Filtrage, stockage et raisonnement sur de grands volumes de triplets RDF ordonnancés

Jérémy Lhez

Gestion de flux RDF et raisonnement : sémantisation, sérialisation et profilage

Jérémy Lhez
Université Paris-Est Marne la Vallée
Laboratoire Informatique Gaspard Monge

13 novembre 2018

Résumé

Avec le développement et la multiplication des appareils connectés dans tous les domaines, de nouvelles solutions pour le traitement de flux de données ont vu le jour. Cette thèse s'inscrit dans ce contexte : elle a été réalisée dans le cadre du projet FUI Waves, une plateforme de traitement de flux distribués. Le cas d'usage pour le développement a été la gestion des données provenant d'un réseau de distribution d'eau potable, plus précisément la détection d'anomalies dans les mesures de qualité et leur contextualisation par rapport à des données extérieures.

Plusieurs contributions ont été réalisées et intégrées à différentes étapes du projet, leur évaluation et les publications liées témoignant de leur pertinence. Celles-ci se basent sur une ontologie que j'ai spécifiée depuis des échanges avec les experts du domaine travaillant chez le partenaire métier du projet. L'utilisation de données géographiques a permis de réaliser un système de profilage visant à améliorer le processus de contextualisation des erreurs. Un encodage de l'ontologie adapté au traitement de flux de données RDF a été développé pour supporter les inférences de RDFS enrichies de `owl:sameAs`. Conjointement, un formalisme compressé de représentation des flux (PatBin) a été conçu et implanté dans la plateforme. Il se fonde sur la régularité des motifs des flux entrants. Enfin, un langage de requêtage a été développé à partir de ce formalisme. Il intègre une stratégie de raisonnement se basant sur la matérialisation et la réécriture de requêtes. Enfin, à partir de déductions provenant d'un d'apprentissage automatique, un outil de génération de requêtes a été implanté. Ces différentes contributions ont été évaluées sur des jeux de données concrets du domaine ainsi que sur des jeux d'essai synthétiques.

Résumé

With the developpement and the expansion of connected devices in every domain, several projects on stream processing have been developped. This thesis has been realized as part of the FUI Waves, a reasoning stream processing engine distributed. The use case for the developement was the processing of data streamed from a potable water distribution network, more specifically the detection of anomalies in the quality measures and their contextualisation using external data.

Several contributions have been realized and integrated in different stages of the project, with evaluations and publications witnessing their relevance. These contributions use an ontology that has been designed thanks to a collaboration with domain experts working for our water data management project partner. The use of geographical data allowed to realize a profiling system aiming at improving the anomaly contextualisation process. An ontology encoding approach, adapted to RDF stream processing, has been developed to support RDFS inferences enriched with `owl:sameAs`. Conjointly, a compressed formalism (PatBin) has been designed to represent streams. PatBin is based on the regularity of patterns found in incoming streams. Moreover, a query language has been conceived from PatBin, namely PatBinQL. It integrates a reasoning strategy that combines both materialization and query rewritting. Finally, given deductions coming from a Waves machine learning component, a query generation tool has been developed. These diferent contributions have been evaluated on both real-world and synthetic datasets.

Remerciements

Je tiens à remercier spécialement mon encadrant, Olivier Curé, pour toute l'aide qu'il m'a apporté du début à la fin de ces quatre années, que ce soit pour l'écriture, le développement, les pistes de recherche, et bien d'autres choses. Je remercie également nos partenaires au sein des différents organismes ayant contribué au projet Waves avec lesquels j'ai eu l'occasion de travailler, particulièrement Xiangnan Ren et Badre Belabbess, pour nos nombreux articles rédigés en association. Au sein de l'équipe du LIGM, je remercie les nombreux enseignants que j'ai pu assister pour les heures de cours que j'ai donné, notamment Claire David, qui m'a beaucoup conseillé et accompagné pour l'organisation de mon poste d'ATER en quatrième année. Je remercie évidemment tous ceux qui liront cette thèse, évaluateurs, rapporteurs, relecteurs ou simples curieux, pour leur appréciation de mon travail. Enfin, je tiens à saluer ma famille et mes amis, pour m'avoir soutenu tout au long de ces années, et m'avoir aidé pour la relecture finale.

Abréviations

API	Application Programming Interface
CEP	Complex Event Processing
CUAHSI	Consortium of Universities for the Advancement of Hydrologic Science, Inc.
DRed	Delete and Rederive
EGC	Extraction et Gestion de Connaissances
ERI	Efficient RDF Interchange
IdO	Internet des Objets
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
LOD	Linked Open Data
NLP	Natural Language Processing
OGC	Open Geospatial Consortium
OSM	OpenStreetMap
OWL	Web Ontology Language
POI	Point Of Interest
QUDT	Quantity, Unit, Dimension and Type
RDF	Resource Description Framework
RDSZ	RDF Differential Stream Compressor based on Zlib

RGPD	Règlement européen de protection des données personnelles
RSS	Rich Site Summary
SIG	Système d'Information Géographique
SKOS	Simple Knowledge Organization System
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SSN	Semantic Sensor Network
SWE	Sensor Web Enablement
SWEET	Semantic Web for Earth and Environmental Technology
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Liste des tableaux

Table des figures

Table des matières

1

Introduction

L'émergence et l'expansion de l'internet des objets (IdO) permet de développer de nouvelles applications et ouvre de nouvelles branches de recherche. Avec l'augmentation des réseaux de capteurs et de leur précision, il devient possible et même nécessaire de créer des méthodes d'analyse afin de détecter des anomalies dans les mesures. Il s'agit précisément de l'un des objectif principaux du projet FUI [1] Waves, dans le cadre duquel j'ai réalisé cette thèse. Le but du projet est d'identifier des anomalies (*e.g.*, des fuites) dans un réseau de distribution d'eau potable, et d'identifier celles qui ne sont pas liées à des problèmes matériels en se basant sur l'étude d'événements extérieurs.

Waves [2] a été appliqué au cas du pilotage hydraulique d'un réseau. La raréfaction des ressources d'eau potable dans certaines parties du monde, croisée avec la croissance de la population mondiale, et simultanément une concentration de cette population dans des mégapoles concourent à ce que l'eau soit une ressource précieuse, et éventuellement disputée. Utiliser la capacité de traitement de la plateforme pour mieux gérer le pilotage des réseaux en utilisant tous les nouveaux flux de données hétérogènes disponibles, non traités actuellement et contribuer ainsi à économiser de l'eau potable est également en soi un objectif du projet. L'application de Waves à ce cas d'usage a essentiellement pour objectif de vérifier la généricité du prototype de la plateforme et sa capacité à s'intégrer facilement dans un SI industriel existant et souvent contraint. Le principal verrou du projet, et qui me concerne directement, est le raisonnement dans un contexte de streaming. Ce dernier

1. Fonds Unique Interministériel
2. https ://www.waves-rsp.org/

a comme objectif de produire de nouvelles connaissances à partir d'ontologies statiques (car étant amenées à être rarement modifiées) et des données dynamiques. Les considérations de la vélocité et de la volumétrie des données impliquent une optimisation des opérations de raisonnement. Ainsi il est indispensable de recevoir des données brutes, de les sémantiser et de les exploiter au sein d'un mécanisme d'inférence dans un temps restreint et ce même si les données arrivent à une très grande vitesse et des volumes très importants. Les partenaires du projet sont Atos (coordinateur), Data Publica, l'ISEP, le LIGM et Suez.

Le partenaire du projet apportant son expertise métier et ses jeux de données dans la gestion de tels réseaux est Suez, une multinationale dont le réseau d'eau potable en France est constitué de 100 000km de canalisations, 12 millions de clients et plus de 3 000 capteurs. Ses infrastructures sont en croissance constante (en particulier le nombre de capteurs déployés), et la compagnie gère des réseaux de différentes échelles au niveau international.

De nos jours, les fuites d'eau potable dans le monde représentent une perte d'environ 32 milliards de dollars chaque année. Outre les pertes financières, elles posent également des problèmes de maintenance, puisque 90% d'entre elles sont invisibles, et sont très difficiles à identifier et donc à réparer. Suez bénéficie depuis plusieurs années d'une gamme de logiciels baptisée Aquadvanced, dont l'un d'eux (AQUADVANCED® Réseaux d'eau) permet une gestion efficace des données provenant du réseau ; il s'agit avant tout d'une solution d'optimisation. Cet outil utilise des méthodes statistiques et nécessite l'évaluation d'un opérateur afin de déterminer la validité des anomalies détectées ; en cela, l'outil est limité et non automatisé. Il est conçu uniquement pour le cas d'usage de gestion de réseaux de distribution d'eau potable et ne peut être employé pour d'autres types de réseaux ou flux de données. D'ailleurs, l'utilisation exclusive d'approches statistiques, en opposition à l'exploitation de la sémantique, ne permet pas à Aquadvanced d'effectuer certains traitements, tels que l'interprétation du contexte d'une mesure anormale.

Le projet Waves reprend l'objectif de détection d'anomalies en fonction de nombreux critères (fuite d'eau, goût/odeur, niveaux de produits chimiques, pH, etc.), mais se base sur la sémantique, et vise en plus à rattacher des événements extérieurs aux anomalies détectées, afin d'éviter les faux-positifs (dans notre cas, un événement détecté en tant que fuite, alors que ce n'en est pas une). Un point prépondérant pour qualifier les pertinences des origines probables est la gestion des mesures problématiques identifiées, en étu-

diant leur contexte géographique et spatial, ainsi que la présence de facteurs extérieurs (météorologie, climat...). Les capteurs ont une position géographique connue et effectuent des mesures à des moments prédéterminés, qui peuvent être impactées par différents paramètres (événements, météo, etc.). Par exemple, l'épreuve sportive de course à pied Paris-Versailles se déroule chaque année à la fin de l'été, et traverse plusieurs zones urbaines entre Paris et Versailles. Il s'agit d'un événement annuel, régulier, qui peut être la cause de mesures de consommations inhabituelle (car les participants ont besoin de s'hydrater). Les composantes géographiques et temporelles des événements les rendent plus ou moins faciles à identifier en tant que source d'anomalies. Autre exemple, le Hellfest, l'un des plus grands festivals de musique français, est aussi un événement annuel, qui se déroule dans une zone majoritairement agricole, et regroupe des dizaines de milliers de personnes en juin ; c'est une combinaison de facteurs qui peut impacter très fortement la consommation en eau potable. Dans ces deux cas toutefois, la surconsommation ou les dépassements de seuils dans les mesures ne sont pas liés à un problème matériel (*e.g.*, rupture d'une canalisation), mais à une situation exceptionnelle.

La première contribution évoquée dans cette thèse est un système de profilage de zones géographiques qui vise à déterminer les proportions de divers types de surfaces dans un secteur spécifique (zone naturelle, industrielle, résidentielle...). L'algorithme se base sur deux types de données cartographiques, les points d'intérêt et les polygones, et calcule le résultat le plus approprié au secteur. Ces informations sont extraites d'OpenStreetMap, un projet de carte libre du monde, qui s'avère le plus complet et le plus précis (il sera détaillé, avec les alternatives que nous avons considérées, dans les chapitres suivants). L'objectif est de déterminer le type de surface majoritaire (zone agricole, résidentielle, industrielle...), afin de rechercher les événements potentiellement sources d'anomalie les plus pertinents, et d'ajuster le score de responsabilité qu'on leur attribue. En effet, selon le type de surface majoritaire, la consommation d'un secteur sera plus ou moins affectée par certains événements : un marathon aura un fort impact de consommation si il passe dans une zone naturelle, d'ordinaire sans consommateurs. Le profilage en lui-même peut être ajusté en utilisant divers fichiers de configuration, la hiérarchie de données cartographiques pertinentes pour son élaboration, ou encore en modifiant la méthode de combinaison des approches par points d'intérêt et polygones. La localisation des capteurs, donnée statique connue dans la plupart des projets, est primordiale à l'établissement du profilage.

Une autre particularité de la gestion de l'IdO est la gestion des flux de

données des mesures émises par les capteurs localisés sur le réseau (*e.g.*, dans notre projet : pression, débit, turbidité, pH, niveau de chlore, etc.). Il s'agit donc de données dynamiques, en comparaison aux données contextuelles géographiques, qui sont elles relativement statiques et ne changent qu'occasionnellement, *e.g.*, transformation d'une zone agricole en une zone immobilière. A l'émission, les capteurs émettent ces données sous différents formats (généralement CSV, bien que d'autres formats soient possibles) ; il est nécessaire de les convertir vers un format commun pour un traitement plus efficace. C'est lors de cette conversion que nous effectuons une sémantisation des données, *i.e.*, nous lions les données à de la connaissance du domaine, qui permettra d'interpréter, et donc de raisonner, sur les flux de données émis par le réseau. Pour représenter les flux et les connaissances, notre choix s'est porté sur les technologies du Web Sémantique. Elles vont nous permettre de supporter une intégration de données efficace en bénéficiant de nombreux jeux de données ouverts, de standards reconnus par le W3C (RDF, SPARQL, RDFS, OWL), de librairies bien établies dans la communauté (Jena) et d'applications diverses (RDF stores, raisonneurs). Tout ces atouts ne sont pas limités à un seul cas d'usage et contribuent également à la portabilité du projet.

Afin de sémantiser les données, nous avons conçu une ontologie à partir de bases de connaissances existantes propres à des domaines spécifiques ; le détail de nos sources est disponible dans le chapitre 2. Chacune de ces bases de connaissances a été étudiée en détail afin de juger de son expressivité et de ses limites : dans chaque cas, les concepts et propriétés pertinents pour le projet ont été retenus et combinés afin d'obtenir une représentation adéquate et couvrant tous les aspects de notre cas d'usage. Il a fallu en utiliser plusieurs afin de couvrir les différentes spécificités, et seule une petite partie de chaque ontologie a été retenue : la représentation de la gestion de qualité de l'eau potable est quelque chose de spécifique, il ne s'agit pas d'un ensemble de notion très étendu. Néanmoins, ma tâche a été d'examiner chacune des ontologies afin de juger de leur contenu, d'identifier quelle partie pouvait être réutilisée dans le projet, et comment les interconnecter. Dans certains cas, lorsque plusieurs choix étaient possibles, il a fallu évaluer quel choix était le plus pertinent pour le projet. Il faut également mentionner que certaines ontologies n'étaient pas accessibles, ou ont du être converties depuis leur format originel. Malgré le caractère *open source* du projet, notre partenaire expert du cas d'usage n'a pas tenu à ce que l'ensemble de l'ontologie soit fourni dans le projet ; toutefois, un échantillon des données statiques et dynamiques est disponible en annexe B.

Dans Waves, les flux sont distribués pour permettre un passage à l'échelle et supporter un traitement plus efficace, afin de raisonner et de requêter sur les données de façon optimisée ; mais il faut sélectionner une sérialisation RDF adaptée et efficace pour toutes ces caractéristiques. La deuxième contribution majeure de ma thèse concerne la définition d'un tel format, dénoté PatBin (une compression des termes *Pattern* et *Binding*), une sérialisation prenant en compte ces différents besoins. Il s'agit d'une méthode de compression de graphes RDF, avec support natif d'inférence, qui présente de nombreux avantages par rapport aux systèmes existants. PatBin est basé sur un système d'encodage existant, que nous avons étendu pour de meilleures performances. Dans une dernière contribution, je présente une extension liée à PatBin. Elle correspond à un langage de requête, baptisé PatBinQL : il permet de requêter sur des données encodées sans avoir à décompresser les graphes. Elle s'intègre dans un composant de Waves, RAMSSES, qui apprend de manière automatique depuis des flux. A partir de cet apprentissage, il est possible de générer automatiquement des requêtes au format PatBinQL. Cette génération aide les opérateurs de la plateforme Waves pour la définition de requêtes pertinentes à exécuter sur la plateforme ; elle se base sur l'ontologie développée dans le cadre du projet pour extraire des informations pertinentes d'un fichier fourni selon un certain format, et tente de produire une requête pertinente en fonction des éléments identifiés.

La plateforme ainsi développée dans le cadre du projet FUI Waves est générique, distribuée et portable, et a donc un certain potentiel pour l'ensemble des applications impliquant de l'IdO et nécessitant un traitement sémantique. La figure 1 donne un aperçu de l'architecture globale du projet Waves. Les données provenant des capteurs sont dans un premier temps filtrées pour retirer les valeurs manquantes liées à des problèmes sur le réseau (panne de capteur, transmission impossible...), puis converties et compressées suivant notre encodage PatBin, après quoi des requêtes (éventuellement générées grâce à notre approche) sont exécutées pour identifier des anomalies potentielles. Un module d'apprentissage agit en parallèle pour rechercher les événements potentiellement à l'origine des anomalies, et ajuste ses décisions en fonction de facteurs spatio-temporels, d'analyse de texte, et de notre système de profilage. Le résultat final, comportant l'anomalie et sa contextualisation, sont enfin soumis aux opérateurs pour leur appréciation.

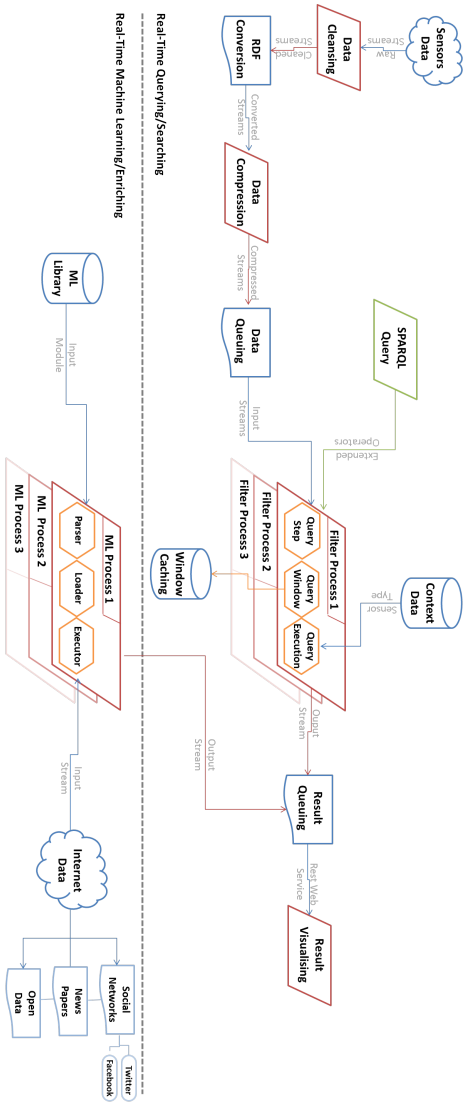


FIGURE 1 – Architecture du projet Waves

2

Connaissances requises

Ce chapitre présente les connaissances requises pour la bonne compréhension des contributions détaillées dans cette thèse. L'intégralité des outils présentés est liée à ces connaissances, en les réutilisant, les améliorant, ou même en s'en servant comme base pour l'implémentation. La première section traite des technologies du web sémantique, qui sont au cœur du projet puisqu'il s'agit de raisonner sur des flux de données. La partie suivante détaille le Linked Open Data, un vaste ensemble de bases de connaissances libres d'accès en ligne, qui est utilisé pour évaluer la pertinence des origines potentielles d'anomalies dans Waves ; d'autres sources de données de nature cartographiques sont listées dans la section suivante, et sont utilisées dans le même but. Dans les sections suivantes se trouve la liste des sources de données événementielles, qui doivent être analysées et traitées afin d'identifier les origines potentielles des anomalies détectées par le projet. Enfin vient la présentation de LiteMat, un système d'encodage pour les composants des graphes de connaissances qui a été développé au sein du LIGM. J'ai utilisé ce projet dans le développement de la sérialisation PatBin ainsi que le requêtage sur PatBinQL. J'ai également étendu PatBin pour le support de la propriété `owl:sameAs`, qui permet d'idetifier des concepts différents faisant référence à un même élément, comme expliqué par la suite (section 5).

2.1 Technologies du web sémantique

Le web sémantique est une extension du web actuel : il est parfois nommé web 3.0 et correspond à une évolution majeure des techniques et usages du

W3C (World Wide Web Consortium).

A l'origine, le web était constitué de pages simples, avec pour seul objectif d'afficher de l'information ; ce n'était que les débuts d'internet, avec des débits faibles et des machines peu performantes. Après plusieurs années, on a commencé à parler de web 2.0, la première évolution majeure du web, avec pour caractéristique principale l'interaction avec les utilisateurs. Cela correspond à l'essor des réseaux sociaux, des wikis et du "*crowdsourcing*". Cette transformation d'un web où l'internaute passait essentiellement son temps à lire du contenu à un web où il fournit directement (ou indirectement) du contenu est à l'origine du mouvement *Big Data*, où d'importantes quantités de données doivent être gérées.

Le web sémantique, tel que défini par Tim Berners Lee [1], est l'évolution du web 1.0. Son innovation principale est de permettre la réutilisation de données, en en facilitant la recherche, la combinaison et l'utilisation. Pour cela, les données disponibles sont organisées en un réseau sémantique, une structure sémantique organisée par le biais de métadonnées. Les métadonnées sont des données décrivant d'autres données : ainsi, on peut obtenir des informations sur chaque donnée annotée, ce qui facilite sa recherche (par exemple, on peut spécifier que la donnée Steven Spielberg, associée à un film, correspond à un nom de réalisateur). La sémantisation des données facilite leur utilisation à la fois pour l'utilisateur et pour la machine.

Une ontologie est constituée d'un ensemble de termes structurés permettant de représenter des connaissances ; elle peut être modélisée sous forme de graphes et réemployée librement par les utilisateurs [2]. Pour des projets importants, il est possible d'étendre une ontologie en la combinant avec des concepts issus d'autres ontologies, afin de créer un modèle de données qui convient au cas d'utilisation. Des règles peuvent être définies afin d'établir les liens pouvant être effectués, afin de conserver une structure logique et correcte. De plus, il est possible d'extraire des informations d'une ontologie de manière ciblée en fonction de certains critères. La figure 2 montre un exemple d'ontologie simple, permettant de représenter les ressources humaines au sein d'une université. Il ne s'agit que d'une hiérarchie, mais elle peut être réemployée ou étendue de diverses façons. Par exemple, on pourrait s'en servir comme base pour représenter des cours, en précisant que des cours ont un responsable et un professeur en charge, et s'applique à des étudiants.

L'ontologie représente donc la conceptualisation de connaissances, ce que l'on appelle la Tbox (terminologie). A cela s'ajoutent les assertions, les déclarations liées à l'ontologie, qui sont regroupées dans ce que l'on appelle la

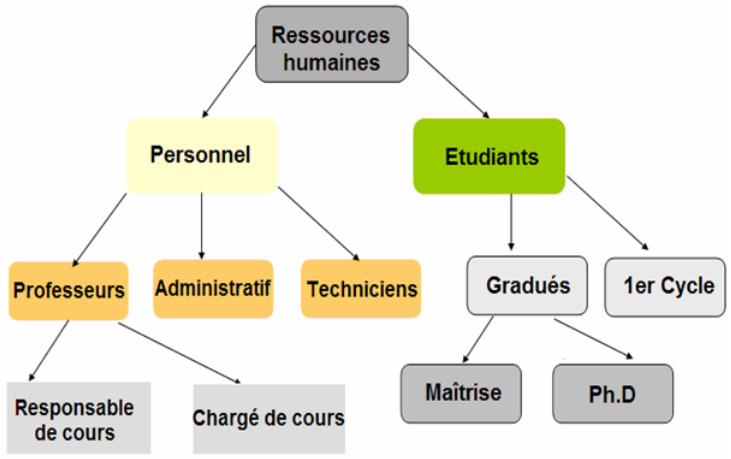


FIGURE 2 – Exemple simple d'ontologie

Abox (assertions). Une base de connaissances est donc formée d'une Abox et d'une Tbox : elle contient des graphes de connaissances, ainsi que les règles permettant de les exploiter. Si l'on reprend l'exemple figure 2, la Tbox regroupera les informations représentées, indiquant par exemple qu'un professeur fait partie du personnel, et que personnel est disjoint d'étudiants. Avec la Abox, on pourra préciser que Alice est une professeur. En utilisant les règles de la Tbox, on pourra raisonner pour déduire qu'Alice fait donc partie du personnel, mais qu'elle n'est pas une élève.

Le format RDF (Resource Description Framework) est le langage de base du web sémantique. En tant que modèle de données, il permet d'établir un graphe représentant les ressources du web et leurs métadonnées. Un document RDF représente les informations sous forme de triplets composés d'un sujet, d'un prédicat et d'un objet, dénoté (s, p, o). Le sujet représente une ressource, à laquelle une propriété peut être appliquée, le prédicat, et cette propriété a pour valeur un objet. Les concepts et les propriétés sont représentés par des URI (*Uniformed Resource Identifier*, les URL sont des sous-ensembles d'URI), représentant une ressource qu'ils définissent, les littéraux sont des valeurs de format standardisé (numérique, chaîne de caractère, date, etc). Certains éléments peuvent être représentés par des nœuds vides, lorsqu'ils n'identifient pas de ressources spécifiques. La signature du langage, soit les combinaisons d'éléments valides permettant de représenter des triplets RDF, est $U \cup B \times U \times U \cup B \cup L$. Avec L, U et B respectivement des

(a) RDF/XML

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
<rdf:Description rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">
    <dc:title>Tony Benn</dc:title>
    <dc:publisher>Wikipedia</dc:publisher>
</rdf:Description>
</rdf:RDF>
```

(b) Turtle

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://en.wikipedia.org/wiki/Tony_Benn>
dc:title "Tony Benn" ;
    dc:publisher "Wikipedia" .
```

(c) Graphe

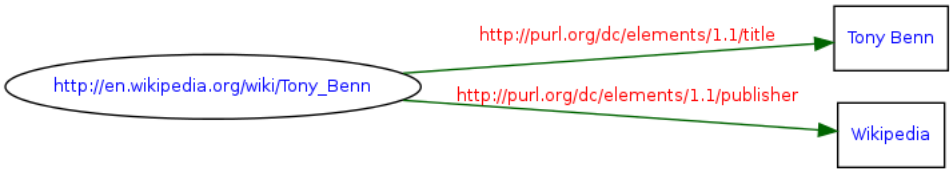


FIGURE 3 – Exemples de représentations RDF

ensembles de littéraux, d'URIs et de nœuds vides disjoints. Les nœuds vides sont disjoints car ils représentent des notions non-définies.

La figure 3 donne un exemple de représentation RDF sous trois formats différents. Il est possible de représenter des triplets RDF au sein de données XML (Extensible Markup Language), grâce à l'ajout de métadonnées : cela donne la syntaxe RDF/XML (exemple a). Il existe d'autres formats, comme la syntaxe Turtle (exemple (b)), où les triplets sont représentés directement, et aucune transformation n'est nécessaire : le sujet, le prédicat et l'objet sont définis les uns à la suite des autres (séparés par un "."), et il n'est pas nécessaire de gérer des balises comme en XML. Pour une représentation plus visuelle, on peut également modéliser le graphe RDF (exemple (c)), manuellement ou via divers outils.

La structure du format RDF est très générique et sert de base à divers vocabulaires et schémas dédiés à des cas d'usage précis, qui peuvent

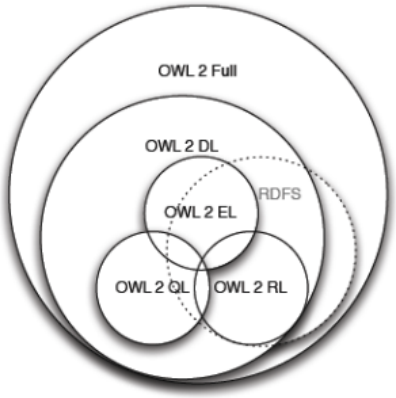


FIGURE 4 – Comparaison d'expressivité des fragments de OWL

être réutilisés et combinés pour servir à d'autres applications. Parmi les plus utilisés, le vocabulaire RDF Schema (RDFS) permet de définir des hiérarchies de concepts et de propriétés en utilisant respectivement les propriétés `rdfs:subClassOf` et `rdfs:subPropertyOf`. De même, il permet de spécifier les domaine et co-domaine des propriétés, en exploitant respectivement les propriétés `rdfs:domain` et `rdfs:range`. Ce langage sera le plus adapté pour représenter l'ontologie de la figure 2, par exemple.

OWL (Web Ontology Language) représente une famille des langages de représentations de connaissances essentiellement basée sur les logiques de description [3]. Elle peut être divisée en plusieurs fragments, chacun avec une expressivité qui lui est propre. Par exemple, OWL DL a été conçu pour fournir une expressivité maximum, tout en conservant la complétude, la décidabilité, et l'application de raisonnements d'algorithmes. OWL Lite, en revanche, est une version simple pour les usages nécessitant une classification hiérarchique et des contraintes simples. Avec l'apparition de OWL2 (en 2012), 3 fragments ayant une complexité polynomiale sont apparus. Il se nomment OWLRL, OWLEL et OWLQL. Par contre, OWL Full n'est pas décidable. La figure 4 présente une comparaison de l'expressivité des différents fragments de OWL, tout en situant le langage RDFS.

