Gestion de graphes de connaissances dans l'informatique en périphérie : gestion de flux, autonomie et adaptabilité

Joffrey de Oliveira

Thèse de doctorat

Gestion de graphes de connaissances dans l'informatique en périphérie : gestion de flux, autonomie et adaptabilité

présentée par DE OLIVEIRA Joffrey

soutenue le 11 décembre 2023

Guillaume Blin	Professeur à l'Université de Bordeaux	Rapporteur
Philippe Calvez	Directeur du laboratoire CSAI CRIGEN Engie	Co-encadrant de thèse
Olivier Curé	Professeur à l'Université Gustave Eiffel	Directeur de thèse
Hubert Naacke	Maître de conférences HDR	Examinateur
Nathalie Pernelle	Professeure à l'Université Paris 13	Rapporteur

Remerciements

Je tiens tout d'abord à remercier plusieurs personnes de m'avoir accompagné pendant ces années de thèse.

Je souhaite remercier mon directeur de thèse, le Professeur Olivier CURÉ, et à mon Co-encandrant le Docteur Philippe CALVEZ pour leur expertise et leur soutien constant tout au long de mes recherches. Leurs conseils éclairés et leur engagement ont été d'une valeur inestimable.

Je tiens également à remercier la Professeure Nathalie PERNELLE et le Professeur Guillaume BLIN, de l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse.

Mes remerciements s'étendent également à L'université Gustave Eiffel pour avoir fourni les ressources et l'environnement propice à la réalisation de cette recherche.

Je remercie aussi mes collègues, Christophe CALLE, Weiqin XU et Maxime THOUVENOT, pour leur contributions indispensables à ma thèse.

Je tient enfin à remercier ma famille et mes amis pour leur soutien infaillible, leurs encouragements et leur compréhension pendant ces trois années.

Table des matières

Table des figures

Liste des tableaux

Résumé

Les travaux de recherche menés dans le cadre de cette thèse de doctorat se situent à l'interface du Web sémantique, des bases de données et de l'informatique en périphérie (généralement dénotée Edge computing). En effet, notre objectif est de concevoir, développer et évaluer un système de gestion de bases de données (SGBD) basé sur le modèle de données Resource Description Framework (RDF) du W3C, qui doit être adapté aux terminaux que l'on trouve dans l'informatique périphérique.

Les applications possibles d'un tel système sont nombreuses et couvrent un large éventail de secteurs tels que l'industrie, la finance et la médecine, pour n'en citer que quelques-uns. Pour preuve, le sujet de cette thèse a été défini avec l'équipe du laboratoire d'informatique et d'intelligence artificielle (CSAI) du ENGIE Lab CRIGEN. Ce dernier est le centre de recherche et de développement d'ENGIE dédié aux gaz verts (hydrogène, biogaz et gaz liquéfiés), aux nouveaux usages de l'énergie dans les villes et les bâtiments, à l'industrie et aux technologies émergentes (numérique et intelligence artificielle, drones et robots, nanotechnologies et capteurs). Le CSAI a financé cette thèse dans le cadre d'une collaboration de type CIFRE.

Les fonctionnalités d'un système satisfaisant ces caractéristiques doivent permettre de détecter de manière pertinente et efficace des anomalies et des situations exceptionnelles depuis des mesures provenant de capteurs et/ou actuateurs. Dans un contexte industriel, cela peut correspondre à la détection de mesures, par exemple de pression ou de débit sur un réseau de distribution de gaz, trop élevées qui pourraient potentiellement compromettre des infrastructures ou même la sécurité des individus. Le mode opératoire de cette détection doit se faire au travers d'une approche conviviale pour permettre au plus grand nombre d'utilisateurs, y compris les non-programmeurs, de décrire les situations à risque. L'approche doit donc être déclarative, et non procédurale, et doit donc s'appuyer sur un langage de requêtes, par exemple SPARQL.

Nos estimons que l'apport des technologies du Web sémantique peut être prépondérant dans un tel contexte. En effet, la capacité à inférer des conséquences implicites depuis des données et connaissances explicites constitue un moyen de créer de nouveaux services qui se distinguent par leur aptitude à s'ajuster aux circonstances rencontrées et à prendre des décisions de manière autonome. Cela peut se traduire par la génération de nouvelles requêtes dans certaines situations alarmantes ou bien en définissant un sous-graphe minimal

de connaissances dont une instance de notre SGBD a besoin pour répondre à l'ensemble de ses requêtes.

La conception d'un tel SGBD doit également prendre en compte les contraintes inhérentes de l'informatique en périphérie, c'est-à-dire les limites en terme de capacité de calcul, de stockage, de bande passante et parfois énergétique (lorsque le terminal est alimenté par un panneau solaire ou bien une batterie). Il convient donc de faire des choix architecturaux et technologiques satisfaisant ces limitations. Concernant la représentation des données et connaissances, notre choix de conception s'est porté sur les structures de données succinctes (SDS) qui offrent, entre autres, les avantages d'être très compactes et ne nécessitant pas de décompression lors du requêtage. De même, il a été nécessaire d'intégrer la gestion de flux de données au sein de notre SGBD, par exemple avec le support du fenêtrage dans des requêtes SPARQL continues, et des différents services supportés par notre système. Enfin, la détection d'anomalies étant un domaine où les connaissances peuvent évoluer, nous avons intégrer le support des modifications au niveau des graphes de connaissances stockés sur les instances des clients de notre SGBD. Ce support se traduit par une extension de certaines structures SDS utilisées dans notre prototype.

Abstract

The research work carried out as part of this PhD thesis lies at the interface between the Semantic Web, databases and edge computing. Indeed, our objective is to design, develop and evaluate a database management system (DBMS) based on the W3C Resource Description Framework (RDF) data model, which must be adapted to the terminals found in Edge computing.

The possible applications of such a system are numerous and cover a wide range of sectors such as industry, finance and medicine, to name but a few. As proof of this, the subject of this thesis was defined with the team from the Computer Science and Artificial Intelligence Laboratory (CSAI) at ENGIE Lab CRIGEN. The latter is ENGIE's research and development centre dedicated to green gases (hydrogen, biogas and liquefied gases), new uses of energy in cities and buildings, industry and emerging technologies (digital and artificial intelligence, drones and robots, nanotechnologies and sensors). CSAI financed this thesis as part of a CIFRE-type collaboration.

