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

Partitionnement des données RDF au sein de RDF_QDAG : garantir le passage à l'échelle sans perdre la connectivité du graphe

Réalisé par :

- Hamadi Mohammed
- Boussalah Salim

Présenté le 29 Septembre 2022 devant le jury composé de :

- *M. CHOUITI SIDI MOHAMMED* *(Président)*
- *Mme. EL YEBDRI ZEYNEB* (Examinatrice)
- *M. MATALLAH HOUCINE* *(Encadrant)*
- M. MESMOUDI AMIN *(*Co-encadrant*)*

Année universitaire : 2021-2022

Remerciements

Avant tout, nous remercions ALLAH le tout miséricordieux, de nous avoir accordé la force, la patience et le savoir durant tout notre cursus universitaire ; grâce à LUI nous avons surmonté toutes les difficultés et sommes arrivés à ce succès.

Nous tenons à remercier ensuite, M. Hocine MATALLAH pour son soutien et l'aide qu'il a fournie avant et pendant toute la durée de la réalisation et la finalisation de ce mémoire.

Nous remercions ensuite notre encadrant M. Amin MESMOUDI, pour ses conseils, ses orientations et son encadrement afin de mener à bien notre projet. Nous le remercions également pour sa disponibilité et la qualité de ses conseils.

Aussi remercions nous les membres de l'équipe IDD du LIAS, plus particulièrement M.YOUSSEFI Houssemdine et M.Saidi Boumedienne qui nous ont largement aidés tant sur le plan personnel que professionnel.

Nous remercions notamment Mme. EL YEBDRI ZEYNEB d'avoir accepté d'examiner notre travail et M. CHOUITI SIDI MOHAMMED d'avoir accepté de présider ce jury.

Nous remercions tous les enseignants de notre département d'informatique pour leurs efforts fournis et leurs savoirs transmis.

Un grand merci à nos parents pour leurs amours, leurs conseils ainsi que leurs soutiens inconditionnels, à la fois morals et économiques, qui nous ont permis de réaliser les études que nous voulions et par conséquent ce mémoire.

Et enfin nous adressons nos remerciements et reconnaissances envers nos amis et collègues qui nous ont apporté leur soutien moral et intellectuel tout au long de notre démarche.

Dédicaces

A mes très chers parents, qu'aucune dédicace et aucun mot ne pourrait exprimer à leur juste valeur, la gratitude et l'amour que je vous porte.

A mes chers sœur et frère , Sarra , Abd Rahman

A mes encadrants M. Matallah Houcine et M. Mesmoudi Amine, à la fois pour l'aide et le soutien qu'ils ont fournis pour la réalisation et la finalisation du PFE, c'était vraiment un plaisir de travailler avec eux.

A tous les autres enseignants qui enseignaient soigneusement, et avec plaisir, et que nous avons senti leurs efforts, d'ailleurs ces efforts nous ont motivé a donné beaucoup plus pendant le parcours universitaire et en dehors de celui-là, et qui nous ont transmis une formation que je considère parmi les meilleures au niveau international (Mr. TADLAOUI Mohammed, Mme.Seladji Yasmine ,Mr. CHIKH Azzedine ,Mme HALFAOUI Amel ...)

A tous mes amis qui sont toujours en contact avec moi, et ceux qui ne le sont pas à cause du destin qui nous sépare, mais qu'ils sont toujours gravés dans mon cœur et mes souvenirs, avec les quels j'ai passé des moments de joie et de bonheur, ce qui m'a bien aidé pour avancer dans mes études.

A mes chers amies Guermoudi Nadir et mon binôme Bousalah Salim pour les moments inoubliable que j'ai eu avec eux

A toute ma famille.

Hamadi Mohammed

Dédicaces

A ma famille et tous mes amis.

Résumé

Dans le but de valider notre Master en Génie Logiciel, nous avons rejoint l'équipe du laboratoire LIAS à Poiters pour leur proposer une solution de fragmentation pour leur système de gestion RDF. Nous avons utilisé pour cela une approche basée sur le framework Spark et nous avons réussi à implémenter un composant de partitionnement capable de garantir le passage à l'échelle tout en conservant la connectivité du graphe RDF.

Mots-clés : Big data, Système de gestion de données, Transfert de données, Passage à l'échelle , Fragmentation,Partitionnement..

Table des matières

Table des figures

Liste des tableaux

Table des abréviation

Abréviation	Signification
HTML	Hypertext Markup Language
API	Application Programming Interface
XPATH	XML Path Language
CSS	Cascading Style Sheets
NLP	Natural Language Processing
ESPN	Entertainment Sport Programming Network Incorporated
XML	Extensible markup language
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
XHR	XML Http Request
HTTP	Hypertext Transfer Protocol
RDF	Resource Description Framework
URI	Uniform Resource Identifier
QDAG	Querying Data As Graphs
SPARQL	Simple Protocol and Rdf Query Language
RISC	Reduced Instruction Set Computing
NT	N-Triples
ETL	Extraction-Transformation-Loading
ID	Identifiant
IOT	Internet of things
SQL	Structured Query Language
IDE	Integrated Development Environment
PDF	Portable Document Format
CSV	Comma Separated Values
IDD	Ingénierie des Données et des modèles

Chapitre 1

Introduction

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 améliorer l'un des composants principaux de ce Triplestore.

1.1 Contexte et problématique, objectifs

L'interconnexion massive des données, des informations et des connaissances a permis de créer ce qu'on appelle aujourd'hui les graphes de connaissances [15]. Rapidement, ce concept a été étendu à d'autres domaines (e.g., média [1], industrie automobile [42] et pharmaceutique [2], biologie [27]). Les graphes de connaissances permettent aujourd'hui d'apporter une certaine sémantique pour donner un contexte et des relations aux données, tout en fournissant un cadre standard pour l'intégration, l'unification, l'analyse et le partage des données [3]. Dans ce contexte, le Resource Description Framework (RDF) 1 et SPARQL 2 se sont distingués respectivement pour représenter les données et les interroger.

La nécessité de gérer et interroger efficacement les données RDF a conduit au développement de nouveaux systèmes, appelés "Triplestores", qui sont conçus spécialement pour traiter ce format de données. Malgré le nombre important de Triplestores proposés

1. https ://www.slideshare.net/ConnectedDataLondon/ten-years-of-linked-data-at-the-bbc
2. https ://www.ontotext.com/knowledgehub/case-studies/pharma-company-uses-ontotexts-smarter-search-across-siloed-data/
3. https ://hbr.org/sponsored/2019/04/how-third-party-information-can-enhance-data-analytics

dans la littérature, il est difficile de trouver une solution qui offre des temps de réponse acceptables pour des requêtes SPARQL lorsqu'il s'agit de gérer plusieurs milliards de triplets RDF [22]. En fait, ces systèmes s'appuient sur des techniques de stockage, d'évaluation et d'optimisation de requêtes qui ne sont pas en adéquation avec la nature des données RDF à cause de l'absence d'un schéma explicite de données. En outre, les Triplestores existants peinent aussi à offrir des garanties sur les performances, même quand il s'agit de traiter des requêtes avec des opérateurs standards (e.g., BGP, WildCard et agrégation). Le seul système permettant de garantir un certain compromis entre passage à l'échelle et performances est, à notre connaissance, l'approche centralisée RDF_QDAG [22] (https ://qdag.projets.univ-poitiers.fr/qdag/), basée sur l'exploration de graphe et la fragmentation.

La création des fragments est un processus très important pour RDF_QDAG, il permet en effet de garantir la prise en charge des jeux de données regroupant plusieurs milliards de triples RDF. Malheureusement, la version actuelle pose des problèmes de performances et ne permet pas de profiter de la totalité des ressources matérielles disponibles. Il est devenu urgent d'avoir une nouvelle stratégie de fragmentation qui prend en compte les architectures matérielles modernes de type "Cluster computing", surtout avec l'arrivée d'une version parallèle de RDF_QDAG qui intègre un modèle de calcul parallèle et un mécanisme de transfert de résultats intermédiaires.

Dans ce projet, nous avons étudié comment faire évoluer l'outil de fragmentation de RDF_QDAG afin de garantir le passage à l'échelle en termes de plateforme de déploiement mais aussi en termes de volume de données.

1.2 Intégration de l'équipe RDF_QDAG

L'équipe du projet RDF_QDAG regroupe plusieurs chercheurs issus d'universités Françaises, algériennes et tunisiennes (`https://qdag.projets.univ-poitiers.fr/qdag/contact`). D'ailleurs, trois thèses de doctorat ont été soutenues dans le cadre de ce projet tandis que trois autres thèses sont en cours. Une vingtaine d'étudiants en licence et en Master ont travaillé sur ce projet afin de valider leurs diplômes.

Pour notre projet de fin d'étude, nous avons travaillé étroitement avec deux doctorants : Houssameddine Yousfi qui travaille sur la prise en compte des opérateurs spatiaux

lors de l'évaluation des requêtes SPARQL et Boumedien Saidi qui travaille sur une version parallèle de RDF_QDAG. Notre proposition va les aider pour valider leurs travaux de recherche. Nous avons par ailleurs essayé d'améliorer l'approche proposée dans le cadre de la thèse de Jorge Galicia [14] en terme de passage à l'échelle.

1.3 Organisation du manuscrit

La suite du manuscrit est organisée comme suit : nous commençons par présenter les concepts de base liés à la représentation et l'interrogation de données RDF dans le chapitre 2. Nous y donnons aussi un aperçu sur les techniques de gestion de données RDF existantes. Le chapitre 3 présente un aperçu du système RDF_QDAG. Nous présentons en détail la partie fragmentation, qui pose actuellement des problèmes de performances. Nous présentons la conception de notre solution dans le chapitre 4. Le chapitre 5 est consacré aux choix techniques que nous avons considérés pour implémenter notre solution. Nous y présentons également nos résultats préliminaires. Nous présentons notre démarche liée à la gestion du projet dans le chapitre 6. Et enfin, nous finissons ce manuscrit par une conclusion et quelques perspectives sur ce travail.

Chapitre 2

Synthèse bibliographique

2.1 Introduction

Dans ce chapitre, nous donnons quelques détails sur la représentation des données avec RDF. Nous présentons également le langage SPARQL qui est dédié à l'interrogation de ce type de données. Une synthèse sur les outils de gestion de données RDF est faite en fin de chapitre.

2.2 Représentation des données avec RDF

Le framework RDF (Ressource Description Framework) permet de fournir un modèle pour représenter les données et les métadonnées du Web. Il est considéré aujourd'hui comme une norme W3C et émerge comme un modèle de données pour le Web de données et le Web sémantique. Cela est dû au fait qu'il fournit une organisation logique définie en termes de certaines structures de données, ce qui facilite la représentation, l'accès et la spécification de contraintes et de relations entre objets d'intérêt dans un domaine d'application donné.

Le Resource Description Framework (RDF) est un modèle standard pour l'échange de données sur le Web. Les ressources décrites par RDF peuvent être n'importe quoi, y compris des documents, des personnes, des objets physiques et des concepts abstraits.

RDF décrit les données en identifiant les ressources, en leur attribuant des propriétés et en établissant des relations entre elles. Une ressource est définie à l'aide d'identificateurs - appelés identificateurs de ressources internationalisés (IRIs) - ou littéraux. Une relation

entre deux ressources est connue sous le nom de triplet (<sujet> < prédicat> <objet>), faisant une relation dirigée du < sujet > vers < objet > en utilisant le < prédicat>. Cette structure de liaison forme un multigraphe dirigé et étiqueté, où les arêtes représentent le lien nommé entre deux ressources, représentées par les nœuds de graphe.

L'unité de base d'information dans RDF est donnée comme un triplet (s, p, o), composé d'un sujet (s), d'un prédicat (p), et un objet (o).

— **Le sujet :** La ressource sur laquelle une affirmation est faite. Seuls les URI et les nœuds vides sont autorisés à être utilisés comme sujet d'un triplet.

— **Le prédicat :** Un attribut d'une ressource ou d'une relation binaire qui relie cette ressource à une autre. Seuls les URI sont valides pour être utilisés comme prédicat d'un triplet.

— **L'objet :** Généralement, la valeur de l'attribut ou une autre ressource. Les objets valides sont des URI et des nœuds vides, mais aussi des chaînes de caractères. Ces chaînes, sont également appelées littéraux.

Chaque triplet représente un fait sur une chose qu'on veut décrire (c-à-d., le sujet, qui est également appelé la ressource), sur une propriété spécifique (c.-à-d., le prédicat), et avec une valeur donnée (c.-à-d., l'objet). Voici un exemple RDF :

```
L'équipe d'Algérie a gagné la CAN
```

Pour cet exemple : "L'équipe d'Algérie" représente le sujet, "a gagné" représente le prédicat tant dis que "la CAN" représente l'objet.

Graphes RDF

Un ensemble de triplets est donc considéré par la norme RDF comme un multigraphe orienté étiqueté dénoté par $G = \langle V_c, L_V, E, L_E \rangle$ où V_c est un ensemble de sommets correspondant à tous les sujets et objets, L_V est un ensemble d'étiquettes lié aux sommets, E est un ensemble d'arêtes orientées qui relient les sujets et objets correspondants, et L_E est un ensemble d'étiquettes d'arêtes. Étant donné une arête $e \in E$, son étiquette d'arête est sa propriété. La figure 2.1 montre une représentation à base de graphes d'un ensemble de triplets RDF liées à **expliquer l'exemple**.

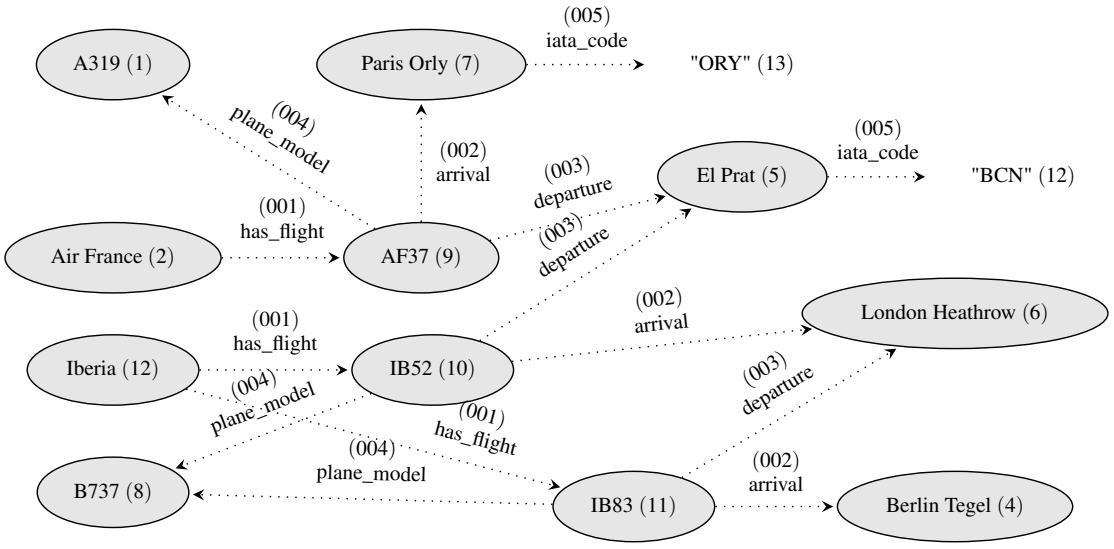


FIGURE 2.1 – Exemple de graphe RDF *G*

2.2.1 Les syntaxes de RDF

Pour sérialiser les données RDF dans des fichiers à des fins de stockage, de partage, d'analyse ou de traitement, des syntaxes analysables par une machine sont nécessaires. Plusieurs syntaxes standards ont été proposées au fil des ans, chacune avec ses propres avantages et inconvénients qui les rendent adaptées à différents contextes.

RDF/XML

Le premier format de sérialisation RDF (recommandée par le W3C en 1999 [**?**]) est RDF/XML [**?**], qui est basé sur la norme XML. Afin de sérialiser les données RDF (graphe) en tant que fichier XML, les nœuds et les arcs doivent être représentés par des noms d'éléments, des noms d'attributs, des contenus d'éléments et des contenus d'attributs XML, comme montré dans la Figure 2.2.

N-triple

La syntaxe N-Triples est normalisée pour la première fois en 2004 [**?**] et mise à jour en 2014 [**?**]. Elle a été motivée par la nécessité de construire des parseurs (lecteurs) RDF simples. En effet, la norme N-Triples est une syntaxe RDF basée sur les lignes, c'est-à-dire que chaque triplet est entièrement écrit sur une ligne. Par conséquent, chaque élément RDF d'un triplet - sujet, prédicat et objet - doit être écrit sans aucune sorte d'abréviation

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:local="http://www.isae-ensma.fr/vocabulaire#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#Paris_Orly">
    <local:iata_code>ORY</local:iata_code>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#AF37">
    <local:arrival rdf:resource="http://www.isae-ensma.fr/air_traffic#Paris_Orly"/>
    <local:departure rdf:resource="http://www.isae-ensma.fr/air_traffic#El_Prat"/>
    <local:plane_model rdf:resource="http://www.isae-ensma.fr/air_traffic#A319"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#IB83">
    <local:arrival rdf:resource="http://www.isae-ensma.fr/air_traffic#Berlin_Tegel"/>
    <local:plane_model rdf:resource="http://www.isae-ensma.fr/air_traffic#B737"/>
    <local:departure rdf:resource="http://www.isae-ensma.fr/air_traffic#London_Heathrow"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#El_Prat">
    <local:iata_code>BCN</local:iata_code>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#Iberia">
    <local:has_flight rdf:resource="http://www.isae-ensma.fr/air_traffic#IB52"/>
    <local:has_flight rdf:resource="http://www.isae-ensma.fr/air_traffic#IB83"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#IB52">
    <local:plane_model rdf:resource="http://www.isae-ensma.fr/air_traffic#B737"/>
    <local:departure rdf:resource="http://www.isae-ensma.fr/air_traffic#El_Prat"/>
    <local:arrival rdf:resource="http://www.isae-ensma.fr/air_traffic#London_Heathrow"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.isae-ensma.fr/air_traffic#AirFrance">
    <local:has_flight rdf:resource="http://www.isae-ensma.fr/air_traffic#AF37"/>
  </rdf:Description>
</rdf:RDF>
```

FIGURE 2.2 – Syntaxe concrète RDF (RDF/XML)

comme les préfixes. Les champs RDF sont ensuite séparés par des espaces ou des tabulations et chaque déclaration se termine toujours par un point.

Plus précisément, les URI sont écrits entre des crochets ($< >$), les nœuds vides sont précédés de $(\_ :)$ et les littéraux sont placés entre guillemets (””). Chaque assertion est présentée sur une seule ligne terminée par un point (.). La figure suivante montre un exemple de la représentation syntaxique N-triple de la Figure 2.1.

L'avantage de cette syntaxe est sa simplicité de sérialisation, d'analyse et de traitement, mais l'utilisation des IRI complets est moins conviviale pour l'homme.

Turtle

Turtle a été normalisé pour la première fois en 2014 dans le cadre de la norme RDF 1.1 [5]. Il fait partie d'une famille de syntaxes similaire à celle de N-Triples, les deux étant inspirées de Notation3 (N3) [6]. Une simple assertion dans Turtle est une séquence

de sujet, prédicat et objet séparés par des espaces ou des tabulations et terminée par un point, comme montré dans la Figure 2.4.