S'appuyant sur le modèle de données RDF, le SKOS, ou système simple d'organisation des connaissances (*Simple Knowledge Organization System*), permet de représenter des thésaurus ou des classifications. Il s'agit d'une recommandation du W3C ayant pour but de permettre la publication simple de vocabulaires structurés, s'intégrant dans des environnements sémantiques

qui utilisent des langages RDF, tel que OWL. A la différence de ce dernier, SKOS est une alternative plus simple, nécessitant un niveau d'expertise moindre dans la construction d'ontologies, tout en permettant une transition vers les technologies sémantiques. Le résultat obtenu en RDF permet d'éventuelles extensions ou intégrations ultérieures par la suite, si besoin.

Le langage SPARQL [4] (SPARQL Protocol and RDF Query Language) est un langage de requêtes qui permet la recherche, l'ajout, la modification et la suppression de données RDF. Tout comme le langage SQL (Structured Query Language), il s'agit d'un langage de requête déclaratif. Néanmoins, l'utilisation de bases de connaissances permet d'obtenir des données/connaissances implicites depuis des connaissances explicites par le biais de raisonnement. On retrouve donc des mots clés communs comme SELECT, FROM et WHERE dans les deux langages. Dans SPARQL, la clause WHERE est a minima composée d'un motif de graphe (graph pattern) dont la signature correspond à $U \cup B \cup V \times U \cup V \times U \cup B \cup L \cup V$ où U, B, L sont identiques à notre définition précédente de RDF, et V est un ensemble de variables disjointes de U, B et L. Un ensemble de triplets de ce type est nommé Basic Graph Pattern (BGP).

Question :
Quels sont les noms et les adresses email des personnes dans la base de connaissance ?

Requête correspondante :

```
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>

SELECT *
WHERE {
    ?person foaf:name ?name .
    ?person foaf:mbox ?email .
}
```

(Dans la requête, ?person ?name et ?email sont des variables)

FIGURE 5 – Exemple de requête SPARQL simple

La figure 5 présente une requête SPARQL simple, l'équivalent d'une sélection en SQL. L'utilisation de triplets permet de spécifier à la fois le résultat à afficher et les critères de recherche. Ici, le résultat est un ensemble de tuples, avec toutes les personnes ayant un nom et une adresse mail dans la base de connaissance. SPARQL a connu une extension (SPARQL 1.1), et a été utilisé à de nombreuses occasions comme base de départ pour le traitement de flux. Nous reviendrons en détail sur cet aspect en section 3.2.

Il y a donc de nombreux outils du web sémantique permettant de représenter et de manipuler des connaissances, que ce soit pour établir des liens entre diverses informations, ou pour rechercher des informations précises. Malgré les avantages du web sémantique, on retrouve des problématiques liées à la manipulation de vastes quantités d'informations, du fait de la nature même du raisonnement. Certaines informations sont représentées de façon directe, d'autres doivent être déduite. Par exemple, si une personne A a pour ancêtre une personne B, et que B a pour ancêtre C, alors A a pour ancêtre C, car "avoir un ancêtre" est une propriété transitive. Mais cette information doit être déduite ; et cette déduction va en entraîner d'autres, et ainsi de suite. Dans les bases de connaissances volumineuses, c'est une problématique importante. Même dans le cas du traitement de flux, comme dans notre projet, des connaissances peuvent être déduites de certains événements, mais il faut pouvoir les conserver pour les mesures suivantes. Cela implique en plus de décider de quand leur validité expire, voire si il est possible de les re-dériver à partir d'autres informations. Raisonner sur des données RDF est toujours une importante problématique de recherche à l'heure actuelle.

2.2 Le Linked Open Data

Le *Linked Open Data* (LOD), ou données ouvertes et liées en français, représente l'association de données liées et de données libres accessibles sur internet. C'est un domaine très vaste : en 2011, il regroupait déjà plus de 31 milliards de triplets RDF, interconnectés par plus de 504 millions de liens RDF. En 2015, le nombre de sources de données avait déjà quadruplé. La figure 6 propose une représentation graphique du LOD en 2017 où chaque nœud du graphe représente lui-même un graphe.

L'interconnexion des données permet d'avoir accès à diverses sources d'informations de manière directe, et leur structure les rend requêtables aisément via des API ou des langages de requête adaptés (on peut citer Virtuoso [5],

un logiciel largement utilisé par les applications du web sémantique). Leur aspect libre assure une utilisation par un maximum de personnes et d'applications. Nous allons détailler ici les principales sources de données du LOD, en citant notamment celles utilisées pour notre projet.

2.2.1 Bases de connaissances

Les bases de connaissances contiennent des informations sur divers sujets, qui sont enregistrées de manière structurée, suivant un schéma bien précis. Il existe divers moyen de les requêter pour en extraire des informations pertinentes. Nous allons prendre l'exemple de trois bases de connaissances parmi les plus emblématiques ; elle sont les plus connues en raison de leur valeur, leur domaine, leur nombre d'utilisateur et la validité des informations qu'elles contiennent.

DBpedia [6] est un projet universitaire et communautaire permettant l'exploration et l'extraction de données dérivées de wikipedia. Les données essentielles de wikipedia, contenues dans les infobox, sont représentées suivant une structure sémantique précise : pour chaque page, les informations disponibles sont ainsi organisées et annotées pour être requêtées efficacement. Les données de DBpedia sont globalement statiques, exception faites des mises à jour et ajouts de la part de la communauté. Cette base regroupe des connaissances statiques et générales sur un très grand nombre de sujets ; elle peut être utilisée pour faire le lien entre diverses sources (grâce aux références), ou fournir des compléments d'information pour préciser un raisonnement en tant que base de connaissance externe.

Wikidata [7] est un projet sœur de wikipedia, édité collaborativement et hébergée par la Wikimedia [1] : la base centralise les données utilisées par divers projets de la fondation. Le projet est un successeur de DBpedia, basé sur plusieurs sources au lieu d'une seule ; son objectif principal est l'amélioration de la représentation des données à grande échelle, pour faciliter l'extraction des données. Comme le projet tire ses données de différentes origines, il peut y avoir des problèmes de données dupliquées, absentes ou sous diverses versions.

1. https ://commons.wikimedia.org/wiki/Accueil

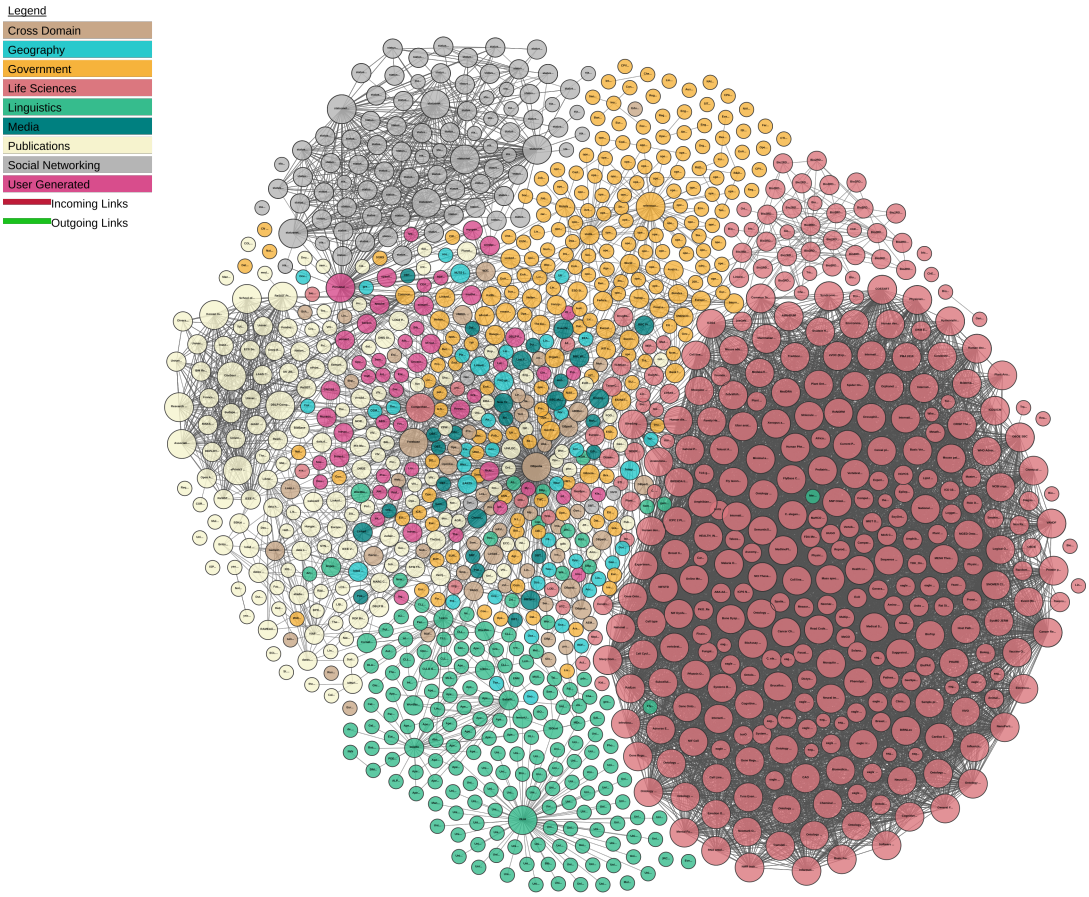


FIGURE 6 – Diagramme du cloud du Linking Open Data (2017)

2.2.2 Ontologies du domaine de Waves

Afin d'organiser les informations extraites dans notre projet, nous avions besoin d'une terminologie adaptée à notre cas d'usage, pour pouvoir requêter efficacement sur les données. Nous avons donc établi un état de l'art des ontologies disponibles en ligne qui nous semblaient pertinentes pour notre cas d'usage, afin de les combiner pour obtenir une représentation adaptée à nos attentes. Nous allons présenter celles qui nous semblent les plus pertinentes pour pourvoir représenter les connaissances concernant des réseaux de capteurs et les critères de qualité de l'eau potable.

L'ontologie SSN [8] (Semantic Sensor Network Ontology) est une référence : développée par un groupe du W3C, elle permet de décrire les capteurs et leurs observations. Il s'agit d'une ontologie globale, avec un haut niveau d'abstraction, conçue pour être associée à d'autres ontologies de domaine, pour mieux spécifier les concepts représentés. Ses concepts sont principalement liés aux informations physiques concernant le matériel (position du capteur, type, ...) et elle ne permet donc pas la gestion avancée des mesures.

L'OGC (*Open Geospatial Consortiunm*) a mis au point des standards pour les capteurs connectés (SWE : Sensor Web Enablement). Cela permet d'annoter aisément les capteurs et leurs mesures, et d'accéder par la suite à leurs données et aux métadonnées qui leurs sont associées. Plusieurs personnes ayant travaillé sur SSN sont également membres du consortium. Toutefois, ces standards ne sont pas alignés sur les technologies du web sémantique, et peuvent poser problème lors de la création et de la maintenance de graphes de données interconnectées. Dans notre projet, ou plusieurs sources de données sont utilisées pour notre ontologie, et ou de bonnes performances de raisonnement sont attendues, ce n'est pas une solution adaptée.

La terminologie SWEET [9] (Semantic Web for Earth and Environmental Technology), développée par la NASA, est constitué de plus de 6.000 concepts répartis en 200 ontologies OWL. Les concepts représentés sont répartis sous plusieurs catégories : les opérations (physiques, mathématiques, biologiques, chimiques), les relations (chimiques, physiques, humaines), les représentations (données, temps, espace), ... SWEET est un projet très étendu, mais comme les mesures des capteurs sont spécifiquement liées à la qualité de l'eau potable, les concepts fournis par la terminologie ne conviennent pas toujours.

Le consortium d'universités pour l'avancement de la science hydrolo-

gique [2] (CUAHSI) est une organisation de recherche composée de plus de cent universités américaines. Le groupe reçoit le soutien de la fondation nationale de la science pour leur travail sur la science de l'eau aux États Unis. Leur ontologie contient environ cinq mille concepts représentant les informations liées à des mesures (physiques, chimiques et biologiques) de qualité de l'eau, effectuées en un point et à un instant précis. Sa représentation tabulaire la rend simple à représenter et à utiliser : c'est cette ontologie que nous avons retenue afin de représenter les propriétés mesurées par les capteurs. En effet, les critères de qualité de l'eau qu'elle représente correspondent tout à fait à ceux employés pour l'eau potable.

QUDT [10] (pour quantités, unités, dimensions et types) est une organisation à but non lucratif créée afin de fournir des spécifications sémantiques pour les unités de mesure et les types de quantités ou de dimensions. L'ontologie est divisée en plusieurs modules pour chacune de ces spécifications, et nous l'avons utilisée pour modéliser les unités de mesures des capteurs.

D'autres projets moins significatifs ont été notés, mais rejetés en raison de leurs différences avec notre cas d'usage, ou de difficultés de réutilisation des données. Le projet InWaterSense [11] par exemple, a déployé un réseau de quatorze capteurs sans fil sur une rivière du Kosovo afin de pouvoir suivre en temps réel l'évolution de paramètres de qualité de l'eau. Un portail web permet d'accéder au réseau, aux mesures et aux positions des capteurs (certains sont statiques, d'autres dynamiques). Le projet présente de nombreuses similarités avec notre cas d'usage, mais l'ontologie employée n'est pas disponible en ligne. De plus, certains types de mesure diffèrent pour chaque cas (il n'y a pas de pression à mesurer dans une rivière).

2.3 Bases de données et de connaissances géographiques

Les bases de connaissances géographiques permettent d'obtenir diverses données cartographiques dans le cas de l'étude de lieux. Ce genre de données externes est très souvent utilisé conjointement à d'autres sources d'informations en tant que complément pour inférer de nouvelles connaissances. Ces dernières nous seront nécessaires pour supporter notre module de contextualisation géographique et les raisonnements associés.

2. https ://www.cuahsi.org/

HERE[3] est un logiciel de cartographie gratuit, accessible à partir d'un ordinateur, un téléphone ou d'une tablette. Il s'agit d'un système GPS (Global Positionning System) offrant une navigation vocale, pour les piétons et les conducteurs, dans plus de cent pays. Malheureusement, si l'application est gratuite et accessible à la plupart des systèmes d'exploitation, l'accès aux données elles-mêmes est payant.

GeoNames [12] est une base de connaissances géographiques gratuite, contenant plus de 8 millions de noms géographiques qui correspondent à plus de 6,5 millions de lieux sur terre. Ces noms sont classés en 9 catégories principales et 645 sous-catégories, et des données basiques sont associées à chacun de ces noms (latitude, longitude, code postal, ...). La base de données est communautaire : n'importe quel utilisateur peut effectuer des ajouts ou des modifications consultables. GeoNames fait également partie du LOD, mais c'est sa composante géographique qui est plus intéressante pour nous.

OpenStreetMap [13] (OSM) est un projet international fondé en 2004 dans le but de créer une carte libre du monde. Il s'agit également d'un projet communautaire : les utilisateurs enregistrés peuvent ajouter ou modifier des données géographiques afin d'enrichir la base, ce qui en fait un outil de plus en plus complet. En 2017, on recense plus de 3,5 milliards de points d'intérêt, et plus de 300 millions de polygones ; le nombre d'utilisateurs enregistrés a dépassé les trois millions fin 2016. Chaque point, tracé ou polygone ajouté contient en plus un ensemble d'étiquettes permettant de l'identifier (tant visuellement que par requêtes) et d'obtenir ses caractéristiques. La qualité des données provenant d'OSM est un sujet sensible [14], et il existe divers outils de comparaison cartographique en ligne (Geofabrik[4],...), qui montrent qu'OpenStreetMap est le projet de cartographie libre le plus complet et détaillé disponible ; c'est pourquoi nous l'utilisons dans Waves pour le traitement des sources d'anomalie.

2.4 Sources de données événementielles

Si des anomalies peuvent être détectées en analysant les données des capteurs, il est en revanche impossible d'en identifier l'origine avec les seules données du réseau, car aucune des informations sur le réseau n'est à l'origine de l'anomalie (sauf en cas de capteur endommagé, mais ce problème est

3. https ://wego.here.com
4. http ://tools.geofabrik.de/mc/

détectable dès le filtrage des données). Il est nécessaire d'étudier des flux de données événementielles extérieures, pour repérer des origines potentielles d'erreur (telles que des sinistres ou des rencontres sportives, culturelles), puis d'établir la pertinence de ces événements considérant le type d'anomalie (un incendie sera plus pertinent qu'une exposition, par exemple).

Les flux RSS (Rich Site Summary) provenant des sites d'informations constituent les sources de données les plus intéressantes. Elles sont structurées, fiables dans la plupart des cas, et leur contenu a un but purement informatif. Il existe des sources d'information à différents niveaux (international, régional, départemental...) et à diverses fréquences (quotidien, hebdomadaire...) ; la sélection des sources dépend donc du cas d'usage. Seuls les événements relativement importants sont recensés dans l'actualité, il n'est donc pas garanti de trouver des éléments pertinents ; toutefois, ceux qui sont identifiés seront généralement très riches en information.

OpenAgenda [5] est une solution d'agenda en ligne qui permet d'organiser et de diffuser des événements librement. Les agendas sont utilisables par tous, et il est possible d'accéder à leurs informations via une API ou bien directement depuis l'URL de l'agenda. Il existe un service similaire, Eventful [6], qui recense les événements (concerts, rencontres sportives, séances cinématographiques...) à proximité d'un lieu donné. Eventful compte plus de 22 millions d'utilisateurs enregistrés et son contenu est accessible de diverses façons : en ligne, par mail, sur téléphone ou via une API. Malheureusement, au cours du développement de notre projet, le site a été bloqué pour des raisons d'infractions au RGPD (Règlement européen de protection des données personnelles) et doit être modifié pour être à nouveau accessible.

Les réseaux sociaux, bien que moins fiables et moins structurés que les sources de données précédentes, constituent de véritables mines d'information. Twitter et Facebook, les deux acteurs principaux de ce domaine, génèrent chaque jour d'importantes quantités de données accessibles via divers outils [15] [16]. Une telle quantité d'information nécessite un traitement plus poussé afin d'extraire les données pertinentes à un cas d'usage spécifique. En effet, bien que les statuts facebook et les tweets soient bien plus concis qu'un article de journal, leur fiabilité est souvent douteuse. Par ailleurs, le traitement du texte des messages est également plus complexe (orthographe, homonymie...).

5. https ://openagenda.com/
6. http ://eventful.com/

A la différence des autres sources de données mentionnées dans cette section, DBpedia [6] ne représente pas un flux de données en soi. Toutefois, le projet contient certaines informations géolocalisées et dont on peut extraire une composante temporelle, *e.g.,* les fêtes nationales et/ou religieuses, des événements se déroulant dans certaines grandes villes, ou la page d'un événement régulier dont on connaît la date et la localisation.

OpenWeatherMap [7] est un service en ligne fournissant des données météorologiques de différentes origines : relevés météo officiels, données brutes provenant d'aéroports, de stations radar, etc. Il ne s'agit pas d'événements directement responsables d'anomalies (sauf en cas de conditions climatiques exceptionnelles), mais ces données permettent néanmoins d'ajuster le classements des sources pertinentes identifiées : *p.ex.*, une canicule améliorera le classement d'un événement sportif en extérieur.

2.5 Litemat, une compression intelligente

LiteMat est un système d'encodage pour les concepts et les propriétés des graphes RDF ; il permet de compresser et d'encoder la sémantique de leurs hiérarchies. Une première version de l'algorithme, se limitant à des bases de connaissances statiques et au support des inférences RDFS, a été présentée dans [17] et ne constitue pas une contribution de cette thèse ; toutefois, il constitue la base de départ de l'algorithme utilisé pour le raisonnement, et doit donc être détaillé précisément.

2.5.1 Principe de l'algorithme

LiteMat permet de compresser aussi bien la ABox (boîte assertionnelle, c'est-à-dire l'ensemble des instances) que la TBox (la boîte terminologique, c'est-à-dire la spécification du schéma) d'une base de connaissances, et s'applique aux propriétés et aux concepts (il s'agit toutefois d'encodages séparés). Le principe est d'attribuer à chaque concept ou propriété un identifiant numérique afin de le compresser. Chaque identifiant est généré à partir d'une structure binaire permettant de conserver les hiérarchies de propriétés et de concepts. Pour cela, la structure binaire d'un concept (respectivement d'une propriété) est préfixée par l'encodage de son super-concept (resp. de sa

7. https ://openweathermap.org/

super-propriété), en plus de l'identifiant binaire de la propriété ou du concept courant. Cela permet de conserver l'arborescence des concepts et propriétés.

Chaque structure binaire est donc composée de trois éléments : un préfixe, correspondant à l'identifiant de la structure parent, et un identifiant binaire qui lui est propre ; on attribue ce dernier de manière incrémentale. Le dernier élément est la série de 0 nécessaires à la normalisation des structures (afin qu'elles aient toutes le même nombre de bits). Les structures doivent donc être créées étape par étape, en commençant par la racine de la hiérarchie du graphe, pour construire les préfixes au fur et à mesure ; l'encodage est terminé lorsque toutes les feuilles du graphe ont un identifiant. La normalisation peut alors être réalisée, car on connaît la taille de la structure la plus longue. On peut alors transformer ces dernières en un identifiant entier numérique, plus compact.

Cet identifiant permet la réécriture de requêtes de manière simple, en utilisant un opérateur de filtrage. En effet, grâce à la manière dont LiteMat créé les identifiants, on sait que tous les identifiants des sous-classes d'un concept seront d'une valeur supérieure à l'identifiant du concept en question. Mais leur valeur sera également inférieure à celle de l'identifiant suivant du niveau supérieur dans la hiérarchie : cela nous permet d'établir un intervalle de valeurs qui contiendra obligatoirement tous les identifiants des sous-classes du concept d'origine (cela fonctionne de la même manière pour les propriétés). Cet intervalle permet de réécrire la requête SPARQL avec une clause FILTER, qui évite une explosion combinatoire de la taille de la réécriture et ainsi la rend plus lisible et compacte. Par ailleurs, en utilisant directement les identifiants des concepts et propriétés dans les données et les requêtes, il n'est pas nécessaire de décompresser : on gagne donc en temps d'exécution.

Cette méthode de compression peut devenir problématique lorsque les ontologies utilisées ne sont pas totalement statiques, mais peuvent être occasionnellement modifiées. En effet, le choix du nombre de bits pour l'identifiant binaire d'un concept (ou d'une propriété) dépend du nombre de concepts (respectivement de propriétés) déjà présents dans l'ontologie à un niveau donné de la hiérarchie correspondante : des ajouts éventuels à l'ontologie peuvent nécessiter l'ajout de bits supplémentaires, ce qui fausse toutes les structures binaires. Une approche naïve serait de recalculer l'encodage des concepts ou propriétés puis de ré-encoder tous les triplets de la ABox. Cette solution n'est pas viable s'il y a une haute fréquence des modifications (plusieurs par jour par exemple), car peu performante. Nous avons envisagé deux solutions différentes pour traiter ce genre de situation. Dans le cas où l'on connaît l'on-

tologie, il est envisageable de penser qu'on sait comment elle peut évoluer : il est donc possible d'ajouter (si nécessaire) un nombre de bits suffisant pour parer toute éventualité. Si on ignore les possibilités d'évolution de l'ontologie, nous proposons d'augmenter d'un certain facteur (par exemple doubler) le nombre de bits attribués de base. Cela permet d'avoir une grande marge pour les sous-concepts (et propriétés) nombreux, et une marge plus faible pour ceux qui le sont moins ; nous partons du principe que les arborescences peu nombreuses sont peu susceptibles de changer. Dans la suite de cette thèse, nous mettrons en évidence que la modification de la TBox pose moins de problème dans un contexte de gestion de flux.

2.5.2 Exemple d'utilisation

La figure 7 présente un cas d'usage de LiteMat. L'exemple est basé sur un cas d'usage du projet Waves. On peut y voir qu'un appareil de mesure (SensingDevice) appartient à la famille des capteurs (Sensor) ; un capteur lui-même est un objet physique (PhysicalObject) qui, de manière très générale, est également une sous-classe de `owl:Thing`. Il ne s'agit que d'une branche de l'ontologie : un moteur (Engine) est aussi un objet physique, mais on compte également des objets sociaux (SocialObject). Tous ces concepts sont extraits de l'ontologie SSN et de celle de DBpedia.

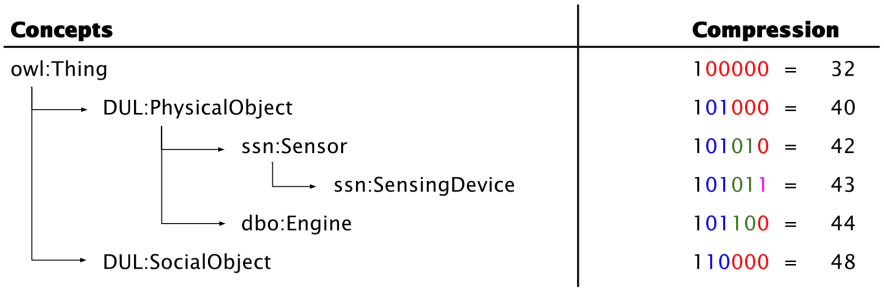


FIGURE 7 – Exemple d'encodage avec LiteMat

L'encodage commence avec `owl:Thing`, le super-concept de la hiérarchie de concept. Il n'y a qu'un seul concept, et on peut donc l'encoder avec un bit (en noir sur la figure) ; nous lui donnons la valeur 1. Les sous-concepts de Thing sont `DUL:PhysicalObject` et `DUL:SocialObject` : il y a deux

Requête de base

```
SELECT ?x
WHERE {
?x rdf:type DUL:PhysicalObject .
}
```

Réécriture LiteMat

```
SELECT ?x
WHERE {
?x rdf:type ?y .
    FILTER (?y >= 40 && ?y < 48)
}
```

Réécriture standard

```
SELECT ?x
WHERE {
?x rdf:type DUL:PhysicalObject
} UNION {
?x rdf:type ssn:Sensor
} UNION {
?x rdf:type ssn:SensingDevice
} UNION {
?x rdf:type dbo:Engine
}
```

FIGURE 8 – Réécriture de requête (2)

concepts, nous avons donc besoin de deux bits (car 0 n'est pas un identifiant valide). Nous donnons donc les valeurs 01 et 10 respectivement (en bleu) aux concepts, mais nous gardons l'identifiant de `owl:Thing` (1) comme préfixe. L'algorithme continue de se dérouler de manière récursive : l'étape suivante nécessite toujours deux bits (verts) pour `ssn:Sensor` et `dbo:Engine`, respectivement 01 et 10, toujours en ajoutant le préfixe du super-concept : 101. Enfin, `ssn:SensingDevice` ne nécessite qu'un seul bit (violet). Une fois l'encodage de chaque concept terminé, il faut normaliser la longueur des identifiants, en ajoutant des 0 (en rouge) pour obtenir des tailles identiques. Les structures binaires ainsi générées sont enfin être converties en un entier décimal, qui peut ensuite être utilisé en tant que filtre pour réécrire la requête (figure 8).

La valeur 40 correspond à l'identifiant de `DUL:PhysicalObject`. 48 est l'identifiant de `DUL:SocialObject`, qui est le concept suivant au même niveau de la hiérarchie ; on l'obtient en incrémentant la structure de `DUL:PhysicalObject` de 1. Concrètement, `DUL:PhysicalObject` a pour structure 101, en retirant la normalisation ; en incrémentant, on obtient 110, et la normalisation nous donne l'entier 48, soit l'identifiant de `DUL:SocialObject` (exclus dans le filtre). La hiérarchie de concepts est bien conservée et peut être utilisée sans besoin de décompresser, ni d'accéder à la TBox.

Ainsi, l'utilisation de LiteMat permet de remplacer les éléments des triplets par des identifiants, bien plus compacts (surtout dans le cas d'URI). L'encodage facilite également la réécriture de requêtes, car il conserve la hié-

rarchie de concepts (ou de propriétés) de l'ontologie. Les modifications de l'ontologie peuvent forcer le ré-encodage total de la Abox et de la Tbox, dans le cas ou le nombre de bits prévus pour un certain niveau de concept n'est pas suffisant, mais cela n'a qu'un impact moindre dans un cas de traitement de flux. En effet les requêtes sont éphémères ; elle sont exécutées sur des fenêtres de données, et la modification d'une requête n'impacte pas les fenêtres précédentes.

3

État de l'art

Ce chapitre présente l'état de l'art dans les domaines (1) de la sémantisation et la sérialisation de flux RDF, (2) des langages de requêtes continues basés sur SPARQL, (3) des systèmes de profilage et (4) de l'utilisation de résumés des triplets RDF. L'objectif est de recenser l'ensemble des projets les plus importants réalisés dans ces domaines, que nous avons étudié, mais que nous n'avons pas retenu. Dans plusieurs cas, nous nous sommes inspirés de leur principe ou de l'idée de base qu'ils employaient afin de réaliser certains de nos composants, d'une manière plus adaptée et/ou plus performante pour notre projet.

3.1 Sérialisation de flux RDF

Dans cette section, nous présentons les principaux systèmes proposant une sérialisation adaptée aux flux RDF.

