 = " + varIndex2);
11                 return false;
12             }
13 
14             String valueStr1 = values[varIndex1].trim();
15             String valueStr2 = values[varIndex2].trim();
16 
17             if (valueStr1.isEmpty() || valueStr2.isEmpty()) {
18                 System.err.println("Valeur non numerique detectee:
      valueStr1 = " + valueStr1 + ", valueStr2 = " + valueStr2);
19                 return false;
20             }
21 
22             if (!applyOperator(valueStr1, valueStr2, operator)) {
23                 return false;
24             }
25         }
26         return true;
27     }
```

Listing 17: Méthode Java FilterVerification()

- extractValues(String line) : Cette méthode extrait les valeurs à partir d'une ligne de résultats. Elle sépare les valeurs en utilisant le caractère '—' comme séparateur et filtre les valeurs vides.

```
1   public static String[] extractValues(String line) {
2         return Arrays.stream(line.split("\\|"))
3                      .filter(value -> !value.isEmpty())
4                      .toArray(String[]::new);
5     }
```

Listing 18: Méthode Java extractValues()

- Méthode applyOperator : Cette méthode applique un opérateur de comparaison entre deux valeurs. Elle gère à la fois les comparaisons simples et les comparaisons spatiales.

```
    private boolean applyOperator(String value1, String value2, String operator) {
    switch (operator) {
        case ">":
            return Integer.parseInt(value1) > Integer.parseInt(value2);
        case "<":
            return Integer.parseInt(value1) < Integer.parseInt(value2);
        case ">=":
            return Integer.parseInt(value1) >= Integer.parseInt(value2);
        case "<=":
            return Integer.parseInt(value1) <= Integer.parseInt(value2);
        case "=":
        case "==":
            return value1.equals(value2);
        case "!=":
            return !value1.equals(value2);
        case "INTERSECTS":
            return IntersectFilter.doGeometriesIntersect(value1, value2);
        case "WITHIN":
            return WithinFilter.doGeometriesWithin(value1, value2);
        case "DISJOINT":
            return DisjointFilter.doGeometriesDisjoint(value1, value2);
        case "CONTAINS":
            return ContainsFilter.doGeometriesContains(value1, value2);
        case "OVERLAPS":
            return OverlapsFilter.doGeometriesOverlaps(value1, value2);
        case "TOUCHES":
            return TouchesFilter.doGeometriesTouches(value1, value2);
        case "EQUALS":
            return EqualsFilter.doGeometriesEquals(value1, value2);
        default:
            return false;
    }
}
```

Listing 19: Méthode Java applyOperator()

- Exemple d'Implémentation des Méthodes Spatiales :

```
public static boolean doGeometriesIntersect(String wkt1, String wkt2) {
        GeoFactory factory = DependencyManager.getGeometryFactory();
        try {
            Geometry geom1 = factory.parseWKT(wkt1);
            Geometry geom2 = factory.parseWKT(wkt2);
            return geom1.intersects(geom2);
        } catch (WKTParsingException e) {
            Logger.getLogger(IntersectFilter.class.getName()).log(Level.SEVERE, "WKT Parsing Exception", e);
            return false;
        }
```

Listing 20: Méthode Java doGeometriesIntersect()

**Description de la Méthode doGeometriesIntersect(String wkt1, String wkt2):** La méthode doGeometriesIntersect est une méthode statique utilisée pour déterminer si deux géométries spécifiées en Well-Known Text (WKT) s'intersectent. Voici une description détaillée de son fonctionnement :

(a) Paramètres :
- wkt1 : une chaîne de caractères représentant la première géométrie en format WKT.
- wkt2 : une chaîne de caractères représentant la deuxième géométrie en format WKT.

(b) Retour :
- La méthode retourne un booléen (true ou false). Elle retourne true si les deux géométries s'intersectent, sinon elle retourne false.

(c) Fonctionnement :
- La méthode utilise une instance de GeoFactory obtenue via le DependencyManager pour parser les chaînes WKT en objets Geometry.
- Elle tente de parser les deux chaînes de caractères WKT en objets Geometry.
- Elle utilise la méthode intersects de l'objet Geometry pour vérifier si les deux géométries s'intersectent.