The functionalities of a system satisfying these characteristics must enable anomalies and exceptional situations to be detected in a relevant and effective way from measurements taken by sensors and/or actuators. In an industrial context, this could mean detecting excessively high measurements, for example of pressure or flow rate in a gas distribution network, which could potentially compromise infrastructure or even the safety of individuals. This detection must be carried out using a user-friendly approach to enable as many users as possible, including non-programmers, to describe risk situations. The approach must therefore be declarative, not procedural, and must be based on a query language, such as SPARQL.

We believe that Semantic Web technologies can make a major contribution in this context. Indeed, the ability to infer implicit consequences from explicit data and knowledge is a means of creating new services that are distinguished by their ability to adjust to the circumstances encountered and to make autonomous decisions. This can be achieved by generating new queries in certain alarming situations, or by defining a minimal sub-graph of knowledge that an instance of our DBMS needs in order to respond to all of its queries.

The design of such a DBMS must also take into account the inherent constraints of Edge computing, i.e. the limits in terms of computing capacity, storage, bandwidth and sometimes energy (when the terminal is powered by a solar panel or a battery). Architectural and technological choices must therefore be made to

meet these limitations. With regard to the representation of data and knowledge, our design choice fell on succinct data structures (SDS), which offer, among other advantages, the fact that they are very compact and do not require decompression during querying. Similarly, it was necessary to integrate data flow management within our DBMS, for example with support for windowing in continuous SPARQL queries, and for the various services supported by our system. Finally, as anomaly detection is an area where knowledge can evolve, we have integrated support for modifications to the knowledge graphs stored on the client instances of our DBMS. This support translates into an extension of certain SDS structures used in our prototype.

Chapitre 1

Introduction

1.1 Contexte

Depuis ses débuts dans les années 90, le World Wide Web (WWW) a connu un certain nombre d'évolutions. Initialement créé pour rendre les informations accessibles aux humains, il fait désormais l'objet d'efforts visant à rendre les données plus structurées, connectées et porteuses de sens. Le principal objectif de cette plateforme est d'ajouter de la sémantique aux ressources du Web et ainsi de permettre aux machines d'interpréter les données d'une manière plus intelligente. Ces initiatives forment ce que l'on appelle le Web sémantique [11].

En parallèle au développement de cette extension du Web, le nombre d'appareils connectés n'a cessé de croître. Cette vision de l'internet des objets (IoT)[7] a posé un défi majeur pour la gestion et le traitement efficace de gigantesques quantités de données générées par ces appareils ainsi que pour la maintenance

des réseaux informatiques. Pour résoudre une partie de ces problèmes est apparu le concept d'informatique en périphérie (ou Edge Computing)[47]. Cette approche consiste à traiter les données au plus près de la source à partir de laquelle elles sont générées, limitant ainsi le transfert systématique des données vers l'informatique en nuage (Cloud computing).

Il est alors apparu nécessaire d'étendre les technologies du Web sémantique, et en particulier les graphes de connaissances (Knowledge Graph - KG) [26], dans le contexte de l'IoT et de l'Edge computing. Cette approche doit favoriser un traitement des données plus intelligent et une prise de décision rapide et autonome dès la périphérie des systèmes.

Mes premiers travaux sur le Web sémantique ont eu lieu lors de mon stage de master 2 au sein de la multinationale ENGIE. Ces travaux ont abouti à une publication [21] sur la création semi-automatisée de données sémantiques à partir de données conventionnelles, e.g., au format csv. A la suite de ce stage, ENGIE a financé ma thèse de doctorat, dans le cadre d'une collaboration du type CIFRE, dont ce document est l'aboutissement.

Dans cette thèse, nous aborderons les questions relatives à l'utilisation du Web sémantique dans le contexte de l'informatique en périphérie. Dans la prochaine section nous présenterons notre problématique sous la forme de trois questions de recherche.

1.2 Problématique

Le contexte de l'informatique en périphérie impose plusieurs contraintes, en particulier en ce qui concerne les machines utilisées, leur nombre et leur capacités. Ces dernières disposent généralement de peu de mémoire vive, de capacité réduite de stockage secondaire, d'une puissance de calcul limitée. La bande passante est généralement restreinte et l'autonomie énergétique est souvent limitée par l'utilisation de batteries ou de panneaux solaires.

En constrate à ces caractéristiues attendues, les systèmes de gestion de bases de données (SGBD) utilisés dans le cadre du Web sémantique ont généralement besoin de beaucoup de ressources, i.e., CPU, méméoire, mémoire secondaire. Le format le plus utilisé pour représenter les données sémantiques est le Resource Description Framework (RDF). Il peut être utilisé pour représenter de vastes graphiques dont le stockage et l'indexation nécessitent beaucoup de mémoire. Il est donc nécessaire de créer un SGBD supportant le modèle de données RDF capable de gérer ces graphes et leur évolution, malgré l'espace limité des appareils dédiés au Edge computing. Certains cas d'usages impliquent la lecture et l'analyse de données provenant de capteurs. La gestion de données provenant de capteurs implique de représenter des données graphe qui sont mises à jour fréquemment. Les SGBD RDF classiques gèrent généralement des graphes RDF qui sont relativement statiques et ne sont donc pas adaptés pour les données très dynamiques, orientées transactionnelles provenant de capteurs. Un cas d'usage fréquent dans l'IoT correspond à la détection d'anomalies. Celles-ci sont identifiés à partir de mesures prises par de capteurs et nécessitent parfois une réponse

plus rapide qu'une intervention humaine. Il est donc important, toujours dans le contexte de l'Edge computing, de permettre aux systèmes d'avoir un certain degré d'autonomie dans leur réponse aux anomalies.

C'est pourquoi notre objectif dans cette thèse est de créer un SGBD RDF capable de gérer des flux de données provenant de capteurs et d'actionneurs, ainsi que de supporter une prise de décision autonome en réponse aux anomalies de ces capteurs. Enfin ce système devra être capable de gérer des graphes des connaissances vastes en les adaptant aux appareils à mémoire restreinte du Edge computing.

1.3 Questions de recherche

1.3.1 QR1 : Comment intégrer une gestion avancée de flux de données dans un SGBD RDF en mémoire conçu pour le Edge computing ?