RDSZ [18] (RDF Differential Stream compressor based on Zlib) est une méthode de compression RDF. L'encodeur différentiel utilisé par l'algorithme attribue un identifiant à chaque sujet et chaque objet des triplets RDF, et les enregistre dans une table de couples clefs-valeur. On peut ainsi sérialiser les triplets RDF, en remplaçant les sujets et les objets par leurs identifiants, ou en les mettant à vide lorsque le sujet se répète sur plusieurs triplets. La figure 9 présente le résultat de l'encodage d'une mesure de capteur. Par la suite, RDSZ analyse quelle méthode de compression est la plus efficace entre l'application de l'algorithme Zlib [19] aux triplets de base, ou bien l'application de Zlib à la forme sérialisée. Pour ce qui est de la décompression, la chaîne en entrée

Triplets RDF

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
@prefix qudt: <http://data.nasa.gov/qudt/owl/qudt#>
@prefix ssn: <http://purl.oclc.org/NET/ssnx/ssn/#>
@prefix waves: <http://waves.org/resource#>

waves:event_1j_sh ssn:hasValue waves.obs_1j_sh ;
  ssn:isProducedBy waves:Q_DT01 ;
  ssn:startTime "2015-01-01T01:15:00"^^xsd:dateTime ;
  rdf:type ssn:SensorOutput .

waves:obs_1j_sh   qudt:numericValue 1.3E-1"^^xsd:double ;
  rdf:type ssn:ObservationValue .
```

Résultat de l'encodage

```
?x0 <http://purl.oclc.org/NET/ssnx/ssn/#hasValue> ?x1 .
 <http://purl.oclc.org/NET/ssnx/ssn/#isProducedBy> ?x2 .
 <http://purl.oclc.org/NET/ssnx/ssn/#startTime> ?x3 .
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?x4 .
?x1 <http://data.nasa.gov/qudt/owl/qudt#numericValue> ?x5 .
 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?x6 .
```

Couples clef-valeur

```
?x0 <http://waves.org/resource#event_1j_sh>
?x1 <http://waves.org/resource#obs_1j_sh>
?x2 <http://waves.org/resource#Q_DT01>
?x3 "2015-01-01T01:15:00"^^xsd:dateTime
?x4 <http://purl.oclc.org/NET/ssnx/ssn/#SensorOutput>
?x5 "1.3E-1"^^xsd:double
?x6 <http://purl.oclc.org/NET/ssnx/ssn/#ObservationValue>
```

FIGURE 9 – L'encodage différentiel de RDSZ

est d'abord passée au dé-compresseur Zlib, puis à un démultiplexeur qui décompose la chaîne de caractères sortie de la décompression en une liste d'objets RDF ; il ne reste plus qu'à réutiliser l'encodeur différentiel pour remplacer les identifiants des triplets par les sujets et objets originels. Si les gains de taille sont importants lors de l compression de graphes imposants, les avantages offerts par RDSZ sont moins avantageux pour des graphes plus petits, ou comportant peu d'URI. Par ailleurs, l'encodeur n'a pas été prévu pour les flux de données : les opération de compression et de décompression, dépendantes de la fréquence et de la tailles des éléments dans les flux, peuvent influencer grandement le temps de traitement.

Ztreamy [20] est une plate-forme middleware évolutive pour la distri-

bution de flux de données sémantiques. Elle permet de publier des flux de données, afin que ces dernières soient consommées par d'autres applications ; la plate-forme supporte des opérations telles que le mirroring (duplication de flux pour un traitement en parallèle), la jointure, le partitionnement (séparation d'éléments du flux pour un traitement spécifique) et le filtrage. L'approche évolutive et portable de ce projet le rend adaptable sur un large éventail de cas d'utilisation tels que, par exemple, la ville intelligente (*smart cities*) : gestion de mesures de capteurs physiques, réutilisation par des applications diverses, variété de formats... Le système fonctionne en sélectionnant spécifiquement les données pertinentes depuis les flux via des requêtes sur une longue durée (*long-lived requests*), selon divers critères (source des données, vocabulaire, requête SPARQL...). Les données extraites sont ensuite stockées dans une zone tampon, qui est vidée périodiquement dans le réseau pour éviter les surcharges (cette fonctionnalité est désactivable). Ainsi, Ztreamy est composé de nœuds ayant différent rôle : sources, producteurs et consommateurs de données. Un nœud peut avoir plusieurs rôles (une application en sortie sera consommatrice de données, mais un filtre sera à la fois consommateur et producteur). Les données générées par Zlib sont modélisées avec un en-tête contenant un identifiant, le timestamp de création, le type de donnée et la taille du corps de la donnée, pour une meilleure gestion. L'algorithme Zlib est ensuite utilisé pour compresser le résultat pour de meilleures performances. Bien que la compression soit adaptée aux flux, on retrouve le problème de temps lié à des suites de compressions et de décompressions, dans le cas de traitements multiples appliqués aux données. Bien que l'entête de la forme compressée soit bien pensé, il peut également occasionner de la redondance d'information : par exemple, dans notre cas d'usage, chaque capteur envoie des données à un timestamp précis, et le type de mesure est spécifié dans le graphe envoyé.

ERI [21] est un format de données RDF compressé visant à réduire la quantité de données transmises lors de traitement de flux. Basé sur RDSZ, l'algorithme repose sur le fait que la structure des données dans les flux est très familière au producteur, et qu'elle ne varie pas énormément. ERI considère un flux en tant que suite continue de blocs de triplets RDF. Chaque bloc est divisé en canaux : des canaux structurels, pour encoder les sujets des triplets et les propriétés associées à chacun avec un dictionnaire dynamique de structures, et des canaux de valeurs, pour encoder les valeurs des données concrètes des triplets. Le dictionnaire de structures regroupe l'ensemble des différents groupes de triplets ayant le même sujet : ces groupes

Molécule

```
waves:event_1j_sh ssn:hasValue waves.obs_1j_sh ;
  ssn:isProducedBy waves:Q_DT01 ;
  ssn:startTime "2015-01-01T01:15:00"^^xsd:dateTime ;
  rdf:type ssn:SensorOutput .
```

Molécule

```
waves:event_2j_sh ssn:hasValue
waves.obs_2j_sh ;
  ssn:isProducedBy waves:Q_DT01 ;
  ssn:startTime "2015-01-01T01:30:00"^^xsd:dateTime ;
  rdf:type ssn:SensorOutput .
```

Dictionnaire de structures

```
ssn:hasValue(1)
ssn:isProducedBy(1, waves:Q_DT01)
ssn:startTime(1)
rdf:type(1, ssn:SensorOutput)
```

FIGURE 10 – Dictionnaire de structures pour un événement de Waves

sont appelés des molécules. La figure 10 présente un exemple de dictionnaire généré à partir de deux molécules (en reprenant l'exemple employé figure 9). Diverses opérations sont réalisées pour optimiser la taille des molécules, en évitant les répétitions de prédicat discrets par exemple (couples prédicat-objet identiques sur plusieurs sujets). Les informations concernant les prédicats discrets, ainsi que celles concernant les molécules (métadonnées, compression, configuration...) sont stockées dans des *presets*, fournis par la source de données, ou inférés lors de l'exécution. Un flux ERI est donc une séquence de blocs de molécules, chacune étant multiplexée en plusieurs canaux, le tout formant un ensemble adapté pour des algorithmes de compression standards. Le principe du dictionnaire de structures permet une compression optimisée pour les flux RDF. Mais pour notre cas d'usage, le problème lié aux traitements multiples, nécessitant plusieurs compressions et décompressions, persiste.

3.2 Les langages de requêtes continues

C-SPARQL [22] est l'une des premières contributions dans le domaine du raisonnement sur flux. Le langage supporte les graphes RDF horodatés, l'exécution de requêtes sur des flux de données, et le support d'opérateurs d'agrégats. C-SPARQL exécute les requêtes sur une fenêtre de temps paramétrable, et peut combiner des données extérieures à celles des flux pour de

l'enrichissement. Le langage est essentiellement basé sur SPARQL, à ceci près que l'exécution des requêtes se fait sur les fenêtres du flux : cela permet à la requête d'être évaluée à une certaine fréquence, paramétrable par l'utilisateur. De nombreuses extensions indépendantes ont été réalisés afin d'adapter le C-SPARQL à différents scénarios spécifiques.

Streaming SPARQL [23] est une extension proposée pour le traitement de flux RDF. La contribution est principalement théorique : l'objectif est d'utiliser de l'algèbre relationnelle et temporelle, qui transforme automatiquement les requêtes SPARQL par le biais d'un algorithme. L'approche supporte la propriété de transitivité et semble intéressante mais n'a pas été expérimentée de manière intensive.

CQELS [24] présente de nombreux points communs avec C-SPARQL, adoptant le même modèle de gestion de flux par fenêtre et des opérateurs de requêtage. La principale distinction entre les deux est que l'évaluation de requête de CQELS n'est pas périodique, mais déclenchée par l'arrivée de nouveaux triplets. Le moteur de requêtes intègre lui-même les données des flux et (si besoin) les données externes, sans déléguer à un autre composant ; ce qui permet d'apporter diverses optimisations et techniques de réécriture de requêtes. Une autre implémentation de cette approche, baptisée CQELS cloud, a été réalisée pour les environnements orientés cloud. L'accent a été mis sur l'élasticité et la réduction de l'utilisation du réseau.

Les auteurs de C-SPARQL ont produit une autre contribution, IMaRS [25], qui repose sur la nature du flux et le fonctionnement de C-SPARQL pour calculer la date d'expiration des triplets RDF basés sur les fenêtres de requêtes. Chaque triplet a donc une durée de validité, ce qui permet d'optimiser les mécanismes de mise à jour. C'est ce qui fait la force du système, mais aussi sa faiblesse : il repose sur l'assomption stricte que la date d'expiration de chaque triplet peut être calculée, ce qui peut s'avérer irréalisable (cas de fenêtres de données non temporelles).

Comparé à IMaRS, TrOWL [26] présente deux distinctions majeures : il supporte des ontologies plus expressives (OWL2-DL), et il ne prédit pas le temps d'expiration des triplets à partir de fenêtres temporelles. La principale innovation du système est d'utiliser des approximations syntaxiques pour réduire la complexité du raisonnement. Pour faciliter la rétractation d'information, TrOWL garde la trace des relations entre les faits dérivants et les faits dérivés : cela facilite la mise à jour des connaissances. Malheureusement, il est difficile de comparer TrOWL aux autres approches, à cause de ses spécificités (support d'ontologies plus expressives, fenêtre non-temporelle).

EP-SPARQL [27] est un langage unifié pour le traitement et le raisonnement sur des événements. Il réutilise les opérateurs de fenêtrage de C-SPARQL, mais la base du langage est représentée par un ensemble d'opérateurs qui peuvent être combinés pour exprimer des schémas d'information complexes. Une autre différence majeure est l'horodatage : EP-SPARQL utilise deux timestamps représentant les bornes de l'intervalle de gestion de l'élément. L'une des limites du langage vient de la configuration limitée des schémas d'information, qui est un problème récurrent.

Bien que ces systèmes présentent de nombreux avantages et représentent de grandes avancées en terme de raisonnement sur flux, aucun d'eux ne gère les données de manière compressée, ce qui pose invariablement problème lors du traitement de larges volumes d'informations. Par ailleurs, aucune des approches mentionnées ne gère l'inférence de manière native, une fonctionnalité qui peut se révéler indispensable pour certains cas d'usage, comme ceux que nous rencontrons dans le projet Waves.

3.3 Les systèmes de profilage

Il existe de nombreuses méthodes de profilage à l'heure actuelle, employées dans de nombreuses applications et sur des domaines variés. Toutefois, bien que certaines se basent sur des données géographiques, elles n'utilisent pas les mêmes méthodes que notre système. Nous avons étudié divers travaux ayant des considérations communes à notre méthode : nous avons cherché des projets avec des composantes spatiale, temporelle et sémantique, afin de pouvoir les comparer à notre approche.

La première méthode retenue, [28], récupère les données géographiques relatives aux relais de communication mobile, ainsi que toutes sortes d'informations géolocalisées dans le voisinage de ces relais. Avec ces données, un profilage des relais de communication est établi ; en ce servant de ce profil, le système arrive à généraliser le comportement des personnes appelant grâce aux statistiques d'appel. Ce système de profilage a donc bien une composante géographique, et une composante temporelle (puisque les données sur les appels reçus concernent une période précise). Il repose sur plusieurs sources de données libres, notamment des ontologies géographiques dérivées d'OpenStreetMap. Dans le cadre du raisonnement, [28] utilise les différentes catégories de POI définies par OpenStreetMap dans une ontologie ; chaque concept permettant de caractériser des lieux est accompagné d'un poids re-

flétant son importance dans l'analyse. Toutes ces techniques sont également employées dans notre approche, bien que notre cas d'utilisation ne soit pas le même. La formule finale que nous utilisons est très différente des calculs de cette méthode, car nous combinons les résultats de plusieurs méthodes de profilage, et pas directement des données géographiques.

[29] a pour but la modélisation de l'activité humaine en utilisant des catégories de lieux ; l'algorithme est basé sur une méthode de clustering, appliquée sur les zones géographiques et les données du réseau social Foursquare. Le profilage est réalisé en se basant sur les activités recensées par l'application (coordonnées GPS des restaurants, boutiques, etc...) ; ce principe a donc également une composante temporelle et spatiale. Ici encore, un système de classement basé sur des scores attribués sur les catégories de données est utilisé dans la formule de clustering. Le résultat donne les concentrations de POI de même nature, mais ne permet pas de caractériser de grandes zones : l'objectif ici est d'identifier des groupements d'activité de même nature.

Dans le domaine de la criminalité, on peut mentionner les travaux de [30], qui traite du profilage géographique des criminels en série, et [31], qui sont basés sur les actions des gangs. Il s'agit dans chaque cas d'un profilage géographique, avec une composante temporelle, et pour lesquels l'usage de données extérieures est requis.

Dans chacun des cas, le profilage est réalisé à partir de POI extraits de données en ligne depuis différentes sources d'information. Si notre méthode utilise ce genre d'approche, elle ne s'y limite pas : en effet, nous utilisons deux calculs différents permettant de caractériser les secteurs de consommations, et les combinons pour obtenir un résultat plus précis. Les travaux mentionnés ne font que combiner des sources de données en ligne pour fournir un résultat final, qui n'est pas ré-exploité automatiquement par d'autres applications. De plus, notre algorithme est le seul à utiliser des données de surfaces en plus des POI ; comme nous l'avons exposé précédemment, cela permet de combler le manque de POI représentatifs de certaines zones géographiques.

3.4 L'utilisation de résumés en RDF

Comme nous l'avons vu en chapitre 2, le langage RDF pose encore de nombreuses problématiques de performances à l'heure actuelle, principalement pour raisonner sur de larges volumes de connaissances. Ces besoins d'efficacité se retrouvent dans le traitement de flux de données sémantisées, ou le

temps est un facteur tout aussi important. Il existe de nombreuses méthodes pour tenter de les résoudre, la plupart toujours en développement. L'une des solutions étudiée concerne l'utilisation de résumés RDF : en considérant un graphe RDF, il est possible de réduire le volume d'informations qu'il contient, tout en conservant un maximum de précision, afin de pouvoir le manipuler de manière plus optimisée. Ces résumés peuvent être générés de différentes manières en fonction des systèmes. Cet aspect est important dans Waves car, dans le cas de recherches d'information dans des bases de connaissances externes (comme c'est le cas pour la contextualisation des anomalies dans notre cas d'usage), il est nécessaire d'examiner de larges volumes de données. Sans méthodes de traitement efficaces, le résultat peut s'avérer imprécis et très long à obtenir : il est donc impératif d'utiliser des méthodes permettant d'obtenir l'essentiel des éléments pertinents au sein d'un ensemble d'informations structurées. LiteMat, que nous avons présenté section 2.5, peut être assimilé à un résumé, car il réduit la taille des graphes RDF qu'il encode, en conservant toutes les informations, et en matérialisant les hiérarchies de concepts et de propriétés dans les formes compressées obtenues. Le résultat obtenu nécessite un traitement spécifique pour être réutilisé, mais pas nécessairement une décompression.

SHER [32] est un raisonneur évolutif et hautement expressif qui supporte la plupart des logiques de description de OWL-DL. SHER réalise toutes les inférences possibles lors du chargement des données, (matérialisation), ce qui permet d'éviter le raisonnement lors de l'exécution des requêtes. En outre, le raisonneur utilise une version résumée de la Abox de l'ontologie : le principe est d'associer tous les concepts de la Abox originelle au même ensemble de concepts d'un unique individu du résumé. La cohérence du résumé garantie la cohérence de la Abox d'origine, ce qui rend les vérifications plus rapides.

Une autre approche, détaillée dans [33], est basée sur l'utilisation de résumés et de raffinement. Le résumé est créé à partir de la Abox d'origine, en agrégeant les individus membres d'un même concept. Les requêtes sont ensuite exécutées sur le résumé : en testant un individu du résumé, tous les individus liés sont testés en même temps. Le raffinement est utilisé pour vérifier les problèmes de cohérence : il s'agit de partitionner les individus liés à un individu unique du résumé, puis de relier chaque partition à un nouvel individu du résumé, jusqu'à prouver soit que tous les individus sont des solutions, soit que ce résumé étendu est cohérent. La définition du raffinement doit être bien précise pour conserver des performances efficaces lors de la vérification.

D'autres résumés ont plus pour objectif principal de condenser les ontologies, de les abréger en conservant les concepts principaux (ce qui n'est pas toujours le cas pour un résumé). RDFdigest [34] utilise un algorithme vérifiant la couverture maximale des nœuds les plus importants d'un graphe RDF en se basant également sur les sommets adjacents. Les performances d'exécution de requêtes sur le résumé sont toujours améliorées, et on retrouve des similarités avec les résultats de requêtes sur l'ontologie globale (le nombre de similarités augmente avec la taille du résumé).

4

Contextualisation des secteurs de mesure

4.1 Motivation

Dans de nombreuses applications intégrant de l'analyse de mesures provenant de capteurs, il existe une composante spatio-temporelle importante. Le projet FUI Waves ne fait pas exception à ce fait puisque les positions des capteurs apportent une connaissance importante sur le contexte de la mesure, tout comme son horodatage (date et heure précise d'émission). Par le biais de méthodes mathématiques et statistiques, nous nous servons des flux de données pour détecter les anomalies dans les mesures relevées : il s'agit du premier filtre appliqué dans le cadre du raisonnement (cf. figure 1). Dans ce cas, c'est l'aspect temporel des données qui entre en jeu : la manipulation des données s'effectue en fonction de leur horodatage. Un nouveau flux est produit en sortie, contenant uniquement les mesures erronées détectées. Toutefois, l'aspect spatial des mesures (*e.g.,* la position où a été relevée la mesure) n'est pas utilisée dans la détection de valeurs aberrantes. Cette composante peut être exploitée à une autre étape du traitement des données.

C'est lors de la recherche des origines potentielles de ces anomalies que les données géographiques sont les plus pertinentes, car c'est la localisation de la source d'erreurs qui permettra d'établir où chercher les événements potentiellement responsables. Le module Scouter, détaillé en section 4.6, se charge de cette tâche : les flux événementiels sur lesquels il raisonne sont géolocalisés à proximité des anomalies détectées. L'aspect temporel est également présent, puisque la date précise des événements joue aussi un rôle pour l'établissement

de leur responsabilité de l'anomalie. Comme pour la détection d'anomalies, Scouter utilise diverses méthodes afin de définir une probabilité pour chaque événement extrait, afin de déterminer la cause des irrégularités détectées. Il se base sur le flux d'anomalies généré précédemment, et fournit en sortie les événements jugés responsables, avec leur probabilité.

Le projet Waves peut donc être divisé en deux phases majeures : l'identification des événements pertinents dans les flux d'entrée, et l'établissement de leur probabilité de responsabilité. Pour chaque cas, nous avons consulté notre partenaire expert dans le domaine du traitement dans la qualité de l'eau afin de décider de la manière de procéder. A la suite de nos multiples discussions, nous avons ainsi pu établir diverses méthodes afin de gérer les deux étapes mentionnées de manière efficace. L'une de nos propositions retenues a particulièrement intéressé notre partenaire : un profilage géographique des secteurs de consommation. Notre système de profilage permet d'établir les proportions de différents types de surface présents au sein d'une zone géographique, ce qui est très utile pour l'ajustement de la probabilité de responsabilité des événements.

Nous détaillons par la suite les limites de l'aspect temporel des données du projet, ce qu'apportent les données spatiales en termes de raisonnement, puis nous détaillons le système de profilage géographique implémenté dans Waves.

4.2 Limitation de l'analyse des données temporelles

Les données provenant de capteurs de mesure sont effectuées à intervalle de temps régulier : nous pouvons donc utiliser ces flux de données pour identifier les anomalies dans les mesures. En outre, disposer des horodatages des mesures permet de récupérer d'autres informations temporelles liées, afin de détecter les origines des irrégularités précédemment décelées.

La détection d'anomalies peut être réalisée en employant diverses méthodes mathématiques et/ou statistiques, en fonction du type de mesure étudiée. Cette contribution a majoritairement été réalisée par l'un de nos partenaires, mais nous pouvons dresser une liste des principales méthodes employées. Certaines sont simples, et applicables indépendamment du type de mesure : vérification de dépassement de valeurs seuil, et la comparaisons

de moyennes de mesures. Cette dernière technique peut être effectuée sur des périodes de temps plus ou moins longues ; dans le cadre de notre projet, elle est effectuée généralement durant la nuit, lorsqu'il n'y a pas de risques de gros écarts de consommation. Il existe d'autres méthodes plus complexes, telle que la comparaison deux à deux, qui vise à mettre en parallèle les consommations de deux secteurs sur une longue période en utilisant un graphe. En présence d'éventuels groupes de points isolés, on peut déduire la présence d'un problème dans les mesures, et tenter de le contextualiser.

En plus de ces méthodes, il est possible d'étudier l'évolution des mesures dans le temps, en analysant les relevés suivants afin de déterminer leur importance et de mieux identifier leur origine. Une erreur ponctuelle (par exemple, un dépassement de seuil pour une seule mesure) ne sera vraisemblablement pas jugée critique, tandis qu'une anomalie prolongée (répétée sur plusieurs mesures de suite) sera évaluée plus grave. La visualisation de notre projet permet également de voir si l'anomalie augmente en intensité dans le temps (une baisse de pression de plus en plus élevée), ou si elle est liée à d'autres paramètres (dans le cas d'une fuite, on aura une augmentation de la consommation, et une baisse de pression). Les données ont déjà subi un nettoyage grâce au système de filtrage situé en amont de l'architecture du projet, par conséquent les déficiences des capteurs n'entrent pas en compte : il n'y a que les origines extérieures à prendre en compte (fuite, surconsommation...).

Ces analyses permettent donc d'identifier les incohérences dans les mesures, et de les caractériser ; mais elles n'aident en rien à trouver les causes de ces anomalies. L'étude des données temporelles n'est donc qu'une motivation : il est nécessaire de trouver les origines de ces anomalies. Pour cela, nous avons besoin de caractériser les secteurs de consommation, afin d'orienter nos recherche. Par exemple, supposons que dans notre cas d'usage, nous détections une chute de pression : l'analyse seule des données temporelles nous permet d'en déterminer l'importance, mais pas la source. S'il ne s'agit pas d'une fuite, mais d'un simple écart par rapport à la moyenne habituelle, envoyer des agents sur le terrain serait une perte de temps, financière et de ressources. Mais le simple fait de savoir si l'erreur est due à un problème matériel ou un événement extérieur n'est pas facile. L'un des exemples évoqué par notre partenaire expert du cas d'usage était un écart de valeur causé par la vidange et le remplissage d'une piscine municipale ; l'origine de l'anomalie n'a malheureusement été identifiée qu'après l'intervention d'experts sur le terrain, qui ont échoué à identifier l'origine matérielle du problème. Ce genre de détail peut potentiellement être repéré en analysant les sources de don-

nées extérieures, et en analysant les spécificités géographiques des secteurs de consommation.

4.3 Analyse de données géographiques

Le profilage géographique s'effectue en prenant en compte les caractéristiques des zones étudiées. En calculant la proportion des différents types de surface présentes sur chaque secteur de consommation, nous pouvons établir le niveau d'urbanisation de chaque zone, et ainsi affiner les sources d'origine identifiées. La figure 11 présente l'architecture simplifiée du système de profilage adopté : dans un premier temps, nous extrayons depuis les sources d'information les données géographiques pour la zone étudiée, puis nous récupérons les données pertinentes à notre cas d'usage, en nous servant d'un fichier décrivant les données utiles. A partir de ces dernières, deux types de profilages sont réalisés, en utilisant des éléments différents : les points d'intérêt, et les polygones. Une fois les résultats obtenus, ils sont ajustés voire combinés pour une meilleure précision, à partir des statistiques des données récupérées (afin de juger des plus utiles). Le résultat est ensuite envoyé au module parent, qui se charge de son interprétation. Nous détaillons la sélection et la récupération des données géographiques dans cette section, et nous développons les explications sur les méthodes de profilage dans le chapitre suivant.

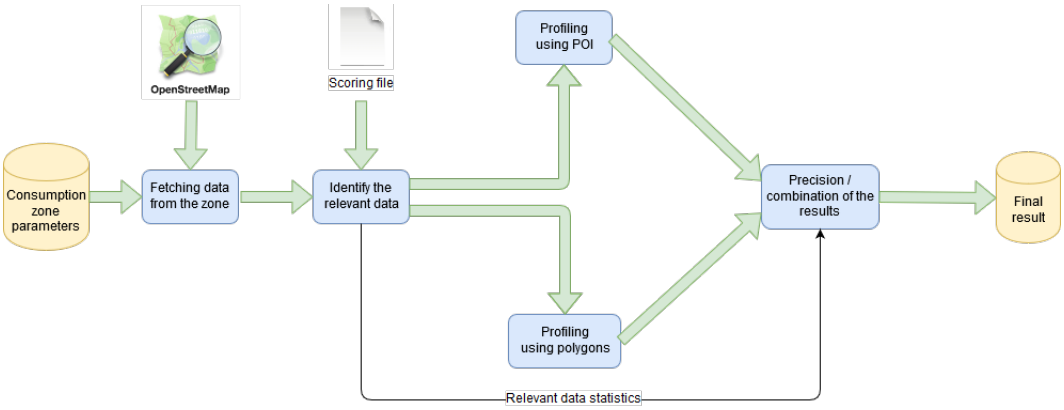


FIGURE 11 – Architecture du système de profilage de Waves

4.3.1 Qualification des sources de données

Il existe de nombreuses sources de données géographiques disponibles en ligne, chacune ayant différentes caractéristiques. Nous avons établi la liste exhaustive des outils que nous avons étudié en section 2.3, et parmi eux nous avons choisi d'utiliser OpenStreetMap. Il s'agit de la source de données la plus complète parmi toutes celles que nous avons étudiées, et semble être la meilleure base géographique gratuite à l'heure actuelle. Par ailleurs, OpenStreetMap a déjà été réutilisé dans de nombreux projets, et sa gestion communautaire la rend extrêmement riche. Nous avons pu la comparer directement à d'autres sources de données via divers outils de comparaison (toujours section 2.3) pour arrêter notre choix, et nous avons examiné plusieurs échantillons de données en téléchargeant plusieurs échantillons sur des secteurs géographiques précis.

Pour cela, nous avons récupéré les points d'intérêt (POI) présents sur notre zone d'étude. Les POI représentent des lieux sur une carte ; il ne s'agit pas forcément de point présentant une très grande utilité (les cabines téléphonique, les boites aux lettres, et même les bancs publics sont des POI), mais plutôt d'éléments représentables par un point (le terme est donc très large). On les trouve en très grand nombre dans les zones urbanisées : pour l'un de nos exemples, une zone peuplée constituée de onze secteurs de consommation, nous avons dénombré plus de 8.500 POI. Ils sont répartis en plusieurs catégories sous OpenStreetMap, en fonction des éléments qu'ils représentent ; ce classement de POI est documenté en ligne [1].

Accéder à OpenStreetMap permet de télécharger des données sous forme de points, mais aussi de lignes (appelées *ways*), et il est également prévu de récupérer des relations (cette fonctionnalité n'est pas encore exploitée à l'heure actuelle). La zone à récupérer est sous forme de *bounding box*, c'est à dire une zone rectangulaire. Une *bounding box* peut se générer très facilement : soit à la main, avec des outils cartographiques (exemple figure 12), soit de manière automatique, en extrayant les coordonnées de latitude et de longitude maximales et minimales parmi tous les points constituant le secteur étudié. Dans notre cas, nous disposons des coordonnées précises des secteurs de consommation dans le fichier de configuration du réseau ; nous n'avons eu qu'à récupérer les latitudes et longitudes minimales et maximales pour chaque secteur, afin de créer la *bounding box* adaptée, regroupant tous les

1. `https://wiki.openstreetmap.org/wiki/Map_Features`

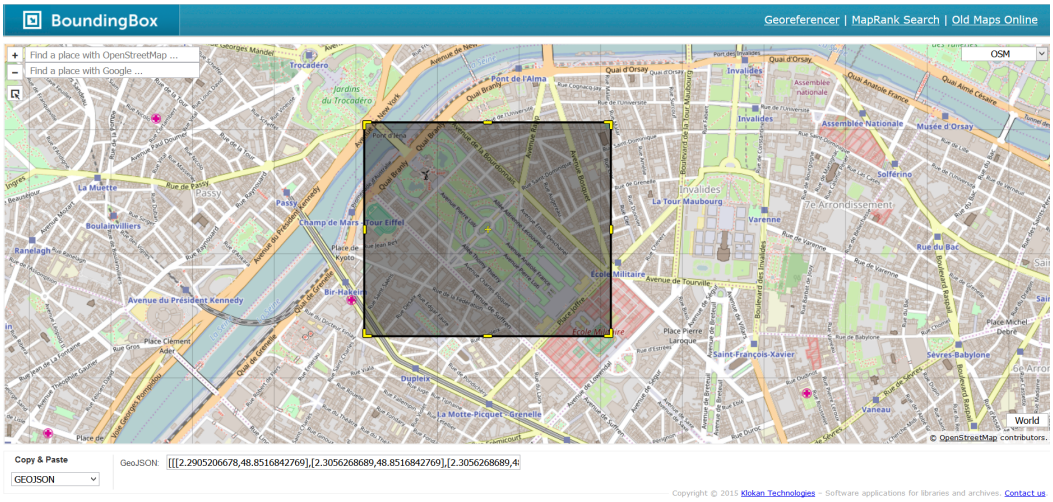


FIGURE 12 – Exemple de bounding box centrée sur Paris

POI qu'il contient.

Chacun des points d'intérêt d'OpenStreetMap est modélisé de la même manière, avec les détails suivants :

— son type (*node* pour les points d'intérêt, *ways* pour les lignes),
— son identifiant,
— sa latitude,
— sa longitude,
— une série de tags le décrivant (absent si le POI fait partie d'une ligne).

Un extrait de données issues d'OpenStreetMap est présenté en figure 13 ; on y retrouve un POI détaillé, ainsi que deux autres plus compacts, servant de limitations à des polygones. Une fois l'extraction effectuée, il est nécessaire d'affiner le résultat obtenu : la bounding box extraite contient plus d'informations que le secteur de consommation (elle englobe le secteur, couvre une zone plus large). Nous avons donc modélisé informatiquement le secteur de consommation afin de pouvoir vérifier pour chaque POI s'il est bel et bien inclus dans le secteur (et pas ajouté par la bounding box), en nous servant de leurs coordonnées géographiques.

Ensuite, nous pouvons examiner le type et la quantité de POI présents sur le secteur de test, en comptant les tags descriptifs rencontrés : *p.ex.*