- En cas d'exception lors du parsing des WKT (par exemple, si le format WKT est incorrect), la méthode capture l'exception et enregistre un message d'erreur dans le journal, puis retourne false.

## 4.8 Résultats et Analyses de l'Implémentation des Stratégies d'Évaluation des Requêtes

### 4.8.1 Introduction :

L'implémentation des stratégies d'évaluation des requêtes dans notre système visait à optimiser le traitement des requêtes SPARQL en assurant une gestion efficace des résultats. Cette section présente les résultats obtenus, les analyses effectuées et les conclusions tirées de cette implémentation.

### 4.8.2 Méthodologie :

Pour évaluer l'efficacité des stratégies mises en œuvre, plusieurs étapes ont été suivies :

1. **Définition des scénarios de test :** Des scénarios de requêtes variés ont été définis, incluant des requêtes simples, des requêtes avec plusieurs graphes, et des requêtes contenant des filtres complexes.

2. **Mise en place de l'environnement de test :** L'environnement de test comprenait une base de données qui contient un ensemble de données représentatif et les outils nécessaires pour mesurer les performances.

3. **Exécution des tests :** Les requêtes ont été exécutées dans l'environnement de test, en recueillant des données sur les temps de réponse, l'utilisation des ressources et la précision des résultats.

4. **Analyse des résultats :** Les données recueillies ont été analysées pour évaluer l'efficacité des stratégies d'évaluation.

### 4.8.3 Analyse

1. **Efficacité des Tampons de Résultats :** L'utilisation des tampons de résultats a été déterminante pour améliorer les performances globales. En minimisant les accès redondants aux données et en optimisant le stockage temporaire, le système a pu gérer plus efficacement les requêtes.

2. **Impact de la Division en Sous-Requêtes:** La division des requêtes en sous-requêtes a permis une exécution plus ciblée et une gestion plus fine des ressources. Cette stratégie a non seulement amélioré les temps de réponse, mais a également facilité l'application de filtres spécifiques.

3. **Gestion des Filtres Complexes:** L'application des filtres complexes sur les résultats combinés a démontré la flexibilité du système à gérer des requêtes avancées. Bien que plus exigeante en termes de traitement, cette fonctionnalité a apporté une valeur ajoutée en termes de précision et de pertinence des résultats.

### 4.8.4 Conclusion:

L'implémentation des stratégies d'évaluation des requêtes a montré des résultats prometteurs en termes de performance et d'efficacité. Les améliorations apportées ont permis de réduire significativement les temps de réponse et d'optimiser la gestion des ressources.

La Figure 11 montre les résultats obtenus lors d'une jointure entre deux graphes RDF, exécutée via la commande docker run sur une interface de commande dans Visual Studio Code.

- **Commande Utilisée:** La commande utilise p_qdag:1.0.2 pour exécuter le script data/queries/watdiv/dolap/T2.in avec les données situées dans le répertoire /data/loaded/watdiv100k.
- **Requête SPARQL du Fichier T2.in:** Voici la requête SPARQL du fichier T2.in:

```
    SELECT ?v3 ?v4 ?v5 WHERE
    {
    ?v0 http://schema.org/isbn ?v3 .
    ?v0 http://schema.org/editor ?v4 .
    ?v1 http://schema.org/isbn ?v5 .
    ?v1 http://schema.org/editor ?v6 .
    }
    FILTERS ?v3 > ?v5;

```

Listing 21: Requête SPARQL

- **Analyse de la Requête:** Cette requête contient deux graphes de requête. Le premier :

```
    ?v0 http://schema.org/isbn ?v3 .
    ?v0 http://schema.org/editor ?v4 .

```

Listing 22: graphe de requete1

Le deuxième :

```
    ?v1 http://schema.org/isbn ?v5 .
    ?v1 http://schema.org/editor ?v6 .

```

Listing 23: graphe de requete2

- **Étapes de l'Exécution:** La figure présente plusieurs étapes et résultats de l'exécution.
  Avant QueryDivider: La requête principale après le parsing.
  Après QueryDivider: Les deux sous-requêtes obtenues après la division de la requête principale analysée.