```
@prefix air_traffic: <http://www.isae-ensma.fr/air_traffic#> .
@prefix local: <http://www.isae-ensma.fr/vocabulaire#> .

air_traffic:AirFrance local:has_flight air_traffic:AF37 .

air_traffic:Iberia local:has_flight air_traffic:IB52,
        air_traffic:IB83 .

air_traffic:AF37 local:arrival air_traffic:Paris_Orly ;
    local:departure air_traffic:El_Prat ;
    local:plane_model air_traffic:A319 .

air_traffic:IB52 local:arrival air_traffic:London_Heathrow ;
    local:departure air_traffic:El_Prat ;
    local:plane_model air_traffic:B737 .

air_traffic:IB83 local:arrival air_traffic:Berlin_Tegel ;
    local:departure air_traffic:London_Heathrow ;
    local:plane_model air_traffic:B737 .

air_traffic:Paris_Orly local:iata_code "ORY" .

air_traffic:El_Prat local:iata_code "BCN" .
```

FIGURE 2.4 – Syntaxe concrète RDF (Turtle)

RDFa

RDFa (RDF in Attributes) a été recommandé par le W3C en 2008 [2] pour intégrer RDF dans des documents XHTML (eXtensible HyperText Markup Language) qui sont similaires aux documents HTML mais avec l'exigence supplémentaire qu'ils soient des documents XML valides. RDFa 1.1 a été conçu par la suite pour assouplir la dépendance aux espaces de noms spécifiques à XML/XHTML, ce qui signifie essentiellement que RDFa peut être intégré dans une gamme plus large de documents HTML, tels que HTML 4.0 ou HTML 5.0. L'avantage de RDFa consiste à permettre aux humains et aux machines de gérer la même page Web. D'ailleurs, des moteurs de recherche tels que Google, Bing, Yandex et Yahoo ! ont adoptés RDFa pour permettre l'extraction de données par des agents. Un exemple de cette syntaxe est montré dans la Figure 2.5.

```
<div xmlns="http://www.w3.org/1999/xhtml"
  prefix="
    rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
    local: http://www.isae-ensma.fr/vocabulaire#
    rdfs: http://www.w3.org/2000/01/rdf-schema#"
  >
  <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#Iberia">
    <div rel="local:has_flight">
      <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#IB83">
        <div rel="local:arrival" resource="http://www.isae-ensma.fr/air_traffic#Berlin_Tegel"></div>
        <div rel="local:plane_model" resource="http://www.isae-ensma.fr/air_traffic#B737"></div>
        <div rel="local:departure" resource="http://www.isae-ensma.fr/air_traffic#London_Heathrow"></div>
      </div>
    </div>
    <div rel="local:has_flight">
      <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#IB52">
        <div rel="local:departure" resource="http://www.isae-ensma.fr/air_traffic#El_Prat"></div>
        <div rel="local:plane_model" resource="http://www.isae-ensma.fr/air_traffic#B737"></div>
        <div rel="local:arrival" resource="http://www.isae-ensma.fr/air_traffic#London_Heathrow"></div>
      </div>
    </div>
  </div>
  <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#AirFrance">
    <div rel="local:has_flight">
      <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#AF37">
        <div rel="local:arrival" resource="http://www.isae-ensma.fr/air_traffic#Paris_Orly"></div>
        <div rel="local:plane_model" resource="http://www.isae-ensma.fr/air_traffic#A319"></div>
        <div rel="local:departure" resource="http://www.isae-ensma.fr/air_traffic#El_Prat"></div>
      </div>
    </div>
  </div>
  <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#El_Prat">
    <div property="local:iata_code" content="BCN"></div>
  </div>
  <div typeof="rdfs:Resource" about="http://www.isae-ensma.fr/air_traffic#Paris_Orly">
    <div property="local:iata_code" content="ORY"></div>
  </div>
</div>
```

FIGURE 2.5 – Syntaxe concrète RDF (RDFa)

2.3 Interrogation de données RDF avec SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) est un langage d'interrogation proposé par le W3C afin de récupérer et de manipuler des données représentées en utilisant le format RDF. SPARQL peut être utilisé pour effectuer des requêtes sur diverses sources de données RDF dont les résultats seront donnés sous forme d'ensembles de triplets ou de graphes RDF.

Dans la figure 2.6, nous présentons une version simplifiée de la syntaxe complète de SPARQL. Principalement, une requête SPARQL peut être divisée en cinq parties. La première partie concerne les déclarations de préfixes facultatifs à l'en-tête pour abréger les URI et rendre le reste de la requête plus lisible. Deuxièmement, la norme autorise plusieurs formes de requêtes, SELECT est la forme la plus populaire, elle permet de

spécifier les variables qui doivent être retournées. Troisièmement, des clauses facultatives sur les sources RDF peuvent être définies pour spécifier le graphe RDF sur lequel la requête est exécutée. Ensuite, la clause WHERE, qui est le cœur de la requête SPARQL, spécifie dans ses termes un ensemble de conditions utilisées pour composer le résultat. Enfin, des modificateurs de solution facultatifs, opérant sur les triplets sélectionnés par les clauses WHERE, peuvent être définis pour affiner la sélection avant de générer les résultats.

Notez que dans une requête SPARQL, il y a deux clauses principales, la première est la spécification de la forme de la requête, alors que la seconde est la clause WHERE.

RequêteSPARQL := [En-tête*] Forme [Jeu_de_données] WHERE { Modèle } Modificateurs

---

En-tête := *PREFIX valeur valeur* | *BASE valeur*

Forme := *SELECT* [*DISTINCT*|*REDUCED*] (*joker*|*var*∗) | *ASK* | *CONSTRUCT var* ∗ | *DESCRIBE*

Jeu_de_données := *FROM valeur* | *FROM Named value*

Modèle := *Modle . Modle* | {*Modle*} *UNION* {*Modle*} | *Modle OPTIONAL* {*Modle*}
| (*valeur*|*var*) (*valeur*|*var*) (*valeur*|*var*) | *FILTER Constraint*

Modificateurs := *LIMIT valeur* | *OFFSET valeur* | *ORDER By* [*ASK*|*DESC*] *var*∗

---

var := (‘?′|‘$′)*valeur*

joker := ′∗′

valeur ∈ *String*

FIGURE 2.6 – Syntaxe de base d’une requête SPARQL

Comme montré dans la Figure 2.6, SPARQL supporte quatre formes de requêtes utilisant les solutions de correspondance des motifs de graphes pour constituer des ensembles de résultats ou des graphes RDF. Les formes de requêtes sont les suivantes :

— SELECT : Retourne un tableau de résultats ou une correspondance BGP.
— ASK (Requêtes booléennes) : retourne un booléen indiquant si la requête a un ou plusieurs résultats (true) ou non (false).
— CONSTRUIRE (Création de graphes) : retourne un graphe RDF basé sur un modèle spécifié dont les variables sont instanciées avec les solutions de la requête.
— DESCRIRE (Décrire les ressources) : retourne un graphe RDF décrivant les ressources trouvées comme solutions pour les variables spécifiées.

Dans le cadre de cette thèse, nous intéressons à l’optimisation des requêtes de type

SELECT, car la complexité du problème de recherche du meilleur plan d'exécution des requêtes est NP-Hard.

La représentation textuelle de la requête de la Figure 2.7 est la suivante :