```
{
  "type": "node",
  "id": 4081916083,
  "lat": 48.8582635,
  "lon": 2.2945161,
  "tags": {
    "addr:city": "Paris",
    "addr:housenumber": "5",
    "addr:postcode": "75007",
    "addr:street": "Avenue Anatole France",
    "name": "Tour Eiffel"
  }
},
{
  "type": "node",
  "id": 4155917782,
  "lat": 48.8571246,
  "lon": 2.2908896
},
{
  "type": "node",
  "id": 4155917783,
  "lat": 48.8571425,
  "lon": 2.2908591
},
```


FIGURE 13 – Extrait de points d'intérêt en région Parisienne

un secteur peut regrouper une trentaine de bancs et trois banques, mais seulement une piscine, et aucun casino. Nous pouvons donc vérifier quels sont les POI présents dans un secteur précis soit en filtrant les POI avec des tags pertinents (*i.e.* utiliser un programme basé sur une Map), soit en vérifiant visuellement les fichiers récupérés dans le cas de secteurs à surface réduite. On peut ainsi vérifier si notre profilage sera judicieux, en observant le nombre de POI d'intérêts et ceux qui sont majoritaires, ou à l'inverse en remarquant le manque de données dans la zone retenue.

4.3.2 Récupération des données

Les données d'OSM peuvent être récupérées et exploitées de diverses façons. La manière la plus simple est d'utiliser une URL de récupération : en utilisant le lien vers l'API d'OSM, il est possible de spécifier les coordonnées d'une bounding box, et d'en récupérer le contenu. Il suffit de fournir les coordonnées (latitude et longitude) du point inférieur gauche du rectangle, et celles du point supérieur droit, ainsi que le type de données à télécharger (points ou lignes). Les deux liens principaux sont ceux de l'API d'OpenStreetMap [2], limitée à de petites zones, et celle d'**Overpass** [3] (également open source), pour des zones plus larges ; ils fournissent les données sur la zone

2. `http://api.openstreetmap.org/api/0.6/map?`
3. `http://www.overpass-api.de/api/interpreter`

fournie au format XML.

Il existe d'autres méthodes de récupération, chacune adaptée à certains cas d'utilisation. **Nominatim**, un autre outil de recherche d'OSM, offre la possibilité d'utiliser d'autres paramètres pour spécifier le résultat, tels que l'adresse à la place de la bounding box. L'outil permet aussi de changer le format auquel sont récupérées les données (html, xml ou json), et le niveau de détail souhaité. Pour réaliser des traitements plus complets, l'application **Osmosis** permet de réaliser plusieurs opérations à la suite en ligne de commande. Cela nécessite le téléchargement des zones de données à étudier (soit via l'une des méthodes précédente, soit en utilisant les extraits de données disponibles en ligne [4]), mais l'application peut gérer les données compressées, ce qui rend les opérations de traitement et de téléchargement plus performantes.

Naturellement, les données d'OpenStreetMap sont également utilisable directement de manière graphique. L'API permet d'intégrer des fragments de cartes dans des applications ou des pages internet, et des extraits de carte peuvent être étudiés dans des SIG (Système d'Information Géographique). Notre approche se base uniquement sur des proportions numériques, indispensables pour le cas de larges volumes de données. Mais l'étude visuelle est également employée et permet d'effectuer diverses vérifications, comme nous le démontrerons par la suite.

Nous avons utilisé l'API Overpass pour réaliser notre système de profilage, car il s'agit de celle qui présente les meilleures performances en terme de vitesse de récupération de données : les URL de récupération d'OpenStreetMap et de Nominatim ont des délais de récupération plus long d'environ une minute. Nous avons également effectué des tests avec Osmosis, qui s'est révélée encore plus lent, même en utilisant directement le code de l'application. Les autres méthodes mentionnées, essentiellement basées sur des représentations graphiques, ne conviennent pas à notre approche.

4.4 Profilage géographique

Les sections suivantes vont détailler le fonctionnement et les évolutions de notre système de profilage. De nombreuses modifications y ont été apportées, et son utilisation a également évoluée au fil du temps ; toutefois, l'architecture globale, présentée dans la figure 11, n'a pas subi de modification majeure.

4. `http://download.geofabrik.de/`

4.4.1 Exploitation de points d'intérêt

Nous avons développé une première méthode de profilage utilisant les points d'intérêt (POI) évoqués précédemment. La méthode de récupération depuis OSM et les vérifications d'appartenance aux secteurs étudiés exposées précédemment furent reprises, mais il nous fallait un moyen de déterminer comment utiliser les POI dans le profilage.

Nous avons choisi d'utiliser la hiérarchie de catégories d'OSM afin de ranger les points d'intérêt dans plusieurs classes, représentant les types de surfaces de consommation, et d'attribuer à chacun de ces POI un poids représentatif de son appartenance à sa classe. Ainsi, le profilage s'effectue en récupérant les points d'intérêt présents dans chaque secteur de consommation, et en additionnant les scores qui leur ont été attribués ; on obtient alors, pour chaque classe établie, une note correspondant à sa représentation en POI dans le secteur. Il ne reste plus qu'à calculer les proportions (les pourcentages) pour chaque classe de surface à partir de leurs notes.

Nous avons commencé par créer un fichier de scores pour les POI, illustré en figure 14. Suite à plusieurs réunions avec notre partenaire spécialiste du domaine de l'eau, nous avons défini cinq classes principales de terrain : industriel, résidentiel, touristique, naturel et agricole. D'autres concepts ont été expérimentés, mais ce sont ces éléments qui sont les plus pertinents et qui offrent les meilleurs résultats. Nous les avons intégré dans la classification d'OpenStreetMap, afin de conserver la hiérarchie de catégories déjà établie ; nous avons également expérimenté une organisation plus détaillée, avec une hiérarchie plus en profondeur, mais le nombre de tags pertinents ne convient pas pour une représentation avancée. En revanche, la classification reprise d'OpenStreetMap est simple d'utilisation, et fournit de bons résultats pour notre cas d'usage. Au cours de nos réunions, nous avons également identifié les points d'intérêt pertinents, les avons répartis dans les classes que nous avions définies, et leur avons attribué un score à chacun. Les classes ont été définies en fonction de notre cas d'usage, en sélectionnant les types de surface les plus distincts au niveau de la consommation d'eau ; nous avons ensuite identifié les POI les plus pertinents, représentatifs de chaque classe. Le résultat est enregistré dans un fichier JSON, qui est un format simple d'utilisation et très compact.

Avec ce fichier, une fois les POI du secteur étudié identifiés, nous pouvons analyser les tags des POI pertinents (ceux sans tags sont ignorés). Nous comparons ces tags à ceux enregistrés dans le fichier de scores, afin de récu-

```
{
        "amenity": {
                "tourism": {
                        "entertainment": {
                                "arts_centre": 0.8,
                                "casino": 1.0,
                                "community_centre": 0.8,
                                "fountain": 0.3,
                                "gambling": 1.0,
                                "planetarium": 0.8,
                                "theatre": 1.0
                        },
                        "other": {
                                "sauna": 0.5,
                                "shower": 0.7,
                                "toilets": 0.2,
                                "water_point": 0.7
                        }
                },
                "natural": {
                        "sustenance": {
                                "bbq": 0.8,
                                "drinking_water": 0.3
                        }
                }
```

FIGURE 14 – Extrait du fichier de scores du profilage de Waves

pérer les notes qui leur ont été attribuées. Lorsque l'un des tags d'un POI correspond à l'un de ceux notés dans le fichier, nous ajoutons son score à celui de la classe de terrain à laquelle il est rattaché. Ainsi, plus il y aura de POI représentatifs d'une classe de surface dans un secteur de consommation, plus sa note sera élevée. Une fois que tous les POI ont été traités, nous calculons les proportions (en pourcentage) de chaque classe de surface, pour un résultat plus facilement interprétable que les simples notes.

Afin de juger de la précision de nos résultats, nous avons commencé par procéder à des examens visuels. Nous avons généré des fichiers KML (*Keyhole Markup Language*) à partir des données du fichier de configuration, pour visualiser les secteurs de consommation avec Google Earth. En combinant cette visualisation avec la couche de données d'OpenStreetMap, nous sommes en mesure de juger de la qualité de notre système. Nous avons procédé à plusieurs ajustements de score pour affiner nos résultats, et nous avons obtenu des résultats pertinents pour certaines classes de terrain (zones touristiques

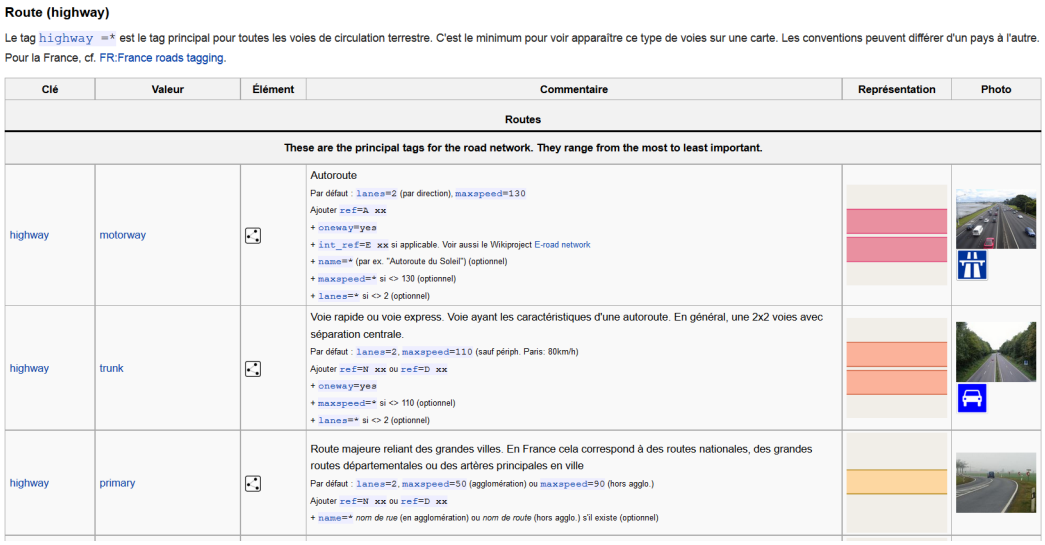
et résidentielles) ; toutefois, certaines surfaces (agricoles et naturelles) ne sont pas du tout représentées par cette méthode. Cela est dû au fait qu'il n'y a que très peu de POI représentatifs de ces zones, et qu'elles ne peuvent donc pas être profilées correctement de cette manière.

4.4.2 Exploitation de polygones

Une seconde méthode de profilage a donc été développée afin de pallier les défauts de la première concernant certains types de surface. Puisque les points d'intérêt ne nous permettent pas de représenter tous les types de surface, nous avons décidé d'établir un autre système de profilage basé cette fois sur les polygones. Ainsi, nous pouvons représenter efficacement les surfaces négligées par la première méthode, et nous pouvons nous baser sur l'aire de polygones pour calculer les proportions finales, et non sur des notes attribuées arbitrairement.

Nous récupérons les données sur les polygones depuis OpenStreetMap en utilisant la même URL que pour les POI, en spécifiant cette fois-ci que nous voulons récupérer les *ways* (ligne). Les lignes sont modélisées de manière similaire aux POI, à l'exception des coordonnées : à la place de la latitude et de la longitude, la représentation de la ligne contient un tableau d'identifiants, ceux des nœuds composant la ligne ; c'est une références aux POI qui n'ont pas de tags. Ceci implique que nous devons également récupérer les données sur les nœuds inclus dans la *bounding box* en plus de celles des lignes, puisque ces informations sont complémentaires. Cela représente une importante quantité de données à récupérer : pour nos tests, les données géographiques représentent en moyenne 30 Mo par secteur, en cumulant les données des POI et des polygones. La taille varie d'un secteur à un autre, et peut devenir bien plus importante si plusieurs secteurs sont récupérés simultanément. Pour cette raison, nous avons inclus la possibilité de persister les fichiers récupérés pour un usage futur : cela évite de télécharger à nouveau l'intégralité des informations si ce n'est pas nécessaire.

Dans les données sur les lignes récupérées, il faut bien différencier les lignes ouvertes (routes, chemins, cours d'eau, etc.) des polygones (composés de lignes fermées) : pour cela, il suffit de vérifier si, dans le tableau des identifiants de nœuds composant la ligne, le premier et le dernier nombre sont identiques (pour signifier la fermeture de la ligne). Nous nous servons ensuite des données sur les POI afin de récupérer les nœuds sans tags (donc constituant les *ways*) inclus dans le secteur de consommation. Une fois ces nœuds



Route (highway)

Le tag `highway =*` est le tag principal pour toutes les voies de circulation terrestre. C'est le minimum pour voir apparaître ce type de voies sur une carte. Les conventions peuvent différer d'un pays à l'autre. Pour la France, cf. FR:France roads tagging.

Clé	Valeur	Élément	Commentaire	Représentation	Photo
Routes					
These are the principal tags for the road network. They range from the most to least important.					
highway	motorway		Autoroute Par défaut : `lanes`=2 (par direction), `maxspeed`=130 Ajouter `ref`=`A xx` + `oneway`=`yes` + `int_ref`=`E xx` si applicable. Voir aussi le Wikiproject E-road network + `name`=* (par ex. "Autoroute du Soleil") (optionnel) + `maxspeed`=* si <> 130 (optionnel) + `lanes`=* si <> 2 (optionnel)		
highway	trunk		Voie rapide ou voie express. Voie ayant les caractéristiques d'une autoroute. En général, une 2x2 voies avec séparation centrale. Par défaut : `lanes`=2, `maxspeed`=110 (sauf périph. Paris: 80km/h) Ajouter `ref`=`N xx` ou `ref`=`D xx` + `oneway`=`yes` + `maxspeed`=* si <> 110 (optionnel) + `lanes`=* si <> 2 (optionnel)		
highway	primary		Route majeure reliant des grandes villes. En France cela correspond à des routes nationales, des grandes routes départementales ou des artères principales en ville Par défaut : `lanes`=2, `maxspeed`=50 (agglomération) ou `maxspeed`=90 (hors agglo.) Ajouter `ref`=`N xx` ou `ref`=`D xx` + `name`=* *nom de rue* (en agglomération) ou *nom de route* (hors agglo.) s'il existe (optionnel)		

FIGURE 15 – Description de lignes représentant des routes dans OSM

identifiés, nous sommes alors en mesure de récupérer leurs coordonnées, et donc de modéliser les polygones inclus dans les secteurs de consommation. Très souvent, les polygones ne sont pas totalement dans le secteur : seule une partie se trouve à l'intérieur, et par conséquent une partie des POI le composant ne sera pas récupérée. Dans ce cas, nous ne modélisons le polygone qu'à partir des nœuds inclus, créant ainsi une figure différente mais appartenant totalement au secteur de consommation. On peut en voir un exemple en figure 16 : la forêt représentée dans OSM n'est pas intégralement incluse dans le secteur de consommation (modélisé en rouge), et sa partie gauche ne sera pas prise en compte pour le profilage. Seuls les nœuds présents dans le secteur (à droite de la délimitation rouge) seront pris en compte pour le calcul de surface.

Nous réutilisons également notre fichier de scores, afin de nous servir des tags notés pour identifier les polygones représentatifs des classes de terrain (la classification établie par OpenStreetMap est valable pour les *nodes* et les *ways*). Toutefois, au lieu d'utiliser les notes établies arbitrairement, nous nous basons sur l'aire des polygones appartenant à chaque classe ; ainsi, le résultat n'est plus arbitraire. Les proportions finales du profilage sont donc

FIGURE 16 – Cas de polygone à la frontière d'un secteur

calculées à partir de surfaces mathématiques.

Les résultats obtenus à l'aide de ce système de profilage sont très satisfaisants pour les types de terrains non représentables avec des POI (surfaces agricoles et naturelles) ; mais comme précédemment, cette méthode révèle ses limites pour les surfaces qui sont peu représentables avec des polygones (quartiers résidentiels, zones touristiques). Il nous fallait donc trouver un moyen de décider dans quel cas chacune des méthodes serait la plus précise, voire les combiner pour les cas mitigés.

4.4.3 Précision des résultats

Pour réaliser le profilage le plus adapté et le plus précis en fonction des secteurs, nous avons choisi d'étudier la consommation de chacun d'eux. L'objectif était de trouver quel type de surface est le plus représentatif, et d'utiliser ensuite la méthode de profilage la plus adaptée pour le détecter et obtenir les proportions exactes.

Nous avons travaillé avec des données historiques fournis par notre partenaire ; sur le projet déployé, il suffit de ré-utiliser les mesures de capteurs qui ont été persistées dans une base de données (mesures utilisées dans certaines méthodes de détection d'anomalie). Les données sont organisées de manière

simple : il y a un répertoire pour chaque type de mesure (pression, débit, taux de chlore...), et un fichier excel pour chaque capteur de mesure (le fichier porte le nom du capteur). La première colonne du fichier contient la date exacte de la mesure, et la seconde la valeur relevée. Un fichier de configuration contient les paramètres du réseau, notamment les unités de mesure des capteurs, les différents secteurs de consommation et la répartition des capteurs sur chaque secteur.

Algorithm 1 Évaluation du rapport de consommation d'un secteur

