Indexation de graphes

Antonin Durey

Master IAGL 2017

Tuteur d'entreprise : Richard Marquis
Tuteur universitaire : Iovka Boneva

Confidentialité

La diffusion de ce document est limitée aux responsables du stage et au jury. Sa consultation par d'autres personnes, notamment d'autres étudiants, est interdite.

Remerciements

En premier lieu, je souhaite remercier mon tuteur entreprise, Richard Marquis, pour sa patience, son implication, son sérieux et ses remarques pertinentes qui m'ont aidé à accomplir un travail de qualité pendant le stage.

Je tiens à remercier Frédéric Potier pour son aide lors de mon stage, notamment sur l'étude des différents outils internes. J'ai apprécié travailler avec lui et son expérience m'a été d'une grande aide.

Je remercie également Alexandre Astier pour son aide sur les fichiers turtle et leur validation, Jean-Philippe Sahut d'Izarn pour son aide sur les requêtes SPARQL et les échanges sur l'outil AMStore et Abdelmonen Feki pour les échanges que j'ai pu avoir avec lui sur l'outil Apollo.

D'une manière plus générale, je remercie l'ensemble du département BIOVIA France, notamment Adrien, Gabriel, Manon, Jean, Benjamin, Brice, Valentin, Jérémy, Marine, Camille, Julien, Jean-Baptiste, François et Emmanuel pour leur accueil et leur sympathie qui ont contribué à faire de mon stage une expérience très bénéfique.

Je souhaite également remercier les enseignants du FIL et du Master IAGL, notamment Martin Monperrus, pour la qualité de la formation que j'ai reçue.

Enfin, je remercie ma tutrice universitaire, Iovka Boneva, pour son suivi régulier et son implication dans mon stage ainsi que dans mon rapport et la préparation de ma soutenance.

Table des matières

Introduction

Actuellement en dernière année d'études en Master IAGL [1], j'ai pour but de m'insérer par la suite dans le monde du travail en travaillant sur des problématiques innovantes et en faisant du développement de prototypes. Lors de ma recherche de stage de fin d'études, j'ai ainsi axé mes recherches sur des missions portant sur ces thématiques.

Mon choix s'est assez rapidement porté sur Dassault Systèmes, qui est une entreprise d'édition de logiciels dans des domaines aussi variés que l'automobile, l'énergie, la construction, l'architecture ou les sciences de la vie. Ma mission s'est déroulée dans ce dernier domaine dans le département BIOVIA, au sein d'une équipe chargée de l'indexation de données. En effectuant mon stage dans cette entreprise, mes attentes étaient principalement basées sur l'innovation, la recherche de solutions et le développement de prototypes.

En biologie, tout est centré autour des données. Certaines tâches comme l'indexation de données peuvent par conséquent varier selon le volume et le type de celles-ci. L'indexation de données est une tâche complexe qui consiste à faire passer des données brutes dans un pipeline de transformation et d'harmonisation pour obtenir en sortie des données formatées dans un index.

Sur le projet sur lequel j'ai travaillé, l'une des sources de données était trop volumineuse, et l'importer dans la mémoire vive faisait saturer cette dernière. Ainsi, il n'était pas possible de réaliser l'indexation de la manière prévue. Dans ce contexte, ma mission a été de réaliser l'étude de différents composants et outils, et choisir celui qui correspondait le plus à ce que nous recherchions pour réaliser l'indexation de ces données.

Ce rapport présente mes travaux pendant ce stage en 4 parties. La première partie présente le contexte de ce stage ainsi sa problématique. Ensuite, la seconde partie est centrée sur les données, outils et technologies utilisés pendant le stage. La troisième partie concerne l'étude des différentes possibilités pour l'indexation des données. Enfin, la quatrième et dernière partie explique comment a été réalisé cette indexation.

1. Infrastructures Applicatives et Génie Logiciel

1 Contexte

1.1 Dassault Systèmes

> *"Dassault Systèmes, the **3D**EXPERIENCE Company, has the mission to provide business and people with **3D**EXPERIENCE universes to imagine sustainable innovations harmonizing product, nature and life. Unveiled in 2012, this purpose has given birth to a broad portfolio of Industry Solution Experiences whose key strengths are in their scientific content and deep understanding of industrial processes. The Company's software portfolio is applicable from Natural Resources to Cities, Transportation, Buildings, Smart Products, Consumer Goods, all the way to biological systems and chemistry".*
>
> Tiré du rapport annuel [1], édition 2016

L'entreprise Dassault Systèmes a été fondée en 1981 par des ingénieurs venant de Dassault Aviation pour créer des produits 3D aujourd'hui très utilisé notamment dans l'automobile. La même année, l'entreprise lance son produit phare, CATIA, logiciel de conception 3D. Au fur et à mesure de son développement et de ses acquisitions, les secteurs de l'entreprise vont se diversifier : gestion de vie de produit avec ENOVIA en 1999, la simulation avec SIMULIA en 2005, la planète et les ressources naturelles avec GEOVIA en 2012, etc.

Aujourd'hui, Dassault Systèmes emploie 12 100 employés dont plus de 3 000 à Vélizy Villacoublay, siège social de l'entreprise situé en région parisienne. Le chiffre d'affaires 2016 s'élève à 3 milliards d'euros, en progression de 7% par rapport à 2015.

1.2 BIOVIA et Eagle

BIOVIA est le département de Dassault Systèmes concernant les biosphères virtuelles et les matériaux. Il est issu de la fusion de l'entreprise Accelrys, rachetée en 2014, et des équipes de bio-informatique existantes au sein de Dassault Systèmes.

BIOVIA a pour but de créer des solutions qui connectent les innovations biologiques et chimiques, et qui aident les sociétés R&D dans ces domaines. Les capacités du département couvrent notamment la simulation, la modélisation biologique et chimique et la gestion de données scientifiques.

Le projet Eagle s'inscrit notamment dans cette démarche de gestion de données. Il a pour but de centraliser des données biologiques de toutes sortes - protéines, gènes, molécules, maladies, etc - et de permettre à des experts de naviguer dans ces données pour créer et modéliser de la connaissance. Les finalités de l'application sont multiples : découverte de maladies, développement de médicaments, etc.

Le projet est développé par 3 équipes :
- une équipe pour développer les outils d'administration,
- une équipe pour développer les fonctionnalités de l'application,
- une équipe pour **développer les pipelines d'indexation** des données pour remplir la base de données de l'application. C'est l'équipe dans laquelle j'ai travaillé.

1. https://www.3ds.com/fileadmin/COMPANY/Investors/Annual-Reports/PDF/2016-3DS-Annual-report-EN.pdf

1.3 L'indexation des données

L'indexation des données est une tâche complexe. Il est nécessaire de modéliser en amont les données qui vont être intégrées dans l'application avant de faire l'indexation. La modélisation de ces différentes données et de leurs liens est présentée ci-dessous. Les noeuds représentent des types de données à stocker dans l'application, et les arcs représentent les liens entre les données.