```
SELECT ?c ?f ?m WHERE {
        ?c <:has_flight>   ?f .          #tp1
        ?f <:departure>    <El Prat> .   #tp2
        ?f <:plane_model>  ?m .          #tp3
}
```

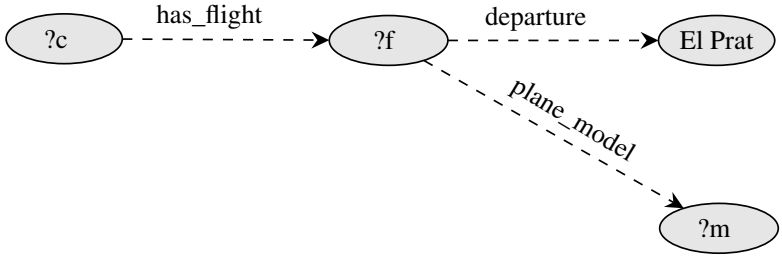


FIGURE 2.7 – Exemple de requêtes SPARQL Q

2.4 Gestion des données RDF : Aperçu

La nécessité de gérer et interroger efficacement les données RDF a conduit au développement de nouveaux systèmes conçus spécialement pour traiter ce format de données. Ces approches peuvent être catégorisées en étant centralisées et distribuées comme l'indique la figure 2.8. Les approches dites centralisés s'appuient sur une seule machine pour gérer les données RDF contrairement aux approches distribuées qui peuvent combiner plusieurs machines connectées avec un réseau informatique.

Deux types d'approches centralisées peuvent être distinguée, celles qui s'appuient sur un système de gestion de données existant et celles qui sont développées en "partant de rien". Le premier type est considéré étant non natif tandis que le deuxième type est considéré comme natif. Les systèmes non natifs sont basés principalement sur les systèmes de gestion de bases de données relationnelles, et peuvent être organisés en trois catégories selon le type de tables utilisée : la table de triplets, la table de propriété et la table binaire. Les approches basées sur la table de triplets proposent de stocker et d'interroger les

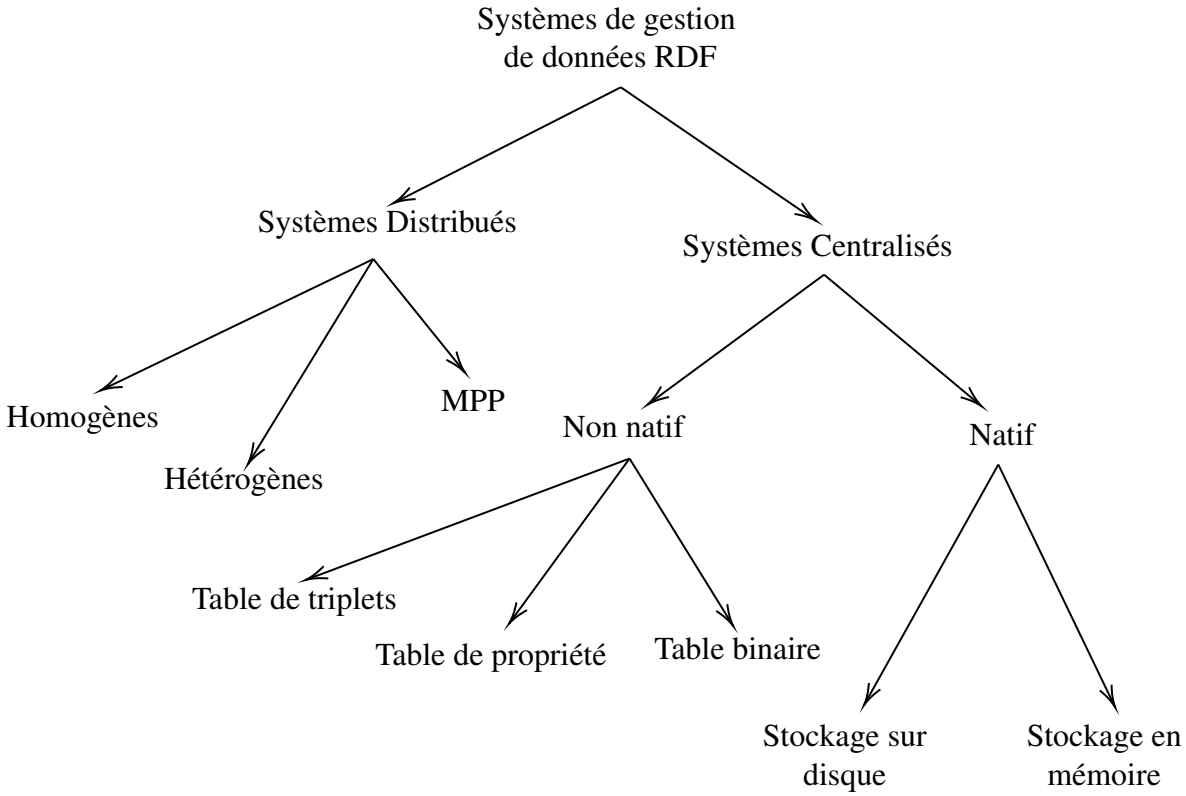


FIGURE 2.8 – Approches de gestion de données RDF

données organisées avec une seule très grande table de trois attributs. De nombreux systèmes adoptent cette approche, à l'image de Redland (2002)[4], qui est l'un des premiers moteurs de stockages des données RDF. 3Store (2003) [17] est arrivé juste après et qui utilise la même stratégie. De plus, 3Store propose de prendre en compte les données RDF représentées avec le vocabulaire XML. En 2005, Oracle [10] a introduit au sein de son SGBD relationnel la fonction SQL "RDF_MATCH", qui permet d'évaluer des requêtes Sparql. Dés 2007, le système Virtuoso [12] a proposé de stocker, en plus des triplets, des quadruplets, qui permettent d'entendre les triplets pour intégrer une information sur le contexte lié à chaque triplet. Le système RDFKB [26], arrivé en 2009, permet de prendre en charge l'inférence au moment du stockage des données plutôt que dans le traitement des requêtes. Enfin, JenaSDB [39], proposé en 2013, a adopté une approche générique permettant de rester indépendant du SGBD utilisé pour stocker physiquement les données.

La deuxième catégorie des systèmes non natifs est basée sur la table de propriétés qui permet de regrouper les triplets qui ont les mêmes propriétés. Chaque groupe est stocké ensuite séparément dans une table relationnelle. Sesame (2002) [7] est le premier système

à adopter cette stratégie de stockage. Il est conçu en effet pour interagir avec n'importe quel système de stockage relationnel. Jena [25], proposé à partir de 2006, est l'un des principales approches adoptant la table de propriété, d'ailleurs ce système s'appuie sur deux types de tables de propriétés, une basée sur le clustering de propriétés et une autre basée sur les classes.

La troisième catégorie des systèmes non natifs est basée sur des tables binaires [1] où les données RDF sont partitionnées verticalement. L'ensemble de triplets est organisé en n tables à deux colonnes, où n est le nombre de propriétés distinctes. Dans chacune de ces tables, la première colonne regroupe les sujets des triplets tandis que la deuxième colonne regroupe les objets des mêmes triplets.

En s'appuyant sur des systèmes existants, nous arrivons à avoir rapidement une solution permettant de gérer les données RDF. Malheureusement, ce type de systèmes représente des inconvénients majeurs, qui sont liés principalement aux performances. En effet, les représentations relationnelles ne permettent pas de prendre en compte la particularité des données RDF et plus précisément la nature graphe. Les techniques classiques d'optimisation deviennent inadaptées lorsqu'il s'agit de requêtes SQL générées à partir de requêtes SPARQL.

Les systèmes natifs sont basés sur une approche spécialement conçue pour la gestion des triplets RDF. Elles correspondent principalement à des approches d'indexation intensive. Les systèmes les plus efficaces sont considérés comme appartenant à cette catégorie. Dans ce contexte, Allegrograph [37] est l'un des premiers systèmes à adopter à la fois la présentation graphe des données et l'indexation intensive. L'évolution de cette catégorie de systèmes s'est accélérée rapidement. En 2005 par exemple, le système YARS [18] a été proposé. Ce dernier s'appuie sur six indexes de type B+Tree. Dans ce même contexte, KOWARI [40] propose une approche hybride combinant les deux types d'arbres AVL et B-tree (au lieu de B+tree utilisé dans YARS). RDFCube (2006) [24] utilise un index de trois dimensions qui correspondent aux sujets, prédicats et objets. GRIN (2007) [36] est le premier système d'indexation RDF à utiliser le partitionnement graphe. Dans cette même catégorie, RDF3X (2008) [29] est considéré comme l'une des approches les plus populaire. Ce système se distingue des autres approche par la proposition d'un système complet reprenant les différents composants classiques d'un SGBD relationnel. L'utilisation intensive des jointures est la base de ce system. TipleT (2009) [13] adopte une technique d'in-

dexation partielle afin de limiter la taille de l'index. iStore (2009) [33] exploite un index de structure construit automatiquement à partir des données. Cet index peut agir comme un schéma, permettant la navigation et l'interrogation de données Web sans schéma. Malgré que ce type d'approche ait été conçu spécialement pour la gestion des données RDF, la majorité des approches peuvent difficilement être étendu pour prendre en compte le vocabulaire de SPARQL. Le passage à l'échelle peut rapidement poser problème quand il s'agit de gérer plusieurs milliards de triplets. Enfin, le dernier système appartenant à cette catégorie est gStore (2011) [43]. Ce système propose de garder la structure graphe en stockant les données RDF sous forme de listes adjacentes. gStore utilise un index binaire de type S-Tree contrairement aux approches existantes qui s'appuient sur des indexes utilisés par les systèmes relationnels. Malheureusement, une telle conception ne permet pas toujours de gérer les jeux de données avec une taille importante. Même coté performances, ce n'est pas toujours évident de répondre efficacement à certains types de requêtes.

Certain système privilégie l'utilisation de la mémoire principale dans la gestion des données RDF. Cela permet en effet, d'améliorer les performances. BRAHMS [21] suit cette logique et permet d'identifier les associations sémantiques entre les différents triplets. Hexastore [38] se base sur l'indexation multidimensionnelle en mémoire principale. BitMat (2009) [3] s'appuie sur une structure matricielle et binaires. Parliament (2009) [23] décrit un schéma de stockage et d'indexation basé sur les listes adjacentes. TripleBit (2013) [41] introduit deux structures d'indexation afin de minimiser le coût de la sélection de l'index lors de l'évaluation de la requête. Malgré que les performances de ce genre d'approches puissent être supérieures par rapport aux approches disques, le passage à l'échelle reste le point faible de ce genre de solution surtout quand il s'agit de gérer plusieurs centaines de millions de triplets.

Afin de garantir le passage à l'échelle, plusieurs approches basées sur le calcul distribué ont été proposée. Deux types de distribution ont été utilisés jusqu'à maintenant, à savoir la distribution homogène et la distribution hétérogène. Un système homogène repose sur le fait que toutes machines du cluster utilisent la même configuration logicielle et matérielle. Dans ce contexte, l'architecture réseau (e.g, type de communication) du système peut jouer un rôle important lors du traitement des requêtes. Deux architectures sont principalement utilisées, à savoir, l'architecture Pair-à-Pair et l'architecture client-Serveur. Concernant le premier type d'architecture, nous pouvons citer RDFPeers

(2006) [8] qui est considéré comme le premier système RDF distribué à utiliser une communication Pair-à-Pair. Il fournit différents composants pour le stockage, l'indexation et l'interrogation à L'aide du langage RDQL. MIDAS-RDF [34] fait aussi partie de cette catégorie. Il est basé sur une structure d'index multidimensionnelle distribuée. MIDAS-RDF propose une récupération rapide des triplets RDF satisfaisant diverses requêtes de modèle en les traduisant en requêtes de plage multidimensionnelles.

L'architecture Client-serveur a été aussi utilisé pour gérer les données RDF. Dans ce type d'architecture, un nœud joue le rôle du maitre afin d'orchestrer l'exécution des tâches assignées aux autres nœuds (considérés comme des esclaves). Parmi les systèmes qui utilisent ce réseau YARS2(2007) [19] qui est basé sur le système Yars. Il fournit des méthodes distribuées d'indexation et d'évaluation de requêtes.

Les systèmes hétérogènes correspondent généralement à un moteur de requêtes fédéré ou à base de médiateur. L'approche fédérée de requêtes a attiré beaucoup d'attention, en raison du potentiel élevé que cela représente pour le développement des points d'entrées SPARQL et du mouvement lié aux données liées, ainsi que de la popularité du langage de requête SPARQL.

Splendid (2011) [16] est l'un des premiers systèmes à adopter une approche fédérée pour répondre à des requêtes SPARQL. FedX [32], proposé aussi en 2011, permet aux utilisateurs du Web sémantique d'intégrer à la demande des points d'entées SPARQL existants. Les systèmes basés sur les médiateurs correspondent principalement aux systèmes OBDA (ontology-based data access) et peu d'entre eux sont capables d'interagir avec plusieurs sources de données existant.

Avec la mouvance du Big Data, un nouveau type d'approches a été proposé. Ces dernières s'appuient sur l'architectures Traitement massivement parallèle (en anglais MPP - Massively Parallel Processing).

Le traitement massivement parallèle est un moyen de croquer d'énormes quantités de données en distribuant le traitement sur des centaines ou des milliers de processeurs, qui peuvent fonctionner dans la même boîte ou dans des ordinateurs séparés et éloignés. Chaque processeur d'un système MPP possède sa propre mémoire, disques, applications et instances du système d'exploitation. Parmi les systèmes qui utilisent cette approche SHARD (2011) [30] qui s'appuie sur Hadoop pour les aspects de persistance des données et de traitement des requêtes, HadoopRDF (2011) [20] combine le framework Hadoop

distribué avec le système RDF centralisé RDF3X. CliqueSquare (2015) [11] limite le nombre de tâches MapReduce et le transfert de données entre les nœuds pendant l'évaluation des requêtes. SparkRDF (2015) [9] est basé sur Spark, il partitionne le graphe RDF en sous-graphe selon les relations et les classes. S2RDF (2016) [31] utilise l'interface relationnelle de Spark pour l'exécution des requêtes et étend le schéma de partitionnement vertical.

L'inconvénient majeur des approches MPP réside dans leurs incapacités de répondre efficacement à certaines requêtes. L'utilisation d'un système parallèle pour traiter des requêtes peut induire un temps supplémentaire qui peut impacter considérablement les performances. Les approches MPP sont conçues pour garantir le passage à l'échelle en termes de volume de données. Elles se trouvent utiles pour gérer plusieurs centaines de téraoctets voir des Petaoctets. Nous nous pesons la question de l'utilité de telles approches dans le contexte du traitement de requêtes RDF.

Aucune des approches n'est en mesure de garantir à la fois les performances et le passage à l'échelle. Quelques résultats expérimentaux liés au passage à l'échelle sont représentés dans la figure 2.9.

	Chargement de données			Support des requêtes wildcards			Support Order-By & Group-By		
	gStore	RDF-3X	Virtuoso	gStore	RDF-3X	Virtuoso	gStore	RDF-3X	Virtuoso
Watdiv100M	✓	✓	✓	✓	✘	✓	✓	✘	✓
Watdiv1B	✘	✓	✓	✘	✘	✓	✘	✘	✓
LUBM500M	✘	✓	✓	✘	✘	✓	✘	✘	✓
LUBM1B	✘	✘	✘	✘	✘	✘	✘	✘	✘
Yago	✘	✓	✓	✘	✘	✓	✘	✘	✓
DBLP	✘	✓	✓	✘	✘	✓	✘	✘	✓

FIGURE 2.9 – Résultats expérimentaux de quelques systèmes

Le seul système permettant de garantir un certain compromis entre passage à l'échelle et performances est le système RDF QDAG [22] (https ://qdag.projets.univ-poitiers.fr/qdag/), basée sur l'exploration de graphe et la fragmentation. Nous présentons ce système dans le chapitre suivant.

2.5 Conclusion

Dans ce chapitre, nous avons donné un aperçu sur les technologies liées à la représentation et l'interrogation de données RDF. Nous nous sommes intéressés au langage