```
1: procedure processConsumptionRatio(sector, configFile)
2:     List(sensor) = extractSensors(sector, configFile)
3:     consumption = 0
4:     for all sensor ∈ List(sensor) do
5:         avg = computeAvgConsumption(File(sensor), 24 * 3600, 7)
6:         if isInput(sensor) then
7:             consumption = consumption + avg
8:         else
9:             consumption = consumption − avg
10:        end if
11:    end for
12:    networkLength = extractNetworkLength(sector, configFile)
13:    return consumption/networkLength
14: end procedure
```

L'algorithme 1 résume l'établissement du calcul du ratio de consommation. Dans un premier temps, nous identifions les capteurs déployés dans chaque secteur, afin de récupérer les fichiers de données correspondant. Nous n'utilisons que les mesures de débit, puisque nous nous intéressons à la consommation uniquement. Ensuite, nous récupérons les valeurs de chaque fichier sur une période de 24 heures, afin d'établir le débit journalier sur chaque capteur. L'étape suivante consiste à calculer la consommation journalière sur chaque secteur : nous additionnons les débits des capteurs en entrée du secteur, et soustrayons les débits en sortie. Seuls les capteurs de débit peuvent être entrants ou sortant, et cela est spécifié dans le fichier de configuration (un capteur frontalier sera en entrée sur un secteur, mais en sortie sur un autre). Ainsi, il est possible qu'un secteur ait une consommation journalière négative ; cela représente les retours d'eau potable, qui peuvent survenir dans certains cas, ou des dérivations du réseau qui perturbent la

distribution normale. Nous obtenons ainsi la consommation journalière pour chaque secteur.

Ensuite, nous récupérons dans le fichier de configuration la longueur totale de canalisations (en kilomètres) pour chaque secteur de consommation. Nous divisons la consommation journalière (la fréquence de réception est d'une valeur toutes les quinze minutes) de chaque secteur par la longueur de son réseau afin d'obtenir le rapport de consommation, qui correspond à la consommation par kilomètre de conduite. Afin de réduire la marge d'erreur dans chaque cas, nous calculons ces valeurs sur sept jours consécutifs afin d'établir une moyenne hebdomadaire ; cela nous permet d'éviter les imprécisions liées aux différences de consommation entre week-end et jours ouvrés. On peut résumer le ratio ainsi : $ratio = (\sum_{t=0}^{1jour} mesures)/longueur\_reseau$

Une fois le rapport de consommation établi pour chacun des secteurs, nous pouvons l'utiliser afin de déterminer la méthode de profilage la plus adaptée. Dans la suite, les valeurs seuils, qui déterminent si un rapport de consommation est faible ou élevé, ont été déterminées en fonction de nos propres estimations, puis ajustées et validées par notre partenaire spécialiste du domaine de l'eau. Nous procédons ainsi :

- Si le rapport de consommation est faible, il y a peu de consommation, ou bien elle est répartie sur une large surface ; on utilise donc la méthode à base de polygones.
- Si le rapport de consommation est élevé, la consommation au kilomètre carré est élevée, et on peut s'attendre à trouver des gros consommateurs. On adopte alors le profilage basé sur des POI.
- Si le rapport de consommation est plutôt moyen, nous combinons les deux méthodes de profilage, afin d'équilibrer les proportions des classes de terrain qui sont difficilement représentables par chacune.

Lors de nos tests, la borne inférieure pour le ratio de consommation a été fixée à 30, et la borne supérieure à 50 (l'unité étant le mètre cube par heure par kilomètre de réseau). Nous avons défini ces valeurs en calculant dans un premier temps les deux types de profilage et le ratio pour chaque secteur, et en vérifiant visuellement, nous avons remarqué à partir de quelles valeurs il convenait de n'utiliser qu'une méthode, et quand il était nécessaire de les combiner. Nous avons présenté nos résultats aux experts de notre client et, en validant les scores finaux, ils ont également validé les bornes que nous avions fixées. Il se peut qu'elles aient à être ajustées en fonction du réseau étudié, et c'est pour cela qu'elles sont accessibles depuis le fichier de configuration.

4.4.4 Examen des résultats

Comme mentionné précédemment, nous avons testé nos différentes méthodes de profilage (et leur combinaison) sur divers échantillons de données. Nous disposons pour nos tests de plusieurs mesures historiques de notre partenaire industriel sur divers réseaux d'eau potable : notre premier cas d'utilisation porte sur la péninsule de Macao, en Chine, le deuxième sur la ville de Versailles et ses environs, et le troisième sur de très nombreux secteurs répartis autour de la Loire. Le format des données varie légèrement dans chaque cas, et il peut ne pas y avoir les mêmes données disponibles. Ainsi, nous disposons seulement du fichier de configuration pour les données autour de la Loire, ce qui ne nous permet pas de réaliser le rapport de consommation. Dans le cas de Macao, nous disposons d'un répertoire regroupant les consommations des plus gros clients de la zone.

Nous avons réalisé nos tests sur toutes les données mises à notre disposition, mais les premiers ajustements et vérifications ont été effectués sur la zone de Versailles. Cela est dû à des raisons pratiques : le réseau de Versailles contient onze secteurs de consommation, simples à étudier et dont on peut trouver d'autres données aisément en lignes. A l'inverse, il est moins aisé de récupérer des informations pour le réseau de Macao (noms chinois, problème de données libres) ; et pour la Loire, il nous manque les historiques de mesures.

Comme mentionné précédemment, nous nous sommes dans un premier temps basés sur des examens visuels pour juger de la précision de nos méthodes. La figure 17 présente les différents secteurs de consommation répartis sur la ville de Versailles et ses environs, superposés par dessus la couche de données d'OpenStreetMap via Google Earth. En activant/désactivant l'affichage des secteurs et de la couche de données, tout en comparant les résultats de notre système de profilage, nous pouvons affiner le fichier de scores, ajuster les seuils du rapport de consommation, et juger de la pertinence de notre approche.

Le tableau 1 présente le résultat des deux méthodes de profilage (avec POI et avec polygones). On note clairement les limites de chacune des méthodes pour certains types de surface. On peut également remarquer leur complémentarité, dans le cas de secteurs à surfaces mitigées : les proportions finales sont moins différentes entre les deux méthodes. Nous avons présenté ces résultats à notre partenaire expert du domaine de l'eau, et il les a jugés très satisfaisants. Nous avons donc intégré nos méthodes de profilage au

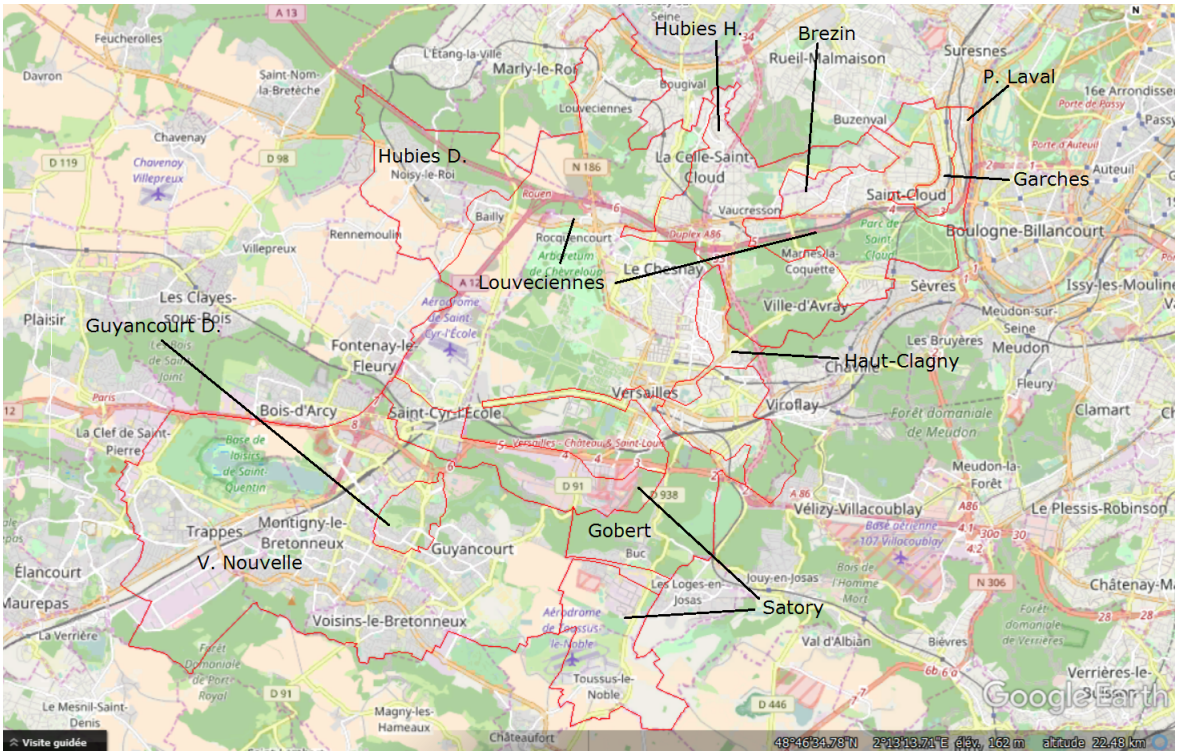


FIGURE 17 – Secteurs de consommation du réseau de Versasilles

module de raisonnement dans le projet.

4.5 Profilage indépendant

Cette première version du profilage géographique se base donc sur des paramètres de configuration statiques et des mesures, en plus des sources de données géographiques. Certains modules ne permettent pas de disposer de telles informations : soit parce qu'elles n'ont pas été représentées, soit parce que le module n'a pas accès à ces informations. Nous avons donc appliqué des modifications au profilage géographique, afin de le rendre exécutable avec OpenStreetMap pour seule source de données.

En effet, sans des échantillons de mesures historiques, il est impossible de

zone	type de surface									
	industrielle		naturelle		résidentielle		touristique		agricole	
	POI	région	POI	région	POI	région	POI	région	POI	région
P Laval	15.22	15.81	3.26	80.93	73.37	3.26	8.15	0.00	0.00	0.00
V Nouvelle	13.41	17.87	1.45	54.76	56.06	19.68	29.08	0.22	0.00	7.47
Hubies D.	4.55	0.00	0.00	3.45	29.55	55.72	65.90	0.02	0.00	40.81
Brezin	0.00	0.00	0.00	80.99	100.00	19.01	0.00	0.00	0.00	0.00
Guy. D	0.00	7.01	0.00	90.16	96.87	0.60	3.13	2.23	0.00	0.00
Louv.	13.31	1.88	2.63	64.29	45.96	11.85	38.10	2.61	0.00	19.37
Hubies H	1.17	0.02	1.31	94.02	35.33	5.48	62.19	0.48	0.00	0.00
H Clagny	20.95	0.45	9.52	69.49	60.95	29.79	8.58	0.27	0.00	0.00
Garches	7.20	0.00	5.95	70.08	27.57	29.76	59.28	0.16	0.00	0.00
Gobert	15.94	0.00	2.72	47.71	24.46	51.62	56.88	0.67	0.00	0.00
Satory	6.81	59.41	0.00	12.49	71.91	13.27	21.28	0.06	0.00	14.77

TABLE 1 – Résultats des deux méthodes de profilage, en pourcentage

réaliser le profilage selon les méthodes évoquées précédemment. Il en va de même pour le fichier de configuration : son absence ne nous permet pas de connaître la répartition de capteurs, ni la taille du réseau, ni même de modéliser les secteurs géographiquement. Nous avons donc développé des méthodes de calcul de ratio alternatives pour pallier ces problèmes, et offrir une autre option en cas de besoin. Les secteurs de consommation, représentés à partir des fichiers de configuration, peuvent être remplacés par des bounding boxes : bien que moins précises, elles sont plus faciles à représenter et permettent de simplifier plusieurs étapes du programme (vérification d'appartenance, calcul d'aire). Cela améliore donc également le temps d'exécution du profilage.

Le ratio de consommation peut lui aussi devenir incalculable, faute de valeurs pour le réaliser. Il nous fallait donc trouver une autre approche pour la précision du résultat, en ne nous basant que sur les données géographiques à notre disposition. Nous avons adopté une approche relativement similaire, en calculant cette fois-ci le ratio de POI au kilomètre carré ; de cette façon, nous avons uniquement besoin de l'aire de la bounding box étudiée, et du nombre de POI qui y sont représentés. Nous avons repris le jeu de données de la ville de Versailles, et comparé les résultats de cette nouvelle méthode avec l'ancienne, après avoir à nouveau ajusté les valeurs des seuils supérieur et inférieur dans le fichier de propriétés. Dans plus de la moitié des cas, les résultats trouvés sont identiques, avec des différences de rapport pour les

zones avec une proportion de zones vides (sans POI, *p.ex.* naturelle ou agricole) approximativement égales : la densité de POI ne permet pas toujours de mettre en valeur ce type de surface. Toutefois, ce n'est pas particulièrement problématique : dans notre cas d'usage, ce sont principalement les zones avec POI qui seront principalement génératrice d'anomalies ; par ailleurs, cette méthode identifie toujours correctement les secteurs majoritairement vides (zones agricoles, forestières, etc).

Sans le détail des secteurs de consommation dans les fichiers de configuration, nous ne disposons plus des dénominations des différentes zones. Ces noms étaient utilisés pour l'enregistrement des données extraites sous forme de fichiers, nous ne pouvons donc plus identifier de manière automatique les zones à modéliser. La solution adoptée a été de ne conserver qu'un seul fichier, et d'ajouter un paramètre afin de demander à l'utilisateur s'il souhaite l'utiliser ou le régénérer ; ainsi, un secteur extrait peut être réutilisé plusieurs fois, puis remplacé par un autre si besoin en ne modifiant qu'une variable.

4.6 Intégration

Notre système de profilage est intégré dans le projet Waves, plus précisément dans le module Scouter [35], chargé de la contextualisation des anomalies précédemment détectées. Nous allons détailler son fonctionnement.

4.6.1 Organisation générale

L'identification des sources d'anomalies, intérêt majeur de notre client au sein du projet, ne pouvait se faire qu'en étudiant des sources de données externes. En effet, les données liées au projet ne concernent que le réseau de distribution, et il est impossible de trouver les origines des anomalies à partir de ces éléments. Nous avons besoin de sources événementielles susceptibles d'être à l'origine des anomalies (*e.g.,* avec des composantes temporelles et spatiales proches) à analyser. Toutefois, repérer les événements pertinents pour notre cas d'usage et juger de leur responsabilité nécessite un traitement spécifique. Car les données événementielles ne sont pas souvent sémantisées : les informations sont contenues dans du texte (articles, posts...) qui doit être analysé. De plus, il faut qualifier la pertinence des éléments retenus.

Le module Scouter [35] se charge d'analyser des flux de données extérieurs au projet, afin d'identifier les éléments qui peuvent avoir causé des anomalies

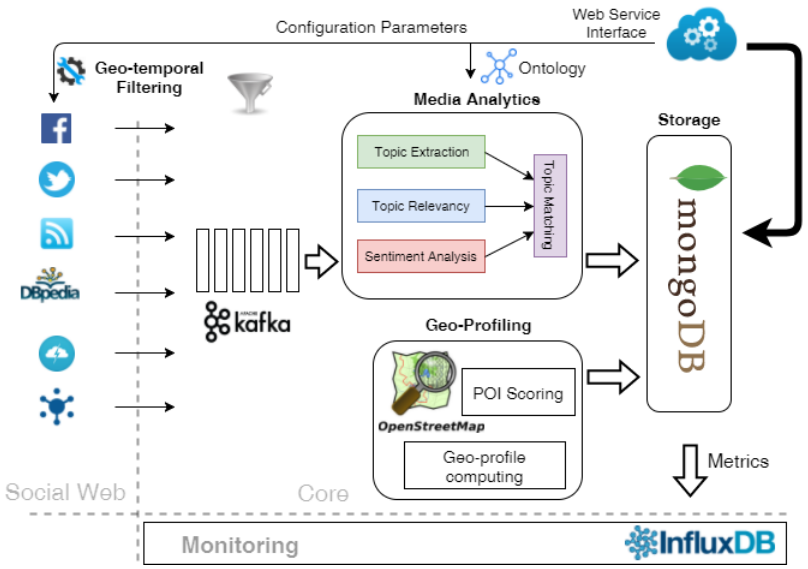


FIGURE 18 – Organisation du module Scouter

sur le réseau de distribution d'eau potable. Son architecture est résumée dans la figure 18 : divers flux de données événementielles sont fournis au module, ce dernier se charge de les analyser selon plusieurs paramètres (extraction de sujets, analyse de sentiments...). Un score de pertinence est ensuite calculé à partir de la quantité et de la qualité des éléments pertinents identifiés, du profilage de la zone géographique concernée, et du type d'anomalie détecté.

Le module est distribué, chaque source de données étant dirigée vers un topic Apache Kafka spécifique ; les résultats peuvent également être stockés pour un usage ultérieur. L'intégralité des composants est paramétrable : le profilage peut être adapté à divers types de surfaces, il est possible d'ajouter ou de supprimer des sources de données événementielles et de modifier la manière dont elles sont analysées. La modularité du composant lui permet de s'adapter à de nombreux cas d'usage.

4.6.2 Analyse de texte

Toutes les données événementielles sont récupérées sous forme de texte (plus ou moins long en fonction de la source), et il faut pour chaque cas

l'examiner pour juger de son importance : les métadonnées ne sont d'aucune utilité, elles ont seulement permis la localisation spatio-temporelle.

La taille de l'information est très limitée pour les posts provenant de Twitter (140 caractères maximum) ou de Facebook ; toutefois, cela peut poser un problème pour les documents de presse (environ une page). Néanmoins, les articles présentent l'avantage d'être structuré en rubriques, avec un regroupement des informations les plus importantes situées au début (titre, chapeau, résumé) ; par conséquent, l'essentiel de l'information est condensé dans la première partie du texte ; le reste ne contient que des détails. C'est pourquoi Scouter ne conserve que les 1200 premiers caractères des articles les plus longs pour analyser l'événement.

Afin d'analyser efficacement des textes de longueur et de contenu variable, on peut recourir au traitement du langage naturel, ou *Natural Language Processing* (NLP) en anglais. Toutefois, la linguistique est un domaine très vaste et toujours en développement de nos jours ; intégrer un traitement développé du langage naturel au module demanderait des connaissances et du temps dont nous ne disposons pas. L'analyse des événements se fait donc de la manière la plus simple et efficace possible, en se basant sur une liste de mots-clés.

4.7 Résumé

Dans ce chapitre, nous avons présenté notre implémentation d'un système de profilage géographique intégré dans Waves. Nous avons démontré l'utilité de l'étude des données spatiales disponibles dans le projet, en montrant les limites des données temporelles en ce qui concerne la contextualisation des anomalies détectées. Notre système de profilage se base sur deux méthodes différentes, l'une basée sur l'étude de points d'intérêt, l'autre impliquant des surfaces de polygones. Ces deux approches peuvent être combinées via diverses méthodes, afin d'obtenir le résultat le plus satisfaisant possible.

L'ensemble de notre système est paramétrable, et peut être configuré pour s'adapter à divers cas d'usage, et fournir une solution optimisée. Les évaluations réalisées ont mis en évidence des performances satisfaisantes ; de même, les données cartographiques utilisées dans notre système ont été soigneusement étudiées, et nous nous sommes assurés de retenir la meilleure source possible parmi celles à notre disposition.

Notre approche a dans un premier temps été présenté au cours d'un work-

shop à une conférence nationale [36] et comme poster à cette même conférence. Le module Scouter, dans lequel il est utilisé, a été publié dans une conférence internationale [37], et a également reçu le prix du meilleur article applicatif à une autre conférence [35].

5

Sémantisation et sérialisation de flux RDF

5.1 Contexte et besoins

Le traitement des données reçues constitue la problématique majeure de l'IdO. Dans le cas d'infrastructures développées et/ou étendues, il est très fréquent que les capteurs produisant les mesures ne partagent pas le même format de données, ni même un modèle de données identique. Cela peut être lié à des différences de versions, mais aussi plus simplement au fait que les mesures effectuées sont différentes, et donc représentées dans un format/modèle adapté en conséquence. Comme indiqué dans la figure 1, le projet Waves effectue une conversion des données reçues dans le modèle de données RDF, *i.e.*, un modèle de graphe labellisé et orienté. Nous avons choisi ce type de modèle afin de faciliter les opérations à réaliser sur les données par la suite (intégration de données, sémantisation, requêtage et raisonnement). En effet, nous utilisons du raisonnement pour l'identification d'anomalies, ainsi que pour la recherche de leurs origines. Par ailleurs, les relèves des capteurs seules ne permettent pas toujours de raisonner de manière efficace, car elle contiennent peu d'éléments pour pouvoir être traitées efficacement (des données compactes sont transmises plus efficacement, comme présenté en figure 19 (a)). Par ailleurs, dans notre cas d'usage, nous avons souvent besoin d'accéder à la configuration du réseau pour la recherche d'anomalie (représentation RDF statique, figure 19 (b)), et de traiter des flux de données extérieurs pour la détection d'origine (sources événementielles). Il y a donc parfois un besoin d'inférence et de gestion de sources multiples pour compléter les flux, ce qui

(a) Extrait de flux :

```
?x1 id "QDT01"
?x1 timestamp "2014-01-01T07:15:00"
?x1 pressureMeasure _:x2
?x2 value 7.03
```

(b) Extrait de la base statique :

```
<http://www.zone-waves.fr/sector#Hubies_Haut> a ssn:Platform ;
rdfs:label "Hubies Haut" .

<http://www.zone-waves.fr/sensor#QDT01> ssn:onPlatform <http://www.zone-waves.fr/sector#Hubies_Haut> .

waves:flow1  a cuahsi:inputFlow .

<http://www.zone-waves.fr/sensor#QDT01> ssn:observes waves:flow1  .

waves:flow1 cuahsi:relatedTo <http://www.zone-waves.fr/sector#Hubies_Haut> .

<http://www.zone-waves.fr/sensor#DT01> a ssn:Sensor ;
rdfs:label "QDT01" ;
wgs84_pos:lat 4.8834914E1 ;
wgs84_pos:long 2.144792E0 .
```

FIGURE 19 – Exemple de données RDF de Waves

justifie encore plus l'usage du RDF.

Toutefois, le raisonnement sur des flux RDF est une problématique de recherche majeure de nos jours, principalement liée au besoin de performance, que ce soit en terme de volume de données ou de vitesse de traitement. Il s'agit d'une opération coûteuse, et son application sur un débit de données important impacte énormément les performances. Aucune des techniques existantes n'est parfaitement adaptée à tous les cas : il n'y a pas de solution globale, chaque approche que nous avons étudiée est adaptée pour certains cas précis (fenêtre de données temporelle, par exemple), mais perdra en performance ou ne sera fonctionnelle dans d'autres cas d'utilisation. Puisque Waves a pour vocation d'être une plateforme portable et adaptative, nous avons étudié les alternatives existantes pour trouver une solution adaptée. Par ailleurs, se baser sur les formats RDF existants (turtle, n3, trig, RDF/XML, ntriples..) n'est pas satisfaisant dans un contexte de gestion de flux : en effet, aucun d'eux ne tient compte du schéma commun des flux entrants, qui reste relativement statique pour chaque capteur. De même, il serait préférable de se

baser sur une approche plus compacte, car aucun des formats mentionnés n'est très efficace pour la représentation de larges volumes de données, *e.g.*, l'omniprésence d'URIs.

Plusieurs solutions ont déjà été expérimentées afin de tenter de manipuler des flux RDF efficacement. Nous avons mentionné dans notre état de l'art l'existence de méthodes de compression qui peuvent être adaptées aux flux, offrent divers avantages. En effet, des éléments plus compacts permettent d'améliorer la fréquence de transmission et de réception. Plusieurs algorithmes ont donc été développés (section 3.1), avec pour chacun une approche différente. Malgré tout, aucune n'est parfaite, elles présentent toutes des défauts, et ne sont généralement pas adaptées au traitement de flux. L'approche de RDSZ attribue des identifiants aux sujets et aux objets différents, ce qui se révèle un mauvais choix pour des flux, dont les littéraux changent très régulièrement. Zstreamy est plus adapté, mais son en-tête peut dans certains cas dupliquer l'horodatage des données, ce qui n'est pas nécessaire, et peut prêter à confusion. Il y a également un processus de compression et de décompression potentiellement plus long. ERI se base sur RDSZ, en apportant des améliorations : dans cette approche, les propriétés sont également encodées, mais le problème des littéraux évoqué précédemment persiste. C'est pour ces raisons que nous avons choisi de développer une nouvelle sérialisation, en reprenant certains principes des travaux étudiés, tout en les adaptant à un contexte de traitement de flux.

En ce qui concerne le raisonnement, deux solutions indépendantes existent afin d'améliorer les résultats des requêtes appliquées sur les flux. La première méthode consiste à matérialiser dans les flux les connaissances manquantes qui peuvent être déduites d'autres sources de données. Les éléments du flux ainsi complétés seront plus susceptibles de répondre aux requêtes continues qui leur sont appliqués. Prenons un cas d'utilisation inspiré de notre cas d'usage : si le graphe issu d'un de ses flux indique qu'un élément est un objet physique, il est aussi possible que l'élément soit, par extension, un capteur, ou bien un moteur ; tout dépend de la précision de la base de connaissances (on peut également avoir des sous-classes de capteurs). On peut donc enrichir le flux avant d'y appliquer la requête, ce qui permet d'obtenir des résultats plus complets ; cela permet de garantir que certaines requêtes seront alors satisfaisables par l'ajout de données provenant d'une base statique (potentiellement liée au projet). Dans Waves, les flux sont le plus compact possible pour favoriser les performances, et la plupart des informations statiques concernant les capteurs (position, unité de mesure, secteur rattaché...) sont stockées dans

une base de connaissances externe ; d'où l'intérêt de l'enrichissement, en cas de requête complexe (besoin de matérialisation). Toutefois, l'enrichissement de graphes peut être une opération coûteuse en performance : il faut s'assurer que ce qui est matérialisé dans les flux soit minimal, de sorte que la requête puisse obtenir la réponse avec un flux surchargé au minimum. Par ailleurs, comme chaque flux conserve sa structure globale pour chaque échantillon, il n'est pas nécessaire de recalculer le besoin de matérialisation pour chaque arrivée de flux. La partie à matérialiser est évaluée à l'initialisation et lors de la modification des flux. S'il n'y a pas de changement, elle peut simplement être ajoutée mécaniquement pour chaque graphe du flux. La matérialisation implique donc une considération de la sémantique du flux et de la requête (puisque la partie à matérialiser dépend des deux).

L'autre solution consiste à réécrire les requêtes, en prenant en compte tous les cas possibles. Cette approche s'oppose donc à la matérialisation : au lieu d'enrichir les flux de données, c'est la requête elle-même qui est reformulée afin de prendre en compte tous les cas possibles. Si l'on se base sur l'exemple précédent, on peut ajouter trois clauses à la requête afin de prendre en compte les différents cas possibles (figure 20 (b)). Cette méthode a l'avantage de préserver les graphes : les performances ne sont donc pas diminuées lors de la manipulation des flux. Néanmoins, la nécessité de réécrire les requêtes provoque un allongement de leur temps d'exécution du fait de la génération de la réécriture et du traitement de toutes les réécritures possibles (*e.g.*, les BGP des unions de la forme réécrite). Cela pose un problème important considérant le caractère continu du traitement de requêtes dans le cadre d'une application de gestion de flux de données.

Pour pallier ces problèmes, nous avons développé un encodage qui combine la réécriture de requêtes et la matérialisation d'information. Dans un premier temps, nous avons étendu LiteMat (présenté en section 2.5), afin de pouvoir supporter les propriétés `owl:sameAs` ; cette amélioration permet à notre sérialisation d'élever le niveau d'expressive supporté par le système et ainsi répondre à un problème de dé-duplication de certains identifiants de la base de faits. Ensuite, nous avons développé PatBin, un encodage qui réutilise les identifiants de LiteMat pour compresser des graphes RDF ; il est composé de deux structures dans lesquelles il est aisé d'ajouter de nouvelles connaissances. Chaque forme encodée avec nos méthodes peut être utilisée et manipulée sans décompression : ainsi, notre algorithme offre un compromis entre la matérialisation et la réécriture de requête sur des formes compactes. Les sections suivantes détaillent nos deux contributions : d'abord l'enrichis-

(a) Requête de base

```
SELECT ?x
WHERE {
?x rdf:type DUL:PhysicalObject .
}
```

(b) Requête réécrite

```
SELECT ?x
WHERE {
?x rdf:type DUL:PhysicalObject
} UNION {
?x rdf:type ssn:Sensor
} UNION {
?x rdf:type ssn:SensingDevice
} UNION {
?x rdf:type dbo:Engine
}
```

FIGURE 20 – Réécriture de requête

sement de LiteMat, et ensuite PatBin, qui permet de raisonner directement sur des formes compressées. Nous terminons par une évaluation de notre approche.

5.2 Extension de LiteMat

LiteMat est un encodage qui s'adapte bien au traitement de flux : comme nous l'avons expliqué dans les sections précédentes, les graphes issus des flux de données ont généralement une structure fixe : les sujets et les prédicats n'évoluent pas, seuls les objets (et plus spécifiquement les littéraux) vont changer d'une mesure à une autre. Utiliser LiteMat pour encoder ces deux premiers éléments offre donc une sérialisation statique pour une partie des flux, ce qui offre un grand avantage. La seule limite à LiteMat apparaît lors de modifications de la Tbox de l'ontologie : cela invalide l'ensemble des identifiants calculés, et impose potentiellement un ré-encodage complet. Dans un contexte de traitement de flux, le problème est moins impactant en terme de perte de performance car les requêtes continues fournissent des résultats qui sont relativement éphémères, c'est-à-dire qu'elles traduisent une situation à un instant donné.

A priori, dans les cas concrets que nous avons rencontrés dans nos expérimentations dans Waves, si l'ontologie change entre deux requêtes, il n'est pas nécessaire de maintenir des correspondances ces deux versions de l'ontologie. Prenons un cas d'usage de raisonnement sur des flux, illustré dans la

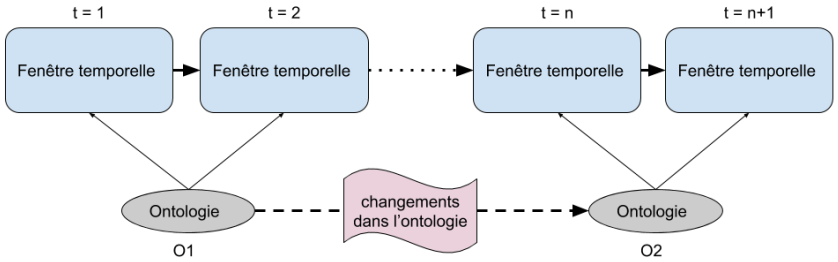


FIGURE 21 – Impact de la mise à jour de l'ontologie sur les flux

figure 21. Dans un premier temps, l'ontologie est à l'état O_1 : les flux sont donc encodés en considérant O_1 à leur arrivée dans Kafka (pour t_1, t_2...). Supposons que l'ontologie change à un instant t_n, et qu'elle passe à l'état O_2 : alors les flux sont encodés à leur arrivée en considérant O_2. Il n'y a pas de conflit, les résultats obtenus en t_1 sont valides avec O_1 et ceux reçus en t_n sont corrects avec O_2. Si les flux doivent être rejoués (c'est un cas d'usage fréquent dans la gestion de flux, pour lequel la capacité de Kafka à faire de la rétention de flux est reconnue), alors on récupérera les échantillons reçus en t_1 (par exemple) et on les ré-encodera avec le dernier état de l'ontologie (O_2 dans l'exemple) ; le résultat obtenu sera différent de celui obtenu à l'instant t_1, mais il sera valide considérant l'ontologie courante. Effectuer l'encodage des données directement à partir des capteurs est techniquement possible, mais cela implique de leur envoyer les nouvelles versions de l'ontologie à chaque mise à jour. Considérant les faibles capacités énergétiques des capteurs sur le long terme, ce n'est pas une solution recommandée.

La propriété `owl:sameAs` est beaucoup utilisée dans le Web Sémantique pour faire le lien entre des individus dans une même base de connaissances ou bien entre base de connaissances, *e.g.*, le Linked Open Data. C'est une propriété très pratique, car il est fréquent que des individus pourtant identiques ne soient pas représentés de la même manière ; identifier et associer des connaissances identiques est un avantage majeur. Toutefois, les `owl:sameAs` ne sont pas évidents à représenter dans le bases de connaissances, ni à gérer lors du requêtage. En effet, représenter des individus similaires peut causer une augmentation considérable des bases de connaissances (*i.e.*, par matérialisation complète d'une clique d'individus reliés pas la propriété `owl:sameAs`).

De plus, l'exécution de requêtes sur des graphes comportant des `owl:sameAs` peut s'avérer délicate si certains liens manquent ou ne sont pas pris en compte, *i.e.*, coût important d'une réécriture de requête.

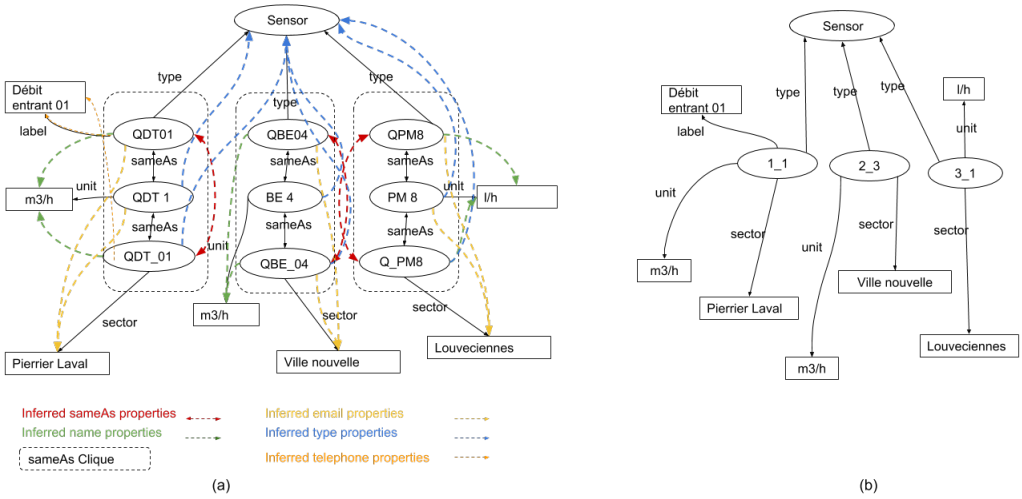


FIGURE 22 – Exemple de traitement des propriétés sameAs par LiteMat

La figure 22 donne un exemple d'utilisation de `owl:sameAs`, en reprenant les données des capteurs de notre cas d'usage. Sur la partie (a), il y a trois dénominations possibles pour chaque capteur, reliées par des propriétés `owl:sameAs`. Chaque représentation n'est reliée qu'à une partie des informations (label, unité ou secteur), et il est nécessaire de dupliquer et matérialiser ces informations pour chaque représentation si l'on veut qu'une requête spécifique à une représentation dispose de tous les triplets qui lui sont liés (ou alors il faut réécrire la requête). On peut voir que le nombre de triplets ajouté de la sorte est assez important, 23 au total pour un exemple aussi simple ; pour une ontologie plus développée et/ou avec plus de propriétés `owl:sameAs`, l'augmentation du nombre de triplets devient vite exponentielle.

Nous avons adopté une représentation alternative dans LiteMat, afin d'éviter le problème de matérialisation excessive. L'ensemble des représentations liées par des propriétés `owl:sameAs` sont regroupées en une seule clique, suivant un certain schéma : chaque représentation au sein d'une clique possède son propre identifiant numérique, et chaque clique possède un identifiant

de référence. Ainsi, une clique respecte l'encodage de LiteMat et bénéficie des avantages expliqués en section 2.5 ; elle conserve les différents concepts en son sein, et chacun peut être identifié séparément des autres en accédant à la clique. La partie (b) de la figure 22 illustre la solution adoptée dans LiteMat à partir de l'exemple (a) (sans compression) : chaque clique est identifiée par son propre identifiant (par ordre croissant de gauche à droite dans l'exemple), et chaque élément de la clique a également un identifiant qui lui est propre (croissant de haut en bas dans l'exemple). Ainsi, l'identifiant de la première clique (noté 1_1) est QDT01, l'identifiant de la deuxième clique (2_3) est QBE_04, et ainsi de suite. Cette correspondance est enregistrée dans une table de hachage, qui peut être accédée par la suite en cas de besoin. Ainsi, chaque élément des cliques est aisément accessible à partir de l'identifiant du représentant.

5.3 Un encodage performant : PatBin

PatBin utilise les identifiants encodés avec LiteMat afin de créer une version compressée de graphes RDF composée de deux éléments : le pattern et le binding. Le premier permet d'encoder la structure du graphe, le second se charge des objets des différents triplets.

5.3.1 Représentation en Pattern-Binding

La motivation derrière cet algorithme vient de l'étude du raisonnement appliqué sur les flux de données : en analysant les opérations effectuées à cette étape du projet, nous avons identifié plusieurs détails intéressants sur lesquels nous avons basé notre encodage.

D'une manière générale, un flux de données RDF provenant d'une source donnée change rarement au cours du temps, c'est-à-dire que les graphes reçus seront globalement identiques : seuls les littéraux (les objets des triplets, *e.g.*, les mesures des capteurs) seront amenés à varier. Cette affirmation est vraie pour la plupart des flux, indépendamment de leur nature : en effet, si la structure d'un flux changeait fréquemment, il ne serait plus possible de lui appliquer des requêtes continues. De même, la structure des requêtes utilise un sous-graphe du flux pour construire sa clause WHERE ; étant exécutée sur une longue période (requêtes continues), cette structure ne varie pas. Cette approche a été exploitée par RDSZ, et nous la reprenons également,

mais d'une façon plus poussée. En effet, nous nous basons sur un encodage intelligent (LiteMat), nous exploitons certaines spécificités des flux RDF, et l'exploitation de notre encodage est différente, comme nous le démontrerons par la suite. Nous avons donc imaginé une structure compressée reprenant les identifiants de LiteMat et exploitant la stabilité des flux. Nous avons divisée la structure en deux catégories, qui donnent le nom à notre algorithme : le pattern, et les bindings.

Le pattern d'un graphe représente sa structure, c'est à dire l'interconnexion entre ses propriétés. Un pattern reprend les identifiants des propriétés, et les organise pour conserver la structure du graphe RDF. Les propriétés des triplets sont séparées par des deux points " :", et les connexions sont représentée par des parenthèses "(", ")" ; on parle de connexion entre deux triplets lorsque l'objet de l'un d'eux est le sujet de l'autre. Cette forme ne conserve pas l'essentiel des concepts du graphe : les sujets, les classes et les nœuds vides, et ce, pour des raisons d'optimisation et d'utilisation de l'encodage. En ce qui concerne les sujets, ils ne sont pas nécessaires, puisque les interconnexions entre les triplets sont préservées dans la forme compressée. Les nœuds vides, quant à eux, sont remplacés simplement par des espaces dans le binding : leur valeur n'est jamais utilisée lors du traitement des flux (ils permettent des liens inter-triplets, déjà modélisés dans le pattern). Les objets, quant à eux, sont bien plus pertinents, mais chaque graphe provenant d'un flux RDF a potentiellement des objets différents. Puisque les patterns ont pour but de conserver une structure stable, les objets sont donc enregistrés dans les bindings : les littéraux conservent leur forme, et les URI de concepts peuvent être représentées sous une forme simplifiée si besoin (l'URI n'a pas d'intérêt dans notre encodage). La forme compressée obtenue est ainsi très simple d'utilisation et de comparaison : son objectif est uniquement de favoriser le raisonnement, de permettre un requêtage rapide, en ne conservant que l'essentiel. Si une forme PatBin est moins expressive qu'un graphe RDF complet, elle est aussi bien plus compacte, et totalement adaptée à du traitement de flux.

En plus des spécificités de l'encodage pour représenter les liens entre les triplets, les éléments du pattern sont classés de manière croissante. Cet ordre est très important, car cela permet à notre représentation d'être déterministe, et cela favorise grandement les opérations de manipulation des formes encodées (notamment pour la recherche de correspondances). Les bindings bénéficient également du déterminisme, car ils sont conçus en suivant le modèle du pattern : pour chaque propriété de celui-ci, le binding va stocker

l'objet qui y est associé, en notant les valeurs des littéraux, et en remplaçant les nœuds vides par des espaces ' '. Chaque valeur est séparée par un point virgule ';', comme pour le format csv. Les bindings ont donc un nombre de valeurs correspondant au nombre de propriétés du pattern auquel ils sont liés, mais leur connexion n'est plus représentée. On peut donc représenter un flux RDF en lui attribuant un pattern, qui joue aussi le rôle d'identifiant et peut être utilisé pour comparer les requêtes qui lui sont applicables. Les graphes peuvent quant à eux être remplacés par des bindings correspondants : on obtient ainsi une représentation très compacte.

Nous considérons que la structure des graphes reçus du flux ne change pas : c'est pour cela que les patterns peuvent servir d'identifiants. En effet, des structures différentes devraient normalement constituer des flux différents, pour recevoir un traitement qui leur est propre. De même, il est difficile d'imaginer que la structure des graphes d'un flux change régulièrement : cela rendrait l'utilisation de requêtes continues bien plus complexes. On peut considérer l'éventualité qu'un flux n'envoie pas un unique graphe, mais plusieurs (soit une forêt) : dans ce cas, il y aura également plusieurs patterns, correspondant à ces graphes, qui serviront d'identifiants. De même, il faudra recevoir plusieurs bindings pour un seul événement (on pourra considérer un caractère séparateur pour les émettre/recevoir en une fois).

La figure 23 présente un exemple d'encodage avec PatBin, en utilisant les identifiants générés par LiteMat. Avec cette forme de compression, nous conservons donc la hiérarchie des propriétés, mais également l'interconnexion entre les triplets. Ainsi, il est possible de raisonner directement sur les formes encodées, sans passer par une étape de décompression (*i.e.*, reconstruction du graphe original). De plus, la manipulation des patterns et bindings est simple : avec le classement des propriétés par ordre croissant, l'ajout, la suppression et la comparaison des encodages sont grandement simplifiés.

5.3.2 Exemple de matérialisation

Nous allons maintenant détailler le processus de matérialisation au sein des graphes compressés avec PatBin. Dans certains cas, les graphes issus des flux (compressés, dans notre cas) ne peuvent pas satisfaire une requête continue : il manque certains triplets dans le graphe. Ces informations manquantes se trouvent généralement dans une base de connaissances extérieure : il s'agit d'informations statiques, qui surchargeraient les flux de données si elles y étaient ajoutées. L'objectif de la matérialisation est d'ajouter le minimum

(a) Graphe RDF de base :