  Résultats affichés: Les résultats sont affichés sous forme d'un produit cartésien.

- **Résultats Obtenus**

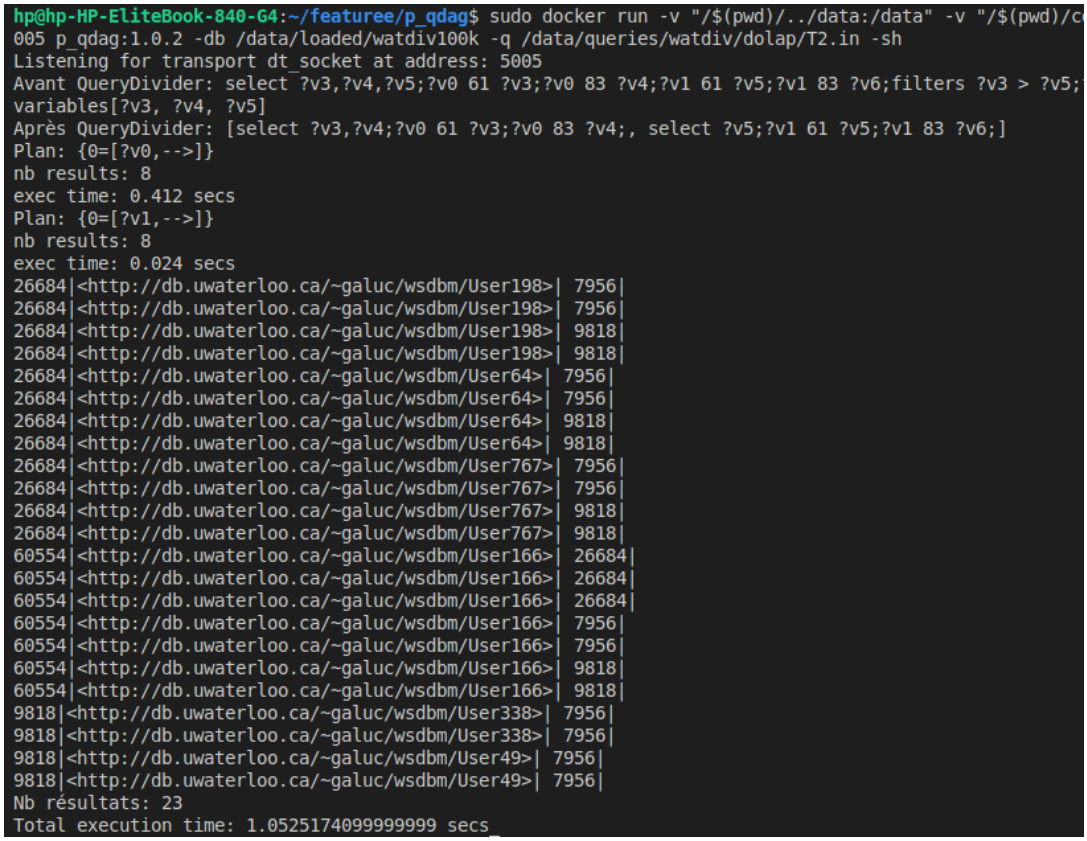



Figure 11: Résultats d'une jointure entre 2 graphes des données simples

Les résultats obtenus sont après l'application de la jointure sur les résultats du produit cartésien entre les deux graphes de requêtes, en sélectionnant uniquement les résultats où ?v3 est supérieur à ?v5.
Les temps de traitement sont notés, avec un temps total d'exécution de 1.0525174999999999 secondes.

Cette figure illustre l'efficacité et les détails de l'exécution d'une jointure complexe entre deux graphes RDF, démontrant à la fois la méthodologie et les résultats pratiques de cette opération.

La figure 12 présente le résultat d'une jointure entre deux graphes pour des données spatiales. Cependant, la figure n'est pas complète en raison de la longueur excessive des résultats. Néanmoins, le scénario d'exécution reste le même.