Dans le contexte d'un environnement Internet des objets (IoT) tel que la détection d'anomalies, l'analyse des données provenant de capteurs doit être réalisée avec une faible latence. En effet, ces capteurs génèrent de manière continue, non délimité et avec généralement une grande vélocité diverses formes de mesures, par exemple des données numériques pouvant correspondre à des températures provenant d'un bâtiment. Un SGBD RDF conçu pour ce contexte doit avoir la capacité à traiter efficacement l'interrogation avancée, c'est-à-dire en prenant en compte des conséquences implicites déduites de données et connaissances explicites, de tels flux de données. Le langage d'interrogation standard SPARQL n'est pas suffisant pour ces cas d'utilisation, et une extension est nécessaire pour permettre l'interrogation des flux de données à l'aide d'une approche de fenêtrage et d'une gestion appropriée de la réception des flux. De même, le mode d'exécution de ces requêtes doit être optimisé car celles-ci interrogent des données statiques, provenant d'une ontologie du domaine, et des données dynamiques, par exemple, les messures provevant des capteurs. Enfin, la capacité de notre SGBD RDF, nommé SuccinctEdge, à supporter des services de raisonnement au moment de l'exécution de requêtes doit être garantie dans cette extension du langage de requête.

Dans les systèmes de gestion de flux de données exécutant des requêtes de manière continue, on retrouve généralement la capacité à joindre des données statiques et dynamiques. Ces données statiques peuvent par exemple correspondre aux caractéristiques des capteurs alors que les données dynamiques correspondraient aux mesures émises par ces capteurs. Chaque requête sur les données provenant de capteurs ayant besoin d'être ré-exécutée, potentiellement plusieurs fois dans un laps de temps défini, e.g., de l'odre de quelques secondes ou minutes. Dans ces conditions, il est primordial pour notre extension streaming de SuccinctEdge de proposer une approche d'optimisation de l'exécution de requêtes.

1.3.2 QR2 : Comment adapter un SGBD RDF pour qu'il réponde aux anomalies de manière autonome en utilisant son KG ?

Dans le cadre de l'IoT, il peut être essentiel de réagir rapidement à la détection d'une anomalie. Cela peut ainsi permettre une intervention humaine dans les plus brefs délais et prévenir des accidents pouvant avoir des conséquences catastrophiques.

Dans certaines situations, la détection d'une fraude ou d'une anomalie peut conduire à la surveillance de nouvelles mesures, auparavant latentes. Pour être le plus réactif possible, le système d'analyse des mesures émises par des capteurs doit présenter une certaine autonomie. Dans un souci de réactivité et d'efficacité, une grande partie de cette gestion autonome des anomalies et risques doit être réalisée au plus près des équipements périphériques. L'interprétation des anomalies relevées et la prise de décision nécessite une approche basée sur la sémantique qui sera capable de déduire des conséquences implicites à partir de faits et de connaissances explicites. Un SGBD RDF conçus pour l'Edge computing avec une certaine autonomie est alors nécessaire.

Cela soulève la question suivante : comment pouvons-nous répondre à ces anomalies de manière autonome, en nous basant sur le graphe de connaissances du système qui a détecté l'anomalie ? Il est également nécessaire d'adapter cette solution aux appareils rencontrés dans le Edge computing et à leur contraintes pour être plus réactif.

1.3.3 QR3 : Comment Adapter des KG aux RDF Stores dédiés à l'Edge computing et supporter leur évolution

Dans un contexte industriel, il est courant de trouver des KG dont le domaine est très étendu, couvrant ainsi beaucoup plus de concepts que ceux réellement utilisés au sein d'un seul système. Lorsque l'on veut charger ces KG sur les appareils rencontrés dans le Edge computing, nous faisons face à un problème de taille mémoire auxquelles même des systèmes de stockage RDF spécialisés ne peuvent résoudre.

Dans plusieurs situations concrètes, nous avons pu observer que l'intégralité du KG n'est pas nécessaire sur une instance donné de notre SGBD RDF, i.e., un client SuccinctEdge. En effet, une instance ne doit répondre qu'à un ensemble prédéfini de requêtes et nécessite donc uniquement un sous-ensemble des connaissances du domaine. Cela soulève donc la question de comment définir un sous-ensemble des connaissances suffisant pour satisfaire un ensemble de requêtes de façon efficace ?

De plus, dans le contexte d'un environnement IoT, les systèmes peuvent avoir des besoins de connaissances dynamiques, par exemple dans le cas où de nouvelles requêtes ont besoin d'être exécutées. Il est donc aussi nécessaire de trouver une solution de maintenance incrémentale des sous-ensembles de connaissances, tenant compte de l'évolution de l'ensemble de requêtes.

1.4 Contributions

Les contributions que nous présentons dans cette section répondent aux trois questions de recherche que nous venons de mentionner. Dans cette section, nous allons énumérer les contributions en fonction de chacune des trois questions. Toutes les implémentations de nos contributions ont été intégrées dans le système SuccinctEdge.

— Concernant QR1 : extension avec Eclipse Mosquitto et support d'un requêtage continu SPARQL intégrant les modes streaming et micro-batch ainsi que différents supports de fenêtrage (fenêtres sautantes, glissantes).

— Concernant QR2 : dans le cadre d'une anomalie donnée, génération de requêtes SPARQL pertinentes à partir du graphe de connaissances du domaine.

— Concernant QR3 : extension des structures de données succinctes (SDS) utilisées dans SuccinctEdge pour un support des modifications et génération automatique de requêtes SPARQL pour la création de sous-graphes de connaissances adapté à chaque client.

1.5 Publications

Durant cette thèse de doctorat, réalisée dans le cadre d'un financement du type CIFRE avec Engie, les contributions mentionnées précédemment ont fait l'objet de plusieurs publications.

— **Knowledge graph stream processing at the edge.** Joffrey de Oliveira, Christophe Callé, Weiqin Xu, Philippe Calvez, Olivier Curé. Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems 2022 (ACM DEBS) : 115-125.

— **Towards Autonomous Anomaly Management Using Semantic Technologies at the Edge.** Joffrey de Oliveira, Christophe Callé, Philippe Calvez, Olivier Curé. IEEE EDGE 2023 : 159-165

— **Adapting Knowledge Graphs to Edge Computing Devices.** Joffrey de Oliveira, Christophe Callé, Olivier Curé. IEEE Big Data 2023

1.6 Organisation de la thèse

Cette thèse est organisée en six chapitres. Les principales contributions sont présentées dans les chapitres trois, quatre et cinq. Les chapitres sont organisés comme suit.

Le premier chapitre présente le contexte et les trois principales questions de recherche abordées dans cette thèse.

Le deuxième chapitre présente les concepts nécessaires à la compréhension des travaux de recherche de la thèse. En particulier les différents standards du Web sémantique, le Edge computing et les structures de données succinctes.