```
_:a <http://xmlns.com/foaf/0.1/name> "Alice" .
_:a <http://xmlns.com/foaf/0.1/knows> _:b .
_:b <http://xmlns.com/foaf/0.1/name> "Bob" .
_:a <http://xmlns.com/foaf/0.1/knows> _:c .
_:c <http://xmlns.com/foaf/0.1/name> "Eve" .
_:b <http://xmlns.com/foaf/0.1/mbox> "bob@gmail.com" .
```

(b) Table d'identifiants de LiteMat :

```
<http://xmlns.com/foaf/0.1/name>  = 23
<http://xmlns.com/foaf/0.1/knows> = 31
<http://xmlns.com/foaf/0.1/mbox>  = 37
```

(c) Conversion intermédiaire :

```
_:a 23 "Alice" .
_:b 23 "Bob" .
_:c 23 "Eve" .
_:a 31 _:b .
_:a 31 _:c .
_:b 37 "bob@gmail.com" .
```

(d) Pattern :

```
23:31:(23:37):31:(23)
```

(e) Binding :

```
"Alice"; ;"Bob";"bob@gmail.com"; ;"Eve"
```

FIGURE 23 – Exemple d'encodage PatBin

d'informations nécessaires au pattern et au binding pour pouvoir rendre la requête satisfaisable. Notre module de matérialisation doit donc être capable de détecter, étant donnée une requête, un flux et une base de connaissances statique, si le flux a besoin d'être enrichi, et si les informations récupérées de la base statique satisfont la requête. Techniquement, il est possible d'interroger plusieurs bases statiques pour récupérer les données manquantes, mais c'est cas à part. En effet, cela nécessiterait d'interroger les bases de connaissances en aveugle, car il est impossible de savoir quelle base détient l'information requise. Mais en premier lieu, il est fort peu pratique d'avoir les données statiques liées à un unique projet dispersées dans plusieurs bases : un regroupement dans une unique base de connaissances est souhaitable.

Afin d'illustrer les différentes étapes de matérialisation dans PatBin, nous allons utiliser un exemple dérivé du projet Waves, illustré en figure 24. Nous utilisons l'extrait d'un flux (en bleu), sur lequel sera appliqué une requête (en rouge) : chaque élément du flux peut être compressé avec PatBin, tout comme la clause WHERE de la requête. Nous avons joint un échantillon du dictionnaire regroupant les identifiants attribués aux propriétés avec LiteMat, afin de permettre une meilleure compréhension. Dans l'exemple, on peut voir que des parenthèses après l'identifiant 56 (*pressureMeasure*) encadrent l'identifiant 42 (*value*) : l'objet de la première propriété joue le rôle de sujet de la seconde. Dans les bindings, on remarque que les variables et/ou les nœuds vides sont remplacés par des espaces ; le classement des identifiants dans le pattern permet de vérifier à quelle propriété ils sont liés.

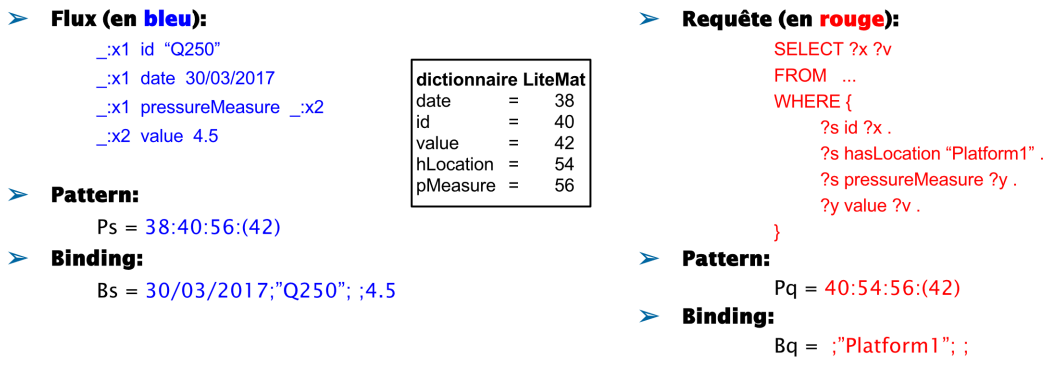


FIGURE 24 – Exemple d'usage de PatBin

On peut remarquer que cet exemple est problématique : la requête ne peut pas être appliquée directement sur le flux. En examinant celle ci, on comprend que l'on cherche à extraire des données reçues l'identifiant et la mesure de chaque capteur situé sur la *Platform1*. Or, ce dernier détail est manquant dans le flux : le graphe ne contient pas d'information quand à la localisation du capteur. Il faut extraire cette donnée de la base de connaissances statique pour enrichir le graphe du flux et ainsi pouvoir répondre à la requête.

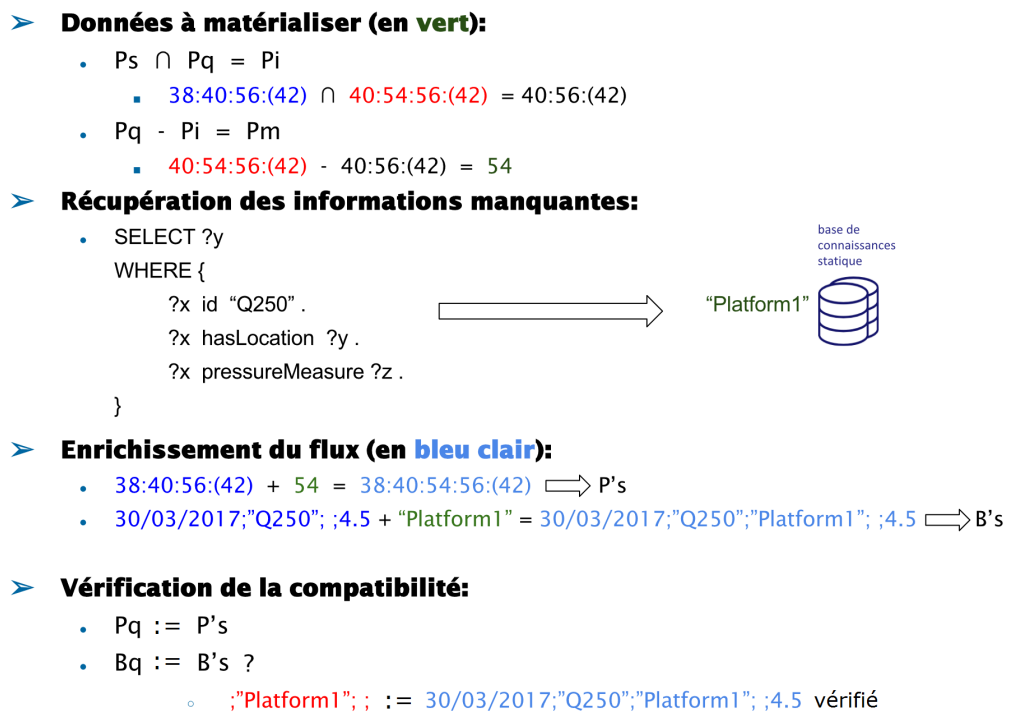


FIGURE 25 – Processus de matérialisation

Pour cela, nous allons utiliser directement les formes compressées du flux et de la clause WHERE de la requête, et matérialiser les triplets nécessaires. Dans un premier temps, il faut identifier quels sont les triplets à matérialiser (les propriétés dans le pattern). En réalisant l'intersection entre le pattern du flux et celui de la requête, nous obtenons la forme commune, c'est-à-dire les propriétés des triplets présents dans chacun des graphes. Puis nous soustrayons cette intersection au graphe de la requête, afin d'obtenir les pro-

priétés des triplets qui sont dans la clause WHERE, mais pas dans le flux : c'est ce que nous devons matérialiser. Le classement des identifiants intervient ici encore, car les opérations d'intersection et de soustraction seraient bien plus complexes sans cela. Les étapes permettant de réaliser l'intersection entre le pattern de requête (P_q) et le pattern de flux (P_s) sont détaillées dans l'algorithme 2. Dans un premier temps, les patterns sont éclatés selon leur séparateur, sauf au niveau des parenthèses : il faut garder l'identifiant de la propriété la précédant (fonction *splitPatternInter*). Ensuite, les listes éclatées sont réduites au fur et à mesure : les éléments communs au flux et à la requête sont ajoutés au résultat : cette étape se fait simplement par comparaison numérique, puisque les éléments sont classés par ordre croissant. Lorsqu'un élément est présent dans l'un des patterns et pas dans l'autre, il est ignoré ; l'algorithme compare les identifiants un à un et retire le plus petit à chaque fois. Si des parenthèses sont rencontrées, un traitement spécifique est appliqué en parallèle (fonction *intersectBranch*) : si la parenthèse et l'identifiant qui lui est associé ne sont pas communs aux deux patterns, l'intégralité de la branche représentée doit être supprimée ; sinon, le traitement normal est effectué. Ces étapes se terminent une fois que l'une des listes éclatées est vide : ce qui reste dans l'autre n'est pas dans l'intersection, et peut donc être ignoré. Enfin, il ne reste plus qu'à nettoyer le résultat à matérialiser (retirer les parenthèses vides et les identifiants associés) pour obtenir un pattern valide (fonction *clean*), qui correspond à l'intersection des patterns passés en argument.

Ensuite vient la soustraction entre le pattern de la requête (P_q) et le pattern d'intersection (P_i), illustrée dans l'algorithme 3. Elle reprend certaines fonctions de l'intersection : les patterns sont encore éclatés, mais chaque élément est isolé, les identifiants comme les parenthèses (fonction *splitPatternInter*). Puis les éléments des listes sont comparés un à un : les éléments identiques sont ignorés, seuls les éléments provenant de la requête et n'étant pas dans l'intersection sont ajoutés au résultat à matérialiser. Les parenthèses sont traitées à part (fonction *substractBranch*), pour des raisons semblables à celles évoquées précédemment : si certaines feuilles d'une branche sont manquantes dans le pattern d'intersection, il faut modéliser leurs nœuds intermédiaires, même s'ils sont communs aux deux patterns. Le parcours s'arrête lorsque les deux listes sont vides : si celle de l'intersection est vide en premier, il faut ajouter le reste de celle de la requête au résultat (car ce n'est pas présent dans l'intersection). Enfin, il ne reste plus qu'à nettoyer le résultat à matérialiser comme précédemment (rajouter les " :" avant les parenthèses

Algorithm 2 Intersection entre deux patterns

```
1:  procedure INTERSECTION(P_q, P_s)
2:      pq_elems = splitPatternInter(P_q)
3:      ps_elems = splitPatternInter(P_s)
4:      materialization = ""
5:      while !pq_elems.isEmpty() and !ps_elems.isEmpty() do
6:          if pq_elems[0].contains("(") then
7:              materialization.append(intersectBranch(pq_elems, ps_elems)
8:              continue
9:          end if
10:         if ps_elems[0] == pq_elems[0] then
11:             materialization.append(pq_elems)
12:             pq_elems.remove(0)
13:             ps_elems.remove(0)
14:         else if ps_elems[0] < pq_elems[0] then
15:             ps_elems.remove(0)
16:         else if ps_elems[0] > pq_elems[0] then
17:             pq_elems.remove(0)
18:         end if
19:     end while
20:     clean(materialization)
21:     return materialization
22: end procedure
```

ouvrantes, retirer les parenthèses vides et les identifiants associés) pour obtenir un pattern valide (fonction *clean*). Le résultat peut alors être renvoyé, il sera vide s'il n'y a pas besoin de matérialisation.

Algorithm 3 Soustraction entre deux patterns

```
1:  procedure SUBSTRACTION(P_q, P_i)
2:      pq_elems = splitPatternSub(P_q)
3:      ps_elems = splitPatternSub(P_i)
4:      materialization = ""
5:      while !pq_elems.isEmpty() and !pi_elems.isEmpty() do
6:          if pq_elems[0].contains("(") then
7:              materialization.append(substractBranch(pq_elems, pi_elems)
8:              continue
9:          end if
10:         if pi_elems[0] == pq_elems[0] then
11:             pq_elems.remove(0)
12:             pi_elems.remove(0)
13:         else
14:             materialization.append(pq_elems[0])
15:             pq_elems.remove(0)
16:         end if
17:     end while
18:     if !pq_elems.isEmpty() then
19:         materialization.append(pq_elems)
20:     end if
21:     clean(materialization)
22:     return materialization
23: end procedure
```

Si il y a un besoin de matérialisation, l'étape suivante consiste à interroger la base de connaissances statique afin d'obtenir les informations manquantes. La requête est construite en prenant les objets des propriétés manquantes en tant que variables pour la clause SELECT, mais il faut aussi compléter le graphe de la clause WHERE. En effet, les seuls triplets des propriétés manquantes ne sont pas suffisants pour spécifier suffisamment le résultat de la requête : dans notre exemple, demander la localisation (*hasLocation*) d'un nœud vide renverrait bien trop de résultats. Le graphe de la requête est donc complété en ajoutant les triplets issus du flux qui sont également présents

dans la base de connaissances statique ; cette information est obtenue en examinant les statistiques de LiteMat, qui recense les occurrences des propriétés statiques. Ces étapes sont illustrées dans l'algorithme 4, qui permet de créer la requête à exécuter à partir d'une forme compressée du flux (PB_s), du pattern à matérialiser (P_m), du dictionnaire LiteMat utilisé pour la compression ($Dico_{LM}$) et de la base de connaissances statiques ($staticKB$). Grâce aux statistiques réalisées par LiteMat lors de l'encodage initial, il est possible de savoir quelles propriétés sont présentes dans la base statique : on peut donc identifier dans le flux les informations qui sont également présentes dans la base statique, et les extraire pour enrichir la requête (fonction *extractStatic*). Ces informations peuvent alors être combinées à la forme à matérialiser pour concevoir le graphe de la clause *WHERE* de la requête (fonction *combinePatBin*). De cette forme, on peut également extraire les variables : il s'agit des objets qui ne sont pas sujets dans le graphe. Avec ces éléments, on peut alors construire notre requête, l'exécuter sur la base statique, et ainsi récupérer les éléments à ajouter aux bindings du flux.

Algorithm 4 Génération de requête pour la matérialisation

1: **procedure** QUERYMATERIALIZATION($PB_s, P_m, Dico_{LM}, staticKB$)
2: $staticPatBin = extractStatic(PB_s, Dico_{LM})$
3: $whereGraph = combinePatBin(P_m, staticPatBin)$
4: $variables = extractVariables(P_m)$
5: $query = buildQuery(variables, whereGraph)$
6: $result = executeQuery(query, staticKB)$
7: **return** $result$
8: **end procedure**

A ce stade, nous disposons donc des identifiants à matérialiser dans le pattern du flux, et des objets (littéraux) à ajouter à ses bindings ; on peut donc enrichir les données reçues (en bleu clair, figure 25). Pour les graphes du flux, on peut donc matérialiser l'identifiant dans le pattern, en respectant l'ordre croissant, et ensuite matérialiser le littéral récupéré dans le binding, en fonction de la position de l'identifiant matérialisé dans le pattern. Avec le flux enrichi, on peut répondre à la requête, car on dispose alors de tous les triplets nécessaires. Certaines données du flux peuvent être hors sujet (comme la *date* dans notre exemple), mais l'essentiel est que toutes les propriétés de la requête se trouvent dans le flux. Ainsi, nous avons pu matérialiser de nouvelles connaissances dans un graphe encodé avec nos deux méthodes. Nous pouvons

aussi bien utiliser la matérialisation que la réécriture de requête, et ce sans jamais passer par une étape de décompression.

5.4 Évaluation

5.4.1 Jeux de données

Les premiers tests de notre approche ont été réalisés sur des jeux de données fournis par notre partenaire sur le projet Waves, et correspondant à notre cas d'usage sur la gestion d'un réseau de distribution d'eau potable. Bien que très utiles au cours des étapes de développement et pour la correction de bugs, ces exemples présentent des inconvénients majeurs : ils sont limités en taille, car nous avons choisi de ne représenter que le strict minimum lors de leur conversion, et ne contiennent pas énormément de données à exploiter. De plus, ils ne peuvent être satisfaisants pour l'évaluation utilisés seuls, car les données de Waves sont d'usage privé, et nous ne pouvons pas dévoiler l'intégralité des informations à notre disposition.

Pour une évaluation plus complète, nous avions besoin de données plus consistantes, et si possible accessibles publiquement. Nous souhaitions stresser notre système avec de gros graphes : les capteurs de Suez n'envoient que de petits graphes, et nous souhaitions tester avec des volumes de données plus imposants. La taille des graphes impacte directement le temps d'encodage, qui peut devenir très long, mais Suez ne s'attend pas à détecter des anomalies à l'échelle de la seconde ou de la minute dans ce cas d'usage. Détecter des anomalies à l'échelle de l'heure serait déjà un avantage concurrentiel très important. Nous avons donc choisi d'utiliser le LUBM [38] (Lehigh University Benchmark), un outil permettant de récupérer des ontologies concernant des universités, et fournissant différentes métriques de performance et requêtes d'exemple. Nous avons généré deux jeux de données de différentes tailles : le premier concernant une université, le second regroupant les informations de dix différentes (dénotés respectivement LUBM 1 et LUBM 10 dans les sections suivantes). La seule différence majeure entre les deux jeux de données est leur taille (la quantité d'informations) : le format reste identique. Les deux exemples de test contiennent 14 classes différentes, et il y a 17000 individus pour le LUBM 1 contre 230000 pour le LUBM 10. Cela nous a permis de tester notre algorithme sur des graphes de taille importante, et de vérifier ses capacités et performances à traiter de larges volumes de données.

Dans les tableaux qui suivront, notre contribution sera testée sur l'exemple de notre projet et sur les jeux de données du LUBM 1 et 10, offrant ainsi trois évaluations différentes pour chaque expérimentation spécifique. Des détails concernant chaque jeu de données sont fournis dans la section suivante ; chacun diffère des autres en termes de taille. Du fait de la confidentialité des données de notre client, nous ne pouvons donner beaucoup de détails les concernant, mais il est possible de trouver de nombreux renseignements sur le LUBM en ligne [1]. Nous avons également mis en ligne le fichier turtle correspondant aux données du LUBM1 [2], mais celui du LUBM10 s'avère trop volumineux pour être uploadé.

5.4.2 Résultats et analyses

Exécution de la compression

Le tableau 2 résume les différentes caractéristiques de nos jeux de données, le temps et les taux de compression, ainsi que le nombre de graphes distincts obtenus après compression.

Les temps de compression sont jugés satisfaisants : plus il y a de triplets, plus le temps d'encodage est important, mais nous sommes capables de compresser près d'un million et demi de triplets en environ trente secondes. Le temps d'exécution passe à seulement trois secondes pour cent mille triplets. Puisque la compression à une telle échelle n'est pas censée être effectuée dans le cadre de flux , ce sont de bons résultats. Dans un scénario idéal, les données sont émises déjà compressées par les capteurs ; ou bien chaque graphe émit est compact et peut être compressé bien plus vite. Le ratio de compression pour l'exemple de Waves est moins intéressant car dans ce cas nous avons laissé notre algorithme générer la table d'encodage de LiteMat pour les propriétés : cela constituait une étape supplémentaire à réaliser. Il est difficile d'établir quelle étape est la plus coûteuse en temps : pour établir le pattern, il est nécessaire de consulter la table d'encodage de LiteMat, puis de trier les triplets en fonction des entiers obtenus. Il n'y a qu'après ces deux étapes que l'on peut établie le pattern, mais c'est aussi vrai pour le binding (car l'ordre est important. Toutefois, l'encodage final du pattern doit aussi tenir compte de la structure du graphe (ajout de parenthèses), et sera donc plus lent à réaliser.

1. http ://swat.cse.lehigh.edu/projects/lubm/
2. https ://github.com/JeremyLhez/PatBin/tree/master/LUBM

Mais comme notre cas d'usage se base sur des flux de données, nous avons également vérifié les différents patterns présents dans le résultat après compression. Dans chaque cas, il y en avait plusieurs distincts au sein de chaque résultat (il s'agissait donc d'une forêt, puisqu'il y a plusieurs graphes) ; dans notre cas d'usage, chaque capteur émet des graphes ayant le même pattern, avec un binding différent pour chaque mesure, comme nous l'avons expliqué précédemment. Dans nos exemples, les patterns distincts sont tous liés d'une certaine manière : certains graphes ont des branches communes, ou sont divisés en plusieurs parties. Il peut y avoir des graphes semblables à d'autres, avec des éléments manquants ou additionnels. Cela représente la diversité des données sur le personnel au sein des universités.

	Waves		**LUBM 1**		**LUBM 10**	
	turtle	**PatBin**	**turtle**	**PatBin**	**turtle**	**PatBin**
Données	822 triplets	90 encodages	103000 t.	12500 enc.	1450000 t.	175000 enc.
Taille	50 Ko	17 Ko	17 Mo	3 Mo	148 Mo	35 Mo
Détails sur la compression						
Temps	787 ms		2.970 ms		30 s	
Graphes	2 distincts		14 distincts		50 distincts	

TABLE 2 – Évaluation de la compression

Taux de compression

La plupart des taux de compression sont exposés dans le tableau 2. Le nombre de triplets varie fortement d'un exemple à l'autre, mais dans chaque cas, nous avons de bons taux de compression. Pour chaque exemple, il y a une grosse différence entre le nombre de triplets originel et le nombre d'encodages obtenu : un encodage représente un couple pattern-binding associé à un graphe (en présence d'une forêt, il y en a plusieurs). L'exemple de Waves est bien plus court que les autres car il ne s'agit que d'un échantillon de données, pris dans un contexte très spécifique : comme nous l'avons mentionné, il a servi principalement aux vérifications basiques de l'algorithme. La différence de taille après compression est également très bonne dans chaque cas ; nous avons retiré les URI des bindings, car elles ne sont pas pertinentes pour notre forme compressée. Elles sont normalement remplacées par des valeurs plus courtes, plus lisibles ; nous ne conservons que les littéraux, les plus susceptibles au changement. La compression peut également être encore améliorée

en ne gardant qu'un échantillon de chaque pattern différent, et en y associant ses bindings correspondants (en utilisant une map).

Formes compressées (LUBM 1)		
Triplets	Bindings	Taille
7	224	8438o
36	275	73Ko
40	55	17Ko
29	258	50Ko
37	1	511o
19	23	3053o
42	13	4366o
39	60	18Ko
35	2	805o
24	9	1801o
22	209	29Ko
13	4689	636Ko
23	5523	927Ko
34	1170	334Ko

TABLE 3 – Détails de compression des données LUBM 1

La répartition des bindings pour chaque pattern n'est pas proportionnelle. Comme nous l'avons expliqué précédemment, tous les patterns sont plus ou moins liés, avec des formes ou des branches communes, et/ou des informations manquantes ou additionnelles. Cela explique que certains sont plus fréquents que d'autres, avec plus de bindings associés. Nous avons résumé la répartition des bindings pour l'exemple du LUBM 1 dans le tableau 3 ; ce n'était pas pertinent de le faire pour l'exemple de Waves, qui ne comprend que deux patterns distincts, et il aurait été trop long de détailler les répartitions des 50 formes différentes du LUBM 10.

Nous n'avons pas ajouté les formes compressées à l'exemple, parce qu'une série d'identifiants encodant des propriétés ne serait pas très significatif. A la place, le tableau donne pour chaque pattern distinct le nombre de triplets dans le graphe qu'il représente, le nombre de bindings qui lui sont associés, et la taille sur disque que l'ensemble (pattern + bindings) représente. Comme on peut le voir, certaines formes sont très communes, et d'autres plus rares, pour les raisons évoquées. Pour le LUBM 10, chaque encodage comprend entre 7 et 46 triplets, avec au maximum 72 195 bindings associés à un pattern (et 2

au minimum). La taille de chaque entrée compressée varie entre 418 octets et 12Mo.

5.5 Intégration

PatBin a été réutilisé pour une autre contribution détaillées dans le chapitre suivant, et l'extension de LiteMat est reprise dans Strider [39], un système de raisonnement sur graphes distribués.

L'organisation de Strider est résumée dans la figure 26. Le système est divisé en deux parties : la première (à gauche sur la figure) reçoit les flux de données, les convertit, et les assigne à un topic Kafka en fonction de leur type de mesure. Ils sont alors compressés suivant l'encodage de LiteMat. En parallèle, la requête peut être décomposée (récupération du graphe de la clause *WHERE*) et compressée, toujours en utilisant LiteMat. Elle peut également être reformulée, en utilisant les identifiants de super-concepts et super-propriétés, comme expliqué en figure 8, et en utilisant la gestion des `owl:sameAs`, comme détaillé dans les sections précédentes. La requête est ensuite exécutée sur le flux, et le résultat est renvoyé si elle est valide.

Le traitement de la requête et le traitement du flux s'effectuent en parallèle, de manière distribuée. Comme chaque opération utilise le même dictionnaire d'encodage (initialisé précédemment et relativement statique), il n'y a pas de problèmes de concurrence. Il y a donc un gain de performance non négligeable en fonction de l'architecture adoptée et du cas d'usage. En outre, la gestion des hiérarchies de propriétés et des `owl:sameAs` permet des opérations plus poussées que la plupart des systèmes existants.

5.6 Résumé

Dans ce chapitre, nous avons présenté une extension d'un encodage intelligent pour les concepts et les prédicats de triplets RDF, LiteMat. Notre amélioration permet de représenter les propriétés `owl:sameAs` de manière compacte et optimisée, tout en conservant les avantages initiaux de l'encodage. Nous avons également détaillé une autre contribution, PatBin, basé sur la compression fournie par LiteMat, qui est essentiellement adapté pour la représentation de flux de données. PatBin permet de raisonner et de manipuler des graphes RDF sans besoin de décompression, et constitue une

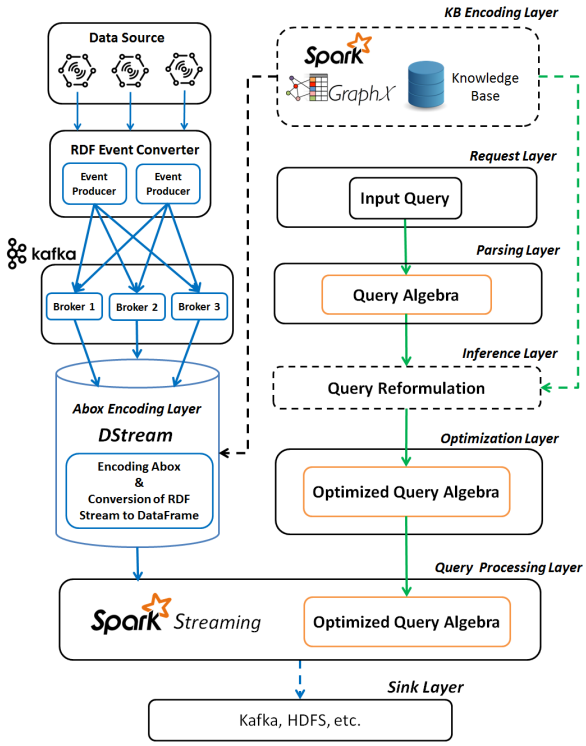


FIGURE 26 – Architecture de Strider

représentation compacte et simple d'utilisation. Nous avons présenté divers algorithmes et exemples d'usage centrés sur la matérialisation, afin d'exposer les avantages de notre approche.

La version initiale de LiteMat a déjà été présenté dans une conférence internationale [17], et sa version étendue a été intégrée dans Strider, l'un des composants du projet Waves, qui fut également reçu à diverses conférences [40] [39]. PatBin, notre algorithme basé sur LiteMat, a également fait l'objet de publications [41] et a été réutilisé dans une autre contribution à cette thèse, détaillée par la suite.

6

Exécution et génération de requêtes pour PatBin

6.1 Motivation

L'implémentation de PatBin nous a permis d'évaluer les performances de compression de l'algorithme, et de programmer la méthode de matérialisation basée sur l'encodage. Grâce à ces étapes, il nous est possible de compresser des flux et des graphes RDF, et de les modifier afin d'y ajouter de nouveaux triplets. Toutefois, la gestion de l'application des requêtes sur des graphes compressés n'est pas totalement assurée : dans nos sections précédentes, nous avons montré comment manipuler le graphe de la clause *WHERE* pour réaliser de la matérialisation et vérifier la validité d'une requête sur un graphe. Mais une requête est souvent constituée de nombreux autres éléments (variables, filtres, agrégats...) qu'il faut également prendre en compte ; or une requête SPARQL standard n'est pas aisément directement applicable sur un graphe encodé.

Nous avons développé un langage de requête inspiré de PatBin, qui reprend notre encodage, en ajoutant des éléments de requête dans le binding. Nous avons également développé une méthode de génération de requêtes à partir de fichiers et d'informations, qui permet d'obtenir une requête compressée ; cette approche a été choisie en raison de la simplicité de la forme compressée de nos requêtes, et pour faciliter les interactions entre les opérateurs et le raisonnement. En effet, il est souvent complexe pour des non-initiés de manipuler des données sémantisées : notre approche vise à résoudre cette difficulté, en plus de permettre la création de requêtes compressées.

Ces deux contributions sont employées par le module RAMSSES, qui sera décrit en fin de chapitre.

6.2 Le requêtage avec PatBin

Comme nous l'avons montré précédemment, PatBin permet de compresser de manière efficace des graphes RDF, et peut être utilisé dans les flux de données. Nous avons également détaillé notre gestion de la matérialisation dans les formes compressées ; il nous reste encore à expliquer comment PatBin peut être utilisé pour le requêtage.

Sous forme compressée, les flux de données ne sont constitués que de bindings : un flux, et donc son motif associé, ne change que très rarement ; seuls les objets du graphe vont varier, *e.g.*, les mesures provenant des capteurs. Une requête SPARQL contient toujours une clause *WHERE*, qui contient un graphe RDF avec des variables : ce graphe peut aussi être converti au format PatBin (comme pour la matérialisation). Le binding sera forcément différent de ceux issus des flux (puisqu'il y aura des variables), mais le pattern peut aussi varier. Il peut y avoir trois cas :

— le pattern du flux et celui de la requête sont identiques,
— certaines propriétés du flux ne sont pas dans la requête,
— certaines propriétés de la requête ne sont pas dans le flux.

Le premier cas est trivial. Le second cas est sans doute le plus courant : il s'agit de situations dans lesquelles certaines informations du flux ne sont pas nécessaires dans la requête. Le troisième cas correspond au besoin de matérialisation, que nous avons traité précédemment. Il est possible que les deux derniers cas soient combinés : une requête qui ne reprend pas tout le graphe du flux, mais ajoute des informations provenant d'une base de connaissances statiques en lien avec les informations du flux.

Toutefois une requête ne se limite pas à la clause *WHERE* ; il y a d'autres éléments qui peuvent être ajoutés pour préciser le résultat (*FILTER...*), exécuter des requêtes sur des ensembles de graphes (agrégats, *HAVING...*), combiner des graphes (*UNION*, *OPTIONAL...*), etc. Tous peuvent être modélisés sous une forme compressée, et traités efficacement pour obtenir le résultat voulu. Le binding de la requête, quant à lui, sera essentiellement vide ; les seuls objets présents seront ceux qui permettent de préciser la requête, et les variables (les nœuds intermédiaires sont laissés vides).

Un exemple simple de requête est présenté dans la figure 27 (b). L'exemple

(a) Extrait de flux compressé