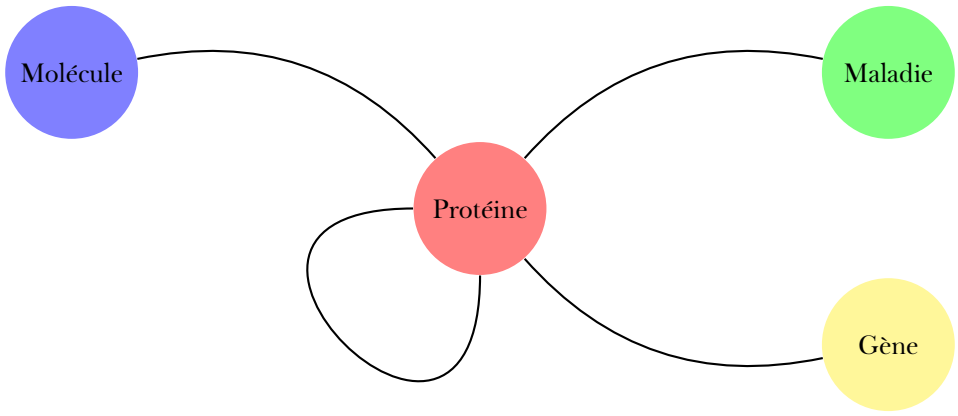


FIGURE 1.1 – Schéma montrant les différentes entités du modèle et leurs liens

Après la modélisation vient l'indexation à proprement parler. C'est l'étape qui concerne l'analyse des fichiers sources, leur décomposition, leur transformation et leur envoi dans l'index qui stocke les données.

Au sein du projet Eagle, l'indexation consiste tout d'abord à parser une source de données en entrée - des fichiers sous différents formats, un flux, etc - et à créer des objets scientifiques. Dans notre cas, un objet scientifique est un élément contenant des propriétés valuées conforme à un corpus RDF. Une fois les objets scientifiques créés, ceux-ci sont à leur tour transformés en documents, puis poussés dans l'outil CloudView qui stocke les données dans un index. Le graphe ci-dessous présente cette série de transformations et de manipulations.

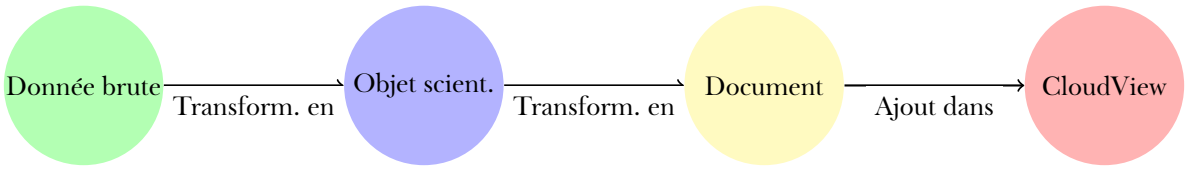


FIGURE 1.2 – Schéma montrant les différentes étapes entre une source de données brutes et un document poussé dans CloudView

La transformation des données brutes en objets scientifiques est gérée par un parser, tandis que la transformation en document puis l'ajout dans CloudView est faite par un connecteur.

1.4 Une problématique d'espace dans la mémoire vive

Pour la majorité des cas, la structure des données publiques fait qu'il n'est nécessaire de charger en mémoire qu'une infime partie des données. Par exemple, les données représentant les protéines sont représentées dans

un fichier les unes à la suite des autres. Par conséquent, il est assez simple d'itérer sur les éléments sans devoir en charger un grand nombre en mémoire à un même instant t.

A l'inverse, les données de ChEMBL stockent plusieurs types d'entités. A chaque type d'entités correspond un fichier. Les différentes entités - et donc fichiers - et les liens entre ces entités sont représentés dans le graphe ci-dessous. Les données que nous cherchons à indexer sont les substances - les molécules - et plusieurs liens entre les substances et les références de protéines, notés dans le graphe UniprotRef.

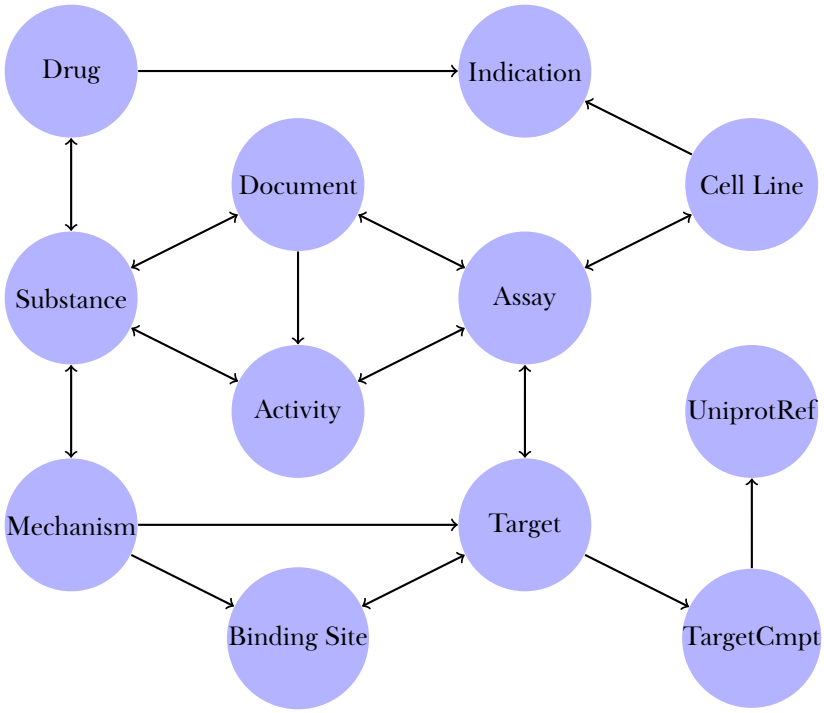


FIGURE 1.3 – Modèle de données ChEMBL : entités et liens

Indexer des liens entre les substances et les références d'Uniprot revient à charger l'ensemble des fichiers concernés par ce lien. Par exemple, si l'on souhaite indexer les liens entre *Substance* et *UniprotRef* en passant par *Document*, *Assay*, *Target* et *TargetCmpt*, cela revient à charger l'ensemble de ces fichiers.

Cela est impossible à cause de la place en mémoire vive que les fichiers concernés prendraient : plusieurs dizaines ou plusieurs centaines de gigaoctets. Il a ainsi fallu réfléchir à plusieurs solutions alternatives : créer certains liens après coup, charger temporairement les fichiers quelque part pour extraire une partie des données, etc.

Ma mission a été d'**étudier plusieurs possibilités pour l'indexation des données ChEMBL, choisir celle qui correspondrait le plus à nos besoins, et réaliser un prototype de parser/connecteur** pour indexer ces données.

2 Environnement technique

2.1 Les données à indexer

Les données à importer dans l'application sont des données publiques. L'avantage de données publiques est qu'elles sont libres d'accès, et mises à jour assez régulièrement. Elles proviennent de 6 bases de données, dont voici la liste :

- UniProt [1] : contient des données sur des protéines. C'est la plus grosse base de données à importer. Elle contient 85 millions de protéines, pour un total de 30 milliards de triplets RDF, soit 150 gigaoctets de données compressées. Elle est stockée au format rdf-xml,
- ChEMBL [2] : contient des données sur des molécules, des expériences chimiques, etc. Elle contient 482 millions de triplets, stockés au format turtle,
- IntAct [3] : contient des données sur des interactions entre protéines,
- EFO [4] : contient des données sur des maladies,
- OpenTargets [5] : contient des données sur des associations entre protéines et maladies,
- NCBI [6] : contient des données sur des gènes.

Pendant le stage, j'ai principalement travaillé avec les données ChEMBL.

2.2 L'outil interne CloudView

CloudView est un outil développé par la société Exalead, rachetée par Dassault Systèmes en 2010. Il permet d'indexer des données via des connecteurs. Les connecteurs natifs CloudView sont disponibles pour un large éventail de sources, telles que des serveurs de fichiers, des serveurs XML, des bases de données, des messageries, des systèmes de gestion de contenu et de collaboration, etc. Ceux-ci peuvent être créés directement via l'interface de CloudView et être configurés en quelques instants.