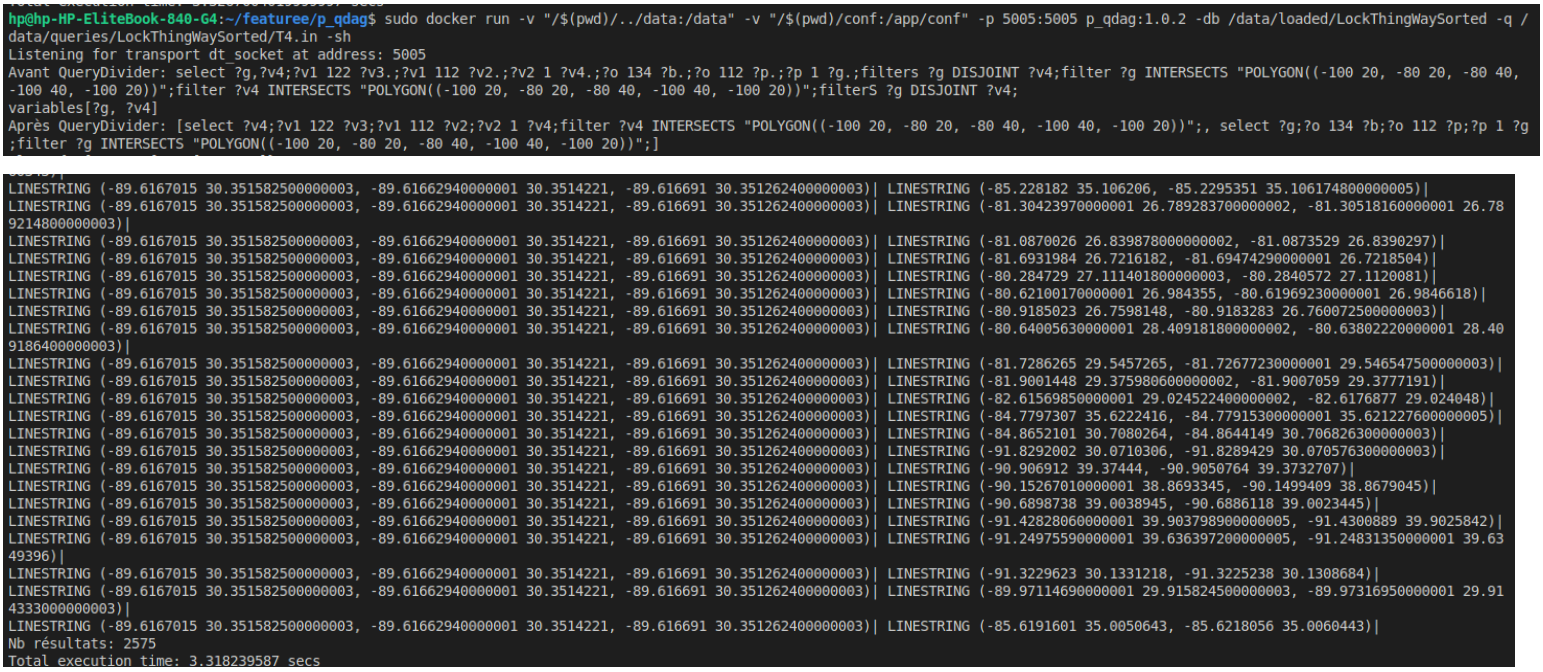



Figure 12: Résultats d'une jointure entre 2 graphes pour des données spatiales

## 4.9 Conclusion:

Ce chapitre a fourni une vue d'ensemble détaillée de l'intégration des jointures dans le TripleStore RDF - QDAG, avec un accent particulier sur les jointures spatiales. Nous avons abordé les différents aspects techniques impliqués dans cette implémentation, y compris la gestion des données géospatiales et les défis associés.

L'utilisation de buffers pour gérer efficacement de grandes quantités de données, ainsi que l'importance du produit cartésien comme base pour les opérations de jointure, ont été mises en avant. Nous avons également décrit les considérations supplémentaires nécessaires pour les jointures spatiales, telles que les index spatiaux et les algorithmes d'optimisation, afin de gérer les calculs géométriques complexes.

La configuration et le débogage du projet RDF QDAG ont été présentés, incluant l'installation des outils requis et l'utilisation de GitHub pour le développement collaboratif. La procédure de parsing et de division des requêtes en sous-requêtes a été expliquée en détail, démontrant comment ces étapes sont essentielles pour préparer les données pour le produit cartésien et les opérations de jointure ultérieures.