SPARQL qui est considéré aujourd'hui comme une recommandation pour l'interrogation de ce type de données. Nous avons donné un aperçu sur les approches existantes permettant de gérer les données RDF. Dans le chapitre suivant, nous donnons quelques détails sur le système RDF_QDAG.

Chapitre 3

Étude de l'existant : RDF_QDAG

3.1 Introduction

Dans cette section, nous décrivons les concepts et techniques clés utilisés dans RDF_QDAG.

3.2 Vue d'ensemble de RDF_QDAG

3.2.1 Représentation des données

L'objectif des concepteurs de RDF_QDAG était de tirer parti de la notion de voisinage entre les nœuds de la représentation logique des graphes RDF, ce qui évite l'utilisation de jointures intensives lors de l'évaluation des requêtes. Ils ont décidé de s'appuyer sur la notion de DataStar, qui permet de regrouper un ensemble de triplets ayant le même sujet ou le même objet. Physiquement, un nœud et ses voisins sont regroupés dans une même entité. On peut également distinguer deux types de DataStar selon le type d'arcs utilisés : entrants et sortants.

RDF_QDAG permet également une représentation native des nœuds basée sur six types : String, Long, Integer, Short, Float et Double. Les données de type chaîne de caractères sont remplacées par des identifiants gérés par RDF_QDAG. Un dictionnaire est créé pour gérer les correspondances <id, String>. Les étoiles de données sont regroupées en fragments reposant sur des ensembles de caractéristiques [28]. Chaque fragment est indexé et compressé séparément. Deux treillis sont utilisés pour indexer les ensembles caractéristiques des fragments.

3.2.2 Évaluation des requêtes

RDF_QDAG prend en charge les requêtes avec BGP, le filtrage WildCard, le filtrage spatial, le regroupement et le tri. L'évaluation de la requête repose sur l'exploration logique du graphe guidée par la requête. Une requête est ensuite divisée en étoiles de requêtes. Pour chaque nœud BGP, les étoiles de requêtes associées sont extraites. Une étoile de requête suit le même principe qu'une étoile de données. Elle regroupe les triplets liés à un sujet ou à un objet dans le BGP.

Un plan d'exécution est basé sur un ordre d'un ensemble donné d'étoiles de requêtes. RDF_QDAG ne prend en compte que les plans garantissant une évaluation par exploration. Une correspondance partielle est effectuée sur un ensemble de fragments pour chaque étoile de requête sélectionnée à l'aide de treillis. Un plan d'exécution contient non seulement des opérateurs de mise en correspondance liés aux étoiles de requêtes, mais également des opérateurs de filtrage (numériques, chaînes de caractère et spatiaux), des opérateurs de tri et de regroupement.

3.3 Principales fonctionnalités de RDF_QDAG

La figure 3.3 illustre l'architecture de RDF_QDAG. Il adopte une conception et une mise en œuvre en couches. Nous expliquons brièvement les idées techniques clés dans chaque couche.

3.3.1 Chargement des données

Cette première couche regroupe quatre composants : Fragmenter, Entity Linker, Dictionary Constructor et Statistics Collector. Le fragmenteur assure l'extraction des étoiles de données et les regroupe en fragments. Le entity linker permet d'associer à chaque nœud l'id du fragment qui regroupe ses arcs entrants et l'id du fragment qui regroupe ses arcs sortants. Ces liens sont nécessaires à la stratégie d'évaluation de RDF_QDAG qui s'appuie sur l'exploration logique des graphes. Le constructeur du dictionnaire permet de remplacer les chaînes de caractères associées aux nœuds du graphe par des identifiants. Enfin, le collecteur de statistiques permet de générer des statistiques sur les valeurs des fragments et les interactions entre elles. Ces statistiques sont très utiles pour les différentes stratégies d'optimisation proposées dans le cadre de RDF_QDAG.

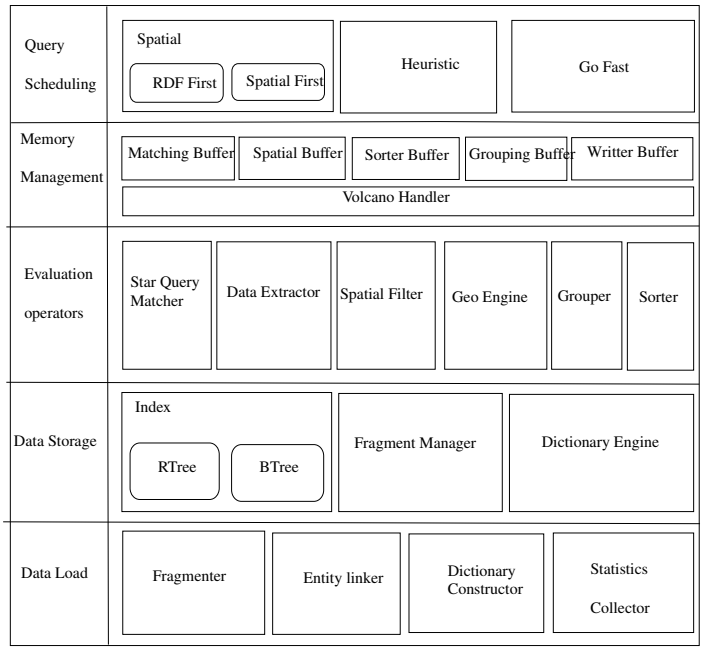


FIGURE 3.1 – Architecture en couches de RDF_QDAG

3.3.2 Stockage des données

Les fragments sont stockés sous la forme d'un B+Tree groupé. RDF_QDAG s'appuie également sur un R-Tree pour gérer les données spatiales. Les différents index sont implémentés en C++ afin d'offrir les meilleures performances possibles. Un gestionnaire de fragments est également intégré afin de gérer l'accès aux fragments. Enfin, le moteur de dictionnaire gère l'encodage/décodage des valeurs des chaînes de caractères. Il offre également la possibilité de rechercher des valeurs String à l'aide d'expressions régulières. La version actuelle de RDF_QDAG repose sur SQLITE 3.

3.3.3 Opérateurs d'évaluation

Pour l'évaluation des requêtes, les concepteurs de RDF_QDAG ont opté pour un traitement personnalisé pour chaque requête. Ce triplestore sélectionne un ensemble d'opérateurs pour créer un flux permettant de répondre à la requête. Pour chaque étoile de requêtes, RDF_QDAG crée un extracteur de données et un matcher d'étoile de requête. L'extracteur de données permet d'interagir avec les fragments afin de charger les étoiles de données pertinentes. Le matcher d'étoile de requête, d'autre part, permet de trouver des correspondances pour les variables mentionnées dans l'étoile de requête. Les filtres numériques sont pris en compte dans le matcher. Des filtres spatiaux et des filtres à base

de chaînes de caractères sont ajoutés au flux dans le cas d'un traitement spatial ou d'un filtrage à base d'expressions régulières. Un filtre spatial doit communiquer avec le moteur géographique et le dictionnaire. RDF_QDAG s'appuie sur CGAL [1] et JTS [2] comme moteurs géographiques. Le filtre à base d'expressions régulières interagit avec le dictionnaire pour décoder les chaînes de caractères. Dans le cas de la clause Group By (ou Order By), un composant de regroupement (tri) est ajouté au flux. RDF_QDAG integere aussi des opérateurs pour envoyer les résultats au client. Trois types sont implémentés selon le type de client utilisé : fichier, réseau et console.

3.3.4 La gestion de mémoire

Après la sélection des composants, RDF_QDAG associe des mémoires tampons à chaque opérateur, qui peut être à la fois producteur/consommateur. Les mémoires tampons sont optimisées pour chaque type d'opérateur. Le tampon du matcher est utilisé pour stocker les résultats partiels des correspondances. Les entrées de regroupement (tri) sont gérées dans un tampon de regroupement (tri). Enfin, le tampon d'écriture gère les données à retourner à l'utilisateur. Chaque tampon a une taille limite. Dans le cas d'un tampon de regroupement (tri), RDF_QDAG utilise le dique pour stocker les objets en excès, car le tri et le regroupement nécessitent les résultats complets.

3.3.5 Planification des requêtes

Cette couche regroupe plusieurs composants afin de fournir plusieurs stratégies d'exécution et d'optimisation. Pour l'exécution, RDF_QDAG s'appuie sur un gestionnaire, basé sur le modèle du volcan. Il gère l'exécution des opérateurs d'évaluation en fonction de règles liées aux taux de remplissage des mémoires tampons. Pour les données spatiales, RDF_QDAG propose deux stratégies d'exécution. La première "RDF first" consiste à traiter d'abord la partie BGP, les filtres spatiaux sont appliqués après avoir trouvé les correspondances avec les variables. D'autre part, la stratégie "Spatial first" permet d'utiliser l'index R-Tree en appliquant un filtre spatial avant de traiter la partie BGP de la requête.

RDF_QDAG propose également plusieurs stratégies d'optimisation. Tout d'abord, il

1. `https://www.cgal.org/`
2. `https://github.com/locationtech/jts`

décide de l'ordre d'évaluation des étoiles de requêtes. Globalement, cet ordre permet d'orienter l'exploration du graphe. Deux stratégies permettent de choisir cet ordre : Une Heuristique basée sur l'approche proposée par Tsialiamanis et al. [35] et GoFast [44]. RDF_QDAG intègre également un algorithme d'élagage permettant d'éliminer les fragments qui ne contribuent pas à la construction des résultats finaux.

3.4 Solution proposée

La contribution que nous avons apportée au système QDAC consiste à proposer une nouvelle approche pour la fragmentation. Grâce à cette approche, basée sur le framework Spark, nous somme arrivés à résoudre le problème du passage à l'échelle qui pénalisait le système pendant la phase de chargement.

Pour cela, nous avons réviser toutes les étapes de fragmentation pour proposer une implémentation compatible avec Spark. Nous avons utilisé le langage fonctionnel Scala pour assurer un code robuste tout en minimisant les erreurs de tests et le débogage.

3.5 Conclusion

Maintenant que nous avons présenté une vue d'ensemble du système QDAC et proposer une solution de partitionnement, nous pourrons passer à la conception de cette solution. Chose qui sera détaillée au prochain chapitre.

Chapitre 4

Conception

4.1 Introduction

L'étude de la version initiale du QDAG nous a permis de rédiger les besoins et les spécifications de ce dernier en termes de chargement et traitement des données. Dans ce chapitre, nous allons détailler la conception de notre solution qui vise à assurer de réglé le problème du chargement et le passage à l'échelle d'une manière fiable et efficace .

4.2 Analyse des besoins

1. Le système doit assurer le chargement des fichiers d'entrée (.nt , .schema) qui ont ses propres structures.
2. Le système doit gérer l'encodage des données d'entrée par des identifiants uniques.
3. A partir des données encodées, le système doit générer les fragments appropris.
4. Le système doit assurer la création d'un dictionnaire de données
5. Le système doit assurer la création des fichiers index et .data et .schema qui correspondent aux fragments générés.

Ces besoins là vont nous permettre de rédiger le diagramme de processus la figure (figure 4.1) représente les différentes fonctionnalités qui doivent être fournies par le système.

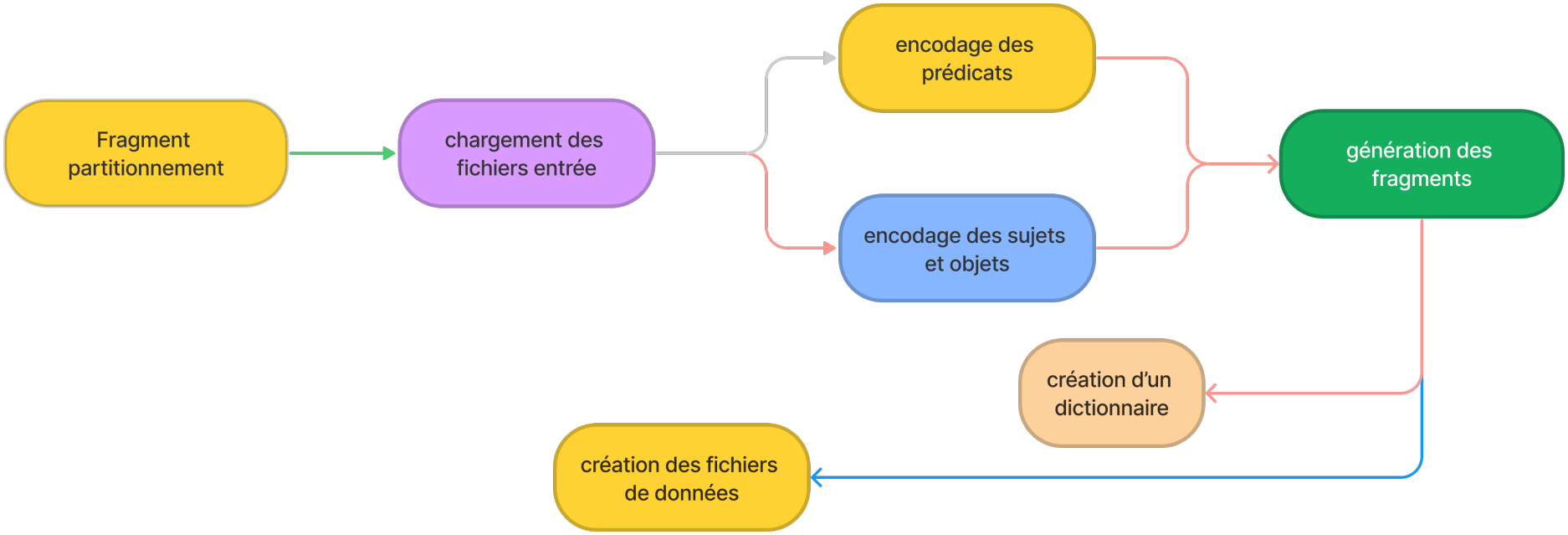


FIGURE 4.1 – Le diagramme de composant

4.3 Conception fonctionnelle

I) Chargement des fichiers d'entrée

1. Le chargement des données sources (.nt et .schema) se fait dans des Data-Frames, les fichiers de type .nt (Table 4.3) ayant la structure [Subject,Predicat,Object] ce qui veut dire qu'ils sont structurés par des triplets (S.P.O), les fichiers .schema (Table 4.2) ayant la structure [Predicat,Range,Domain].

Subject	Predicat	Object
S1	P1	S3
S2	P2	O2
S1	P2	O3
O1	P3	S2
S4	P4	S3
S2	P1	O1
S1	P2	O2
5.2	P4	O1

TABLEAU 4.1 – Les données du fichier .nt

Predicat	range	Domain
P1	String	B
P2	String	String
P3	String	String
P4	String	String

TABLEAU 4.2 – Les données du fichier .schema

II) Encodée les prédicats (Predicat encoding)

2. L'encodage des prédicats est attribuée aux ids uniques pour chaque ligne de prédicat (Table 4.3)

Prédicat	range	Domain	ID
P1	String	B	1
P2	String	String	2
P3	String	String	3
P4	String	String	4

TABLEAU 4.3 – Dataframe des prédicats et ses identifiant

III) Encodée les sujet et les objets (SO encoding)

3. Regrouper les sujets et les objets non numériques dans un dataframe, puis affecter les identifiants à eux.(Table 4.4)

Subject&Object	ID
S1	1
S2	2
S3	3
S4	4
O1	5
O2	6
O3	7

TABLEAU 4.4 – Dataframe du sujet et objet encodee

4. Joindre les dataframes pour rendre un DataFrame comme celles du DataFrame d'entrée (.nt) mais encoder par des identifiants.(Table 4.5)

Subject_ID	Predicat_ID	Object_ID
1	1	3
2	2	6
1	2	7
5	3	2
4	4	3
2	1	5
1	2	6
5.2	4	5

TABLEAU 4.5 – dataframe encodée par des ids

IV) Génération du fragments (Fragment generation)

A/ Génération du fragments SPO

5. Regrouper les triplets (S,P,O) qui ont le même sujet dans une datastar (Figure 4.2), une datastar représente tous les nœuds qui ont la même tête (Figure4.3.5). Vu qu'on doit générer les fragments SPO, les datastar ayant le sujet comme une tête et ses nœuds sont les triplets (Table 4.6) []

head	Triple (S, P, O)
1	[(1,1,3), (1,2,7), (1,2,6)]
2	[(2,2,6), (2,1,5)]
5	[(5,3,2)]
4	[(4,4,3)]
5.2	[(5.2,4,5)]

TABLEAU 4.6 – Les DataStar SPO generer

6. Regrouper les datastar qui ont le même caractéristique (Predicat_id) pour générer des fragments SPO (Figure 4.3), un fragment est tous les datastars qui ont les mêmes valeurs des predicat_id (Table 4.7)

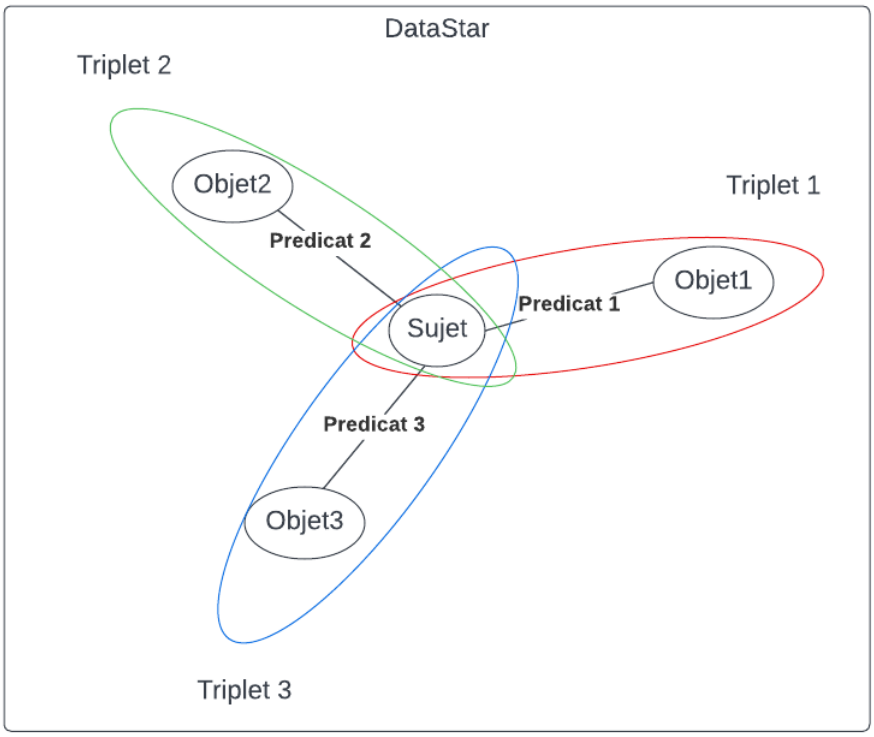


FIGURE 4.2 – Représentation graphique d'un DataStar avec ses triplets

Fragment
D1[(1,1,3), (1,2,7), (1,2,6)],D2 [(2,2,6),(2,1,5)] D5[(5,3,2)] D4[(4,4,3)], D5.2[(5.2,3,5)]

TABLEAU 4.7 – les fragments SPO générer

Clés : D1[(T1,T2)] ça veut dire une datastar qui a une tête 1 et des triplets t1 et t2

Prennent ce fragment à titre d'exemple : D1[(1,1,3), (1,2,7), (1,2,6)],D2 [(2,2,6),(2,1,5)]

7. Découper si c'est nécessaire les fragments spo pour ne pas dépasser une limite paramétrable et ensuite attribuée les ids à eux (Table 4.8)

ID	Fragment
1	D1[(1,1,3), (1,2,7), (1,2,6)],D2 [(2,2,6),(2,1,5)]
2	D5[(5,3,2)]
3	D4[(4,4,3)], D5.2[(5.2,3,5)]

TABLEAU 4.8 – Les fragments SPO encodée

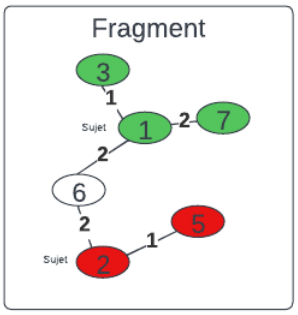


FIGURE 4.3 – Un fragment contient 2 datastar