Il est également possible d'écrire des connecteurs manuellement grâce à des APIs - interface de programmation. L'implémentation de tels éléments se fait en Java. Dans ce cas, le développeur doit effectuer la totalité des tâches via ces APIs : création des documents, envoi dans l'index, configuration éventuelle selon un certain nombre de paramètres, etc.

2.3 Les données du Web sémantique

2.3.1 RDF

Le RDF - Resource Description Framework - est un modèle de graphe créé par le W3C en 1997 destiné à décrire de façon formelle les ressources Web et leurs métadonnées, de manière à permettre le traitement automatique de telles descriptions.

Les ressources RDF sont structurées sous forme de triplets. Chaque triplet est une association *sujet, prédicat, objet* :

- Sujet : représente la ressource à décrire, prend la forme d'une URI.
- Prédicat : représente une propriété applicable au sujet du triplet, prend la forme d'une URI.
- Objet : représente la valeur de la propriété, prend la forme d'un littéral ou d'une URI.

1. http://www.uniprot.org/
2. https://www.ebi.ac.uk/chembl/
3. http://www.ebi.ac.uk/intact/
4. http://www.ebi.ac.uk/efo/
5. https://www.opentargets.org/
6. https://www.ncbi.nlm.nih.gov/

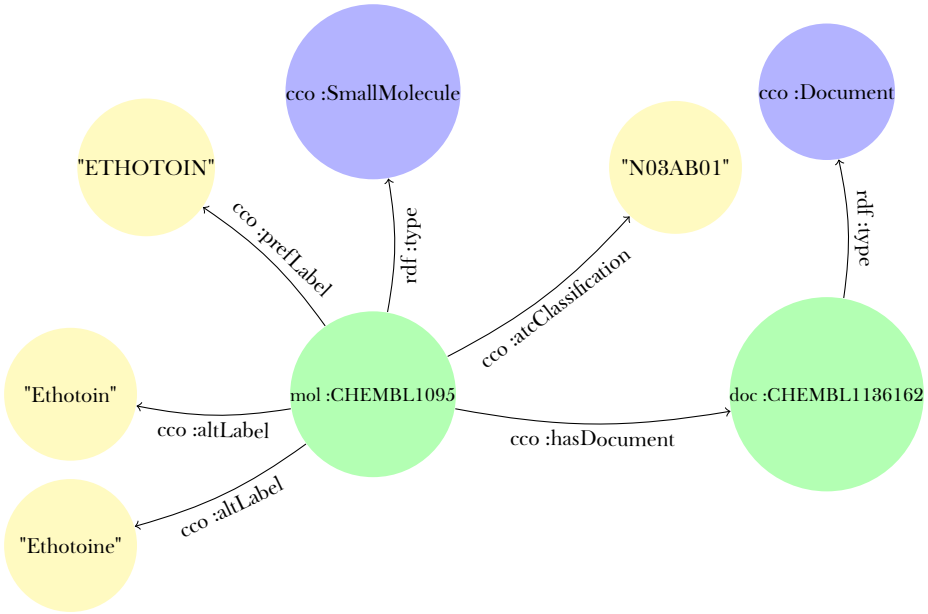
```
@prefix cco:    <http://rdf.ebi.ac.uk/terms/chembl#> .
@prefix mol:    <http://rdf.ebi.ac.uk/resource/chembl/molecule#> .
@prefix doc:    <http://rdf.ebi.ac.uk/resource/chembl/doc#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

mol:CHEMBL1095
    rdf:type            cco:SmallMolecule ;
    cco:prefLabel       "ETHOTOIN" ;
    cco:altLabel        "Ethotoin", "ETHOTOIN", "Peganone" ;
    cco:chemblId        "CHEMBL1095" ;
    cco:hasDocument     doc:CHEMBL1136162, doc:CHEMBL3137667, doc:CHEMBL1158643 ;
    cco:moleculeXref    <http://en.wikipedia.org/wiki/Ethotoin> ;
    cco:atcClassification "N03AB01" .
```

FIGURE 2.1 – Exemple de données RDF représentant une molécule et certains de ses attributs, stockée au format turtle

Dans l'exemple ci-dessous, il existe une ressource qui a pour URI *mol :CHEMBL1095*. Celle-ci est de type *SmallMolecule*. Elle possède plusieurs prédicats ou propriétés :

- cco :prefLabel. A pour valeur "ETHOTOIN",
- cco :altLabel. A pour valeur "Ethotoin", "ETHOTOIN", "Peganone",
- etc.

Comme l'objet d'un triplet peut être une ressource, on peut établir un graphe représentant ces données, dont les noeuds sont des ressources ou des littéraux et les arcs sont des prédicats entre les ressources.



La figure ci-dessus représente un tel type de graphe pour rapport à la molécule présentée précédemment. Les noeuds en jaune sont les littéraux et celles en vert sont les ressources, qui sont ainsi au nombre de 2. Chacune de ces ressources a un type, représenté ici en bleu.

2.3.2 SPARQL

Le SPARQL - sigle récursif signifiant *SPARQL Protocol And RDF Query Language* - est un langage de requêtes sur des données RDF. Il permet d'effectuer plusieurs opérations sur les données RDF comme l'insertion, la suppression, la consultation, etc. L'une des opérations les plus intéressantes est l'opération SELECT.

```
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?molecule
WHERE {
   ?molecule rdf:type cco:SmallMolecule .
}
```

Listing 2.1 – Première requête SPARQL

Dans cette première requête, on cherche à récupérer toutes les ressources dont le type est *cco*:*SmallMolecule*. Lancée sur les données détaillées précédemment, cette requête va nous renvoyer une ressource : *mol :CHEMBL1095*.

```
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?molecule ?altLabel ?prefLabel
WHERE {
   ?molecule rdf:type cco:SmallMolecule .
   ?molecule cco:prefLabel ?prefLabel .
   ?molecule cco:altLabel ?altLabel .
}
```

Listing 2.2 – Seconde requête SPARQL

Dans cette seconde requête, l'objectif est de récupérer pour chaque *SmallMolecule* l'ensemble des valeurs des propriétés *prefLabel* et des *altLabel*. Cette requête va retourner un tableau avec les données suivantes :

molecule	prefLabel	altLabel
mol :CHEMBl1095	"ETHOTOIN"	Ethotoin"
mol :CHEMBl1095	"ETHOTOIN"	EHOTOIN"
mol :CHEMBl1095	"ETHOTOIN"	Peganone"

FIGURE 2.2 – Tableau représentant les résultats obtenus en lançant la requête du *Listing 2.2* sur les données de la *Figure 2.1*.

Pour pouvoir faire des requêtes SPARQL sur des données RDF, il est nécessaire que celles-ci soient stockées dans un triple store. Par la suite, il est possible de faire des requêtes sur ce triple store à l'aide d'une interface, appelé endpoint.

2.3.3 L'entailment

L'entailment est un mécanisme qui permet d'inférer - de déduire - de nouvelles informations à partir d'informations existantes. Par exemple, si l'on a la phrase *"Tous les renards ont le poil roux, mais cet animal n'a pas le poil roux"*, alors on sait que cet animal n'est pas un renard.

Le même système de déduction d'informations existe en RDF. On peut ainsi obtenir de nouveaux triplets à partir de triplets existants et de règles. Ces règles sont au nombre de 13. Les règles les plus courantes et les plus utilisées sont présentées dans le tableau page suivante.