En résumé, ce chapitre a permis de mettre en lumière les étapes clés et les défis techniques de l'implémentation des jointures, en particulier les jointures spatiales, dans RDF QDAG, offrant ainsi une base solide pour les chapitres et développements futurs de ce projet.

# CHAPITRE 05 : Conclusion & Perspectives

# 5 Conclusion et Perspectives

## 5.1 Conclusion

Notre projet de recherche s'est concentré sur l'intégration de la jointure spatiale au sein du tripleStore RDF_QDAG. Le développement a commencé par l'analyse approfondie des requêtes SPARQL, intégrant la division des requêtes en sous-requêtes correspondant à chaque graphe dans la clause WHERE. Nous avons conçu une structure logicielle permettant d'incorporer les instructions ORDER BY, GROUP BY et les filtres sur un seul graphe, essentiels à la gestion des requêtes complexes.

Après la division des requêtes, nous avons appliqué le produit cartésien entre les résultats des sous-requêtes obtenus. Ensuite, nous avons intégré les opérations de jointure en général, avec une attention particulière à la jointure spatiale. Cette dernière a été mise en œuvre en utilisant une clause FILTERS pour affiner les résultats combinés, garantissant ainsi une gestion précise et efficace des données spatiales.

L'accent a été mis sur l'évaluation rigoureuse des requêtes, notamment dans les scénarios impliquant plusieurs graphes ou des filtres entre deux graphes. Nous avons mis en œuvre des mécanismes pour extraire, stocker et combiner les résultats de manière optimisée, tout en assurant la gestion adéquate des filtres complexes avant leur intégration dans les résultats finaux. L'objectif principal était de fournir une solution robuste et performante pour l'affichage des résultats sur la console.

En intégrant ces fonctionnalités avancées, notre travail représente une base aux chercheurs qui travaille sur RDF_QDAG afin d'évaluer leurs contributions. Les résultats obtenus ouvrent de nouvelles perspectives pour l'amélioration continue de l'efficacité et de la précision des requêtes dans les environnements de traitement de données complexes. Ce projet constitue une base solide pour de futures recherches visant à optimiser encore davantage la gestion des requêtes et à explorer de nouvelles possibilités d'application dans divers domaines.

## 5.2 Perspectives :

Ce projet offre plusieurs perspectives prometteuses pour l'avenir :

1. Optimisation des Jointures Spatiales : Une étude approfondie sera menée sur l'évaluation des jointures spatiales, en développant des stratégies spécifiques adaptées aux caractéristiques uniques de RDF_QDAG. L'objectif est d'améliorer significativement les performances et l'efficacité des opérations de jointure et jointure spatiale, ouvrant ainsi la voie à de nouveaux standards de traitement de données spatiales dans les graphes de connaissances.

2. Visualisation et Exploration de Données : Développer des outils et des interfaces de visualisation pour permettre aux utilisateurs d'explorer et d'analyser les données spatiales stockées dans RDF_QDAG de manière intuitive et informative.

3. Optimisation Basée sur les Estimations : En intégrant l'approche Spatial First dans les jointures, nous pouvons améliorer l'optimisation des opérations en fonction des estimations. Cela permettra à l'optimiseur de choisir la meilleure stratégie, que ce soit BGP First ou Spatial First, en fonction des caractéristiques spécifiques des données et des requêtes, assurant ainsi la solution la plus efficace pour chaque situation.

4. Comparaison avec d'Autres Systèmes : Comparer RDF_QDAG avec d'autres systèmes sur le traitement des jointures et jointures spatiales en termes de temps d'exécution et de ressources nécessaires. Cela permettra de situer les performances de RDF QDAG dans un contexte plus large et d'identifier des pistes d'amélioration.

En conclusion, ces perspectives représentent des opportunités stratégiques pour enrichir RDF_QDAG, non seulement en termes de fonctionnalités techniques avancées, mais aussi en ouvrant de nouvelles voies pour l'innovation et la recherche dans le domaine des technologies de données et des graphes de connaissances.