Il présente également un état de l'art, notamment les librairies supportant les SDS, les SGBD RDF, certains classiques et d'autres dédiés au Edge computing, en particulier le système SuccinctEdge qui est au cœur des travaux de recherche de cette thèse. Nous y présentons aussi les éléments fondamentaux des SDS vecteurs de bits RRR (structures de Raman, Raman et Rao [46]) et différents systèmes de communication pour l'Edge computing utilisés dans ces travaux.

Les troisième, quatrième et cinquième chapitres adressent respectivement les questions de recherches QR1, QR2 et QR3. Chacun de ces chapitres présente l'approche adoptée pour résoudre sa question de recherche, complète l'état de l'art entrevu en Chapitre 2 et propose une évaluation de l'extension du système SuccinctEdge.

Le sixième chapitre conclut la thèse en résumant ses contributions et en proposant des pistes de recherches pour des travaux à venir.

Chapitre 2

Prérequis

Ce chapitre fournit les informations nécessaires à la compréhension de cette thèse. Il s'attache donc à présenter le système d'encodage LiteMat qui est utilisé pour la création de dictionnaires pour les concepts, propriétés et instances de nos graphes de connaissances. Cet encodage est utilisé intensivement dans notre système de gestion de graphes de connaissances qui se base sur l'exploitation de structures de données succinctes que nous introduisons également. Nous présentons ensuite le Web sémantique, l'informatique à la périphérie et des outils associés à la gestion de flux de données.

2.1 LiteMat

LiteMat [17] correspond à un schéma d'encodage pour les données RDF et les ontologies RDFS. Dans cette thèse, nous nous concentrons sur le sous-

ensemble ρdf[37] de RDFS, c'est-à-dire les inférences associées aux constructeurs tels que `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`. Dernièrement, certaines extensions de LiteMat ont été développées, par exemple pour le support de l'héritage multiple des concepts, des propriétés transitives, inverses et sameAs [18], mais celles-ci n'ont pas été exploitées dans les travaux de recherche de cette thèse.

Le principal avantage de LiteMat par rapport à d'autres approches d'encodage, par exemple celles que l'on trouve dans la plupart des SGBD RDF [1][48][42][25], est qu'il peut être qualifié de conscient de la sémantique. Cela signifie que les valeurs entières associées aux concepts et aux propriétés de l'ontologie véhiculent la sémantique de leurs hiérarchies respectives. En conséquence, les systèmes qui utilisent LiteMat peuvent efficacement prendre en charge des services de raisonnement.

L'encodage sémantique des concepts et prédicats prend en charge les services de raisonnement généralement requis lors du traitement des requêtes. Par exemple, considérons une requête demandant la valeur de pression des capteurs de type S1. Cela serait exprimé par les deux motifs triples suivants : `?x pressureValue ?v. ?x type S1`. Dans le cas où le concept de capteur $S1$ a n sous-concepts, une reformulation naïve de la requête nécessite l'exécution de l'union de n+1 requêtes. Avec l'encodage sensible à la sémantique de LiteMat, nous sommes en mesure, en utilisant deux opérations de décalage de bits et une addition, de calculer l'intervalle d'identifiants, c'est-à-dire [borne inférieure, borne supérieure), de tous les sous-concepts directs et indirects de $S1$. Ainsi, nous pouvons calculer cette requête avec une simple reformulation : (i) remplacer le concept S1 par une nouvelle variable : `?x type ?newVar` et (ii) introduire une clause de filtre qui contraint les valeurs de cette nouvelle variable : `FILTER (?newVar>=borne inférieure && ?newVar<borne supérieure)`.

Dans cette section, nous ne détaillons pas la génération de l'encodage mais invitons le lecteur intéressé à obtenir des détails dans l'article [17].

2.2 Structures de données succinctes

Les structures de données succinctes (SDS) forment une classe de structures de données capables de gérer et de traiter efficacement de grandes quantités d'informations tout en minimisant les besoins de stockage. Elles permettent également des opérations efficaces de requêtage sans nécessiter de décompression des données. Les structures couramment utilisées comprennent les vecteurs de bits, bit vector (BV) en anglais, et les arbres d'ondelettes, wavelet trees (WT) en anglais, qui sont des arbres binaires, balancés dont les noeuds sont des BVs.

Ces structures prennent en charge trois opérations :

— `access`(i) récupère l'élément stocké à la i$^{\text{ème}}$ position de la structure,

— `rank`$_{\text{v}}$(i) récupère le nombre d'éléments v dans le préfixe de taille i de la structure,

— `select`$_{\text{v}}$(i) récupère la position du i$^{\text{ème}}$ élément v dans la structure.

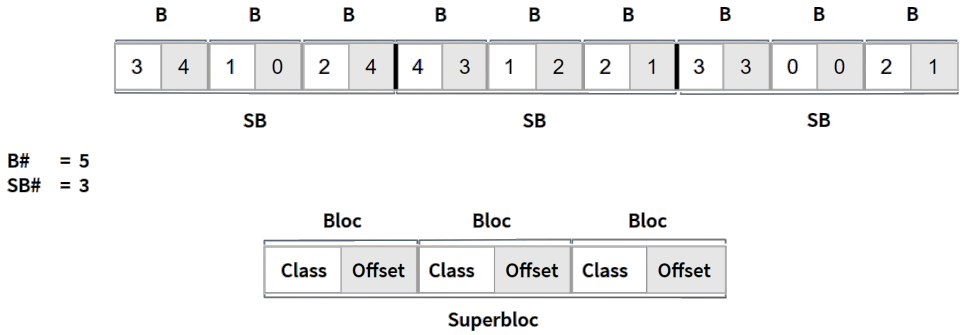

FIGURE 2.1 – Représentation des classes, offsets, blocs et Superblocs au sein d'un BV RRR

2.2.1 Vecteurs de bits RRR

Les vecteurs de bits (BV) sont une structure de données succinctes qui ne permet que de représenter deux éléments, typiquement des 0 et des 1. La représentation RRR[46] permet de représenter les BVs en compressant les données mais aussi en optimisant l'exécution des opérations de `rank` et `select`, afin de les calculer en temps constant. Pour cela, la représentation RRR utilise plusieurs structures dédiées au `rank` et d'autres dédiées au `select`. Ces structures sont définies par deux paramètres qui permettent de minimiser le temps des exécutions des opérations et l'empreinte mémoire des structures.