Numéro de règle	Si les données contiennent	Alors ajouter
rdfs5	UUU rdfs:subPropertyOf VVV . VVV rdfs:subPropertyOf XXX .	UUU rdfs:subPropertyOf XXX .
rdfs6	UUU rdf:type rdf:Property .	UUU rdfs:subPropertyOf UUU .
rdfs7	AAA rdfs:subPropertyOf BBB . UUU AAA YYY .	UUU BBB YYY .
rdfs9	UUU rdfs:subClassOf XXX . VVV rdf:type UUU .	VVV rdf:type XXX .
rdfs10	UUU rdf:type rdfs:Class .	UUU rdfs:subClassOf UUU .
rdfs11	UUU rdfs:subClassOf VVV . VVV rdfs:subClassOf XXX .	UUU rdfs:subClassOf XXX .

FIGURE 2.3 – Tableau représentant les principales règles d'inférence RDF.

Sans entailment, il est nécessaire d'exprimer une requête différemment pour que les classes et propriétés soient prises en compte convenablement. Par exemple, pour les données ChEMBL, la classe Substance possède plusieurs sous classes. Ci-dessous est présentée la requête qui permet de récupérer les ressources dont le type est *cco :Substance* ou dont l'un des surtypes est *cco :Substance*. Cette requête suppose que le triple store supporte l'entailment.

```
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?substance
WHERE {
   ?substance rdf:type cco:Substance .
}
```

Listing 2.3 – Requête SPARQL pour récupérer les substances avec l'entailment intégré dans le triple store

Sans entailment, il est nécessaire d'expliciter les relations entre les classes et de modifier la requête de la manière suivante. L'étoile indique que le solveur de la requête peut passer zéro, une ou plusieurs fois par la relation *rdfs:subClassOf*.

```
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdf: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?substance
WHERE {
   ?substance rdf:type ?type .
   ?type rdfs:subClassOf* cco:Substance .
}
```

Listing 2.4 – Requête SPARQL pour récupérer les substances en explicitant l'entailment dans la requête

2.4 Java

Les pipelines d'indexation sont codés en Java. Une grande architecture déjà présente à mon arrivée contenait de nombreux éléments de code. Cela a pour but d'organiser le code au maximum, de limiter les erreurs et de permettre aux développeurs d'éditer uniquement les fichiers voulus. Tout cela est manipulé grâce à un gestionnaire interne d'arborescence de code. Il existe également un autre outil interne pour compiler du code, générer les dépendances des projets, lancer les tests, etc. Cet outil se nomme mkmk.

Durant mon stage, j'ai également utilisé à de multiples reprises la bibliothèque Apache Jena [7]. C'est une bibliothèque Java qui permet de créer et manipuler des données RDF.

7. https://jena.apache.org/

3 Études de différents outils

3.1 La consolidation de CloudView

La première étape de mon stage a été la prise en main de l'outil CloudView, et l'étude de l'un de ses composant, appelé le serveur de consolidation. Le but était de voir si ce composant pouvait nous aider à réaliser l'indexation des données ChEMBL. Ce composant se place entre les connecteurs et le serveur d'indexation et permet d'effectuer deux opérations supplémentaires sur les documents poussés dans l'index :

— La transformation est une opération qui consiste à créer des relations entre les documents poussés sur le serveur de consolidation. Les documents et leurs relations sont stockés sous forme de graphe, où les documents sont les noeuds et les relations sont des arcs.

— L'agrégation permet de naviguer entre les documents en passant le long des arcs, et de récupérer des valeurs pour créer de nouvelles métadonnées dans les documents.

Comme pour les connecteurs, le code d'un serveur de consolidation est à écrire en Java à l'aide d'APIs CloudView.

3.1.1 Connecteur CSV et consolidation

J'ai commencé à tester CloudView en créant un unique connecteur CSV [1]. Ce connecteur indexe des données présentes dans un fichier CSV, où chaque ligne fait référence à un objet prêt à être indexé. Le but était de simuler le comportement des différents connecteurs en cours de développement. L'étape de consolidation est utile pour créer les liens qui nous intéressent : stocker dans une protéine les identifiants de protéines interagissants avec cette première protéine par exemple.

La création du connecteur CSV s'est effectuée via l'interface de CloudView où il a suffit de renseigner le type du connecteur, le chemin du fichier CSV et l'endroit où les données doivent être envoyées après avoir été indexées : le serveur de consolidation.

Dès le début, j'ai dû faire face à un premier problème : les éléments provenant d'une seule et même source ont tous la même classe de document. Comme notre connecteur indexe des protéines, des gènes et d'autres éléments à partir du fichier CSV, il a fallu résoudre ce comportement en créant depuis le code Java du serveur de consolidation les documents avec leur classe attitrée.

Une fois cela fait, j'ai pu créer un processus de transformation ayant pour tâche de créer les arcs entre les documents selon leurs propriétés comme expliqué précédemment.

Le graphe page suivante montre le résultat obtenu sur l'exemple d'une protéine interagissant avec une autre. Ici, la protéine P00533 interagit de 5 manières différentes avec la protéine P21860. Cette information est stockée dans 5 interactions binaires contenues dans un cluster d'interactions.

La dernière étape a consisté à naviguer le long des arcs pour stocker les informations voulues, comme stocker dans une protéine l'ensemble des protéines interagissant avec cette protéine. Tout cela a été implémenté en Java grâce aux APIs de CloudView.

3.1.2 Consolidation sur les données du connecteur EFO

J'ai par la suite travaillé sur le connecteur EFO qui indexe des maladies. J'ai créé un processus de transformation et d'agrégation pour stocker dans les maladies les identifiants de toutes les maladies ascendantes. Par exemple, la maladie *conjunctival disorder* possède 2 maladies ascendantes : *eye disease*, puis *disease*. Après passage dans le serveur de consolidation, la maladie *conjunctival disorder* possède ainsi une métadonnée appelée *disease ascendant* avec les 2 identifiants des maladies ascendantes.

1. Comma-Separated Values

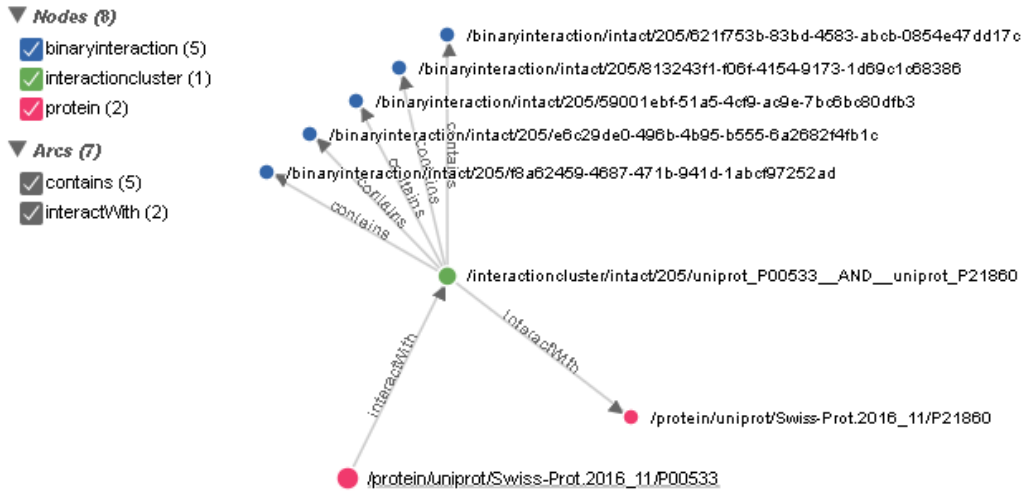


FIGURE 3.1 – Impression d'écran de CloudView concernant la visualisation de documents passés par le serveur de consolidation

Même si le but a été atteint, une métrique nous a particulièrement posé problème : le temps complet du passage des 9 000 documents par le connecteur, le serveur de consolidation et par l'index est de 10 minutes. Dans notre cas, c'est un temps beaucoup trop conséquent au regard de la taille des données que nous devons indexer.