8. Rediriger tous les segments entrants et sortants qui correspondent aux fragments générés, un segment est l'ID du fragment qui correspond à la tête du datastar, ce qui veut dire qu'on doit parcourir tous les datastars qui figurent dans les fragments pour récupérer la tête et puis affecter l'ID du segment approprié comme étant un segment sortant .(Table 4.9)

head	Segment (IN, OUT)
1	(0,1)
2	(0,1)
5	(0,2)
4	(0,3)
5.2	(0,3)

TABLEAU 4.9 – Les segment spo

B/ Génération du fragments OPS

9. Regrouper les triplets (S,P,O) qui ont le même objet dans une datastar. Vu qu'on doit générer les fragments OPS, les datastar ayant l'objet comme une tête et ses nœuds sont les triplets (Table 4.10)

head	Triple (S, P, O)
3	[(1,1,3), (4,4,3)]
6	[(2,2,6), (1,2,6)]
7	[(1,2,7)]
2	[(5,3,2)]
5	[(2,1,5) ,(5.2,4,5)]

TABLEAU 4.10 – Les DataStar OPS generer

10. Regrouper les datastar qui ont le même caractéristique (Predicat_id) pour générer des fragments OPS (Table 4.11)

Fragment
D3[(1,1,3), (4,4,3)], D5[(2,1,5), (5.2,4,5)]
D6[(2,2,6), (1,2,6)], D7[(1,2,7]
D2[(5,3,2)]

TABLEAU 4.11 – Les fragment ops

11. Découper, si nécessaire, les fragments ops pour ne pas dépasser une limite paramétrable et ensuite attribuer les ids uniques différemment à celle des fragments SPO. (Table 4.12)

ID	Fragment
4	D3[(1,1,3), (4,4,3)], D5[(2,1,5), (5.2,4,5)]
5	D6[(2,2,6), (1,2,6)], D7[(1,2,7]
6	D2[(5,3,2)]

TABLEAU 4.12 – Les fragments OPS encodée

12. Rédiger tous les segments entrants et sortants qui correspond les fragments ops générer. Pour y parvenir, on doit parcourir tous les datastars qui figurent dans les fragments ops pour recuperer la tete et puis affectee l'ID du segment approprie comme étant un segment entrant .(Table 4.13)

head	Segment (IN, OUT)
3	(4,0)
5	(4,0)
6	(5,0)
7	(5,0)
2	(6,0)

TABLEAU 4.13 – Les segment spo

V) Génération d'un dictionnaire de données

13. Fusionner les segments entrants et sortants générés par les fragments spo et ops (Table 4.14), puis regrouper les segments qui ont la même tête dans un tuple (Table 4.15)
14. À partir du dataframe des segments le system doit générer un dictionnaire de données qui contient les segments et les valeurs qui correspond les têtes, il suffit une jointure pour les récupérer (Table 4.16)

    Les fichiers .dic ayant le pattern [tete_id tete_value sg_entrant sg_sortant]

head	Segment (IN, OUT)
1	(0,1)
2	(0,1)
5	(0,2)
4	(0,3)
5.2	(0,3)
3	(4,0)
5	(4,0)
6	(5,0)
7	(5,0)
2	(6,0)

TABLEAU 4.14 – les segments Spo et ops fusionnée

head	Segment (IN, OUT)
1	(0,1)
2	(6,1)
5	(4,2)
4	(0,3)
5.2	(0,3)
3	(4,0)
6	(5,0)
7	(5,0)

TABLEAU 4.15 – les segments SPO et OPS fusionnée avec une tête unique

1	S1	0	1
2	S2	6	1
5	O1	4	2
4	S4	0	3
5.2	5.2	0	3
3	S3	4	0
6	O2	5	0
7	O3	5	0

TABLEAU 4.16 – Dictionnaire de données .dic

VI) Génération des fichiers

15. Générer les fichiers .data de chaque fragment SPO ce fichier est nommée par l'identifiant du fragment spo , ces fichiers contient les triplets de chaque datastar du fragment et a chaque triplet on y ajoute le segment entrant qui correspond le sujet ainsi les segments entrant et sortant d'objet.(Table 4.17)

    donc ces fichier ayant la structure

    [subject_id,s_sg_in,predicat_id, object_id,o_sg_in ,o_sg_out]

    Exemple : Prenant le fragment numéro 1

    D1[(1,1,3), (1,2,7), (1,2,6)], D2 [(2,2,6), (2,1,5]

Et voici son processus (Figure 4.4)

1/Parcourir ces triplets spo

2/Charger le segment entrant du sujet

3/Charger le segment entrant et sortant d'objet

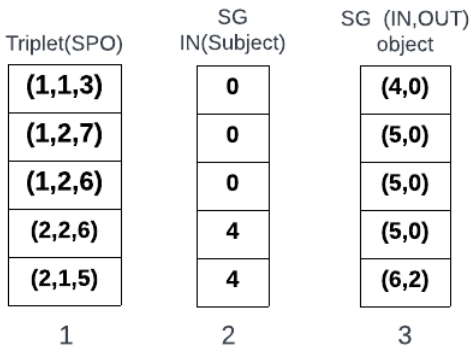


FIGURE 4.4 – Processus de génération des données de fichier .data

1	0	1	3	4	0
1	0	2	7	5	0
1	0	2	6	5	0
2	6	2	6	5	0
2	6	1	5	4	2

TABLEAU 4.17 – fichier 1.data du segment spo 1

16. générer les fichiers .schema de chaque fragment SPO, ce fichier est nommé par l'identifiant du fragment il comporte les domaines de ses predicats et le range de l'un entre eux il suffit de choisir une datastar du fragment

    Exemple : Prenant le datastar D1[(1,1,3), (1,2,7), (1,2,6)] du Fragment 1 (Table **??**)

    Les fichiers .schema ayant le pattern

    [Range [predicat_id domain]]

String	
1	String
2	String

TABLEAU 4.18 – fichier 1.schema du segment spo 1

17. Générer les fichiers .data de chaque fragment OPS, ce fichier est nommée par l'identifiant du fragment OPS , il est remplit par tous les triplets des datastar

du fragment et a chaque triplet en y ajoute le segment sortant qui correspond l'objet ainsi les segments entrant et sortant du sujet.

Prenant le fragment numéro 4 a titre d'exemple

D3[(1,1,3), (4,4,3)], D5[(2,1,5), (5.2,4,5)]

Et voici son processus (Figure 4.5)

1/Parcourir ces triplets SPO

2/Charger le segment sortant d'objet

3/Charger le segment entrant et sortant du sujet

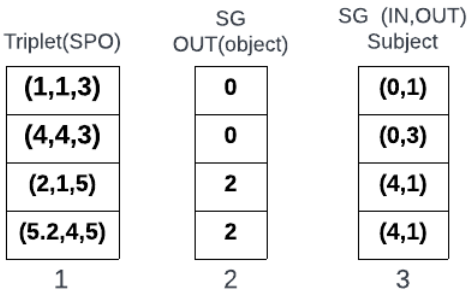


FIGURE 4.5 – Processus de génération des données de fichier .data ops

Les fichiers .data du fragment OPS ayant le pattern (Table 4.19)

[object_id o_sg_out predicat_id subject_id s_sg_in s_sg_out]

3	0	1	1	0	1
3	0	4	4	0	3
5	2	1	2	4	1
5	2	4	5.2	4	1

TABLEAU 4.19 – fichier 4.data du segment OPS 4

18. Générer les fichiers .schema de chaque fragment OPS de la même manière du fragment SPO

Prenant le datastar D3[(1,1,3), (4,4,3)] du Fragment 4 (Table 4.20)

String	
1	String
4	String

TABLEAU 4.20 – fichier 4.schema du segment OPS 4

4.4 Conception dynamique

4.4.1 Chargement et encodage des données d'entrée

La figure 4.6 montre le processus de l'encodage lorsque on reçoit les fichiers base au notre framwork .

D'abord , dés que les dataframes du fichiers source sont chargée les processus doivent respecter l'ordre

1. a partir du dataframe du prédicat chargée on ajoute une colonne pour attribuée les ids et exporter cette dernier dans un fichier .txt
2. après le chargement du dataframe du fichier .nt on regroupe les sujets et les objets dans dataframe ensuite appliquée une distinction pour éliminer les redondants
3. cette étape consiste a attribuée les ids a les sujets et les objets regrouper cette action doit fait aux éléments non numérique seulement et pour les éléments numérique on affecte a eux ses valeurs
4. Lorsque les dataframes sont encodée il nous reste que établir une jointure entre les dataframe encodée et dataframe des triples (.nt) pour collecter les ids des triplets et rendre une dataframe des triplets encode par les ids .

4.4.2 Génération des fragments

1/ La figure 4.7 montre le processus de génération des fragment spo qui est lancée par **FragmentManager**

5. a partir du dataframe des triplets encodée on regroupe les triplets et les predicats qui ont le même sujets dans une dataset , on regroupe les triplets dans une Liste[String,String,String] et les predicats dans une Liste[String] , ensuite pour rendre ce regroupement d'une manière dynamique on reproduit le regroupement précèdent dans une DataSet des objets DataStar et garder la liste des predicat
6. a travers le dataset des triplets regrouper dans des datastar , on regroupe autre fois les datastar qui ont les mêmes caractéristique c-a-d qui ont les mêmes predicat_id sont prendre en considération l'ordre ou la redondances des ids , vu que on a une liste des predicats a travers lui on peut regrouper les datastars dans un fragment.
7. réduire la taille des fragments pour ne dépasse pas une limite paramétrable ensuite attribuée les ids a les fragments générer a nouveaux

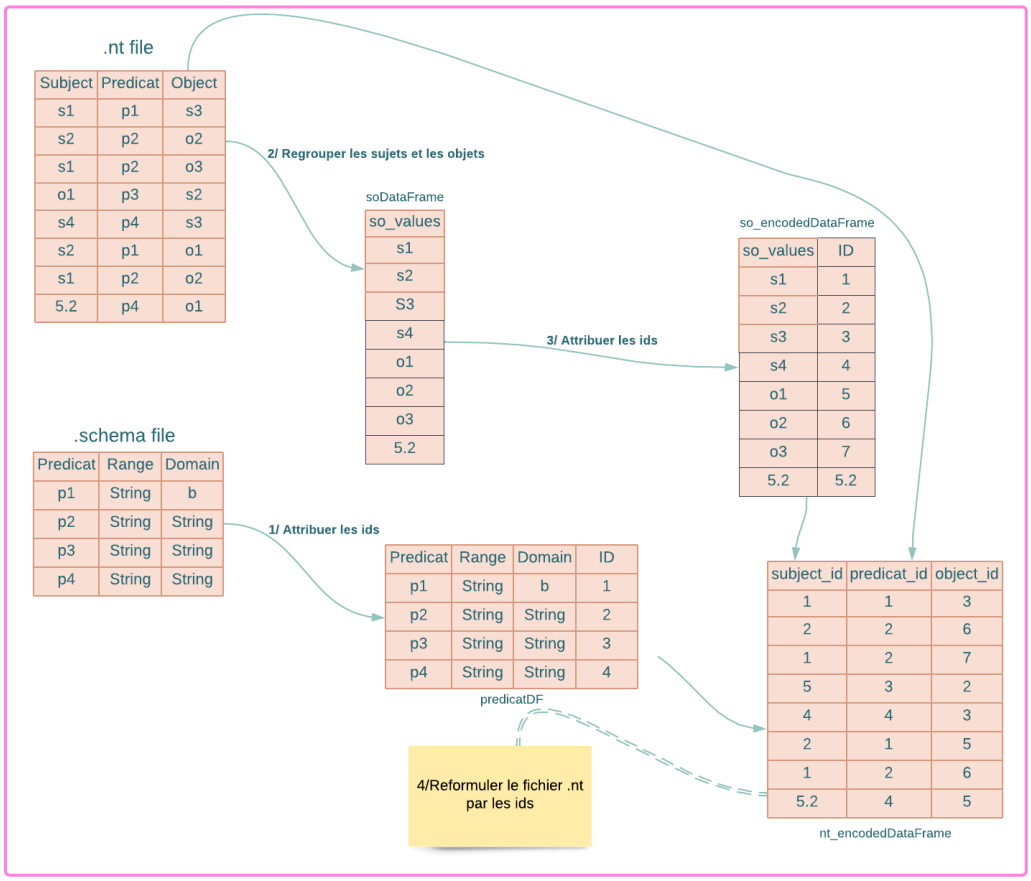


FIGURE 4.6 – Processus de la class EncodeFile

8. construire une dataset qui contient les têtes de chaque datastar et l'id de fragment a laquelle il appartient on l'appelle un segment sortant

2/ La figure 4.8 montre le processus de génération des fragments ops qui est lancée par **FragmentManager**

9. a partir du dataframe des triplets encodée on regroupe les triplets et les predicats qui ont le même objet dans une dataset ,de la meme maniere d'etape 5
10. a travers le dataset des triplets regrouper dans des datastar , on regroupe autre fois les datastar de la meme maniere d'etape 6.
11. réduire la taille des fragments pour ne dépasse pas une limite paramétrable ensuite attribuée les ids non effectués a les fragment spo générer
12. construire une dataset qui contient les têtes de chaque datastar et l'id de fragment a laquelle il appartient , on l'appelle un segment entrant.

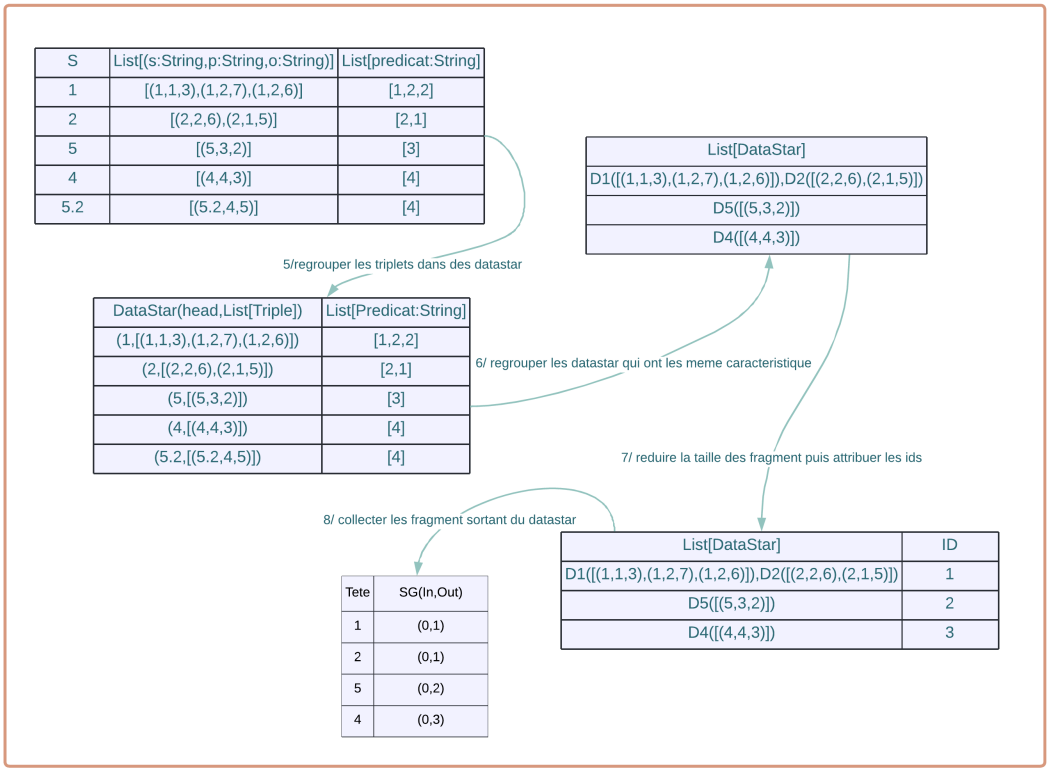


FIGURE 4.7 – Processus de spo fragmentation

4.4.3 Génération des fichiers

La figure 4.9 montre le processus de génération des fichiers des fragment générer

13. Fusioner les segments entrants et sortants des fragments spo et ops ensuite regrouper les segments qui ont la meme tete dans un objet Segment
14. générer le fichier .dic
15. générer les fichiers .data du segment spo
16. générer les fichiers .schema du segment spo
17. générer les fichiers .data du segment ops
18. générer les fichiers .schema du segment ops

4.4.4 Processus complet de Fragmentation

La figure 4.10 illustre un exemple du fonctionnement général de ce composant.

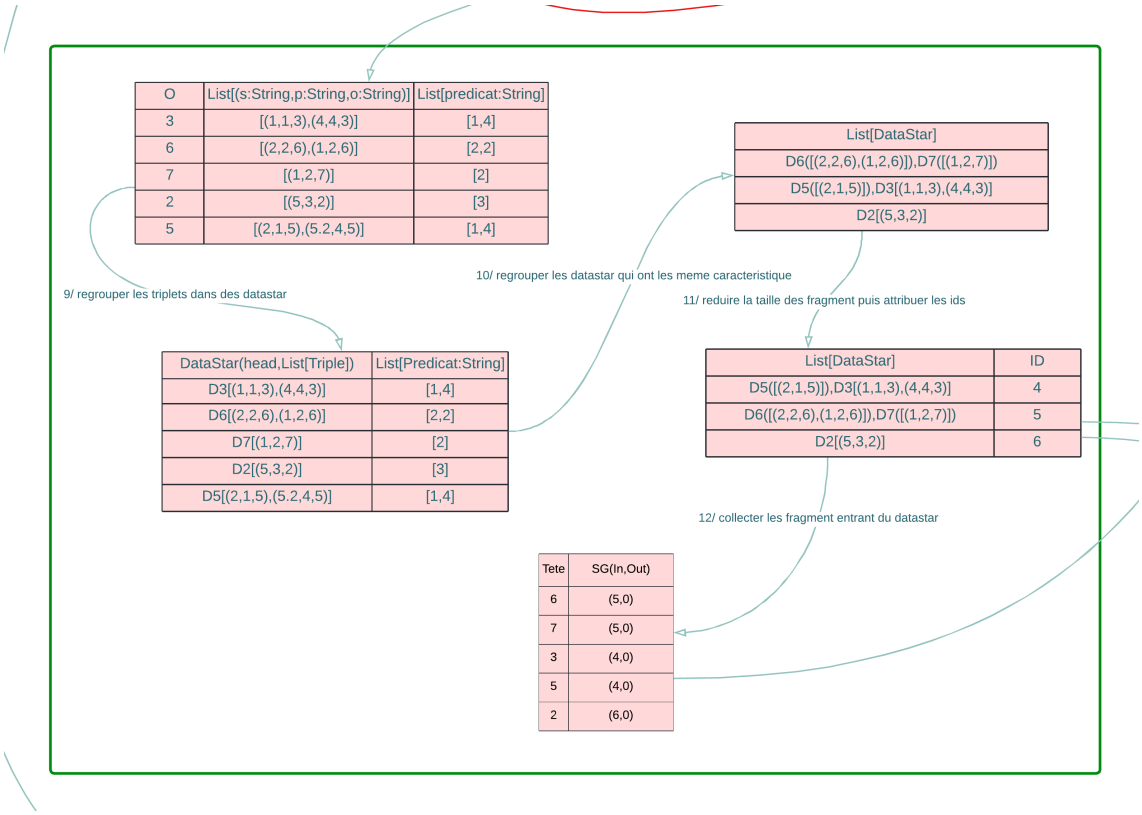


FIGURE 4.8 – Processus de ops fragmentation

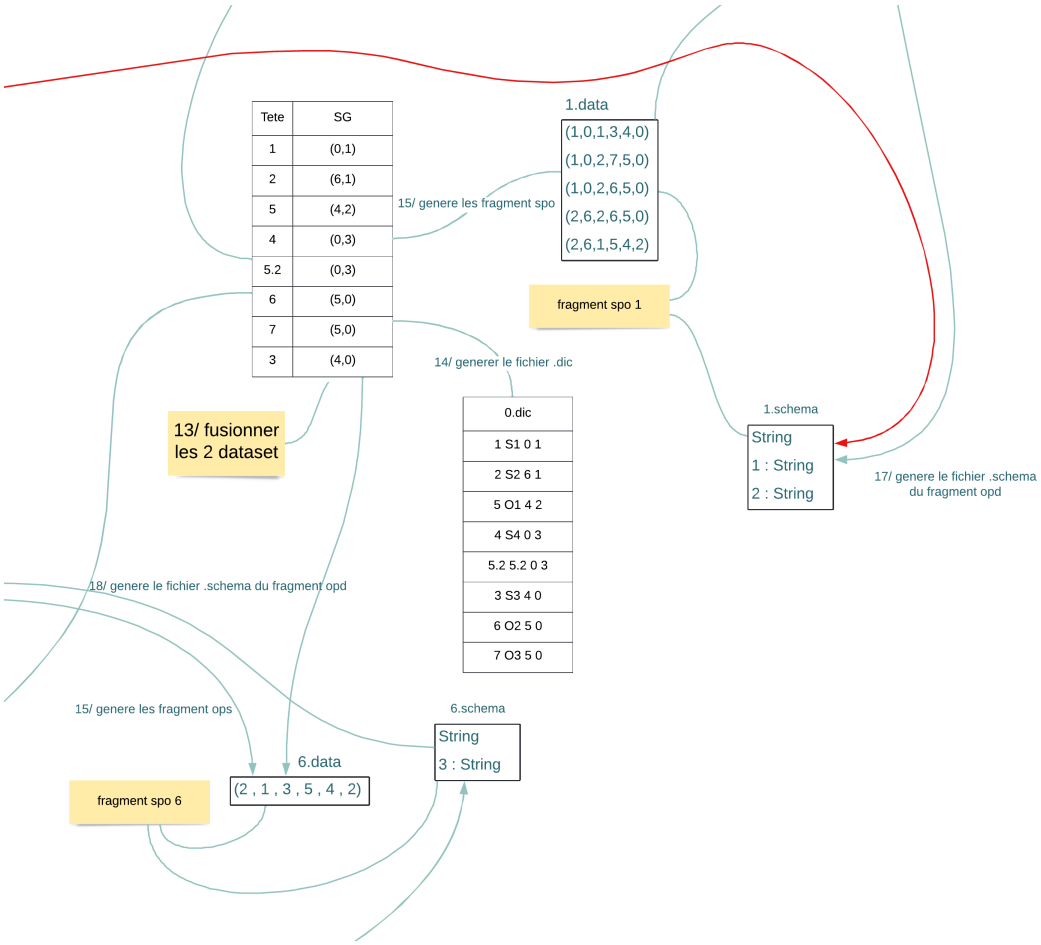
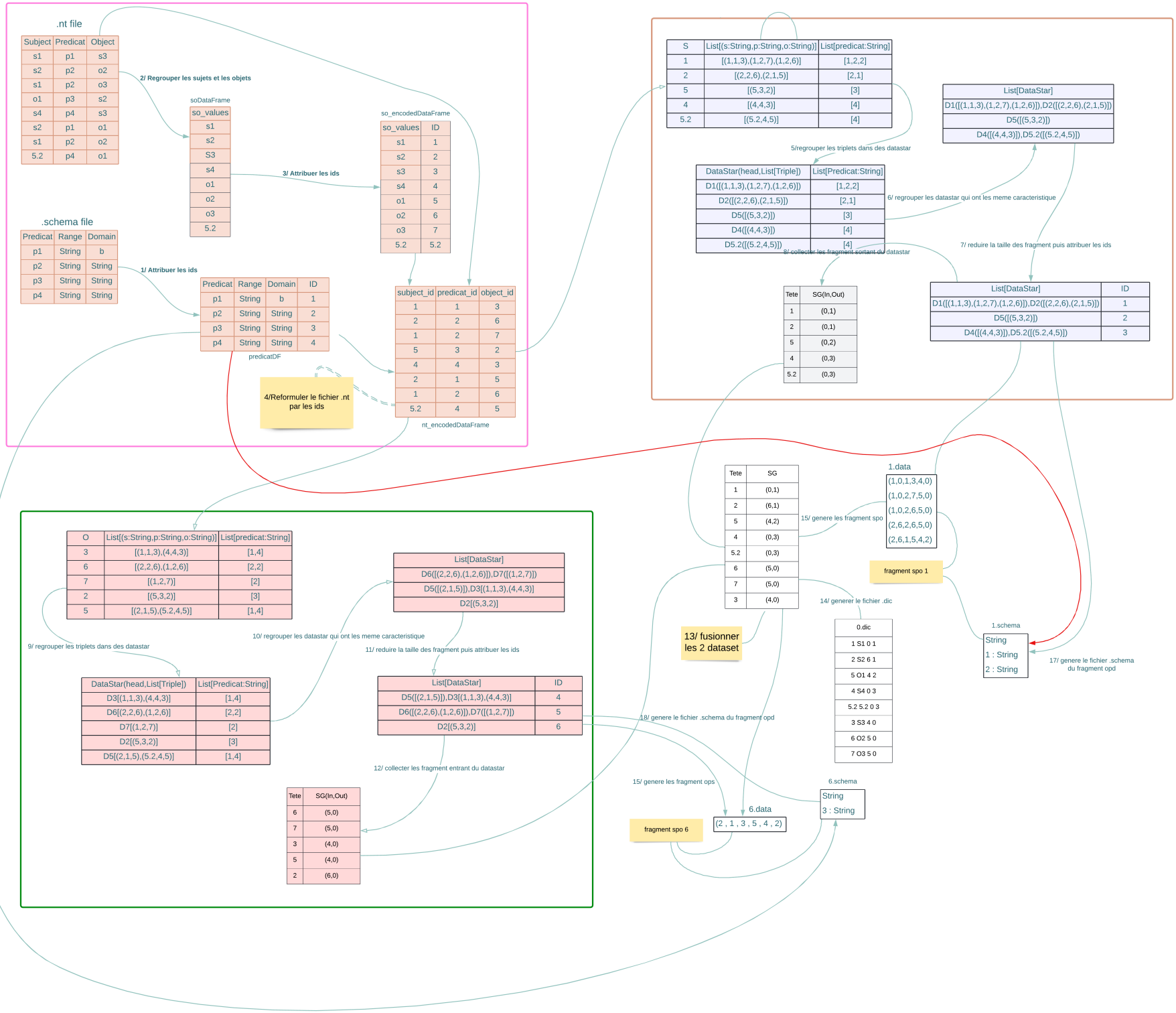


FIGURE 4.9 – Processus de generation des fichier


.nt file
Subject Predicat Object
s1 p1 s3
s2 p2 o2
s1 p2 o3
o1 p3 s2
s4 p4 s3
s2 p1 o1
s1 p2 o2
5.2 p4 o1
2/ Regrouper les sujets et les objets
soDataFrame
so_values
s1
s2
S3
s4
o1
o2
o3
5.2
3/ Attribuer les ids
so_encodedDataFrame
so_values ID
s1 1
s2 2
s3 3
s4 4
o1 5
o2 6
o3 7
5.2 5.2
.schema file
Predicat Range Domain
p1 String b
p2 String String
p3 String String
p4 String String
1/ Attribuer les ids
Predicat Range Domain ID
p1 String b 1
p2 String String 2
p3 String String 3
p4 String String 4
predicatDF
4/Reformuler le fichier .nt par les ids
subject_id predicat_id object_id
1 1 3
2 2 6
1 2 7
5 3 2
4 4 3
2 1 5
1 2 6
5.2 4 5
nt_encodedDataFrame
S List[(s:String,p:String,o:String)] List[predicat:String]
1 [(1,1,3),(1,2,7),(1,2,6)] [1,2,2]
2 [(2,2,6),(2,1,5)] [2,1]
5 [(5,3,2)] [3]
4 [(4,4,3)] [4]
5.2 [(5.2,4,5)] [4]
5/regrouper les triplets dans des datastar
DataStar(head,List[Triple]) List[Predicat:String]
D1([(1,1,3),(1,2,7),(1,2,6)]) [1,2,2]
D2([(2,2,6),(2,1,5)]) [2,1]
D5([(5,3,2)]) [3]
D4([(4,4,3)]) [4]
D5.2([(5.2,4,5)]) [4]
6/ regrouper les datastar qui ont les meme caracteristique
List[DataStar]
D1([(1,1,3),(1,2,7),(1,2,6)]),D2([(2,2,6),(2,1,5)])
D5([(5,3,2)])
D4([(4,4,3)]),D5.2([(5.2,4,5)])
8/ collecter les fragment sortant du datastar
7/ reduire la taille des fragment puis attribuer les ids
Tete SG(In,Out)
1 (0,1)
2 (0,1)
5 (0,2)
4 (0,3)
5.2 (0,3)
List[DataStar] ID
D1([(1,1,3),(1,2,7),(1,2,6)]),D2([(2,2,6),(2,1,5)]) 1
D5([(5,3,2)]) 2
D4([(4,4,3)]),D5.2([(5.2,4,5)]) 3
O List[(s:String,p:String,o:String)] List[predicat:String]
3 [(1,1,3),(4,4,3)] [1,4]
6 [(2,2,6),(1,2,6)] [2,2]
7 [(1,2,7)] [2]
2 [(5,3,2)] [3]
5 [(2,1,5),(5.2,4,5)] [1,4]
List[DataStar]
D6([(2,2,6),(1,2,6)]),D7([(1,2,7)])
D5([(2,1,5)]),D3[(1,1,3),(4,4,3)]
D2[(5,3,2)]
10/ regrouper les datastar qui ont les meme caracteristique
9/ regrouper les triplets dans des datastar
11/ reduire la taille des fragment puis attribuer les ids
DataStar(head,List[Triple]) List[Predicat:String]
D3[(1,1,3),(4,4,3)] [1,4]
D6[(2,2,6),(1,2,6)] [2,2]
D7[(1,2,7)] [2]
D2[(5,3,2)] [3]
D5[(2,1,5),(5.2,4,5)] [1,4]
List[DataStar] ID
D5([(2,1,5)]),D3[(1,1,3),(4,4,3)] 4
D6([(2,2,6),(1,2,6)]),D7([(1,2,7)]) 5
D2[(5,3,2)] 6
12/ collecter les fragment entrant du datastar
Tete SG(In,Out)
6 (5,0)
7 (5,0)
3 (4,0)
5 (4,0)
2 (6,0)
Tete SG
1 (0,1)
2 (6,1)
5 (4,2)
4 (0,3)
5.2 (0,3)
6 (5,0)
7 (5,0)
3 (4,0)
15/ genere les fragment spo
1.data
(1,0,1,3,4,0)
(1,0,2,7,5,0)
(1,0,2,6,5,0)
(2,6,2,6,5,0)
(2,6,1,5,4,2)
fragment spo 1
14/ generer le fichier .dic
0.dic
1 S1 0 1
2 S2 6 1
5 O1 4 2
4 S4 0 3
5.2 5.2 0 3
3 S3 4 0
6 O2 5 0
7 O3 5 0
13/ fusionner les 2 dataset
1.schema
String
1 : String
2 : String
17/ genere le fichier .schema du fragment opd
18/ genere le fichier .schema du fragment opd
15/ genere les fragment ops
6.schema
String
3 : String
6.data
(2 , 1 , 3 , 5 , 4 , 2)
fragment spo 6

4.5 Conclusion

Dans ce chapitre, nous avons spécifié les fonctionnalités de notre framework. Nous avons ainsi proposé une architecture basée sur plusieurs composants génériques, flexibles et extensibles. La réalisation du projet sera abordée dans le chapitre suivant.

Chapitre 5

Réalisation et résultats

5.1 Introduction

Ce chapitre est consacré à la démonstration de notre projet. Pour cela, nous allons d'abord commencer par présenter les deux technologies phares que avons utilisées, à savoir Scala et Spark. Nous passerons ensuite à la présentation des résultats qui démontrent la contribution de nos travaux.

5.2 Technologies utilisées : SCALA et PF

La programmation orientée objet POO a été le paradigme de programmation le plus fréquemment répandu au cours des deux dernières décennies. Plus récemment, le besoin croissant de gérer la concurrence et le traitement multicœur a encouragé les développeurs à revenir sur la *programmation fonctionnelle* PF (délaissée autrefois) et cela grâce aux concepts d'immuabilité et de pureté, qui permettent de partager les données en toute sécurité et de concevoir des applications modulaires et faciles à tester.