Pour les structures de `rank`, ces paramètres sont la taille des blocs (BS) et la taille des superblocs (SBS). Pour un BV en entrée, la représentation RRR va compresser les données d'origine dans une structure en créant un bloc. Pour chaque BS bits du BV original, ces blocs sont ensuite regroupés en superblocs, comprenant SBS blocs. Pour représenter un bloc, la structure utilise deux entiers, le premier étant le nombre de bits définis à 1 dans le bloc, qu'on appelle la "classe", et le second étant utilisé pour reconstruire les données d'origine en se basant sur l'entier précédent, qu'on appelle "l'offset". Pour minimiser la taille de ces entiers, la structure n'écrit que jusqu'au bit le plus significatif de l'entier.

Par exemple, comme illustré dans la Figure 2.1 les bits du BV original sont représentés par une classe de 3 et un offset de 1 dans la représentation RRR. Selon les tailles de bloc utilisées, cette représentation permet de réduire l'espace utilisé par les données d'origine mais dans tout les cas elle permet d'accélérer l'exécution des opérations du type `rank` en connaissant déjà le nombre de bits à 1 dans le bloc. Le fait que l'on puisse retrouver le bloc original à partir de ces données signifie qu'il n'y a aucune perte dans cette compression. De même, il n'est pas nécessaire de décompresser les données même si l'on cherche un `rank` au milieu d'un BV. Pour une taille de bloc donnée, un certain couple classe/offset correspond toujours au même bits dans le BV original. Nous pouvons donc maintenir une structure commune à plusieurs BV où le `rank` dans le bloc à chaque position est déjà précalculé.

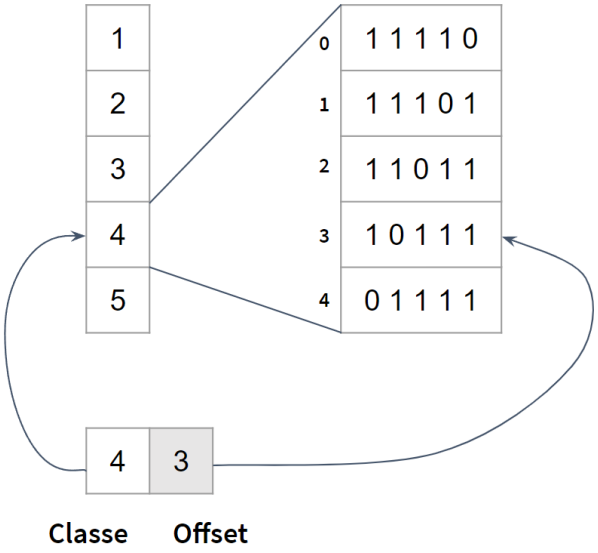

FIGURE 2.2 – Structure pour décompresser les blocs RRR

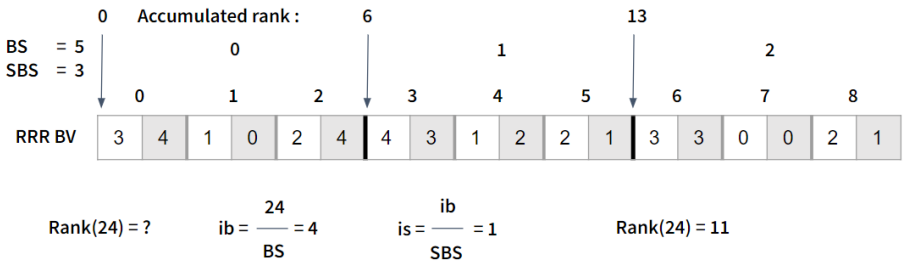

FIGURE 2.3 – Représentation d'un rank sur un BV RRR

Un superbloc (SB) est un groupe de SBS blocs, il sert à stocker le nombre de bits à 1 entre le début du BV et le début du SB sous la forme d'un entier. C'est avec ces deux éléments que la représentation RRR peut répondre aux opérations `rank` en temps constant O(1) pour un BS et un SBS donnés. Une operation $\texttt{rank}_1$(i), s'effectue en calculant dans quel SB et bloc se trouve le i ème bit, soit i/BS, puis on trouve le SB contenant ce bloc en calculant (i/BS)/SBS. Avec ces informations, on peut directement lire le `rank` cumulé au début du SB calculé en consultant l'entier de la structure de `rank`, puis nous ajoutons le premier entier de chaque bloc du SB jusqu'à atteindre le bloc qu'on recherche. Enfin au sein de ce bloc, nous ajoutons le `rank` précalculé pour le bit qu'on recherche, ce qui nous donne la valeur finale qui était recherchée.

Par exemple sur la Figure 2.3, nous calculons un $\texttt{rank}_1(24)$ sur un BV RRR,

le BV utilise un BS de 5 bits et un SBS de 3 blocs. Le système détermine donc que le dernier bit utilisé pour notre rank se trouve dans le bloc 4, puis nous calculons que ce bloc est dans le superbloc 1. Nous initialisons notre résultat sur la valeur de `rank` du SB qui est 6, puis nous ajoutons la classe du bloc 3, premier bloc du SB 1, dont la valeur est 4. Nous ne pouvons pas juste ajouter la classe du superbloc 4 car le bit à 1 qui s'y trouve pourrait être situé après le 24ème bit. Nous vérifions donc le `rank` à la position recherchée dans ce bloc, pour une classe de 1 et un offset de 2, ce qui nous renvoie 1. Le résultat est donc 11.

L'article original de présentation de RRR présente également une structure pour prendre en charge les opérations `select` en temps constant O(1). Néanmoins, la complexité de la structure a conduit la plupart des implémentations existantes de librairies SDS à créer une solution plus simple qui peut tout de même prendre en charge des opérations `select` en temps constant.

Une opération `access` peut être effectuée en accédant au bloc qui contient le bit recherché et en déterminant la valeur du bit précis à partir de sa position dans le bloc et de la classe et offset de ce bloc.