Suite à ces 2 premières tâches, il a été nécessaire d'établir une architecture solide pour que les documents provenant du connecteur CSV ou du connecteur EFO soient traités convenablement. L'architecture est présentée ci-dessous.

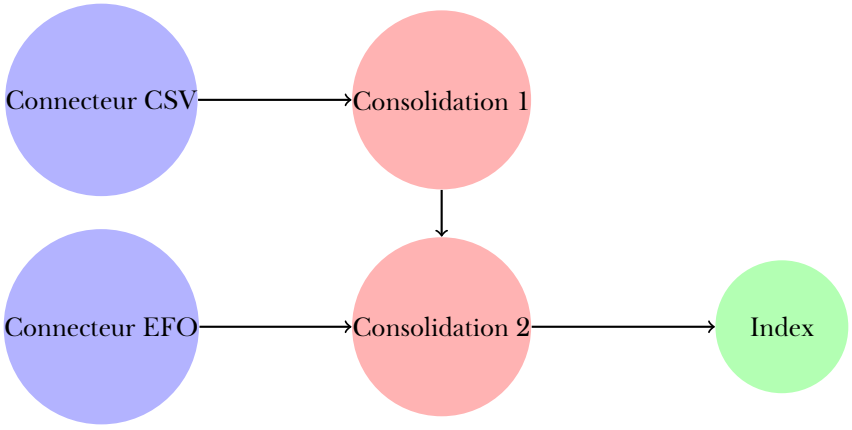


FIGURE 3.2 – Schéma représentant l'enchaînement des différents connecteurs et serveurs de consolidation jusque l'index

Lorsque le connecteur CSV pousse ces données, celles-ci sont envoyés dans le premier serveur de consolidation. C'est lui qui s'occupe d'attribuer à chaque des documents sa classe. Par la suite, les documents sont poussés vers le second serveur de consolidation, qui s'occupe de créer les arcs et les nouvelles métadonnées sur les documents. C'est également sur ce serveur de consolidation que sont poussés les documents provenant du connecteur EFO. Une fois passés par le second serveur de consolidation, les documents sont poussés dans l'index.

3.1.3 Consolidation sur les données du connecteur Uniprot

La dernière étape de mon travail sur la consolidation de CloudView a été de travailler sur le connecteur Uniprot, qui concerne l'indexation des protéines. Le but était de stocker dans un gène la liste des protéines que ce gène encode. Pour cela, à chaque fois qu'une protéine est encodée par un gène, le but était de récupérer l'identifiant du gène pour l'ajouter dans une métadonnée de la protéine.

Ce but s'est heurté à un problème impossible à résoudre : lors de leur passage dans le serveur de consolidation, les documents n'ont connaissance que d'eux-même, à la manière d'un flux qui envoie l'un après l'autre les éléments. A l'indexation des protéines, il était donc impossible de récupérer un gène pour y ajouter des données.

Ce problème, ajouté au temps de traitement beaucoup trop long sur les serveurs de consolidation, nous a amené à **ne pas retenir l'utilisation de la consolidation dans CloudView**.

3.2 Comparaison des outils Apollo, AMStore et Virtuoso

Puisque la consolidation de CloudView ne pouvait pas nous être utile, nous nous sommes dirigés vers l'idée de stocker temporairement les données ChEMBL sur un triple store, d'où nous pourrions faire des requêtes SPARQL pour récupérer les éléments qui nous intéressent. Nous nous sommes tournés vers Apollo et l'AMStore, qui sont 2 outils internes en cours de développement faisant office de triple store. Pour comparer ces outil avec un outil déjà existant, nous avons également étudié Virtuoso [2], triple store existant libre d'utilisation non commerciale.

L'objectif de cette partie de mon stage était de **comparer les temps d'import des données, et les temps d'exécution de différentes requêtes SPARQL** sur les données Uniprot et ChEMBL, sur les outils Virtuoso, AMStore et Apollo. L'étude de ces différents outils devaient ainsi nous permettre de choisir celui qui serait le plus adapté à nos besoins pour réaliser le prototype du parser/connecteur ChEMBL.

3.2.1 Conversion des données en format turtle

Apollo et l'AMStore ne prenant en entrée que des données RDF au format turtle, nous avons d'abord dû penser à un moyen de convertir les données. En effet, seules les données ChEMBL étaient disponibles dès le départ en turtle.

Dans un premier temps, j'ai écrit un convertisseur d'objets scientifiques en format turtle. Pour rappel, un objet scientifique est ce qui est produit par chacun des parsers lors du traitement des fichiers. Le parser Uniprot crée ainsi des objets scientifiques de type protéine, le connecteur EFO crée des maladies, etc.

Ce convertisseur a été testé avec 3 parsers : EFO, Intact et Uniprot. À chaque fois, il a été nécessaire de bien structuré le code dans des framework et modules précis pour pouvoir tester cela. Les tests sont lancés avec l'utilitaire mkmk présenté dans la partie 2.

Le convertisseur d'objets scientifiques en format turtle a été utilisé pour les données EFO, Intact, OpenTargets, NCBI et sur quelques données Uniprot. Comme les données Uniprot étaient trop volumineuses, cela aurait pris trop de temps à les parser pour ensuite écrire les objets scientifiques au format turte. J'ai ainsi écrit un script utilisant un utilitaire de la bibliothèque Jena pour convertir les données Uniprot, disponibles au format xml-rdf.

3.2.2 Discussion avec les équipes

Avant d'obtenir la main sur les outils à tester, nous avons commencé notre approche par plusieurs discussions avec les équipes en interne pour voir quels étaient les niveaux d'avancement de ceux-ci.

2. https://virtuoso.openlinksw.com/

Lors de nos discussions, nous avions notamment appris que l'entailment n'était à l'époque supporté ni par Apollo, ni par l'AMStore. Cela était pour nous un problème conséquent car les requêtes que nous voulions écrire pour indexer les données ChEMBL avaient besoin de ce mécanisme pour fonctionner correctement. De plus, d'autres éléments de syntaxe non supportés au moment de ces discussions nous ont ôté toute solution de secours pour écrire d'une autre manière ces requêtes.

De manière plus générale, les outils étaient à un stade moins avancé que ce que nous espérions, et d'autres éléments rendaient leur utilisation complexe : version pas très stable, erreur de saturation de la mémoire vive à l'import, etc.

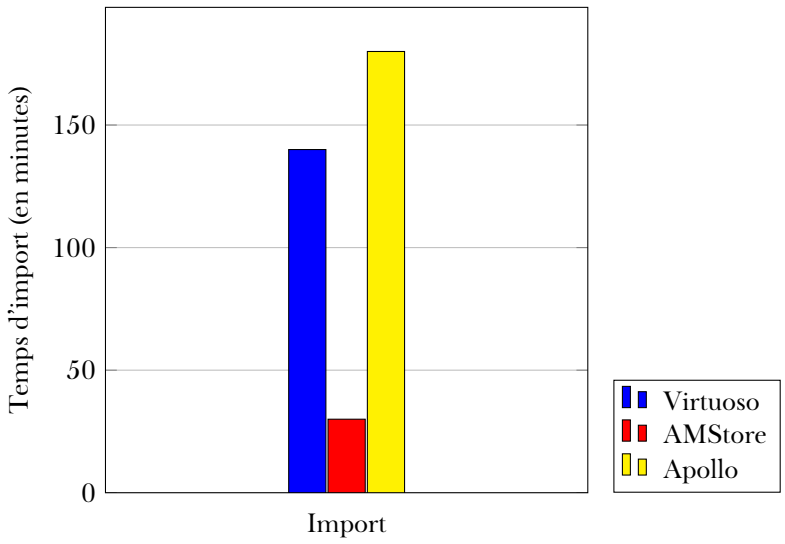
Face à ces problèmes, il avait assez rapidement été décidé **d'écrire le prototype du parser/connecteur ChEMBL à l'aide de Virtuoso, mais de continuer à étudier les outils Apollo et AMStore.** J'ai ainsi travaillé sur ces 2 tâches en parallèle jusque la fin de mon stage.