Nous sommes actuellement dans une période de transition où les développeurs sont familiarisés avec le paradigme POO mais voudraient se perfectionner à la PF qui devient aujourd'hui plus adaptée aux besoins actuels. Ces besoins nous mettent à la recherche d'un langage élégant et puissant qui combine les styles de codage POO et PF pour offrir le meilleur des deux mondes et, donc, contribuer à satisfaire un plus large éventail de domaines.

5.2.1 Présentation

Langage Scala

Scala est un langage de programmation multiparadigme moderne conçu pour exprimer des modèles de programmation courants de manière concise, élégante et sécurisée. Scala a été créé par Martin ODERSKY qui fit publier la première version en 2003. Son nom Scala signifie ‘’scalable language” (langage évolutif). Le langage est ainsi nommé car il a été conçu pour évoluer avec les demandes de ses utilisateurs.

Scala combine harmonieusement les fonctionnalités des langages orientés objet et celles des langages fonctionnels. Il est à typage statique et est basé sur la machine virtuelle Java, il permet donc de générer du Byte Code mais aussi du JavaScript pour le Web.

Scala 3 est une version majeure de ce langage. Elle est apparue dès fin 2012 en apportant des fonctionnalités fondamentales pour la PF. Nous nous basons sur cette version dans le cadre du développement de notre projet.

Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation de type déclaratif mettant l'accent sur les fonctions qui ne dépendent pas de l'état du programme. Le code fonctionnel est plus facile à tester et à réutiliser, plus simple à paralléliser et moins sujet aux bogues. Scala est un langage JVM populaire qui offre un support solide pour PF. Sa syntaxe familière et son interopérabilité transparente avec Java font de Scala un endroit idéal pour la programmation distribuée.

Contrairement aux modèles de programmation impérative qui mettent en avant les changements d'état, en PF cela devient impossible car la mutation des données ne peut être représentée par des évaluations de fonctions. Cette limitation offre une sécurité supplémentaire au code.

C'est la raison pour laquelle notre choix s'est porté depuis le début du projet sur le langage Scala pour l'implémentation du partitionnement au sein d'un système distribué comme RDF_QDAC. Cela nous a offert un certain confort en développement et nous a évité une multitude de tests de débogage au cours des phases de réalisations.

5.2.2 Avantages

Langage hybride

Scala parvient à combiner deux techniques de programmation qui sont considérablement différents : le style orienté objet avec le style fonctionnel. Lors de l'exécution de code sur la JVM, l'approche orientée objet peut être plus performante mais sujette aux erreurs. Une approche fonctionnelle peut être plus lisible et réutilisable mais pas aussi performante. Grâce à des données et des structures immuables, on peut garantir l'exactitude des programmes lorsqu'il s'agit de concurrence. Les données ne changent jamais, on peut donc les partager en toute sécurité. Il sera notamment plus facile à comprendre et à réutiliser car tous ses composants seront indépendants de facteurs externes.

Syntaxe concise

Le style de programmation de Scala est relativement concis, en particulier par rapport à d'autres langages comme Java. Avoir une syntaxe compacte peut augmenter à la fois la productivité et la lisibilité de votre programme.

Flexibilité

Vous pouvez atteindre le même objectif de plusieurs manières en passant progressivement d'une approche à une autre sans s'engager dans un style spécifique dès le début. Par exemple, on peut plonger dans le monde de la programmation fonctionnelle sans aucun engagement à long terme.

Concurrence

Grâce à son utilisation de structures de données immuables et d'un système de type expressif, la gestion de la concurrence est moins sujette aux erreurs que dans d'autres langages. En conséquence, les programmes Scala ont tendance à utiliser les ressources plus efficacement.

Big Data et Spark

Grâce aux fonctionnalités et aux optimisations de Scala au niveau de la compilation, la communauté a développé de nouveaux outils performants pour le traitement du Big

Data. Apache Spark est le plus populaire de ces outils. Grâce à l'évaluation paresseuse (lazy) de Scala, Spark peut effectuer des optimisations au moment de la compilation qui ont d'énormes impacts sur ses performances d'exécution.

5.2.3 Caractéristiques

Orienté Objet

Un langage de programmation orienté objet a une structure basée sur des classes et des sous-classes. Ils ont des comportements bien définis et échangent des informations par des méthodes.

Programmation Fonctionnelles

Les langages fonctionnels fondent toute leur structure sur des fonctions. Les fonctions jouent un rôle primordial en Scala. Cela nous permet d'utiliser la pureté fonctionnelle et ses nombreux avantages ainsi que de passer les fonctions comme paramètres ou valeurs de retour (fonctions d'ordre supérieur), permettant de créer des abstractions puissantes qui simplifient et suppriment la duplication de code d'une manière qui n'est généralement pas aussi facile à réaliser dans une approche orientée objet.

Système de typage robuste

Scala est un langage typé statiquement. Les types définissent la plage de valeurs acceptable pour le calcul. Ainsi, le compilateur peut vérifier au moment de la compilation que le code ne viole aucune contrainte, ce qui le rend plus fiable et moins sujet aux erreurs lors de l'exécution. Scala dispose d'un système d'inférence de type, on n'a donc pas à spécifier le type prévu pour chaque expression, ce qui rend le code moins verbeux. Les langages qui n'ont pas de système d'inférence de type nous obligent à fournir des types explicitement et ont tendance à être plus verbeux. Le système de types de Scala est également assez flexible, on peut réutiliser des types existants ou créer des types personnalisés.

Intégration avec Java

Scala est un langage basé sur la JVM. On peut donc facilement intégrer du code Java dans des programmes rédigés en Scala. La plupart des IDE prennent en charge Scala,

comme IntelliJ IDEA, et peuvent même convertir automatiquement du code Java en Scala.

5.2.4 Environnement

JVM

La JVM est la plate-forme standard pour exécuter des programmes Scala. Sun Microsystems l'a introduit en 1994, il y a plus de 25 ans. Depuis lors, l'industrie s'est largement appuyée sur elle. La communauté Java a également été extrêmement active et a produit une quantité impressionnante de bibliothèques. Grâce à son intégration avec Java, il est possible d'utiliser toutes ces bibliothèques avec Scala.

REPL

Scala REPL est un outil de ligne de commande pour l'évaluation et l'exécution de petits extraits de code. L'acronyme REPL signifie "Read-Evaluate-Print-Loop".

Semblable à Java Shell, Scala REPL est très utile pour les nouveaux arrivants et pour ceux qui souhaitent expérimenter de nouvelles bibliothèques ou fonctionnalités du langage.

Worksheet

Worksheet est comme une session REPL et bénéficient d'un support d'éditeur de première classe : complétion, hyperliens, erreurs interactives au fur et à mesure que vous tapez, etc. Les feuilles de travail utilisent l'extension .worksheet.sc.

STB

On peut créer des projets Scala à l'aide de plusieurs outils de génération tels que Maven, Ant et Gradle, mais sbt est l'outil de génération le plus courant dans la communauté. C'est un outil complexe mais puissant qui gère les dépendances et définit le cycle de construction du code. sbt a sa syntaxe et ses instructions prédéfinies. Il est également possible de charger des plugins, qui sont des programmes écrits en utilisant la syntaxe sbt, pour ajouter de nouvelles commandes ou modifier les comportements existants du processus de construction.

IDE

Scala peut être utilisé avec n'importe quel IDE supportant la JVM, même si IntelliJ de JetBrain est le plus adapté pour le moment. C'est ce dernier qui a été adopter pour la réalisation de notre projet.

5.2.5 Fonctionnalités générales

Déclaration et initialisation des variables

Les variables peuvent être déclarées via les mots clés **var** ou **val**. Le premier permet de modifier la variable après initialisation alors que la deuxième l'interdit.

Inférence de type

Dans Scala, il n'est pas obligatoire de spécifier explicitement le type de données des variables. Le compilateur peut identifier le type de variable en fonction de l'initialisation de la variable par le mécanisme d'inférence de type intégré.

Interpolation de chaîne

L'interpolation de chaîne est le processus de création d'une chaîne à partir des données. L'utilisateur peut incorporer les références de n'importe quelle variable directement dans les littéraux de chaîne traités et formater la chaîne.

5.2.6 Fonctionnalités liées à la POO

Scala intègre des fonctionnalités orientées-objet communes à la plupart des autres langages de ce type et implémente d'autres fonctionnalités particulières à son langage. Nous allons décrire ici ces fonctionnalités en mettant l'accent juste sur celles spécifiques au langage Scala, sans apporter trop de détails sur les fonctionnalités habituelles déjà connues par d'autres langages tel que Python et Java.

Classes et hiérarchie

Les définitions standards de classe et d'instance, comme dans tout autre langage orienté objet, s'appliquent également à Scala. Une classe peut être considérée comme

un modèle pour les instances. En d'autres termes, une instance est générée à partir d'une classe suivant la structure définie dans cette classe. La déclaration syntaxique des classes en Scala se fait grâce au mot clé **class**.

La classe **Any** est l'ancêtre ultime de toute autre classe. Elle engendre deux classes principales : **AnyVal** pour les classes de valeurs comme celles des types primitifs ; et la classe **AnyRef** qui est le parent de toutes les classes normales.

Scala intègre aussi des classes qui sont des "enfants ultimes" dont chaque autre classe en est le parent. Ainsi, la classe **Nothing** est enfant de toutes les classes engendrées par **AnyRef**. De même, la classe **Null** est descendante de toute les classes, sans exception.

Case Classe

Les "case class" ressemblent aux classes régulières avec des méthodes prédéfinies. Elles sont très utiles pour modéliser des données immuables. Leurs intérêts se montre avec l'utilisation des "patern matching". Le mot-clé **case class** est utilisé pour sa déclaration syntaxique.

Singleton

Les classes Scala ne peuvent pas avoir de variables et de méthodes statiques. Au lieu de cela, une classe Scala peut avoir un objet singleton ou un objet compagnon. Vous pouvez utiliser un objet singleton lorsqu'il n'y a besoin que d'une seule instance d'une classe. Un singleton est également une classe Scala mais il n'a qu'une seule instance (c'est-à-dire un objet). L'objet singleton ne peut pas être instancié. Il peut être créé à l'aide du mot-clé **object**. Les fonctions et les variables de l'objet singleton peuvent être appelées directement sans création d'objet.

Objet compagnon

Un objet portant le même nom qu'une classe est appelé un objet compagnon et la classe est appelée une classe compagnon.

Trait

En déclarant une classe avec le mot clé **trait** à la place de **class**, on peut spécifier la signature d'une méthode sans la définir. C'est l'équivalent des "interface" dans d'autres

langages POO comme Java.

Héritage et polymorphisme

Comme avec tout langage POO, la notion de polymorphisme est primordiale. Elle permet de modifier le comportement d'une classe en faisant appel aux méthodes virtuelles grâce aux appelles dynamiques. L'héritage se fait à l'aide du mot clé **extends** et **with**.

5.2.7 Fonctionnalités liées à PF

Immuabilité

L'immuabilité signifie que la valeur d'une variable ne peut pas être modifiée une fois qu'elle est déclarée. Le mot-clé "val" est utilisé pour déclarer des variables immuables, tandis que les variables mutables peuvent être déclarées à l'aide du mot-clé "var". L'immuabilité des données vous aide à contrôler la concurrence lors de la gestion des données.

Evalution Lazy et Eager

La fonction d'évaluation différée (Lazy) permet à l'utilisateur de différer l'exécution de n'importe quelle expression jusqu'à ce qu'elle soit nécessaire à l'aide du mot-clé **lazy**. Lorsque l'expression est déclarée avec le mot-clé **lazy**, elle ne sera exécutée que lorsqu'elle sera appelée explicitement. Les autres fonctions ordinaires sont appelées "eager" car l'évaluation se fait avant l'appelle effectif de la fonction.

Pattern Matching