2.2.2 Wavelet Trees

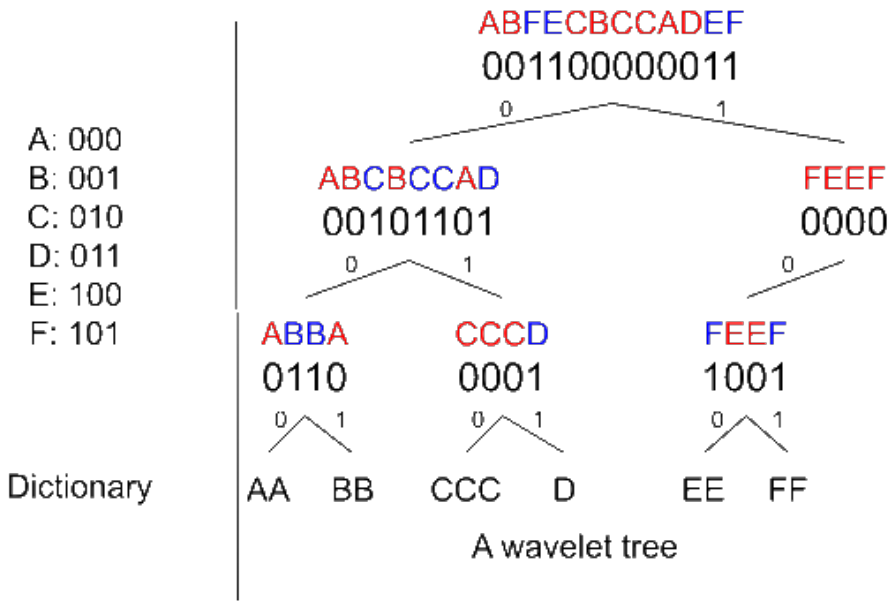

FIGURE 2.4 – Exemple de wavelet tree avec son dictionnaire

Un wavelet tree (WT) est une SDS qui représente ses données sous la forme d'un arbre binaire et balancé. Chaque élément du vocabulaire est représenté par un encodage en binaire. Chaque noeud de l'arbre est un BV contenant la

représentation d'une partie des données, tant que l'on n'est pas dans une feuille, la valeur à 0 ou 1 d'un bit indique s'il est stocké dans le fils gauche ou droit, du noeud actuel. En suivant l'encodage d'un élément à partir de la racine, nous accédons à toutes les occurrences de l'élément dans le WT. Si nous considérons que les 0 sont représentés dans le fils gauche, alors le i ème 0 d'un noeud est représenté par le i ème bit, à 0 ou 1, dans le BV du fils gauche du noeud actuel. Ce principe est le même avec les bits à 1 et le fils droit du noeud actuel. Les opérations de `rank`, `select` et `access` peuvent être décomposées en ces mêmes opérations SDS sur les BVs des noeuds de l'arbre. Ces opérations ne nécessitent que de visiter les noeuds entre la racine et une feuille incluse, et sur chaque noeud visité il n'est nécessaire d'effectuer qu'une opération SDS pour passer au noeud suivant. Cela signifie que si l'arbre est équilibré et en utilisant les BVs RRR pour représenter l'arbre, les opérations de `rank`, `select` et `access` sont calculées en O(log n), où n est la taille du vocabulaire.

Les opérations SDS sur le WT sont décomposées sur les BVs des noeuds comme suit : $\texttt{rank}_{\text{E}}$(i) renvoie le nombre de E parmi les i premiers éléments du WT, pour cela, à partir de la racine nous calculons $\texttt{rank}_{\text{C}}$(i), où C est le bit qui encode l'élément à la racine. On réutilise ensuite la valeur renvoyée par cette opération sur le BV du noeud suivant, tout en progressant dans la lecture du code. Nous répétons cette opération jusqu'à atteindre la feuille où sont stockées les occurrences de E. Le retour de la dernière opération de `rank` sur cette feuille indique combien de E sont présents parmis les i premiers éléments.

L'opération $\texttt{select}_{\text{E}}$(i) renvoie la position dans le WT, du i ème E. Pour cela, nous démarrons le calcul de la feuille contenant les E et nous utilisons l'opération $\texttt{select}_{\text{C}}$(i) où C est le bit qui encode l'élément E à sa feuille. Nous réutilisons ensuite la position renvoyée dans le noeud parent tout en progressant dans la lecture du code. Nous répétons cette opération jusqu'à atteindre la racine où le résultat de la dernière opération sera la position dans le WT du i ème élément E.

L'opération `access(i)` renvoie le i ème élément du WT. Pour cela nous partons de la racine et nous utilisons une opération $\texttt{rank}_{\text{V}}$(i) où V est la valeur du i ème bit du BV, nous poursuivons ensuite dans le noeud indiqué par V, tout en retenant les bits qui ont été lus. Nous réutilisons la valeur renvoyée par l'opération pour la prochaine opération de `rank`. Nous répétons cette opération jusqu'à atteindre une feuille. La dernière opération de `rank` indique ici le dernier bit qui encode l'élément recherché et nous pouvons lire le code complet qui a été accumulé pour renvoyer l'élément recherché.

Par exemple, considérons la séquence de chaînes $ABFECBCCADEF$, où chaque lettre est associée à un identifiant entier dans un ordre croissant, par exemple A et B sont respectivement assignés aux valeurs 0 et 1 (voir le dictionnaire dans la Figure 2.4b). La Figure 2.4c affiche le WT de cette séquence selon cet encodage. Une structure d'arbre est construite à partir de cette séquence de la manière suivante : chaque niveau de cet arbre divise la séquence des nœuds précédents en deux sous-séquences par le bit correspondant. Par exemple, de la racine au premier niveau, $ABFECBCCADEF$ est divisé en $ABCBCCAD$ et $FEEF$ par le premier bit de chaque identifiant. Cette stratégie est appliquée de

manière récursive jusqu'à ce que chaque feuille soit calculée.

2.2.3 Librairies disponibles

Différentes librairies mettant en oeuvre des SDS ont été développées. Parmi celles-ci, on retrouve Pizza&chili [1] (un corpus pour l'indexation compressée), Sux [2] (comporte des implémentations de vecteurs de bits and opérations de `rank` et `select`), Succinct [3] (vecteurs de bits, succinct tries), libcds2 [4] (vecteurs de bits, wavelet trees) et sdsl-lite [5]. Dans un premier temps, ces bibliothèques ont prouvé que les structures de données statiques succinctes pouvaient être pratiques dans le développement d'applications, en plus d'être théoriquement attrayantes.

Certaines de ces librairies ne sont néanmoins pas maintenues. Ainsi, les dernières mises sur le Github de Succinct et Sux datent respectivement de 2013 et 2020. Enfin, le spectre de ces librairies est assez hétérogène, par exemple, elles n'implantent pas toutes les structures du type WTs.