3.2.3 Import des données

Après avoir converti les fichiers, nous avons voulu mesurer les temps d'import des différentes bases dans les différents outils. J'ai donc commencé par importer les fichiers turtle compressés d'Uniprot générés précédemment grâce au script dans Virtuoso. Cela n'a pas fonctionné à cause d'un problème relatif à la transformation des fichiers, et nous n'avons pas pu aller au terme de l'import comme nous le souhaitions. Comme les comparaison de temps d'import n'ont pas été possibles avec les données d'Uniprot, nous avons effectué les imports avec les données ChEMBL, directement accessibles au format turtle.

L'import des données a concerné 1.7 gigaoctet de données compressées au format gzip, ce qui donne environ entre 8 et 12 Go de données décompressés. Pour les 3 outils, l'import se fait en quelques lignes de commandes en spécifiant la liste des fichiers à importer.

Les temps obtenus sont présentés dans l'histogramme ci-dessous.



3.2.4 Les requêtes SPARQL

Par la suite, nous avons écrit quelques requêtes SPARQL à lancer sur les différents outils pour pouvoir comparer les temps d'exécution. Le but était de lancer ces requêtes sur les endpoints, vérifier que le nombre de résultats est identique, et comparer les temps d'exécution.

Le principal problème auquel nous avons fait face était comme présenté précédemment le comportement des outils face à l'entailment. Ce problème m'a souvent amené à réfléchir à des requêtes alternatives qui fonctionneraient sur les 3 outils pour que les temps d'exécution et les nombres de résultats puissent être comparés.

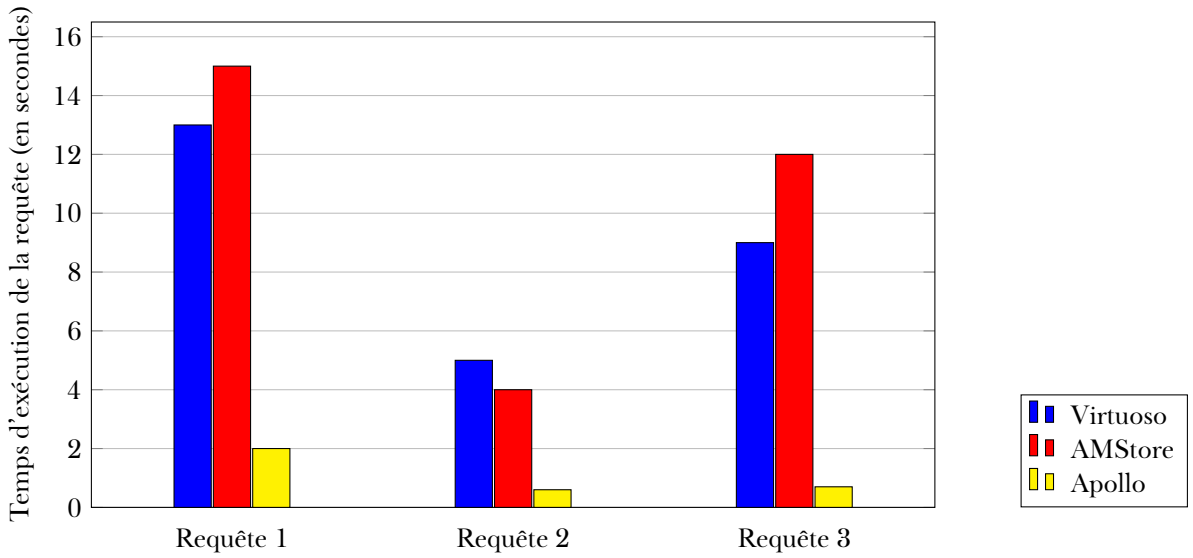
Nous avons envisagé de mettre plusieurs solutions en place, la principale étant la suivante : à chaque fois qu'une requête nécessitait l'utilisation de l'entailment pour résoudre tous les sous types d'un type donné, nous avons écrit l'ensemble des requêtes avec chacun des sous types.

Nous pouvons prendre pour exemple la requête présentée en Annexe 1 qui récupère certaines valeurs des prédicats d'une substance. La solution dans ce cas a été de remplacer *Substance* par chacun des sous types de *Substance*. Ainsi, nous avons écrit 9 requêtes en remplaçant *Substance* par *Antibody*, *CellTherapy*, *Enzyme*, etc.

3.2.5 Résultats

Le niveau de connaissance que nous avions des outils - quasi nul - nous obligeait à communiquer de manière régulière avec les équipes des outils testés pour s'assurer que les tests étaient valables et fonctionnaient convenablement. Les tests ayant été menés pendant la période des vacances d'été, il m'est arrivé de tester les outils d'une certaine manière qui s'avérait par la suite être non comparable par rapport à d'autres résultats obtenus.

Néanmoins, à force de persévérance et de tests, nous avons pu obtenu des résultats partiels qui donnaient une idée de la performance des outils par rapport à certaines requêtes. Certains de ces résultats sont présentés dans les graphiques ci-dessous.



Pour des raisons de confidentialité, il n'est pas possible d'expliciter dans ce rapport les requêtes qui ont permis d'obtenir ces résultats. Ces graphiques donnent néanmoins une tendance globale. Hormis de rares cas, Apollo est l'outil le plus performant où la quasi totalité des requêtes s'effectuent en moins de 5 secondes. Quant à Virtuoso et l'AMStore, les temps d'exécution sont dans le même ordre de grandeur.

Sur certaines requêtes, nous n'avons parfois pas pu obtenir de résultats, dû à une erreur de l'un des outils. Ces erreurs pouvaient être de diverses formes comme une saturation de la mémoire, une requête non supporté, ou une requête tournant dans le vide pendant plusieurs minutes. Même si nous n'avons pas obtenu de temps pour ces requêtes, les informations d'erreur étaient très intéressantes pour les équipes concernées qui ont pu ainsi continuer à améliorer leur outil.

4 Réalisation du parser et du connecteur ChEMBL

4.1 Le prototype

4.1.1 Le modèle de données ChEMBL

Comme vu précédemment, le modèle de données de ChEMBL est complexe et les entités à récupérer sont multiples. Tout d'abord, il était nécessaire de récupérer les informations sur les substances, qui représentent les principales informations de ChEMBL. Par la suite, il était nécessaire de récupérer les liens entre *Substance* et *UniprotRef* en passant par *Document*, *Assay*, *Target* et *TargetCmpt*, puis en passant par *Mechanism*, *Target* et *TargetCmpt*.

L'ensemble des informations à récupérer est modélisé dans le graphe ci-dessous. Les noeuds en rouge foncé - *Substance* et *UniprotRef* - sont les noeuds de départ et d'arrivée des liens. Les noeuds en rouge pâle et les arcs rouges sont les noeuds et arcs par lesquels il est nécessaire de passer.