Le processus de vérification d'un modèle par rapport à une valeur est appelé "patern matching". Une correspondance réussie renvoie une valeur associée au cas. Le patern matching est au cœur de la programmation Scala et il est intégré sur beaucoup de langages modernes. En scala, il est déclaré avec le mot clé **match**.

Collections

Les collections de Scala sont les conteneurs de certains éléments. Ils contiennent le nombre arbitraire d'éléments du même type ou de types différents en fonction de la nature

de la collection. Il existe deux types de collectes : Mutable et Immutable. Les collections les plus utilisées sont : ***List***, ***Set***, ***Map***, ***Tuple***, ***Option***.

Les collections peuvent être itérées à l'aide de la méthode "iterator". La méthode "iterator.hasNext" est utilisée pour déterminer si la collection contient d'autres éléments et l'itérateur. La méthode "next" est utilisée pour accéder aux éléments d'une collection. Les principales méthodes appliquées aux collections sont : ***filter***, ***map***, ***flatMap***, ***distinct***, ***foreach***.

Fonctions imbriquées

Scala permet à l'utilisateur de définir des fonctions à l'intérieur d'une fonction. Ceci est connu sous le nom de fonctions imbriquées et la fonction interne est appelée fonction locale.

Fonctions anonymes

Une fonction anonyme est une fonction qui n'est pas définie avec un nom et qui est créée pour un usage unique. Ces fonctions n'ont pas de nom mais fonctionnent comme une fonction. Des fonctions anonymes peuvent être créées en utilisant le symbole **=>** et par "_" . Elles sont également représentées sous forme de fonctions lambda.

Fonctions d'ordre supérieur

Les fonctions qui acceptent d'autres fonctions comme paramètre ou renvoient une fonction sont appelées fonctions d'ordre supérieur.

Fonction Currying

Le processus de transformation d'une fonction qui prend plusieurs arguments en tant que paramètres en une fonction avec un seul argument en tant que paramètre s'appelle "currying". Le "currying" d'une fonction est principalement utilisé pour créer une fonction partiellement appliquée. Les fonctions partiellement appliquées sont utilisées pour réutiliser une invocation de fonction ou pour conserver certains des paramètres. Dans de tels cas, le nombre de paramètres doit être regroupé sous forme de listes de paramètres. Une seule fonction peut avoir plusieurs listes de paramètres.

5.3 Technologies utilisées : Spark

Apache Spark est un projet de traitement de données distribué qui a été lancé en 2009 par Matei Zaharia à l'Université de Californie. Ce projet compte aujourd'hui plus de 400 contributeurs d'entreprises telles que Facebook, Yahoo !, Intel, Netflix, Databrick, et autres.

Open source et gratuit, le code Spark est écrit en Scala, qui est construit au-dessus de la machine virtuelle Java (JVM) et de l'environnement d'exécution Java. Cela fait de Spark une application multiplateforme capable de fonctionner sur Windows ainsi que sur Linux.

Spark a été fondé comme une alternative à l'utilisation traditionnelle de MapReduce sur Hadoop, qui était jugé inadapté aux requêtes interactives ou aux applications en temps réel à faible latence. Un inconvénient majeur de l'implémentation MapReduce de Hadoop était sa persistance de données intermédiaires sur le disque entre les phases de traitement Map et Reduce.

Spark implémente une structure distribuée, tolérante aux pannes et en mémoire appelée Resilient Distributed Dataset (RDD). Spark maximise l'utilisation de la mémoire sur plusieurs machines, améliorant les performances globales par ordre de grandeur. La parallélisation est invisible à l'utilisateur afin de lui permettre de se concentrer sur le traitement des données

5.3.1 Interfaces

Comme mentionné précédemment, Spark lui-même est écrit en Scala. Il s'exécute sur des machines virtuelles Java (JVM). Spark fournit une prise en charge native des interfaces de programmation, notamment Scala, Python (avec PySpark), Java, R. Mais le langage de prédilection de Spark est bien évidement Scala.

5.3.2 Façons d'utiliser Spark

Les programmes Spark peuvent être exécutés de manière interactive ou soumis sous forme de travaux par lots, y compris des travaux en mini-lots et en micro-lots.

On peut aussi lancer les programmes Spark en **mode local** sur une seule machine. Dans ce mode, tout les composants Spark sont exécutés sur la même JVM. L'intérêt de ce

mode est de procéder au débogage des programmes et de faire les tests unitaires.

Concernant la gestion de ressource, on peut déployer Spark pour fonctionner avec YARN ou Mesos, comme on peut aussi outrepasser tout gestionnaire de ressource tiers et utiliser le gestionnaire de cluster intégré avec Spark (**Standalone** mode).

5.3.3 Architecture

Les composants d'une application Spark sont le pilote (driver), le maître, le gestionnaire de cluster et le ou les exécuteurs, qui s'exécutent sur des nœuds de travail (workers).

Le **driver** est le processus avec lequel le client interagit lors du lancement d'une application Spark, soit via le shells ou via le script spark-submit. Le driver est responsable de la création du contexte Spark et de la planification d'une application en créant un DAG (graphe orienté acyclique) composé de tâches (tasks) et d'étapes (stages).

Le driver communique avec un **gestionnaire de cluster** pour allouer des ressources d'exécution d'application (containers) sur lesquelles les exécuteurs s'exécuteront. Les exécuteurs sont spécifiques à une application donnée et restent actifs jusqu'à l'achèvement de l'application. Les exécuteurs peuvent exécuter de nombreuses tâches dans une application.

5.3.4 MapReduce

MapReduce comprend deux phases de traitement implémentées par le développeur, la phase **Map** et la phase **Reduce**, ainsi qu'une phase Shuffle, qui est implémentée par le framework (dans Hadoop, elle est implémentée par le framework de traitement MapReduce dans une phase appelée phase **Shuffle** and Sort).

5.3.5 RDD

La structure de données fondamentale de Spark est RDD, une collection d'éléments tolérants aux pannes qui peuvent être utilisés en parallèle. Il est résilient car il a une tolérance aux pannes intégrée. Si quelque chose ne va pas, nous pouvons le reconstruire à partir de la source (lignée). Les données sont distribuées en mémoire sur les noeuds worker. Un ensemble de données représente les enregistrements des données.

La plupart des programmes Spark consistent à créer de nouveaux RDD en effectuant des opérations sur les RDD existants.

Caractéristiques

Calcul en mémoire : RDD stocke les résultats intermédiaires dans la mémoire distribuée.

Évaluations paresseuses (lazy evaluation) : dans Spark, toutes les transformations sont paresseuses. Paresseux signifie qu'ils ne calculent pas leurs résultats tant que cela n'est pas nécessaire.

Tolérance aux pannes : RDD reconstruit automatiquement les données perdues à partir de la source en cas d'échec à l'aide du lignage (lineage). Chaque RDD se souvient de la façon dont il a été créé à partir d'autres ensembles de données.

Immuabilité : les RDD sont immuables dans le sens où les données ne peuvent pas être modifiées sur place. Les RDD ne peuvent être modifiés qu'en appliquant des opérations RDD, à savoir la transformation et l'action.

Partitionnement : les données sont divisées en partitions et réparties sur le cluster et exploitées en parallèle.

Transformations

La création d'un nouveau RDD à partir d'un RDD existant est appelée transformation. La chaîne de transformation peut être effectuée une fois que les données sont chargées en mémoire. Un exemple consiste à extraire des champs spécifiques et à filtrer uniquement certains enregistrements.

Actions

Spark ne traite pas les données immédiatement. Il traite les données uniquement lorsqu'une action est appelée. Des exemples d'actions sont les fonctions d'agrégation.

DAG (Graphe orienté acyclic)

DAG dans Apache Spark est un ensemble de sommets et d'arêtes. Les sommets représentent les RDD et les arêtes représentent l'opération à appliquer sur RDD. Chaque

arête du DAG est dirigée d'un sommet à un autre. Spark crée le DAG lorsqu'une action est appelée.

Gestion des pannes

Spark maintient la lignée de chaque RDD (c'est-à-dire le RDD précédent dont il dépend) qui est créé dans DAG pour gérer la tolérance aux pannes. Lorsqu'un nœud tombe en panne, Spark Cluster Manager affecte un autre nœud pour continuer le traitement. Spark le fait en reconstruisant la série d'opérations qu'il doit calculer sur cette partition à partir de la source.

5.3.6 Dataset et Dataframe

Dataset est une nouvelle interface ajoutée dans Spark SQL qui offre tous les avantages RDD avec le moteur d'exécution Spark SQL optimisé. Il est défini comme la distribution de la collecte de données. L'API Dataset est disponible pour Scala et Java. Il n'est pas disponible pour Python, car la nature dynamique de Python offre les avantages de l'API Dataset en tant que fonctionnalité intégrée.

DataFrame est un ensemble de données organisé en colonnes nommées, ce qui facilite l'interrogation. Conceptuellement, le DataFrame équivaut à une table dans n'importe quelle base de données relationnelle. Les DataFrames peuvent être créés à partir de diverses sources telles que des fichiers de données structurés, des sources de données relationnelles externes ou des RDD existants.

5.4 Tests unitaires

Les tests sont très importants pour la réalisation d'une application fiable. Pour cela, nous avons pris la peine de tester chaque fonctionnalité après l'avoir finie. Nous avons aussi procédé à ces tests pour chaque étape d'avancement du projet et par différents jeux de tests.

5.5 Conception

Notre solution est le résultat du rassemblement de plusieurs composants génériques, flexibles, facilement réintégrables et extensibles, dont chacun satisfait une certaine fonctionnalité. Le diagramme de composant de la figure 5.1 montre l'architecture de ce composant

5.5.1 Chargement et encodage des données d'entrée

C'est le principal composant de notre framework. Il permet de charger les données non structurées entrantes et les rendre encodées sous le format des identifiants. Il s'agit d'encapsuler dans un dataframe les prédicats encodées et les exporter dans un fichier predicat.txt ainsi qu'encoder les triplets des fichier .nt dans un dataframe afin de générer une dataframe qui comporte les triplets encodées par les ids. Afin d'y parvenir, on a crée une classe nommée **EncodeFiles** pour gérer l'encodage des données sources.

5.5.2 Génération des fragments

Elle prend la part la plus importante dans le processus de fragmentation et donc le coeur de notre framework. Il s'agit de générer les fragments pour rédiger les fichiers de données. Dans cette architecture le **Triple** représente la structure d'une ligne de données , et **DataStar** représente les triplets qui ont la même tète , Comme la fragmentation est pareil pour les fragment spo et ops, nous l'avons généralisée dans **FragmentBuilder**. Il suffit de lui fournir juste le regroupement approprié du fragment par FragmentManager et donc ce dernier a pour rôle le gestionnaire du fragment spo et ops.

5.5.3 Génération d'un dictionnaire

Il s'agit du composant gestionnaire de la fusion des segments spo , ops et la création d'un dictionnaire, a travers **DictionaryGenerator** une dataset qui contient les objet **Segment** et la tête appropriée représente un dictionnaire est généré.

5.5.4 Création des fichiers

C'est la dernière étape du partitionnement , il s'agit de récupérer les fragments et les segments afin de générer des fichiers .data et .schema approprie , il suffit juste de fournir les datasets a **FileGenerator** pour ce but .

La figure 5.1 montre le diagramme de notre composant .

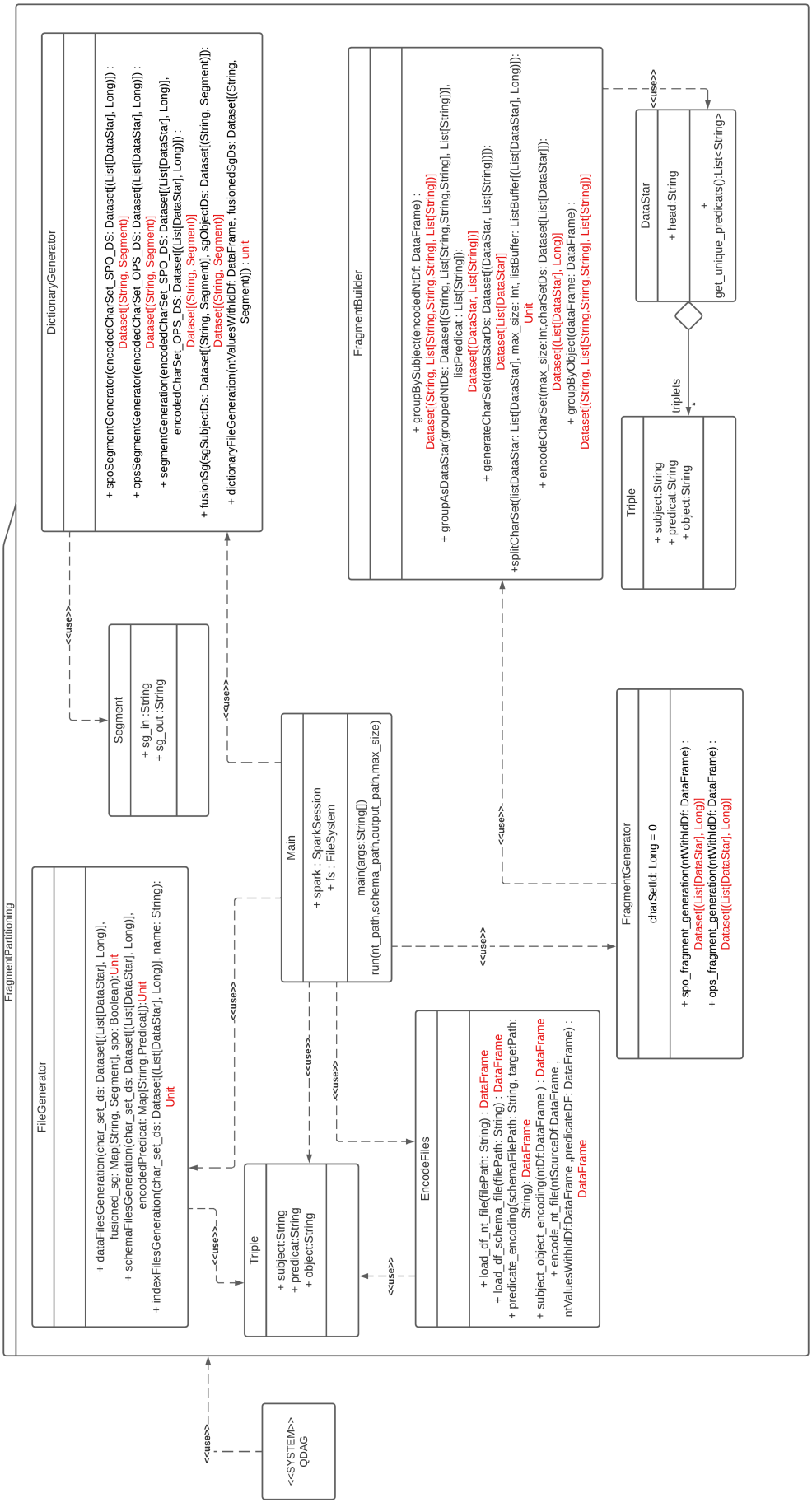


FIGURE 5.1 – Le diagramme de composant

5.6 Résultats

Dans cette section, nous allons découvrir la performance de notre framework . Nous allons déployer notre solution dans SPARK cluster et on vas utiliser HDFS YARN pour stockage des données générer . Nous allons utiliser une machine master, 4 machines workers

5.6.1 Le deploit du framework

D'abord, on génère une application jar via intellij ensuite on dépolit l'application dans la machine master via la commande SCP (Figure 5.2)

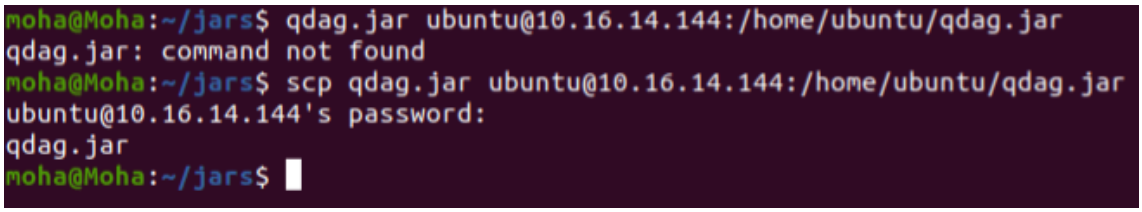


FIGURE 5.2 – L'execution de la commande SCP

5.6.2 Le lancement du framework

Avec la commande spark-submit nous allons lancer notre application sur le cluster. Pour cela on doit fournir les paramètres nécessaires pour qu'il se lance correctement, voici les configuration a fournir

1/ on doit fournir le package de la class qui a la méthode main

–class com.fragment.spark.app.MainApp

2/ le mode de déploiement

–deploy-mode cluster

3/ le chemin du fichier .jar dans la machine local

4/ les paramètres personnaliser qui doivent capter par la méthode main

dans notre cas on doit fournir 4 paramètres pour qu'il se lance , la figure 5.3 montre la commande utilisée