Parmi, les bibliothèques disponibles, sdsl-lite[22] est certainement la plus complète, la plus testée et utilisée. Elle propose des implémentations optimisées de BVs et WTs qui se révèlent particulièrement essentielles pour notre implantation de SGBD RDF dédié à l'informatique en périphérie. Elle inclut notamment une implémentation des BV basée sur la représentation RRR. Cette implémentation se concentre sur la compression des données et la structure de l'opération de `rank` mais n'inclut pas la structure de `select` originale. En fait, les opérations de `select` sont réalisées par une recherche binaire sur les données compressées du BV, ce qui offre des performances acceptables pour la plupart des cas. Nous avons préféré cette librairie à libcds2 en raison de sa documentation plus détaillée et de sa dernière maintenance plus récente, 2013 pour libcds2 et 2019 pour sdsl-lite.

2.3 Web sémantique

Le "Web sémantique" est un concept qui vise à enrichir et à améliorer la manière dont les informations sont organisées, partagées et interprétées sur le World Wide Web (W3C). Proposé par Tim Berners-Lee, le Web sémantique vise à permettre aux machines l'interprétation des données disponibles sur le Web [11]. Cela va donc bien au-delà du simple affichage de pages Web de la première version du Web, que l'on peut qualifier de Web syntaxique. En effet, le Web sémantique ne s'adresse plus qu'aux utilisateurs humains mais également à des agents et services informatiques. Contrairement au Web conventionnel, où l'essentiel du code (HTML) avait pour objectif une représentation dans un navigateur internet, le Web sémantique met l'accent sur la sémantique des données

1. http ://pizzachili.dcc.uchile.cl/
2. https ://github.com/vigna/sux
3. https ://github.com/ot/succinct
4. https ://github.com/fclaude/libcds2
5. https ://github.com/simongog/sdsl-lite

dans l'objectif de permettre une interprétation de celles-ci. Cette approche permet aux machines de traiter et de raisonner sur les informations et connaissances disponibles sur le Web.

Le Web sémantique est basé sur diverses technologies et normes qui ont été développées au sein du W3C et ont entraîné des recommandations. Celles-ci concernent la modélisation de données (RDF) et d'ontologies, i.e. RDF Schema (RDFS) et la famille des Web Ontology Languages (OWL), ainsi que le requêtage de données RDF à l'aide du langage SPARQL. Ces normes et langages seront présentés dans la suite de cette section.

2.3.1 Resource Description Framework

Le Resource Description Framework(RDF) est un modèle de données prenant la forme d'un multigraphe dirigé étiqueté. En supposant des ensembles infinis disjoints I (IRI pour Internationalized Resource Identifiers), B (nœuds vides) et L (littéraux), un triplet (S, P, O) $\in$ (I $\cup$ B) x I x (I $\cup$ B $\cup$ L) est appelé un triplet RDF où S, P et O représentent respectivement le sujet, le prédicat et l'objet de ce triplet.

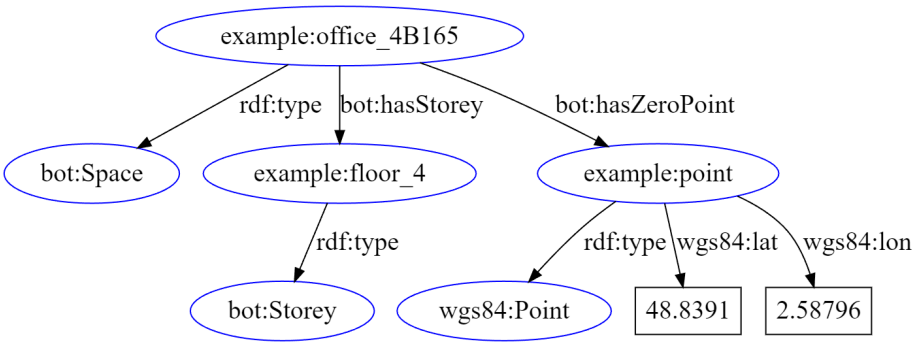

FIGURE 2.5 – Exemple de graphe RDF d'un bureau

La Figure 2.5 présente un exemple de graphe RDF où les IRIs sont représentées par des cercles bleus, les relations sont étiquetées avec leur prédicat et les littéraux sont représentés par des rectangles. Cet exemple représente des informations sur le bureau 4B165, en particulier sa position géographique et l'étage où il se trouve, en utilisant l'ontologie BOT [6] et le vocabulaire RDF WGS84 [7].

Un graphe nommé [12] est un sous-ensemble de triplets dans un graphe RDF. Il peut y avoir plusieurs graphes nommés à l'intérieur d'un graphe RDF, contenant éventuellement des triplets partagés entre plusieurs de ces graphes. Ces collections de triplets sont plus faciles à manipuler en tant que groupe que la somme de leurs parties, ce qui rend plus efficace leur suppression ou leur mise à

6. https ://w3c-lbd-cg.github.io/bot/
7. https ://www.w3.org/2003/01/geo/

jour dans un graphe, en tant que groupe. Le langage de requête SPARQL prend en charge l'interrogation de graphes nommés à l'aide de la clause FROM.

2.3.2 Ontologies et bases de connaissances

Il existe plusieurs langages permettant de représenter et de décrire des connaissances sous forme de graphes RDF. Ces langages facilitent l'échange et l'intégration de données sémantiques entre différentes applications et systèmes.

Ces langages diffèrent en terme du niveau d'expressivité supporté. Le language RDF Schema (RDFS) est une extension de RDF qui fournit des éléments pour définir des vocabulaires et des hiérarchies de classes et de propriétés. Avec RDFS, il est possible de définir des sous-classes, des sous-propriétés, et de restreindre les objets et sujets des propriétés à certaines classes. Cela permet d'organiser les ressources en classes et de spécifier des relations entre les classes et les propriétés. C'est l'ensemble de ces relations entre différents termes d'un vocabulaire qui forment une ontologie, une spécification explicite qui définis comment représenter une connaissance.[23]

Dans ce travail, nous considérons le système déductif RDF minimal ρdf qui a été défini et étudié théoriquement dans [38]. En résumé, *ρdf* utilise les propriétés `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:range` et `rdfs:domain` pour effectuer des inférences. Une propriété RDF est définie comme une relation entre une ressource sujet et une ressource objet. RDFS permet de décrire ces relations en termes de classes de ressources auxquelles elles s'appliquent en spécifiant la classe du sujet (c'est-à-dire le domaine) et la classe de l'objet (c'est-à-dire le co-domaine ou range en anglais) du prédicat correspondant. Les propriétés `rdfs:range` et `rdfs:domain` permettent respectivement d'indiquer que le sujet et l'objet d'une certaine `rdf:Property` doivent être une instance d'une certaine `rdfs:Class`. La propriété `rdfs:subClassOf` est utilisée pour indiquer qu'une classe donnée (`rdfs:Class`) est une sous-classe d'une autre classe. De même, en utilisant la propriété `rdfs:subPropertyOf`, on peut indiquer que toute paire de ressources (c'est-à-dire sujet et objet) liées par une certaine propriété est également liée par une autre propriété.