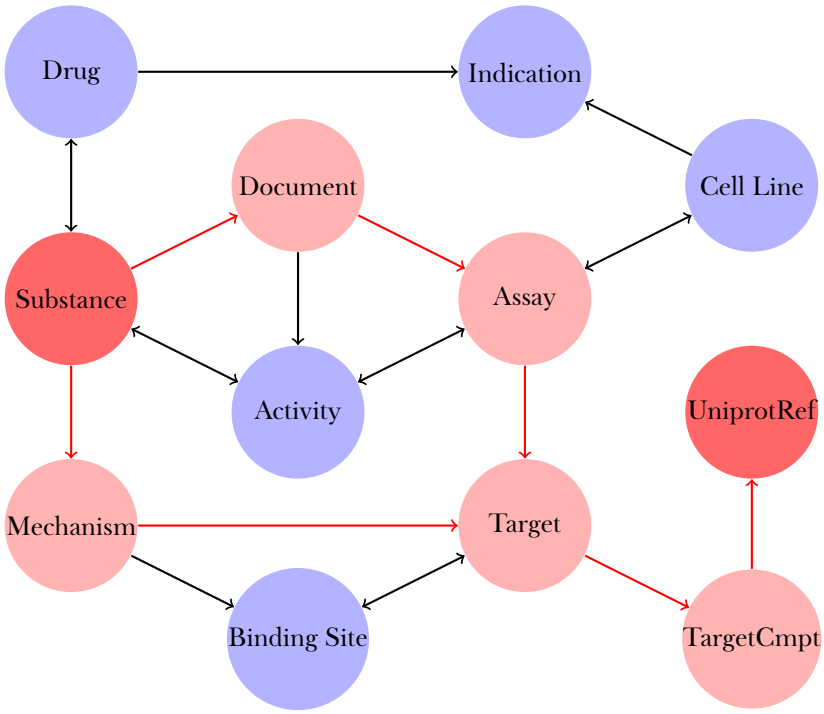


FIGURE 4.1 – Modèle de données ChEMBL : entités et liens

Une fois les éléments à récupérer identifiés, je suis passé à l'écriture des requêtes SPARQL.

4.1.2 Les requêtes SPARQL

Les requêtes SPARQL à écrire pour récupérer les données qui nous intéressent sont au nombre de 3. La première requête est celle qui permet de récupérer les informations sur les substances. Cette requête liste les propriétés et les valeurs des propriétés pour chaque substance, et c'est le code Java qui traite les valeurs des prédicats selon ces derniers. Cette requête est présentée page suivante.

Après avoir testé cette requête, il s'est avéré que nous n'arrivions pas à obtenir tous les résultats attendus - plus de 55 millions - mais seulement une fraction. Cela est vraisemblablement dû à un paramètre de Virtuoso qui limite le nombre de résultat d'une requête. Je n'ai jamais su corriger ce problème et j'ai donc dû trouver une autre manière de procéder.

À la requête initialement prévue, j'ai ajouté l'élément *FILTER REGEX*(?*s*, "*%PATTERN%*"). pour filter les URLs des ressources selon un pattern spécifique. L'élément %PATTERN% est modifié en Java dans une boucle. Successivement, il prend les valeurs CHEMBL10, CHEMBL11, etc, jusque CHEMBL99. Cela permet de récupérer en 90 fois les éléments voulus sans avoir de problème lié à une limite quelconque.

```
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?s ?p ?o
WHERE {
        ?s rdf:type ?type .
        ?type rdfs:subClassOf* cco:Substance .

        {
                ?s ?p ?o .
        } UNION {
                ?s <http://semanticscience.org/resource/SIO_000008> ?sio .
                ?sio ?p ?o .
        }
} ORDER BY ?s
```

Listing 4.1 – Requête pour récupérer les substances

La seconde requête concerne les liens entre une substance et une référence de protéine via des ressources de type *Document*, *Assay*, *Target* et *TargetCmpt*. Sur les éléments intermédiaires, la requête va récupérer l'identifiant pubmed du document, la source et le type de l'assay. En arrivant sur *TargetCmpt* et sa propriété *cco:targetCmptXref*, il est nécessaire de ne prendre que les objets dont la classe est *UniprotRef*.

Cette requête renvoie beaucoup d'éléments, et nous avons donc eu le même problème de limite que pour la requête précédente. La même solution avec le *FILTER REGEX* y a donc été appliquée.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX bibo: <http://purl.org/ontology/bibo/>

SELECT ?substance ?pubmedid ?sourceLabel ?assayType ?protein
WHERE {
    ?substance rdf:type ?type .
    ?type rdfs:subClassOf* cco:Substance .
    ?substance cco:hasDocument ?document .
    ?document bibo:pmid ?pubmedid .
    ?document cco:hasAssay ?assay .
    ?assay cco:assayType ?assayType .
    ?assay cco:hasSource ?source .
    ?source rdfs:label ?sourceLabel .
    ?assay cco:hasTarget ?target .
    ?target cco:hasTargetComponent ?targetcomponent .
    ?targetcomponent cco:targetCmptXref ?protein .
    ?protein rdf:type cco:UniprotRef .
}
```

Listing 4.2 – Requête pour récupérer les liens en passant par Document

La troisième requête traite des liens entre une substance et une référence de protéine en passant par *Mechanism*, *Target* et *TargetCmpt*. Sur les éléments intermédiaires, la requête va récupérer le type d'action et la description du mécanisme. De la même manière que précédemment, on ne récupère à la fin que les objets dont la classe est *UniprotRef*.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>

SELECT ?substance ?actiontype ?description ?protein
WHERE {
    ?substance rdf:type ?type .
    ?type rdfs:subClassOf* cco:Substance .
    ?substance cco:hasMechanism ?mechanism .
    ?mechanism cco:mechanismActionType ?actiontype .
    ?mechanism cco:mechanismDescription ?description .
    ?mechanism cco:hasTarget ?target .
    ?target cco:hasTargetComponent ?targetcomponent .
    ?targetcomponent cco:targetCmptXref ?protein .
    ?protein rdf:type cco:UniprotRef .
}
```

Listing 4.3 – Requête pour récupérer les liens en passant par Mechanism

4.1.3 Le parser

Comme expliqué précédemment, le parser est le code qui va, à partir de données en entrée, récupérer les informations souhaitées, les formater et les renvoyer sous forme d'objets scientifiques. Contrairement aux autres parsers, nous n'avons pas directement de fichiers en entrée mais un endpoint SPARQL sur lequel il est possible de faire des requêtes et de récupérer des données.

Le parser a un comportement bien défini selon la requête. Pour la première requête, chaque résultat correspond à une propriété d'une substance et sa valeur. Il y a ainsi plusieurs résultats pour reconstituer une substance. L'algorithme ci-dessous explique les opérations à faire lors du traitement de la requête concernant les substances.

Algorithm 1: Algorithme qui traite les résultats de la requête concernant les substances

Data: La requête SPARQL *query* à lancer en entrée

$resultSet \leftarrow LaunchQuery(query)$

while $HasNext(resultSet)$ **do**
  $querySolution \leftarrow Next(resultSet)$
  $newId \leftarrow GetSubject(querySolution)$

  if $newId \neq oldId$ **then**
    $FireObjectChange(scientificObject)$
    $oldId \leftarrow newId$

  end
  $AddPropertyAndValueToSO(scientificObjet, querySolution)$

end

À l'inverse, chacun des résultats des 2 requêtes concernant les liens entre une substance et une référence d'Uniprot sont des résultats à part entière qui amènent la création d'une annotation. Le traitement est donc le suivant :

Algorithm 2: Algorithme qui traite les résultats de la requête concernant les substances

Data: La requête SPARQL *query* à lancer en entrée

$resultSet \leftarrow LaunchQuery(query)$

while $HasNext(resultSet)$ **do**
  $querySolution \leftarrow Next(resultSet)$
  $AddAllPropertyAndValueToSO(scientificObjet, querySolution)$

  $FireObjectChange(scientificObject)$
end

4.1.4 Le connecteur

Le connecteur est le **code** qui se connecte à CloudView, qui lance le parser, et qui, une fois un objet scientifique récupéré, s'occupe de le pousser dans l'index de CloudView. Comme les étapes sont toujours les mêmes, le code existant dans l'architecture effectuait déjà l'immense majorité de ces tâches. Je n'ai donc eu aucune difficulté à écrire le code du connecteur.