```
31:(5:8):40:(44:(49:53):66:(12:74):67:(12:74))
 ;"Postal Bank";"Bank#041"; ; ;450;"US$"; ;"Dupond";"CCP XXX"; ;"Durand";"CCP YYY"
```

(b) Requête compressée basique

```
31:(8):40:(44:(49:53))
 ;"Bank#041"; ; ;?var1;"US$"
```

(c) Requête compressée avec filtre

```
31:(8):40:(44:(49:53))
 ;"Bank#041"; ; ;?var1>300;"US$"
```

(d) Requête compressée avec agrégat

```
31:(8):40:(44:(49:53))
 ;"Bank#041"; ; ;?var1?SUM;"US$"
```

FIGURE 27 – Exemples de requêtes compressées

utilisé ici se base sur un flux de transactions financières ; un extrait du flux est présenté en (a). La requête de base renvoie simplement le montant des transactions en dollars américains effectuées par la banque 041. Certaines parties du graphe du flux sont ignorées car elles ne sont pas pertinentes dans notre cas. La variable est préfixée par un point d'interrogation, afin d'être retrouvée aisément : les littéraux dans un triplet RDF ne sont jamais préfixés ainsi.

Les filtres peuvent être ajoutés à la suite des variables, comme montré en figure. 27 (c). Il est possible d'opérer un filtrage sur plusieurs variables, mais les filtres seront tous vérifiés lors de l'exécution de la requête (comme s'ils étaient séparés par une conjonction). On pourrait ajouter des opérateurs booléens à la suite de la représentation pour préciser le traitement à lui appliquer : par exemple, rajouter une barre (|) pour représenter une disjonction et un plus (+) pour une conjonction. Mais l'ordre des variables dans le binding peut varier par rapport à une requête SPARQL, ce qui complique l'opération. En outre, même avec ce genre de représentation, il est impossible d'utiliser des parenthèses pour définir davantage de priorités.

Les agrégats peuvent être ajoutés de la même manière, comme précisé dans la partie (c). Toutefois, la requête s'effectue alors sur un ensemble d'éléments du flux avant de renvoyer sa réponse ; ce paramètre doit être défini ailleurs, et ré-utilisé pour la requête. Il est possible également d'ajouter des

HAVING aux agrégats, en reprenant le format du filtre, et en l'ajoutant à la suite de l'agrégat (dans notre exemple, on aurait ?var1 ?SUM>500 pour une somme de transactions supérieure à 500).

Plusieurs opérateurs du langage SPARQL combinent les résultats de graphes RDF au sein d'une même requête : *UNION*, *INTERSECT*, mais aussi *OPTIONAL*, etc. Dans notre cas, chacun de ces graphes peut être représenté par sa forme compressée, puis les résultats affinés en fonction de l'opérateur utilisé. Il s'agit du seul cas où un élément de la requête n'est pas modélisé au sein de la forme compressée.

6.3 Exécution de requêtes

L'exécution de requêtes est résumée dans l'algorithme 5 ; on considère qu'il s'agit de la toute première exécution, qui comporte son initialisation. Un exemple complet est également présenté, étape par étape, en Fig. 28.

C'est à cette étape que la validité de la requête est vérifiée (qu'elle est bien applicable sur le flux), et que l'on effectue la matérialisation (si besoin) ; toutes les modifications effectuées à ce moment sont enregistrées pour être appliquées lors de l'exécution de la requête sur les échantillons suivants. Ainsi, si la requête est valide, son application future prendra moins de temps (les vérifications initiales ne seront plus nécessaires). Pour que l'algorithme fonctionne, il faut naturellement fournir la compression PatBin de la requête, mais également un échantillon PatBin provenant du flux de données. C'est le pattern qui est le plus important ; le binding peut éventuellement être factice, et géré plus tard (on peut séparer la gestion de la première exécution de la requête des exécutions suivantes). Les flux (séries de bindings) sont stockés en mémoire principale afin d'optimiser le traitement des requêtes ; il est possible de faire persister les flux par sérialisation si un traitement ultérieur est nécessaire.

Il faut dans un premier temps initialiser une structure de résultat, qui permettra de renvoyer les variables attendues, mais aussi d'enregistrer l'évolution des agrégats, qui doivent se faire sur plusieurs échantillons. Ensuite vient l'étape mentionnée en section 6.2, pour comparer les patterns respectifs de la requête et du flux. En effectuant l'intersection des encodages, on obtient P_i, soit le pattern commun au flux et à la requête (Fig. 28 a). Puis, en soustrayant P_i au pattern de la requête, on obtient P_m : ce qui reste à matérialiser (figure 28 b). En effet, si certains prédicats (par extension, des

triplets) sont présents dans la requête, mais pas dans le flux, il y a un besoin de matérialisation. On peut résoudre cette étape en requêtant une base de connaissance statique liée au projet, qui contiendra les informations manquantes (Fig. 28 c), et les ajouter au pattern, et à chaque binding qui arrivera du flux (le pattern donnera la position des objets issus de la base statique dans le binding). Ensuite vient le traitement du cas inverse : si la taille du pattern de flux est supérieure à celle du pattern de requête, c'est qu'il y a certains éléments du flux qui ne sont pas nécessaire à la requête ; on peut donc les retirer, et normaliser les deux patterns pour qu'ils soient identiques (Fig. 28 d). Notons qu'à chaque étape, le retrait (respectivement l'ajout) d'un élément d'un pattern entraîne le retrait d'un élément du binding à la position correspondante (les séparateurs du pattern sont des :, ceux du binding des ;).

Lors de la comparaison entre les éléments des patterns, il est possible d'utiliser les bénéfices de l'encodage de LiteMat afin de réécrire la requête. Cette étape se déroule lors du calcul de l'intersection : pour chaque élément encodé du flux étant strictement supérieur à la valeur à la position correspondante dans la requête, il est possible que l'élément en question corresponde à une sous-propriété de l'élément de la requête. Cette vérification peut s'effectuer en se basant sur les spécificités de LiteMat (nombre de bits pour l'encodage) : si une sous-propriété de l'identifiant de la requête correspond à l'identifiant du flux, cet identifiant est adopté en remplacement dans la requête. La réécriture ne s'effectue qu'une fois pour toute, à l'initialisation, et elle n'est valide que pour l'application de la requête sur ce flux spécifique (elle devra potentiellement être réécrite d'une autre manière pour d'autres flux).

A cette étape, pour que la requête puisse être exécutée, les patterns de la requête et du flux doivent être identiques ; les étapes précédentes ne sont pas toujours nécessaires, mais elles s'assurent que les encodages correspondent au maximum. Il faut également s'assurer que le binding de la requête est *inclus* dans celui du flux, c'est-à-dire que les littéraux spécifiés dans la requête sont présents dans le binding du flux, et qu'ils correspondent. Si ces deux conditions ne sont pas remplies, alors soit le graphe du flux est différent de celui de la requête (il n'est donc pas possible de l'appliquer), soit les littéraux spécifiés ne sont pas identiques, et dans ce cas les éléments fournis ne correspondent pas. L'exécution renvoie alors *null*, pour préciser le problème.

Si ces conditions sont vérifiées, le binding du flux est comparé à celui de la requête, d'où on extrait les filtres (sur les variables et sur les agrégats). Si ces filtres sont valides, enfin, on peut extraire les variables du flux : il suffit

Algorithm 5 Exécution complète de requête

```
1: procedure QueryExecution(P_q, B_q, P_s, B_s)
2:     result = initialize(B_q)
3:     P_i = P_q ∩ P_s
4:     P_m = P_q − P_i
5:     if P_m is not empty then
6:         P_s, B_s = materialize(P_s, B_s, P_m)
7:     end if
8:     if P_s > P_q then
9:         normalize(P_s, P_q)
10:    end if
11:    if P_s == P_q and B_q ⊂ B_s then
12:        if verifyFilters(B_q, B_s) then
13:            update(result)
14:            return result
15:        end if
16:    end if
17:    return null
18: end procedure
```

Extrait de flux compressé

```
31:(5:8):40:(44:(49:53):66:(12:74):67:(12:74))
 ;"Postal Bank";"Bank#041"; ; ;450;"US$"; ;"Dupond";"CCP XXX"; ;"Durand";"CCP YYY"
```

Requête compressée (nécessitant matérialisation)

```
31:(8:23):40:(44:(49:53))
 ;"Bank#041";"Paris"; ; ;?var1;"US$"
```

(a) Calcul de $P_s \cap P_q = P_i$

```
31:(8):40:(44:(49:53))
```

(b) Calcul de $P_q - P_i = P_m$ (forme statique à matérialiser dans le flux)

```
31:(23)
 ;"Paris"
```

(c) Nouveau P_s, B_s

```
31:(5:8:23):40:(44:(49:53):66:(12:74):67:(12:74))
 ;"Postal Bank";"Bank#041";"Paris"; ; ;450;"US$"; ;"Dupond";"CCP XXX"; ;"Durand";"CCP YYY"
```

(d) Normalisation ($P_s = P_q, B_q \subset B_s$?)