# *Table des abréviations*

| Abréviation | Signification |
|---|---|
| QDAG | Querying Data As Graphs |
| RDF | Resource Description Framework |
| SPARQL | Simple Protocol and Rdf Query Language |
| QS | Star Query |
| POO | Programmation Orientée Objet |
| WKT | Well-Known Text |
| BGP-First | Basic Graph Pattern First |
| MBR | Minimum Bounding Rectangle |

# References


[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-store databases. *Communications of the ACM*, 53(4):117–121, 2007.

[2] Anonyme. Interrogation et analyse efficiente des données du web sémantique. *Synthèse de la thèse en français*, 2024. Saturation: La littérature propose deux approches principales pour l'interrogation des données en présence de contraintes sémantiques, ce qui rend explicite les informations qui peuvent être déduites, ou utiliser les contraintes pour remodeler la question. Nous formalisons un cadre commun pour comparer les deux approches, tout en améliorant l'état de l'art pour chacun.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. `http://agilemanifesto.org/`, 2001.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[5] Mohammed El Bachir Bouchenaki. Intégration des filtres spatiaux au sein du triplestore rdf_qdag. Mémoire de fin d'études pour l'obtention du diplôme de master en informatique, option: Génie logiciel (g.l), Université Abou Bakr Belkaid – Tlemcen, Faculté des Sciences, Département d'Informatique, 2023. Présenté le 26/06/2023 devant le jury composé de : Amine Belabed (Président), Houcine Matallah (Encadrant), Nadir Guermoudi (Co-encadrant), Amina Benosman (Examinatrice).

[6] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, 2010.

[7] Fabien Gandon. *Graphes RDF et leur Manipulation pour la Gestion de Connaissances*. Mémoire d'habilitation à diriger les recherches, Université Nice Sophia Antipolis, 2008. soutenue le Mercredi 5 novembre 2008 par Fabien L. Gandon devant un jury composé de Président : Pierre Bernhard, Rapporteur : Nathalie Aussenac-Gilles, Rapporteur : Marie-Laure Mugnier, Rapporteur : Vincent Quint, Examinateur : Bertrand Braunschweig, Examinateur : Amedeo Napoli.

[8] Claudio Gutierrez and Juan F. Sequeda. Knowledge graphs. *Communications of the ACM*, 64(3):96–104, March 2021.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[10] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[11] Sonoo Jaiswal. Docker tutorial. `https://www.javatpoint.com/docker-tutorial`, 2024. Education: MCA, M.Tech; Age: 31; Native Place: Ayodhya, Uttar Pradesh, India; Current Address: Noida, Uttar Pradesh, India; Marital Status: Unmarried; Company: Tpoint Tech; Website: `www.javatpoint.com`; Number of Employees: 300+; Language: Hindi, English; Nationality: Indian; Region: Asia-Pacific (APAC).

[12] Sonoo Jaiswal. Vscode for ubuntu tutorial. `https://www.javatpoint.com/vscode-for-ubuntu`, 2024. Education: MCA, M.Tech; Age: 31; Native Place: Ayodhya, Uttar Pradesh, India; Current Address: Noida, Uttar Pradesh, India; Marital Status: Unmarried; Company: Tpoint Tech; Website: `www.javatpoint.com`; Number of Employees: 300+; Language: Hindi, English; Nationality: Indian; Region: Asia-Pacific (APAC).

[13] Wilhelm Karl Joseph Killing. Contributions to the theories of lie algebras, lie groups, and non-euclidean geometry, 1923. Wilhelm Karl Joseph Killing (10 mai 1847 – 11 février 1923) était un mathématicien allemand connu pour ses nombreuses contributions aux théories des algèbres de Lie et des groupes de Lie et à la géométrie non euclidienne.

[14] José Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):Article 16, 2009.

[15] Yann Rivault, Nolwenn Le Meur, and Olivier Dameron. Intégration et exploration des bases de données médico-administratives grâce aux technologies du web sémantique. Slides sur PowerPoint, disponibles sur SlideShare, 2017. EA 7449 REPERES, EHESP, Rennes. Dyliss, UMR 6074 IRISA, Rennes. Thèse financée par l'ANSM. Consulté le 21 juin 2024.