```
ubuntu@master:~$ spark-submit \
> --master yarn --deploy-mode cluster \
> /home/ubuntu/qdagPart.jar \
> /user/ubuntu/rawdata/watdiv100k/watdiv_100k.nt \
> /user/ubuntu/rawdata/watdiv100k/watdiv_100k.schema \
> /user/ubuntu/output \
> 40
```

FIGURE 5.3 – Lancement d'application sur le cluster

5.6.3 Résultat d'exécution

Nous avons fournis 2 fichier en paramètre : le fichier WATDIV_100K.NT comporte 100 milles ligne de données, et le fichier WATDIV_100K.SCHEMA notre framework a générer les fragments sous forme de fichiers dans 57 S (Figure 5.4) en mode cluster. Nous avons lancé ensuite notre framework en mode local mais cette fois ci en fournissant le fichier WATDIV1M.NT qui comporte 1 million de lignes, le temps d'exécution pour générer les fragment était 206 S (Figure 5.5)

User:	ubuntu
Name:	qdag.jar
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Fri Sep 23 20:29:40 +0000 2022
Launched:	Fri Sep 23 20:29:40 +0000 2022
Finished:	Fri Sep 23 20:30:38 +0000 2022
Elapsed:	57sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

FIGURE 5.4 – Résultat d'exécution du fichier 100k en cluster

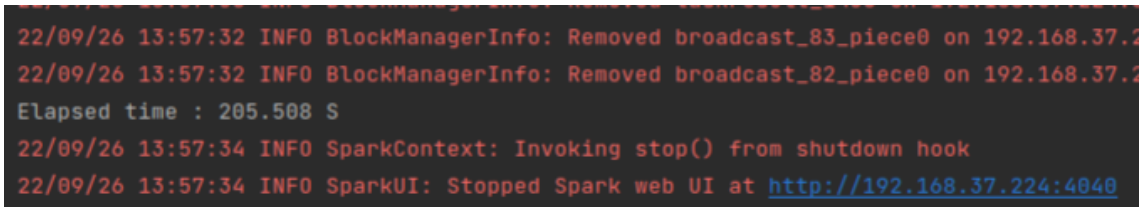


FIGURE 5.5 – Résultat d'exécution du fichier 1M en local

5.7 Diagramme de Gantt

La figure(5.6) montre la gestion de notre projet depuis le début vers la fin via le diagramme de gantt.

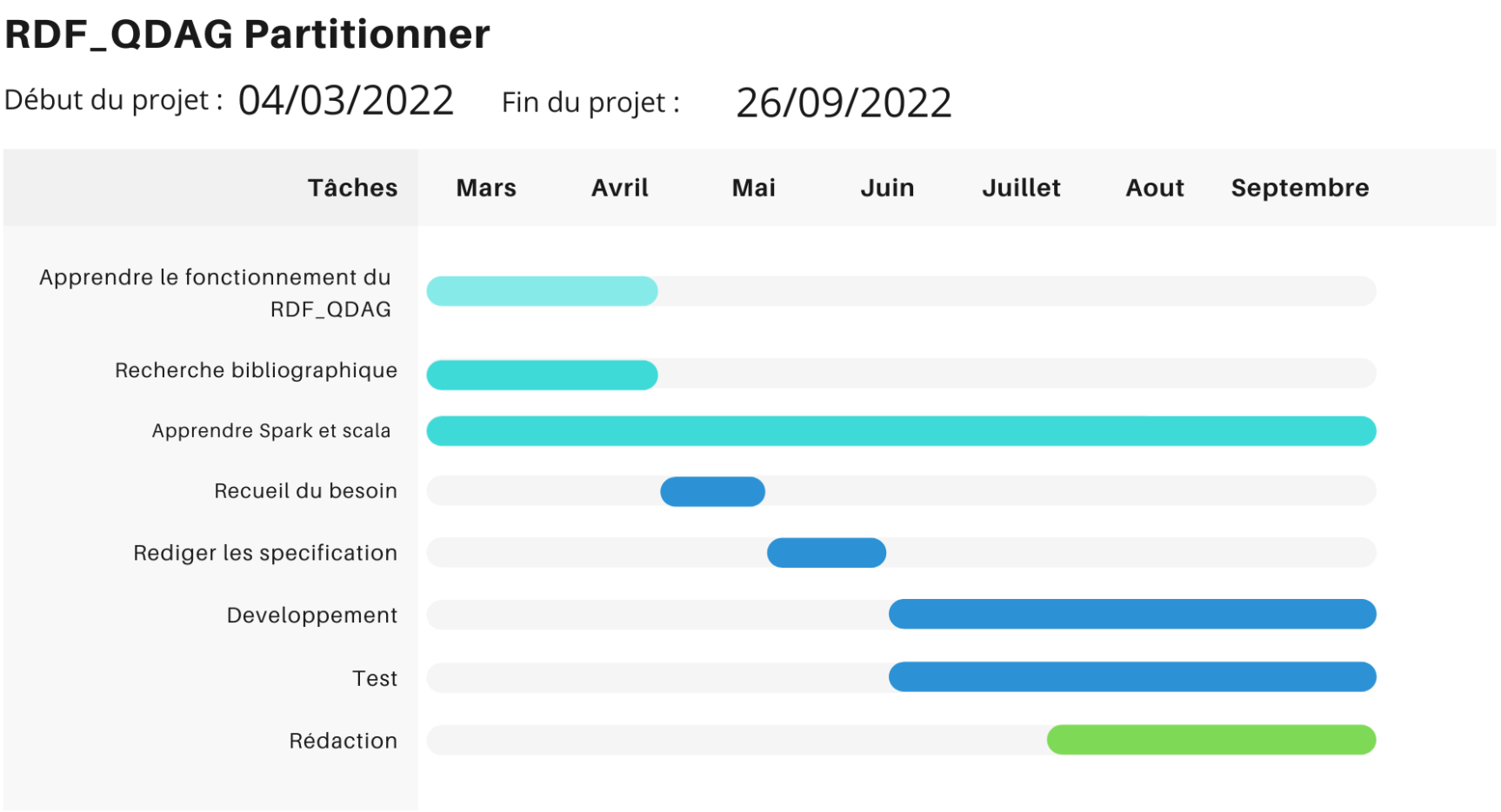


FIGURE 5.6 – Le diagramme de gantt

5.8 Conclusion

Dans ce chapitre, nous avons présenté les différent technologies utilisées afin d'accomplir notre travail. Nous avons aussi présenté quelques captures d'écran, montrant le fonctionnement global de notre application. Le chapitre suivant sera consacré à la description des outils de collaboration ainsi que notre planning de projet.

Chapitre 6

Gestion de projet

6.1 Introduction

Afin de mener à bien notre travail tout au long des différentes phases de conception et de réalisation, nous avons eu recours à certains outils qui facilitent la gestion des projets. Nous allons présenter dans ce chapitre ces produits ainsi que leurs principales fonctionnalités.

6.2 Outils collaboratifs

6.2.1 GitKraken

Généralement, lorsqu'un développeur travaille dans un environnement d'équipe, de nombreux développeurs peuvent se focaliser sur la même application. Cela rend tout à fait possible, et même probable, qu'un conflit de fusion se produise lors de l'extraction des derniers "commits" dans notre branche de travail. La plupart des développeurs redoutent d'avoir à parcourir des fichiers pour trouver les conflits, puis passent des heures à essayer de déterminer ce qu'il faut conserver et ce qu'il faut supprimer. Une solution de contrôle de version s'impose alors d'elle-même.

Git est l'outil phare pour le contrôle de version. Même si le choix était évident en ce qui concerne l'adoption de Git pour notre projet, celui-ci est disponible par défaut en ligne de commande. On avait besoin d'un client plus visuel avec une interface graphique conviviale pour faciliter notre travail.

GitKraken offre en ce moment les meilleures fonctionnalités pour utiliser Git d'une façon intuitive et faciliter le développement dans un environnement d'équipe. Gratuit et open source, il est adapté pour toute les tailles de projet.

6.2.2 Trello

Trello est une application de gestion de travail collaboratif conçue pour suivre les projets d'équipe, mettre en évidence les tâches en cours, montrer à qui elles sont affectées et détailler les progrès vers l'achèvement. Il est basé sur la gestion des cartes : chaque carte peut contenir un large éventail d'informations sur les tâches, notamment une description textuelle, des pièces jointes, etc. Les cartes individuelles dans les listes contiennent des informations sur une tâche spécifique et peuvent être déplacées d'une liste à l'autre selon les besoins.

Cet outil nous a été très bénéfique dans la gestion du planning dès le début du projet et jusqu'à son achèvement, rendant la collaboration plus fluide et la coordination des tâches plus explicite.

6.2.3 WebEx

Etant donné que l'équipe de travail avec laquelle nous collaborions se situait en France, la totalité de nos échanges se faisait à distance. Nous avons choisi l'outil WebEx de chez Cisco pour assurer nos réunions de suivi hebdomadaires en visioconférence et pour nos communications exceptionnelles.

WebEx offre une panoplie complète d'outils pour la collaboration en équipe telles que les réunions en ligne, la messagerie d'équipe et le partage de fichiers. Il existe en version Web, Mobile et Desktop. Durant notre projet, WebEx nous a apporter une grande valeur ajoutée par rapport à d'autres outils non spécialisés tel que WhatsApp ou Viber. Par exemple, Grâce à Webex Meetings, on créait des salles personnelles que d'autres utilisateurs peuvent personnaliser, programmer des réunions ou laisser ouvertes pour des réunions ad hoc.

6.2.4 Google Drive

Durant la phase de recherche bibliographique, nous utilisions massivement Google Drive pour le partage de documentation. Ce service cloud nous permet de stocker, synchroniser et partager une quantité considérable de données avant d'être obligé de passer à la version payante.

6.2.5 OverLeaf

La rédaction du présent manuscrit ainsi que la génération du document final en PDF sont réalisées par l'outil Overleaf. C'est un outil d'écriture et de publication collaboratif LaTeX. Il est accessible en ligne et rend l'ensemble du processus de rédaction, d'édition et de publication de documents scientifiques beaucoup plus rapide et plus facile. Cette application Web nous a aidé à lever la complexité de rédaction en Latex tout en facilitant le processus de collaboration par plusieurs fonctionnalités très utiles.

6.3 Conclusion

Véritable levier de croissance, le travail collaboratif prend de plus en plus d'ampleur dans le quotidien professionnel et éducatif, et les outils sont là pour venir le favoriser. Même pour un projet de petite taille comme le nôtre, ces outils nous ont été d'un grand secours durant notre travail dès lors qu'un groupe de deux personnes ou plus se penchait sur la même tâche.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion

Notre projet de fin d'étude s'inscrit dans le cadre d'une collaboration entre le laboratoire LRIT de l'université de Tlemcen et le laboratoire LIAS de l'Université de Poitiers. Il a pour objectif de proposer un cadre théorique et pratique pour la gestion des données RDF. Nous avons alors intégré l'équipe de développement du Triplestore RDF_QDAG. Nous nous sommes intéressés par l'outil de fragmentation qui posait problème en termes de passage à l'échelle. Nous avons proposé une nouvelle approche basée sur le framework Spark. Nous avons dû revisiter les différentes étapes de fragmentation afin de respecter les contraintes imposées par Spark. Nous avons mené une étude expérimentale, en utilisant un vrai cluster Spark, qui a confirmé le potentiel de notre approche.

7.2 Perspectives

Ce travail ouvre plusieurs perspectives permettant d'améliorer le système RDF_QDAG. En effet, ce système est par nature distribuable du fait qu'il s'appuie sur un processus de fragmentation physique de données.

La première piste que nous avons retenu consiste à améliorer le processus de fragmentation pour rendre la gestion des fragments dynamiques en changeant la structure et l'emplacement des fragments en fonction des jeux de requêtes. Ce problème étant NP-difficile, il serait donc intéressant de considérer des techniques d'apprentissage automatique non-supervisées (e.g., clustering, factorisation de matrices) afin de prendre en charge

la réplication de données et recommander de nouvelles stratégies de fragmentation.

Bibliographie

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. Rdfa in xhtml : Syntax and processing, Oct 2008.

[3] Medha Atre, Jagannathan Srinivasan, and James A. Hendler. Bitmat : A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC2008), Karlsruhe, Germany, October 28,*, 2008.

[4] David Beckett. The design and implementation of the redland RDF application framework. *Computer Networks*, 39(5) :577–588, 2002.

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Rdf 1.1 turtle, Feb 2014.

[6] Tim Berners-Lee and Dan Connolly. Notation3 (n3) : A readable rdf syntax, Mar 2011.

[7] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame : A generic architecture for storing and querying RDF and RDF schema. In *The Semantic Web - ISWC,First International Semantic Web Conference, Italy, June 9-12*, pages 54–68, 2002.

[8] Min Cai and Martin R. Frank. Rdfpeers : a scalable distributed RDF repository based on a structured peer-to-peer network. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference*

on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, pages 650–657. ACM, 2004.

[9] Xi Chen, Huajun Chen, Ningyu Zhang, and Songyang Zhang. Sparkrdf : Elastic discreted RDF graph processing engine with distributed memory. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6-9, 2015 - Volume I*, pages 292–300. IEEE Computer Society, 2015.

[10] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based RDF querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1216–1227, 2005.

[11] Benjamin Djahandideh, François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. Cliquesquare in action : Flat plans for massively parallel RDF queries. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1432–1435. IEEE Computer Society, 2015.

[12] Orri Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1) :3–8, 2012.

[13] George H. L. Fletcher and Peter W. Beck. Scalable indexing of RDF graphs for efficient join processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 1513–1516, 2009.

[14] Jorge Galicia. *Revisiting Data Partitioning for Scalable RDF Graph Processing. (Revisiter le partitionnement des données pour le traitement scalable des graphes RDF)*. PhD thesis, École Nationale Supérieure de Mécanique et d'Aérotechique, Poitiers, France, 2021.

[15] Jose Manuel Gomez-Perez, Jeff Z Pan, Guido Vetere, and Honghan Wu. Enterprise knowledge graph : An introduction. In *Exploiting linked data and knowledge graphs in large organisations*, pages 1–14. Springer, 2017.

[16] Olaf Görlitz and Steffen Staab. SPLENDID : SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, 2011.

[17] Stephen Harris and Nicholas Gibbins. 3store : Efficient bulk RDF storage. In *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, 2003.

[18] Andreas Harth and Stefan Decker. Optimized index structures for querying RDF from the web. In *Third Latin American Web Congress (LA-Web 2005), 1 October - 2 November 2005, Buenos Aires, Argentina*, pages 71–80, 2005.

[19] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2 : A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, pages 211–224, 2007.

[20] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.*, 4(11) :1123–1134, 2011.

[21] Maciej Janik and Krys Kochut. BRAHMS : A workbench RDF store and high performance memory system for semantic association discovery. In *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC, Galway, Ireland, November 6-10, 2005, Proceedings*, pages 431–445, 2005.

[22] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, Mohand-Saïd Hacid, and Emmanuel Coquery. Combining graph exploration and fragmentation for scalable rdf query processing. *Information Systems Frontiers*, 23(1) :165–183, 2021.

[23] Dave Kolas, Ian Emmons, and Mike Dean. Efficient linked-list rdf indexing in parliament. 10 2009.

[24] Akiyoshi Matono, Said Mirza Pahlevi, and Isao Kojima. Rdfcube : A p2p-based three-dimensional index for structural joins on distributed triple stores. In *Databases, Information Systems, and Peer-to-Peer Computing, International Workshops,*

DBISP2P 2005/2006, Trondheim, Norway, August 28-29, 2005, Seoul, Korea, September 11, 2006, Revised Selected Papers, pages 323–330, 2006.

[25] Brian McBride. Jena : A semantic web toolkit. *IEEE Internet computing*, (6) :55–59, 2002.

[26] James P. McGlothlin and Latifur R. Khan. RDFKB : efficient support for RDF inference queries and knowledge management. In Bipin C. Desai, Domenico Saccà, and Sergio Greco, editors, *International Database Engineering and Applications Symposium (IDEAS 2009), September 16-18, 2009, Cetraro, Calabria, Italy*, ACM International Conference Proceeding Series, pages 259–266. ACM, 2009.

[27] Sameh K Mohamed, Aayah Nounu, and Vít Nováček. Biological applications of knowledge graph embedding models. *Briefings in bioinformatics*, 22(2) :1679–1693, 2021.

[28] Thomas Neumann and Guido Moerkotte. Characteristic sets : Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE)*, pages 984–994, 2011.

[29] Thomas Neumann and Gerhard Weikum. Rdf-3x : a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1) :647–659, 2008.

[30] Kurt Rohloff and Richard E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In Tevfik Kosar, editor, *DIDC'11, Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, San Jose, CA, USA, June 8, 2011*, pages 35–44. ACM, 2011.

[31] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF : RDF querying with SPARQL on spark. *Proc. VLDB Endow.*, 9(10) :804–815, 2016.

[32] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx : Optimization techniques for federated query processing on linked data. In *International semantic web conference*, pages 601–616. Springer, 2011.

[33] Thanh Tran, GUNTER Ladwig, and Sebastian Rudolph. Istore : efficient rdf data management using structure indexes for general graph structured data. *Institute AIFB, Karlsruhe Institute of Technology*, 2009.

[34] George Tsatsanifos, Dimitris Sacharidis, and Timos K. Sellis. On enhancing scalability for distributed RDF/S stores. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 141–152. ACM, 2011.

[35] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. Heuristics-based query optimisation for SPARQL. In *15th EDBT, Berlin, Germany, March 27-30*, pages 324–335, 2012.

[36] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. GRIN : A graph based RDF index. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1465–1470, 2007.

[37] Mark Watson. *Scripting intelligence : Web 3.0 information gathering and processing*. Apress, 2009.

[38] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore : sextuple indexing for semantic web data management. *Proceedings of VLDB*, 1(1) :1008–1019, 2008.

[39] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8*, pages 131–150, 2003.

[40] David Wood. Kowari : A platform for semantic web storage and analysis. In *In XTech 2005 Conference*, pages 05–0402, 2005.

[41] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit : a fast and compact system for large scale RDF data. *PVLDB*, 6(7) :517–528, 2013.

[42] Mingxiong Zhao, Han Wang, Jin Guo, Di Liu, Cheng Xie, Qing Liu, and Zhibo Cheng. Construction of an industrial knowledge graph for unstructured chinese text learning. *Applied Sciences*, 9(13) :2720, 2019.

[43] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gstore : answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8) :482–493, 2011.

[44] Ishaq Zouaghi, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, and Taoufik Aguili. Gofast : Graph-based optimization for efficient and scalable query evaluation. *Information Systems*, 99 :101738, 2021.

Résumé

Dans le but de valider notre Master en Génie Logiciel, nous avons rejoint l'équipe du la- boratoire LIAS à Poiters pour leur proposer une solution de fragmentation pour leur sys- tème de gestion RDF. Nous avons utilisé pour cela une approche basée sur le framework Spark et nous avons réussi à implémenter un composant de partitionnement capable de garantir le passage à l'échelle tout en conservant la connectivité du graphe RDF. Mot-clé : Big data, Système de gestion de données, Transfert de données, Passage à l'échelle

Mots-clés : Big data, Système de gestion de données, Transfert de données, Passage à l'échelle,Fragmentation,Partitionnement.

Abstract

In order to validate our Master in Software Engineering, we joined the LIAS laboratory team in Poiters to offer them a fragmentation solution for their RDF management system. For this, we used an approach based on the Spark framework and we succeeded in implementing a partitioning component capable of guaranteeing scalability while maintaining the connectivity of the RDF graph.

Keywords : Big data, Data Management System, Data transfer, Scalability,Fragmentation,Partitioning

ملخص

من أجل التحقق من درجة الماجستير في هندسة البرمجيات ، انضممنا إلى فريق مختبر *LIAS* في *Poiters* لنقدم لهم حل تجزئة لنظام إدارة *RDF* الخاص بهم. لهذا ، استخدمنا نهجًا يعتمد على إطار عمل *Spark* ونجحنا في تنفيذ مكون تجزيء قادر على ضمان قابلية التوسع مع الحفاظ على اتصال الرسم البياني *RDF* .

الكلمات المفتاحية : البيانات الضخمة، نظام معالجة البيانات، تحويل البيانات، تغير المقاييس، تجزئة ،التقسيم