```
31:(8:23):40:(44:(49:53))
 ;"Bank#041";"Paris"; ; ;450;"US$"
```

FIGURE 28 – Vérification de validité de requête

de récupérer les bindings correspondants à la position des variables (puisqu'à cette étape, les bindings ont la même longueur). Les valeurs correspondant aux agrégats sont récupérées de la même manière, mais stockées différemment et non retournées, car il faudra plusieurs échantillons avant de les calculer. Ce résultat est ensuite renvoyé à l'utilisateur.

Lorsque la première exécution de requête est remplie avec succès, certaines vérifications ne sont plus nécessaires : les trois premières conditions de l'algorithme (lignes 5, 8, 11) seront toujours valides tant que le pattern du flux ne varie pas, et on peut traiter le PatBin du flux toujours de la même manière après la première étape (matérialisation/normalisation). Le stockage des modifications à effectuer pour la requête exécutée permet donc de gagner du temps. En revanche, il est toujours nécessaire de vérifier les filtres appliqués au binding (ligne 12), car les littéraux vont varier d'un échantillon à l'autre. Le résultat renvoyé par la procédure d'exécution sera mis à jour à

chaque fois, avec des variables retournées différentes, et des agrégats dont les valeurs vont s'accumuler. L'utilisateur peut choisir de calculer le résultat des agrégats lorsque le nombre de valeurs rassemblées sera satisfaisant.

Dans le cas d'un besoin de groupement (*GROUP BY* de SPARQL, variables et agrégats à retourner en même temps), la structure qui stocke les valeurs des agrégats au fur et à mesure va, en plus de l'opération finale à exécuter (*SUM*, *AVG*, *MAX*, ...), enregistrer les variables de groupement liées à l'agrégat. Il y aura donc une liste de structures pour chaque série de valeurs de variables différentes ; lorsque l'utilisateur demande le résultat, l'intégralité des variables enregistrées sont renvoyées, et les groupements qui leur sont associées sont calculés.

6.4 Génération des requêtes

La manière la plus évidente de générer des requêtes compressées au format PatBin est bien sûr de convertir directement les graphes reçus et la requête SPARQL, afin de gagner en temps d'exécution. Comme nous l'avons vu précédemment, le flux sera simplement constitué d'une liste de bindings, et la requête sera elle-même convertie en un couple pattern/binding (ou plusieurs, en fonction de la complexité de la requête et des opérateurs employés). Mais nous avons également travaillé sur un moyen de générer des requêtes à partir de données d'un format étranger au Web Sémantique. L'objectif est de permettre à n'importe quel utilisateur d'interroger un flux (ou une base de connaissances) en fournissant les informations et les restrictions qu'il désire dans un fichier respectant un certain format ; la requête est alors générée automatiquement, sans que l'utilisateur n'aie à manipuler de RDF.

L'architecture de notre générateur est présentée en figure 29. Nous avons commencé par travailler sur des fichiers tabulaires de type excel, qui possèdent généralement une structure similaire : les données sont organisées par colonne, chaque colonne étant identifiée par un alias, et regroupant un seul type de données. Cette approche fonctionne également avec du csv, tsv ou autres données tabulaires indiquant le nom des colonnes sur la première ligne, ce qui permet d'adapter notre système à d'autres formats. Par conséquent, nous pouvons identifier à partir des labels de colonne et/ou des données qu'elles regroupent les concepts RDF que l'utilisateur va vouloir récupérer pour spécifier le résultat, ainsi que les variables à renvoyer. Pour cela, il nous faut nous baser sur la TBox de l'ontologie de l'utilisateur, et sur

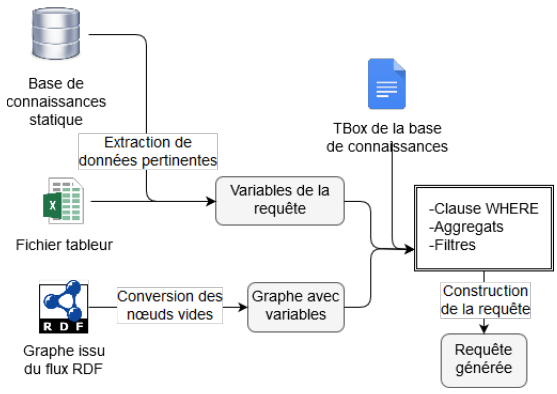


FIGURE 29 – Processus de génération de requêtes

un échantillon de flux (pattern et binding) : il n'y a qu'ainsi que l'on peut établir une correspondance précise. Un autre avantage des fichiers tableur, c'est qu'ils permettent d'utiliser diverses formules mathématiques, qui correspondent aux agrégats et aux filtres du langage SPARQL ; il est donc possible de construire la requête élément par élément, en décomposant le fichier fourni.

Chacun des éléments identifiés dans le fichier d'origine constitue une partie de la requête compressée. La première étape est de modéliser la clause *WHERE*, en se servant des concepts et des variables pertinents identifiés via les alias de colonnes. En utilisant l'extrait de flux, nous pouvons extraire du graphe du flux (représenté par le pattern) les branches correspondant aux variables et concepts identifiés dans le fichier, c'est-à-dire le pattern minimal allant de la racine jusqu'au concept (ou à la variable). Le 'binding' correspondant sera principalement vide, avec seulement la variable ou le concept le plus à droite.

Une fois que les branches associées ont été générées, on peut les combiner pour recréer un sous-graphe du flux, qui servira de pattern de base pour la clause *WHERE* de notre requête. Le 'binding' peut être créé en parallèle, et contiendra l'intégralité des concepts et variables identifiés, à leur place. Chacun des autres éléments identifiés dans le fichier fourni par l'utilisateur peut être ajouté : agrégat, filtre, etc. Une fois que tout a été ajouté, on peut ensuite l'exécuter sur le flux, et fournir le résultat à l'utilisateur. La figure 30 présente un exemple de génération de requête à partir d'un extrait de flux

du projet Waves. Trois triplets ont été retenus, grâce à l'identification de prédicats dans le fichier d'entrée : pour ceux en rouge, on va vouloir récupérer leurs objets, qui serviront donc de variables pour notre requête. La propriété en bleu ne sert qu'à spécifier le graphe de la clause *WHERE* ; d'ailleurs, on voit bien qu'il s'agit d'un sous-graphe minimal de celui du flux, ne reprenant que les informations essentielles. Enfin, l'identification d'une valeur seuil nous a permis d'ajouter un filtre, en vert. La requête est présentée sous sa forme décompressée pour une meilleure compréhension, mais elle est normalement générée directement compressée (en dessous).

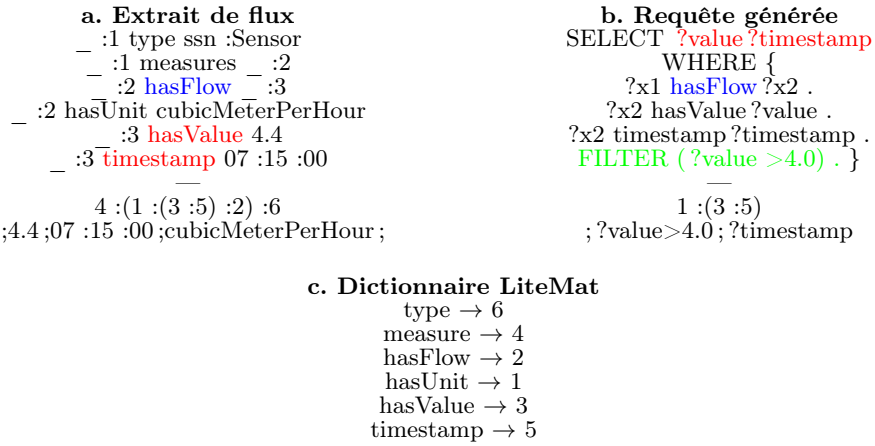


FIGURE 30 – Exemple de génération de requête

Il est possible que le résultat obtenu ne convienne pas : soit parce que la requête n'est pas assez précise, mais peut-être aussi parce qu'il manque des éléments. Nous avons conçu un système de feedback pour l'utilisateur, afin qu'il puisse modifier le résultat, ou la requête, selon ses désirs. Notre méthode de construction de la requête compressée étape par étape nous permet de sélectionner les éléments à ajouter ou à retirer, telles les pièces d'un puzzle. Il nous est possible de dévoiler à l'utilisateur tous les éléments qui ont été utilisés pour la création de la requête : il ne s'agit toujours pas de RDF, mais plutôt des mots-clés et des expressions qui ont mené au résultat final par sélection. Si certains éléments ne conviennent pas, ou sont jugés trop sélectifs, l'utilisateur peut les retirer. De même, si certaines précisions sont manquantes, il peut les ajouter ; le générateur de requête se chargera d'adapter le nouvel élément pour la requête compressée.

Le feedback est particulièrement adapté pour gérer certains opérateurs SPARQL compliqués à trouver dans des fichiers : par exemple, l'utilisateur peut remarquer qu'une partie du résultat obtenu est optionnel, et le signaler comme tel ; le générateur prendra donc en compte deux graphes différents pour la requête, l'un d'eux étant *OPTIONAL*. Afin d'aider l'utilisateur à mieux comprendre l'implication de chaque élément identifié dans le fichier fourni, le feedback s'effectue en classant chacun des composants en fonction de son rôle au sein de la requête. Les labels de colonne, formant les variables et les concepts essentiels au graphe de la requête, peuvent être regroupés dans une catégorie *éléments du flux*. De la même manière, on peut regrouper les agrégats dans une catégorie *groupe de valeurs*, et toutes les opérations de filtrage (*FILTER*, *HAVING*) dans une catégorie *filtrage*, et ainsi de suite. La désignation regroupant les éléments de tri (*ORDER* notamment) est plus développée que les autres, car elle fournit également à l'utilisateur des options pour limiter les résultats renvoyés, ce qui correspond aux opérateurs *LIMIT* et *OFFSET* de SPARQL. Le nombre d'échantillons vérifiés par les agrégats, ou par des requêtes continues, peut aussi être ajusté à ce stade.

6.5 Extensions considérées

Nous avons imaginé diverses extensions et améliorations possibles pour PatBinQL. Nous avons traité la génération de requêtes dans les sections précédentes, mais il est également possible de générer directement de simples graphes RDF compressés à partir des informations du fichier de l'utilisateur. Ainsi, on pourrait éventuellement recréer des flux de bindings à partir de données excel structurées, puis générer des requêtes à partir d'autres données fournies, et recréer ainsi une chaîne de raisonnement à partir d'un format extérieur au web sémantique. On peut également envisager une alternative sans flux, avec seulement des graphes RDF statiques : PatBin est également applicable pour ce type de données.

Pour étendre l'exploitation du format excel, on peut également vérifier les codes couleur employés dans certaines cases, qui correspondent potentiellement à des seuils de valeurs pertinents pour notre requête. Cela nécessiterait un nouvel ajout à la section de feedback, et il faudrait également tenir compte des nuances de couleurs pour la gestion des valeurs (teintes de vert pour les valeurs supérieures, teintes de rouge pour les inférieures, par exemple). D'autres particularités des fichiers excel peuvent certainement être exploitées

pour PatBinQL ; il faut seulement découvrir une correspondance avec un élément de requête SPARQL, puis trouver comment parser les valeurs pour les utiliser au mieux, et les représenter dans la compression.

Nous avons étudié le cas de fichiers tableurs comme structure de données pour créer la requête pour notre cas d'usage, mais il est tout à fait possible de se servir d'autres formats. Le JSON, par exemple, nous semble être une très bonne alternative, et offre une structure simple et aisément utilisable pour le générateur de requêtes. Il y a plusieurs possibilités pour la représentation des informations pertinentes : on peut reprendre la structure du feedback pour faciliter le traitement, ou bien employer une hiérarchie de clefs et de valeurs qui nous permettra de retrouver la structure du graphe de la requête.

D'autres formats sont certainement envisageables, la seule considération que nous avons à l'esprit est que l'utilisateur n'a pas besoin de connaissance dans le domaine du Web Sémantique ; la structure proposée doit donc être à la fois suffisamment simple pour être accessible aux opérateurs, mais également suffisamment complète pour modéliser un maximum d'informations. Le format CSV par exemple, a l'avantage d'être simple, mais ne permet pas forcément de représenter l'ensemble des informations pour des requêtes complètes. D'un autre côté, il s'agit du format idéal pour représenter une compression RDF utilisant PatBin, grâce à ses séparateurs. Les point-virgules permettent de représenter les bindings, donc en supposant qu'un utilisateur soit familier avec la syntaxe de PatBinQL, et que la requête à exécuter corresponde exactement au graphe du flux, on peut directement écrire la requête en csv. C'est un cas extrêmement simple (pas de processus de génération de requête), mais applicable pour peu de cas d'utilisation.

6.6 Évaluation

6.6.1 Jeux de données

Notre approche étant basée sur PatBin, détaillé dans la section 5.4.2, nous avons utilisé les mêmes jeux de données afin de réaliser notre évaluation. Les requêtes utilisées dans les exemples suivants sont disponibles en annexe A. Les mêmes requêtes sont utilisées pour le LUBM 1 et le 10, avec seulement un ajustement des préfixes si nécessaire.

Exécution de requêtes PatBinQL

Nous avons exécuté diverses requêtes afin de vérifier les performances de PatBinQL sur les données d'exemple compressées avec PatBin. Les requêtes sont de complexité différente, afin de vérifier les performances de manière globale : la première est basique, avec seulement une clause *WHERE*, la seconde comporte en plus une clause *FILTER*, et la dernière réalise une agrégation (*MAX*, *COUNT*, *SUM*...). Pour la comparaison, nous avons exécuté les mêmes requêtes non-compressées en utilisant Apache Jena, dans un environnement Java. Les performances sont présentées dans le tableau 4.

	Waves		**LUBM 1**		**LUBM 10**	
	Jena	PatBinQL	Jena	PatBinQL	Jena	PatBinQL
Requête simple	61 ms	5 ms	120 ms	60 ms	615 ms	270 ms
Requête filtrée	85 ms	5 ms	250 ms	127 ms	785 ms	482 ms
Requête agrégée	75 ms	6 ms	160 ms	120 ms	625 ms	440 ms

TABLE 4 – Évaluation de l'exécution de requêtes

Chaque mesure prend en compte l'affichage du résultat, mais ignore les étapes d'initialisation. Pour Jena, cela signifie que nous mesurons l'exécution après avoir initialisé le modèle avec le graphe en entrée ; pour PatBinQL, tout est déjà compressé, et l'étape d'initialisation (vérifiant les besoins de matérialisation et de normalisation) a déjà été effectuée. Les tests ont été exécutés en simulant une approche de flux : SPARQL doit examiner tous les graphes entrants, alors que PatBinQL se contente de ceux correspondant au pattern qui lui a été fourni. Comme nous pouvons le voir, les résultats de notre approche sont encourageants ; de plus, en cas d'un besoin de matérialisation, notre algorithme peut inférer les données nécessaires directement, alors qu'avec JENA cela nécessite des étapes supplémentaires, plus complexes.

Nous n'avons pas réalisé d'évaluation précise sur la matérialisation avec PatBinQL ; les données de Waves sont d'usage privé, et nous ne pouvons pas matérialiser d'information dans les jeux de données du LUBM, car nous ne les générons pas directement. Il serait éventuellement possible de modifier les fichiers récupérés afin de séparer les données en deux bases de connaissances et ainsi pouvoir simuler un processus de matérialisation, mais un tel procédé, sur une telle quantité de triplets, serait extrêmement long et complexe.

Génération de requêtes

Afin d'évaluer la pertinence de notre générateur de requêtes, nous l'avons appliqué sur divers exemples de notre cas d'utilisation, pour tenter de reproduire certains des résultats utilisés pour la détection d'anomalies dans notre projet. Nous nous sommes basés sur les fichiers fournis par nos clients et des échantillons de flux pour générer plusieurs requêtes, les exécuter, puis présenter les résultats à des experts de notre client. L'objectif était d'essayer de trouver les limites de notre approche, et de vérifier si les éventuelles modifications suggérées peuvent être appliquées en utilisant notre méthode de feedback.

Evaluateur	Requêtes						
	1	2	3	4	5	6	7
1	✓	✓	✓	✓	✓	×	✓
2	✓	✓	✓	✓	×	✓	×
3	✓	✓	✓	×	✓	×	×
4	✓	✓	✓	✓	✓	×	×
5	✓	✓	✓	✓	✓	✓	×
Complexité des requêtes							
Triplets	2	3	3	3	4	3	4
Filtre	×	×	×	×	×	✓	×
Agrégats	×	×	×	×	×	×	✓

TABLE 5 – Évaluation des requêtes générées

Le tableau 5 résume l'évaluation de sept requêtes générées par notre système, spécifiant leur qualité et leur complexité. Cinq experts du domaine de notre client ont vérifié nos résultats, afin de juger de la pertinence de nos requêtes. Les trois premières étaient relativement simples, adaptées à plusieurs flux indépendamment de leur type, et furent jugées satisfaisantes. Les deux suivantes étaient plus spécifiques, mais donnaient également de bons résultats : seul l'un des experts critiquait un paramètre manquant. Nous avons vérifié le fichier en entrée, et constaté qu'un paramètre pertinent n'avait pas été identifié du fait de sa position inhabituelle (notre générateur se base principalement sur les alias de colonnes du tableur) : cela fut réglé rapidement. Les dernières requêtes étaient plus complexes, et visaient à obtenir un résultat basé sur des cellules spécifiques (pour créer un agrégat ou un filtre). Seulement la moitié des opérateurs ont jugé ces requêtes acceptables, alors

qu'elles n'étaient pas assez pertinentes pour d'autres. Comme nous l'avons mentionné précédemment, les principales difficultés rencontrées par notre générateur concernent l'ajout et l'évaluation d'agrégats, de filtres et autres étapes évoluées par rapport à des requêtes basiques.

L'évaluation des opérateurs est stockée dans la première moitié du tableau, tandis que la seconde précise la complexité des requêtes. Nous avons spécifié dans chaque cas le nombre de triplets pertinents identifiés dans le fichier d'entrée pour la génération du graphe de la clause *WHERE*, ainsi que les cas d'utilisation de *FILTER* et d'agrégats. Plus le nombre de triplets identifiés est grand, plus le sous-graphe de la requête sera conséquent. Des difficultés peuvent survenir pour la génération de requêtes plus complètes, mais jusqu'à présent nous n'avons rencontré aucun problème ne pouvant être résolu par notre étape de feedback.

Le tableau 6 présente divers gains de performances pour chaque zone avec une requête générée pour les archives fournies par notre client (une même requête peut être appliquée pour plusieurs zones géographiques, puisque les anomalies se détectent de la même manière). La première colonne donne les pourcentages de mesures retirés, c'est-à-dire les échantillons du flux qui ne répondent pas à la requête générée. La colonne *Triplets* indique le nombre de triplets du flux examinés par la requête, sans tenir compte des triplets non pertinents ; cela correspond à l'étape de normalisation de la section 6.3. Enfin, la dernière colonne représente le gain final de taille pour les flux pertinents : par exemple, pour la zone 1, 29% du million de triplets reçus est conservé une fois le processus d'exécution de requête effectué. L'efficacité de nos requêtes est donc bien identifiable.

Zones	Mesures	Triplets	Taille
Zone géographique 1 (France)	75%	50%	29%
Zone géographique 2 (France)	66%	52%	32%
Zone géographique 3 (Asie)	72%	46%	27%

TABLE 6 – Gains de performance dus à la génération de requêtes

6.7 RAMSSES, une approche générale

Le composant RAMSSES [42] est chargé de la détection d'anomalies dans le projet. Il intègre notre encodage pour les graphes RDF et les requêtes, et

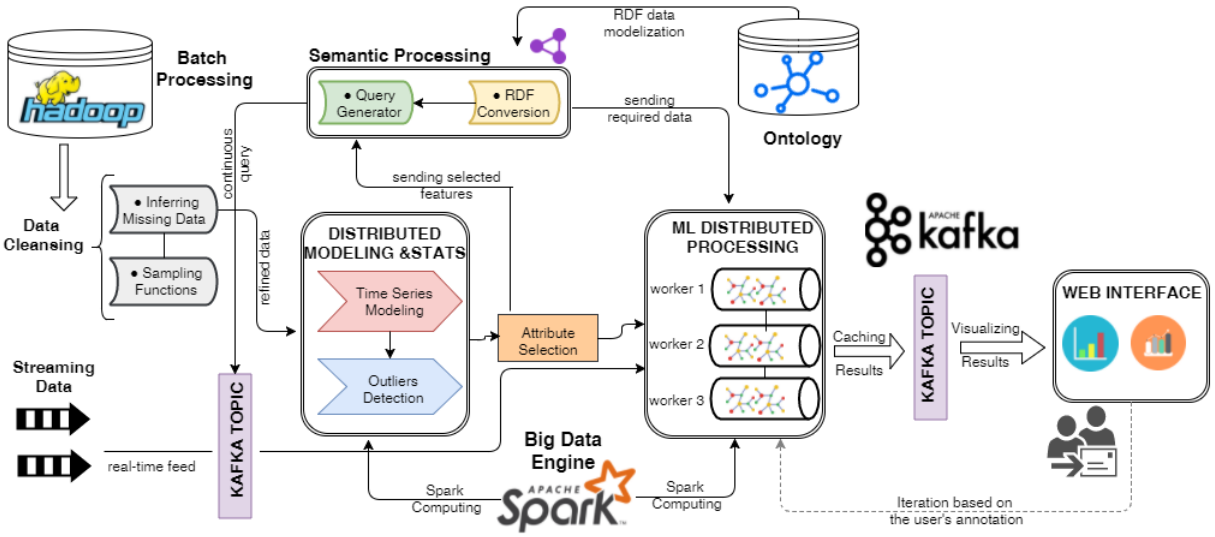


FIGURE 31 – L'architecture de RAMSSES

peut donc se servir de notre générateur pour appliquer de nouveaux traitements aux flux de données.

6.7.1 Une approche automatisée

RAMSSES reprend le principe de base de la suite de logiciels Aquadvanced de notre client, avec de nombreuses différences et améliorations. Son objectif principal est de pouvoir lancer des modèles intensifs d'apprentissage automatique sur des flux massifs de données en temps réel. Dans le cadre du projet Waves, le composant applique plusieurs méthodes mathématiques et statistiques sur les flux de données, et peut également générer des requêtes pour extraire des valeurs aberrantes. La combinaison de ces diverses méthodes permet d'identifier les plus efficaces pour le traitement de chaque flux spécifique, et ainsi détecter un maximum d'anomalies dans les mesures.

La figure 31 présente l'architecture du composant RAMSSES. Comme pour Scouter, ce composant a été conçu de façon modulable et distribuée, afin de pouvoir être réemployé pour un maximum de cas d'usage. Les méthodes d'apprentissage et les approches mathématiques et statistiques peuvent être configurées par l'utilisateur, et le système peut être reconfiguré pour tenir

compte de nouvelles ontologies à partir desquelles raisonner. Dans notre cas d'usage, il s'agit du composant principal chargé de la détection d'anomalies ; la plupart des autres composants majeurs (*e.g.* Scouter) se basent sur ses résultats.

6.8 Résumé

Nous avons présenté dans ce chapitre un langage de requête, PatBinQL, basé sur notre encodage détaillé dans le chapitre précédent (PatBin). Notre approche reprend les avantages de ce dernier, et ajoute divers éléments dans le binding pour spécifier le résultat attendu. Nous avons également développé un outil de génération pour ces requêtes compressées, basé sur des fichiers au format commun, afin d'être accessible pour des opérateurs étrangers aux méthodes de raisonnement du web sémantique.

Notre langage de requête a l'avantage d'être simple, et de reprendre l'essentiel des clauses utilisées dans le langage SPARQL. Il peut être généré à partir d'informations extraites depuis des fichiers fournis par l'opérateur, eux-même configurables. La requête générée peut être ajustée via un système de feedback développé, présentant à l'utilisateur les différents éléments pris en compte. L'ensemble permet d'obtenir un résultat compact et précis.

Nos contributions ont été intégrées dans le composant RAMSSES, qui fournit la base du projet Waves. Il a fait l'objet d'une publication ayant reçu le prix du meilleur article applicatif dans une conférence nationale [42]. Une autre publication détaillant le composant PatBinQL a été soumis à une conférence internationale reconnue dans le domaine de l'ingénierie et de la maintenance de connaissances.

7

Conclusion

Cette thèse s'est effectuée dans le cadre du projet FUI Waves, un système de raisonnement sur flux distribués, adaptatif et portable. Le cas d'usage pour le développement est la détection et la contextualisation d'anomalies dans les mesures de capteurs d'un réseau de distribution d'eau potable. Nous avons présenté plusieurs contributions, intégrées à différentes étapes du projet : d'abord au niveau de la contextualisation des anomalies, pour mieux identifier les origines potentielles d'erreurs dans les mesures, et ainsi éviter les faux positifs. Puis les travaux suivants se sont focalisés sur le raisonnement et ses performances, qui est réalisé au moment de la détection d'anomalies ; nous avons ensuite étendu le résultat obtenu à cette étape pour fournir d'autres outils aux opérateurs finaux. Chacune des contribution a été intégrée dans un composant majeur du projet (*i.e.* Scouter, RAMSSES et Strider).

Nous avons d'abord évoqué la nécessité d'utiliser la composante géographique du cas d'usage, après avoir analysé les limites des données temporelles. En première partie, nous avons détaillé le fonctionnement de notre système de profilage de zones géographiques, qui a pour objectif d'aider à contextualiser les anomalies détectées. Notre approche calcule deux résultats à partir de données cartographiques sous forme de nœuds et de polygones, puis décide de la manière de les combiner en utilisant une méthode mathématique. Pour ce dernier point, plusieurs approches ont été évaluées afin d'améliorer le résultat final et de s'adapter aux données disponibles : en se basant sur la moyenne de consommation au kilomètre carré, la longueur du réseau par rapport à la taille du secteur et la densité de points d'intérêts. Le système est paramétrable pour ajuster les résultats et s'adapter à divers cas d'usage, et ses performances ont été évaluées satisfaisantes. Il est employé dans le

module Scouter du projet Waves, le composant chargé de la détection des origines potentielles des anomalies.

Ensuite, nous avons présenté notre travail réalisé dans le cadre de la sémantisation des flux. La première étape fut de développer l'ontologie adaptée pour notre cas d'usage, qui fut composée de concepts et de propriétés d'autres représentations et projets existants. Le choix fut motivé en fonction de la qualité des sources étudiées, et de leur intérêt pour notre cas d'usage : le résultat étant utilisé par l'intégralité des composants du projet, il fallait obtenir un résultat le plus précis et pertinent possible, reprenant tous les aspects des données à représenter. Avec cette classification, les données du projet peuvent être représentées au format RDF : cela nous permet de raisonner et de requêter efficacement sur les flux ou les données statiques, de faire de l'inférence ou d'utiliser des connaissances externes. Par ailleurs, il fallait définir un format commun pour toutes les données entrantes, et pouvant être réutilisé pour d'autres cas d'usage. Mais nous avions besoin d'effectuer ce traitement de la manière la plus optimisée possible puisque nous gérons des flux de données. Nous avons choisi de ré-employer un algorithme existant déjà développé par notre équipe de recherche, LiteMat, que nous avons étendu pour supporter les propriétés `owl:sameAs`. Cette amélioration rend l'algorithme bien plus performant pour le traitement des liens inter bases de connaissances, chose importante pour notre projet (qui peut chercher de l'enrichissement dans des sources de données externes), et effectue des optimisations dans le cas où le même objet est représenté par différents identifiants (un cas fréquent dès lors qu'un référentiel est développé conjointement par un ensemble de collaborateurs). Il fut intégré dans Strider, un système de raisonnement sur graphes qui effectue les requêtes pour la détection d'anomalies dans le projet Waves.

A partir de LiteMat, nous avons développé un système de compression de graphes RDF baptisé PatBin, spécialement adapté à la gestion de flux d'informations sémantisées. LiteMat est extrêmement performant dans l'encodage de propriétés et de concepts seuls, mais il est moins performant pour la représentation de graphes complets : son approche n'offre aucun gain dans les représentations de littéraux ou de nœuds vides, ce qui est problématique dans le cas de flux RDF (nouveau littéraux et nœuds à chaque mesure). C'est pour cela que nous avons choisi de réutiliser l'encodage au sein d'un autre format : PatBin. Notre approche reprend les identifiants de LiteMat et encode les graphes RDF en deux structures distinctes, le pattern et le binding, d'où son nom. Le pattern est composé des identifiants des propriétés encodées, avec des séparateurs représentant les différents niveaux du graphe

et les liens entre les triplets originaux ; les identifiants sont triés par ordre croissant pour obtenir une forme déterministe. Le binding est quant à lui constitué des objets du graphe, littéraux comme concepts ou nœuds vides, séparés par des points-virgules ; ils suivent l'ordre des propriétés auxquelles ils sont rattachés dans le pattern. Les sujets sont ignorés car ils ne sont pas pertinents dans le cas de flux de données, et la structure du graphe est quoi qu'il en soit maintenue dans le pattern. La forme compressée obtenue permet de raisonner directement sur les données, sans décompression nécessaire, et permet de réaliser notamment la matérialisation d'éléments manquants de manière simple.

PatBinQL est un langage de requête développé à partir de PatBin. Comme l'encodage de base permet la manipulation de graphes sans décompression, nous avons développé un moyen de requêter en suivant le même principe. En plus du langage de requêtes, nous avons implémenté un système permettant de générer directement des requêtes compressées à partir de fichiers fournis par l'utilisateur, un expert du domaine de l'eau potable dans notre cas d'usage. Notre approche a été motivée par le fait que le web sémantique et ses spécificités ne sont pas encore bien intégrées au sein des entreprises. Dans de nombreux cas, réaliser des requêtes sur des données sémantisées peut s'avérer complexe ; dans le cas d'une réutilisation du projet également, il peut être difficile de créer de nouvelles requêtes sans une bonne connaissance des données existantes. L'utilisation de fichiers d'un format commun d'utilisation permet de développer de nouvelles requêtes et de faire évoluer le raisonnement. Nous avons ajouté un système de feedback développé, afin que le résultat obtenu soit le plus optimisé possible. Par comparaison, PatBinQL est plus performant que des requêtes SPARQL exécutées avec JENA ; nous avons également recueilli des avis positifs concernant la pertinence et les résultats des requêtes générées pour notre client. Les deux composants sont intégrés dans RAMSSES, le module de Waves chargé de la contextualisation des anomalies.

Ainsi, les trois contributions détaillées dans cette thèse ont été intégrées dans le projet auquel elles sont liées. Toutes ont fourni des résultats, qui ont été évalués satisfaisants soit par des opérateurs de l'expert du cas d'usage, soit après comparaison avec des systèmes similaires existant. Le projet Waves a été conçu pour être portable, réutilisable pour d'autres cas d'usage, et c'est aussi le cas des composants en son sein : chacun peut être réutilisé, individuellement ou associé à d'autres, au sein d'autres projets. Leurs performances et leur paramétrisation, qui ont été détaillées dans cette thèse, sont

leur principaux avantages. Par ailleurs, bien que nous ayons mis l'accent sur le traitement de flux, qui était la problématique principale de notre projet, il est tout à fait possible de réutiliser Waves ou ses composants pour des données totalement statiques. L'intégralité de nos travaux a été validée par des publications acceptées et présentées dans des conférences nationales et internationales. Le composant Scouter a même reçu le prix pour le meilleur article applicatif, à la conférence nationale Extraction et Gestion de Connaissances de 2018.

Puisque l'ensemble de nos contributions, tout comme le projet dans lequel elles sont utilisées, sont paramétrables et ont pour but d'être utilisée dans différents cas d'usage, l'une de nos perspectives futures est de réemployer ce travail dans d'autres domaines. Comme nous l'avons détaillé, Waves utilise des ontologies du Web Sémantique, et nous avons développé celle du projet à partir de classifications existantes. Rien n'est codé directement "en dur" dans le projet : il est possible de créer une ontologie pour un autre cas d'usage, et de s'en servir pour paramétrer les composants du projet. Il faut alors spécifier les données statiques et/ou externes qui seront potentiellement utilisées, paramétrer le projet en conséquence, et également entrer les requêtes exécutées aux différentes étapes de déroulement. Cela représente un certain travail, qui aurait néanmoins été nécessaire avec toute autre application (ou en travaillant *from scratch*). Waves fournit un socle totalement paramétrable et optimisable avec de surcroît de bonnes performances d'exécution ; sa réutilisation par d'autres opérateurs, pour divers scénarios, serait bénéfique pour nous sous différents aspects, et nous permettrait également d'envisager de nouvelles extensions.

Pour des travaux et extensions futurs, nous envisageons d'étendre à nouveau LiteMat avec d'autres constructeurs. Il serait très bénéfique de permettre à notre encodage de supporter les propriétés inverses, transitives, fonctionnelles ou inverse fonctionnelles. Nous avons réfléchi à plusieurs théories plus ou moins évoluées, notamment en ajoutant des tables d'encodage, mais il ne s'agit que d'ébauches. Le développement de ces extensions et leur amélioration constitue une piste intéressante qui reste à suivre. L'amélioration de PatBinQL et du générateur de requêtes sont envisageables. En effet, notre langage de requête convient très bien pour des scénarios classiques, mais il peut se révéler limité pour des raisonnements avancés, nécessitant certaines clauses très complexes. L'ajout de constructeurs et d'opérateurs est donc notre priorité das ce domaine, afin d'étendre l'expressivité de notre langage. Nous avons également mentionné dans les sections précédentes la

possibilité d'utiliser d'autres formats de fichiers en entrée pour le générateur de requêtes. Une plus grande diversité de formats d'entrée utilisable rendrait le générateur plus flexible, et pourrait également permettre la modélisation plus facile de certains éléments de la requête.

A Annexe : Raisonnement

Requête basique

```
SELECT ?x ?y WHERE {
      ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?b .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#telephone> "xxx-xxx-xxxx" .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#researchInterest> "Research20" .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#name> ?x .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#emailAddress> ?y .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#undergraduateDegreeFrom> ?c .
}
```

Requête avec filtre

```
SELECT ?x ?v WHERE {
      ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?b .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#telephone> ?v .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#researchInterest> ?c .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#name> ?x .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#emailAddress> ?y .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#undergraduateDegreeFrom> ?d .
      ?d <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?z .
      FILTER regex(str(?v), "xxx-xxx-xxxx") .
}
```

Requête avec agrégat

```
SELECT ?x (COUNT(?z) AS ?count) WHERE {
      ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?b .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#telephone> "xxx-xxx-xxxx" .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#researchInterest> ?c .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#name> ?x .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#emailAddress> ?y .
      ?a <http://swat.cse.lehigh.edu/onto/univ-bench.owl#undergraduateDegreeFrom> ?d .
      ?d <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?z .
} GROUP BY ?x
```

FIGURE 32 – Requêtes exécutées pour les tests du LUBM

B Annexe : Extrait de l'ontologie de Waves

Extrait de la base statique :

```
@prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#> .
@prefix ssn:<http://purl.oclc.org/NET/ssnx/ssn#> .
@prefix qudt:<http://data.nasa.gov/qudt/owl/qudt#> .
@prefix cuahsi:<http://his.cuahsi.org/ontology/cuahsi#>  .
@prefix wgs84_pos:<http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix waves:<http://www.waves.org/ontology#> .

<http://www.zone-waves.fr/2016/sector#Haut-Clagny> a ssn:Platform ;
     rdfs:label "Haut-Clagny" .

<http://www.zone-waves.fr/2016/sensor#QS_HC1_Dep1> ssn:onPlatform <http://www.zone-waves.fr/2016/sector#Haut-Clagny> .

waves:flow24  a cuahsi:inputFlow .

<http://www.zone-waves.fr/2016/sensor#QS_HC1_Dep1> ssn:observes waves:flow24  .

waves:flow24 cuahsi:relatedTo <http://www.zone-waves.fr/2016/sector#Haut-Clagny> .

<http://www.zone-waves.fr/2016/sensor#QS_HC1_Dep1> a ssn:Sensor ;
     rdfs:label "QS HC1 Dep1" ;
     wgs84_pos:lat 4.8816025E1 ;
     wgs84_pos:long 2.150636E0 .
```

Extrait de flux :

```
_:1 measures _:2
_:2 hasFlow _:3
_:2 hasUnit qudt:cubicMeterPerHour
_:3 hasValue 13.7
_:3 timestamp 12:30:00
```

FIGURE 33 – Extrait des ontologies du projet Waves

Bibliographie

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. a new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 284(5) :1–5, 2001.

[2] Thomas R Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2) :199–220, 1993.

[3] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. In *Handbook on Ontologies*, pages 3–28. 2004.

[4] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.

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

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia : A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.

[7] Denny Vrandečić and Markus Krötzsch. Wikidata : A free collaborative knowledge base. *Communications of the ACM*, 57 :78–85, 2014.

[8] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web semantics : science, services and agents on the World Wide Web*, 17 :25–32, 2012.

[9] Robert G Raskin and Michael J Pan. Knowledge representation in the semantic web for earth and environmental terminology (sweet). *Computers & geosciences*, 31(9) :1119–1125, 2005.

[10] Ralph Hodgson and Paul J Keller. Qudt-quantities, units, dimensions and data types in owl and xml. *Online (September 2011) http ://www. qudt. org*, page 34, 2011.

[11] Edmond Jajaga, Lule Ahmedi, and Figene Ahmedi. An expert system for water quality monitoring based on ontology. In *Research Conference on Metadata and Semantics Research*, pages 89–100. Springer, 2015.

[12] Marc Wick. Geonames. *GeoNames Geographical Database*, 2011.

[13] Mordechai Haklay and Patrick Weber. Openstreetmap : User-generated street maps. *IEEE Pervasive Computing*, 7(4) :12–18, 2008.

[14] Peter Mooney, Padraig Corcoran, and Adam C Winstanley. Towards quality metrics for openstreetmap. In *Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems*, pages 514–517. ACM, 2010.

[15] Y Yamamoto. Twitter4j. *Internet : http ://twitter4j. org/en/index. html,[March, 03, 2014]*, 2008.

[16] Mark Allen. Restfb. *RestFB-A Lightweight Java Facebook Graph API and Old REST API Client*, 2012.

[17] Olivier Cure, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. Litemat : a scalable, cost-efficient inference encoding scheme for large rdf graphs. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1823–1830. IEEE, 2015.

[18] Norberto Fernández, Jesús Arias, Luis Sánchez, Damaris Fuentes-Lorenzo, and Óscar Corcho. Rdsz : an approach for lossless rdf stream compression. In *European Semantic Web Conference*, pages 52–67. Springer, 2014.

[19] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

[20] Jesus Arias Fisteus, Norberto Fernandez Garcia, Luis Sanchez Fernandez, and Damaris Fuentes-Lorenzo. Ztreamy : A middleware for publishing semantic streams on the web. *Web Semantics : Science, Services and Agents on the World Wide Web*, 25 :16–23, 2014.

[21] Javier D Fernández, Alejandro Llaves, and Oscar Corcho. Efficient rdf interchange (eri) format for rdf data streams. In *International Semantic Web Conference*, pages 244–259. Springer, 2014.

[22] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying rdf streams with c-sparql. *ACM SIGMOD Record*, 39(1) :20–26, 2010.

[23] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming sparql-extending sparql to process data streams. *The Semantic Web : Research and Applications*, pages 448–462, 2008.

[24] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *International Semantic Web Conference*, pages 280–297. Springer, 2013.

[25] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Extended Semantic Web Conference*, pages 1–15. Springer, 2010.

[26] Edward Thomas, Jeff Z Pan, and Yuan Ren. Trowl : Tractable owl 2 reasoning infrastructure. In *Extended Semantic Web Conference*, pages 431–435. Springer, 2010.

[27] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql : a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[28] Krzysztof Węcel. Linked geodata for profiling of telco users. *Studia Ekonomiczne*, 234 :199–213, 2015.

[29] Anastasios Noulas, Salvatore Scellato, Cecilia Mascolo, and Massimiliano Pontil. Exploiting semantic annotations for clustering geographic areas and users in location-based social networks. In *Fifth International AAAI Conference on Weblogs and Social Media*, 2011.

[30] Brent Snook, David Canter, and Craig Bennell. Predicting the home location of serial offenders : A preliminary comparison of the accuracy of human judges with a geographic profiling system. *Behavioral sciences & the law*, 20(1-2) :109–118, 2002.

[31] D Kim Rossmo. *Geographic profiling*. CRC press, 1999.

[32] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Edith Schonberg, and Kavitha Srinivas. Scalable highly expressive reasoner (sher). *Web Semantics : Science, Services and Agents on the World Wide Web*, 7(4) :357–361, 2009.

[33] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Aaron Kershenbaum, Edith Schonberg, Kavitha Srinivas, and Li Ma. Scalable semantic retrie-

val through summarization and refinement. In *AAAI*, volume 7, pages 299–304, 2007.

[34] Georgia Troullinou, Haridimos Kondylakis, Evangelia Daskalaki, and Dimitris Plexousakis. Rdf digest : Efficient summarization of rdf/s kbs. In *European Semantic Web Conference*, pages 119–134. Springer, 2015.

[35] Badre Belabbess, Jérémy Lhez, Musab Bairat, and Olivier Curé. Contextualisation de singularités en temps-réel par extraction de connaissances du web des données. In *Extraction et Gestion des Connaissances, EGC 2018, Paris, France, January 23-26, 2018*, pages 59–70, 2018.

[36] Jérémy Lhez and Olivier Curé. Profilage sémantique et probabiliste de zones géographiques. In *Extraction et Gestion des Connaissances, EGC*, 2017.

[37] Badre Belabbess, Musab Bairat, Jérémy Lhez, Zakaria Khattabi, Yufan Zheng, and Olivier Curé. Scouter : A stream processing web analyzer to contextualize singularities. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 624–631, 2018.

[38] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm : A benchmark for owl knowledge base systems. *Web Semantics : Science, Services and Agents on the World Wide Web*, 3(2-3) :158–182, 2005.

[39] Xiangnan Ren, Olivier Curé, Li Ke, Jérémy Lhez, Badre Belabbess, Tendry Randriamalala, Yufan Zheng, and Gabriel Képéklian. Strider : An adaptive, inference-enabled distributed RDF stream processing engine. *PVLDB*, 10(12) :1905–1908, 2017.

[40] Xiangnan Ren, Olivier Curé, Hubert Naacke, Jérémy Lhez, and Li Ke. Strider$^{\mathrm{r}}$: Massive and distributed RDF graph stream reasoning. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3358–3367, 2017.

[41] Jérémy Lhez, Xiangnan Ren, Badre Belabbess, and Olivier Curé. A compressed, inference-enabled encoding scheme for RDF stream processing. In *The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part II*, pages 79–93, 2017.

[42] Badre Belabbess, Musab Bairat, Jérémy Lhez, and Olivier Curé. Détection de singularités en temps-réel par combinaison d'apprentissage

automatique et web sémantique basés sur spark. In *Extraction et Gestion des Connaissances, EGC 2018, Paris, France, January 23-26, 2018*, pages 375–376, 2018.