[16] Philippe Rouilhan. Sujet, prédicat, objet, concept chez frege. *Histoire Épistémologie Langage*, 6(1):91–99, 1984. Logique et grammaire, sous la direction de Suzanne Bachelard.

[17] Ken Schwaber and Jeff Sutherland. The scrum guide. `https://www.scrumguides.org/scrum-guide.html`, 2020. Scrum.org.

[18] Santiago Timón-Reina, Mariano Rincón, and Rafael Martínez-Tomás. An overview of graph databases and their applications in the biomedical domain. *Database*, 2021:baab026, 2021.

[19] Wikimedia Foundation. SPARQL Protocol and RDF Query Language/Requêtes de lecture. Consulté le 22 juin 2024, depuis `https://fr.wikiversity.org/wiki/SPARQL_Protocol_and_RDF_Query_Language/Requtes_de_lecture`, n.d. Accessed: 22 juin 2024.

[20] Houssameddine YOUSFI. *THÈSE LMD : Ingénierie des Systèmes d'Information et de connaissances et Aide à la décision*. Doctorat, Université Abou-Bekr Belkaid Tlemcen, École Nationale Supérieure de Mécanique et d'Aérotechnique, Tlemcen, Algérie, décembre 2023. Soutenue publiquement le 07 décembre 2023 à Tlemcen devant le jury composé de : CHIKH Azeddine, MATALLAH Houcine, HADJALI Allel, DEBBAT Fatima, D'ORAZIO Laurent, HACID Mohand-Saïd, MESMOUDI Amin.

[21] Kun Zhao and Beng Chin Ooi. Spatial query optimization techniques - a survey. *ACM Computing Surveys (CSUR)*, 45(4):Article 48, 2013.

## Résumé

Afin de finaliser notre Master en Génie Logiciel, nous avons intégré l'équipe du laboratoire LIAS à Poitiers pour intégrer la jointure spatiale au sein des tripleStores RDF_QDAG. Nous avons analysé les requêtes SPARQL et développé une structure logicielle pour gérer les requêtes complexes en utilisant les instructions ORDER BY, GROUP BY et les filtres. Nous avons appliqué le produit cartésien aux résultats des sous-requêtes et intégré les opérations de jointure, en mettant l'accent sur la jointure spatiale avec des clauses FILTERS. Ce travail a permis de prendre en compte des requêtes complexes et constitue une base solide pour les recherches futures.



***Mots clés :*** Graphe de connaissance, Langage SPARQL, Triplestores, Jointure, Jointure spatial, Données spatiales.



## Abstract

To complete our Master's degree in Software Engineering, we joined the LIAS laboratory team in Poitiers to integrate spatial join within the RDF_QDAG tripleStores. We analyzed SPARQL queries and developed a software structure to handle complex queries using ORDER BY, GROUP BY, and filters. We applied the Cartesian product to subquery results and integrated join operations, focusing on spatial joins with FILTERS clauses. This work has enabled us to consider complex queries and provides a solid basis for future research.



***Keywords:*** Knowledge Graph, SPARQL Language, Triplestores, Join, Spatial join, Spatial Data.



## ملخص

لإكمال درجة الماجستير في هندسة البرمجيات، انضممنا إلى فريق مختبر LIAS في بواتييه لدمج الوصلة المكانية ضمن المخازن الثلاثية RDF_QDAG. وقمنا بتحليل استعلامات SPARQL وطورنا بنية برمجية للتعامل مع الاستعلامات المعقدة باستخدام ORDER BY و GROUP BY والمرشحات. قمنا بتطبيق المنتج الديكارتي على نتائج الاستعلام الفرعي وعمليات الربط المتكاملة، مع التركيز على الوصلات المكانية باستخدام بنود FILTERS. مكّننا هذا العمل من النظر في الاستعلامات المعقدة ووفر أساسًا متينًا للأبحاث المستقبلية.



***الكلمات المفتاحية***

الرسم البياني المعرفي، ولغة SPARQL، والمخازن الثلاثية، والتحسين، والبيانات المكانية الزمانية.