Le connecteur prend en entrée les paramètres de connexion à CloudView - adresse, port, utilisateur, mot de passe - l'URL de l'endpoint SPARQL et le chemin du fichier contenant les requêtes SPARQL.

4.1.5 Les métriques

L'ensemble du code et des requêtes ci-dessus a fonctionné correctement. Voici quelques métriques mesurées pendant le déroulement :

- Pour la création des substances, les 90 requêtes retournent 55 millions de résultats. 1.6 million de substances sont créées, en 5h. Ce temps inclut le temps d'exécution des 90 requêtes, le temps de traitement des résultats et d'envoi des documents CloudView dans l'index.
- Pour la création des annotations de documents, 7 000 000 annotations sont créées en 20 minutes.
- Pour la création des annotations de mécanismes, 10 000 annotations en 20 secondes.

Comme le but premier était de vérifier que l'implémentation choisie allait être correcte, le code du prototype comprend un grand nombre d'éléments codés en dur. Une fois que celui-ci a fonctionné convenablement, je me suis penché sur les améliorations successives qui pouvaient être faites.

4.2 Les améliorations successives

4.2.1 Le fichier de configuration

Un grand nombre d'éléments étaient implémentés en dur dans le code du prototype. Les améliorations de ce côté ont consisté à rendre flexible un plus grand nombre d'éléments.

En premier lieu, la position des requêtes dans le fichier SPARQL était figée. Ces positions ont été mises dans un fichier de configuration qui permet le changement de position des requêtes.

Dans un second temps, les paramètres autrefois nécessaires au connecteur - l'URL de l'endpoint SPARQL et le chemin de fichiers contenant les requêtes SPARQL - ont été également mis dans le fichier de configuration. Ainsi, le connecteur n'a plus besoin que du chemin du fichier de configuration comme paramètre d'entrée, hormis les paramètres de connexion à CloudView.

J'avais pendant un temps pensé à spécifier dans le fichier de configuration la correspondance entre une propriété d'une substance de ChEMBL et une propriété d'un objet scientifique. Cela aurait permis de rendre le parser beaucoup plus flexible et résistant face aux éventuels changements de modélisation chez ChEMBL.

Après réflexion, nous avons convenu que cela n'était pas judicieux. En effet, les données étant publiques, elles sont probablement utilisées par un grand nombre d'entreprises et d'organisations. Ainsi, il est peu probable qu'un changement d'URL de sujet ou de prédicat intervienne puisque cela affecterait un grand nombre de projets.

Néanmoins, si un tel changement devait intervenir, il est également probable qu'il faille reprendre les requêtes SPARQL pour les réécrire et se replonger dans le code pour vérifier que les propriétés et valeurs sont traitées correctement.

Enfin, en laissant le mapping entre les propriétés de ChEMBL et les propriétés des objets scientifiques en dur dans l'implémentation du parser, nous avons ainsi suivi ce qui a été fait dans les autres parsers.

4.2.2 Optimisation des requêtes SPARQL

L'optimisation des requêtes SPARQL a surtout concerné la requête de récupération des substances, de leurs propriétés et de leurs valeurs. A cause du problème de limite mentionné précédemment, nous avons dû passer outre ce problème en modifiant la manière de traiter les requêtes dans l'implémentation du parser.

Bien que je n'ai jamais réussi à enlever cette limite, j'ai néanmoins travaillé sur les requêtes et leur temps de calcul pour être certain de lancer la requête la plus optimale possible. J'ai ainsi écrit 5 requêtes qui sont autant de manières différentes de récupérer les informations des substances.

Les différentes requêtes m'ont permis de comparer le nombre de résultats retournés par la requête et le temps d'exécution mis par la requête pour retourner ces résultats. Ainsi, je suis passé de 55 millions de résultats retournés en 5h à 6 millions de résultats retournés en 50 minutes. Cela a grandement contribué à l'amélioration de parser/connecteur ChEMBL, et c'est après ces améliorations qu'une première version a été livré dans l'application.

Tout ne reste néanmoins pas parfait dans le prototype, et il revient désormais à mes collègues de reprendre mon travail pour améliorer le code existant.

Conclusion

Effectué chez Dassault Systèmes au sein du département BIOVIA, mon stage a pris place dans un projet de stockage de connaissance. En travaillant dans l'équipe qui s'occupait de l'indexation des données, mon stage avait pour but de répondre à une problématique d'espace mémoire. J'ai ainsi étudié des données RDF sur des données biologiques et chimiques, et j'ai testé différents composants et outils pour faire de l'indexation de données et de réaliser un prototype de pipeline d'indexation avec l'outil le plus adapté.

L'analyse de la consolidation box de CloudView, d'Apollo et de Virtuoso m'a permis de mettre en évidence les avantages et les inconvénients de chaque outil en terme de facilité d'utilisation, de volume de données et de performances. Les performances mesurées montraient qu'Apollo était l'outil le plus performant, devant l'AM-Store et Virtuoso. Cependant, l'outil était encore instable et nous avons donc choisi de réaliser le prototype du pipeline d'indexation sur Virtuoso.

Ce prototype de pipeline d'indexation consiste en 2 éléments : un parser et un connecteur. Ces éléments ont été développés en Java selon une architecture bien définie pour répondre aux besoins de l'application et pour être les plus performants possibles. Bien qu'une première phase d'amélioration ait déjà été effectuée, il est encore nécessaire de poursuivre l'amélioration du pipeline et des performances des outils internes pour pouvoir un jour être capable de les utiliser.

Ce stage m'a permis découvrir le cadre d'une grande entreprise et des contraintes qui en découlent - choix technologiques imposés, processus à respecter. De plus, cela m'a permis de mettre davantage en évidence mon intérêt pour la R&D et le développement de prototypes. En effet, cette partie du stage m'a fortement plu et j'ai apprécié de me retrouver confronté à des problématiques industrielles innovantes avec de fortes contraintes.

Annexe 1 : Requête SPARQL des Substances

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX cco: <http://rdf.ebi.ac.uk/terms/chembl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sio: <http://semanticscience.org/resource/>

SELECT ?s ?prefLabel ?altLabel ?exactMatch ?chemblId ?xref ?substanceType
?canonical_smiles_value ?standard_inchi_value ?standard_inchi_key_value
WHERE {
        ?s rdf:type cco:Substance .
        ?s skos:prefLabel ?prefLabel .
        ?s skos:altLabel ?altLabel .
        ?s skos:exactMatch ?exactMatch .
        ?s cco:chemblId ?chemblId .
        ?s cco:moleculeXref ?xref .
        ?s cco:substanceType ?substanceType .

        ?s sio:SIO_000008 ?canonical_smiles .
    ?canonical_smiles rdf:type sio:CHEMINF_000018 .
    ?canonical_smiles sio:SIO_000300 ?canonical_smiles_value .

    ?s sio:SIO_000008 ?standard_inchi .
    ?standard_inchi rdf:type sio:CHEMINF_000113 .
    ?standard_inchi sio:SIO_000300 ?standard_inchi_value .

    ?s sio:SIO_000008 ?standard_inchi_key .
    ?standard_inchi_key rdf:type sio:CHEMINF_000059 .
    ?standard_inchi_key sio:SIO_000300 ?standard_inchi_key_value .
}
```

Listing 4.4 – Cinquième requête pour récupérer les substances