Le Web Ontology Language (OWL) est un langage d'ontologie plus expressif que RDFS. Il est capable de réaliser des inférences plus avancées, comme l'inférence sur la symétrie ou la transitivité de propriétés. Il permet aussi d'exprimer des cardinalités. Dans sa version d'origine, OWL proposait trois sous-langages, chacun offrant un niveau de complexité croissant : OWL Lite, OWL DL et OWL Full. Par la suite, la famille de langages de OWL a été étendue pour intégrer des nouvelles versions où le raisonnement est praticable (tractable en anglais) avec une complexité polynomiale des inférences : OWL QL (pour Query Language), OWL RL (pour Rule language) et OWL EL (pour la logique de description EL). Ces langages d'ontologies permettent de définir les relations entre des termes mais c'est seulement en y ajoutant un ensemble de données que nous obtenons une base de connaissances.

Une base de connaissances, Knowledge base (KB) en anglais, est composée d'une ontologie, également appelée boîte terminologique (TBox), et d'une base

de faits, également appelée boîte assertionnelle (ABox). Dans le contexte RDF, un KG correspond à la représentation d'une KB sous forme d'un graphe de données RDF.

2.3.3 SPARQL

Pour interroger des graphes RDF, aussi bien une ABox qu'une TBox„ il est nécessaire d'utiliser un langage de requêtage. A cet effet, le W3C propose le langage SPARQL [8] dont l'acronyme correspond à "SPARQL Protocol and RDF Query Language". La syntaxe SPARQL suit l'approche SELECT-FROM-WHERE du langage SQL. La clause `SELECT` est utilisé pour spécifier les variables (distinguées) que l'on souhaite récupérer dans le résultat de la requête. Les variables sont des symboles précédés d'un point d'interrogation, qui représentent les valeurs recherchées. La clause `FROM` est utilisé pour spécifier la source de données (un graphe) à partir de laquelle la requête va récupérer les informations. Il permet d'indiquer le graphe RDF sur lequel la requête doit s'exécuter. La clause `WHERE` est utilisé pour définir les motifs de triplet, ou triple pattern (TP) en anglais, que les données doivent satisfaire pour être incluses dans le résultat. Les motifs décrivent les relations entre les sujets, les prédicats et les objets et peuvent contenir des variables, qui peuvent être présentes ou non dans la clause SELECT. L'ensemble des TPs de la clause forment un motif de graphe de base, basic graph pattern (BGP) en anglais. C'est ce BGP qui sera comparé au graphe RDF pour déterminer le résultat de la requête. Il existe d'autres clauses dans le langage pour spécifier l'inclusion ou l'exclusion de certains TPs, la clause `FILTER` par exemple, permet d'appliquer des conditions supplémentaires sur les données récupérées à l'aide d'expressions booléennes, de comparaisons ou de fonctions s'appliquant sur les variables des TPs.

Intuitivement, le traitement des requêtes SPARQL consiste à renvoyer les assignations de variables distinguées correspondant au BGP dans un graphe RDF donné. Le traitement SPARQL est basé sur la correspondance de motifs de graphe. En plus des ensembles I, B et L, la signature d'un TP en SPARQL nécessite V, un ensemble infini de variables. Nous pouvons définir récursivement un motif de graphe SPARQL comme suit : (i) un triplet $gp \in (I \cup B \cup V)$ x $(I \cup V)$ x $(I \cup V \cup B \cup L)$ est un motif de graphe SPARQL, (ii) si gp_1 et gp_2 sont des motifs de graphe, alors $(gp_1.gp_2)$ représente un groupe de motifs de graphe qui doivent tous correspondre, (gp_1 `UNION` gp_2), qui représente des alternatives de motifs, sont des motifs de graphe, et (iii) si gp est un motif de graphe et C est une condition intégrée, alors l'expression (gp `FILTER` C) est un motif de graphe qui permet de restreindre les solutions d'une correspondance de motifs de graphe en fonction de l'expression C.

2.3.4 Flux de données et requêtes SPARQL continues

Le C-SPARQL (Continuous SPARQL) est une extension du langage de requête SPARQL, spécialement conçue pour le traitement de flux de données RDF

8. https ://www.w3.org/TR/sparql11-overview/

en temps réel. Pour ce faire le C-SPARQL utilise de nouveaux mots-clés, par exemple, STREAM est utilisé pour déclarer les flux de données RDF qui seront utilisés dans la requête. Il permet de spécifier les sources de flux en temps réel à partir desquelles les données seront obtenues. En C-SPARQL les flux de données sont traités avec des fenêtres glissantes, dont la taille de pas est définie par le mot-clé STEP, On indique la taille des fenêtres avec le mot-clé RANGE.

Bien que d'autres extensions continues de SPARQL aient été développées, CQELS [28], Etalis [4], C-SPARQL reste la principale source d'inspiration pour notre propre solution d'interrogation de flux de données RDF.

2.3.5 SHACL

Le langage de contraintes SHACL [9] (Shapes Constraint Language) est une recommendation du W3C permettant de valider des graphes RDF par rapport à un ensemble de contraintes. Pour ce faire, SHACL utilise des formes et des graphes RDF qui décrivent un ensemble de conditions utilisées pour valider des données. Les graphes RDF décrivant les contraintes sont appelés "graphes de formes" et les graphes RDF contenant les données à valider sont appelés "graphes de données". Les formes SHACL peuvent être utilisées pour déduire des triplets RDF à partir des triplets explicites dans le graphe de données en utilisant des règles SHACL [10].

2.3.6 RDF stores

Il existe plusieurs études [3],[49] et ouvrages [15] sur les SGBD adoptant le modèle de données RDF. Dans cette section, notre attention se porte sur les RDF stores spécifiquement conçus pour l'informatique en périphérie (Edge computing), ainsi que sur les bibliothèques qui facilitent le développement de tels systèmes ou qui intègrent des RDF stores légers pouvant être intégrés dans une application.

Librairies pour le développement d'un RDF store

Apache Jena [11] est une bibliothèque et une boîte à outils open source pour la création d'applications Web sémantique et données liées (Linked data). C'est l'une des bibliothèques les plus utilisées en raison de son implémentation sur la machine virtuelle Java (JVM), son large spectre de fonctionnalités et à sa maturité.

Apache Jena inclut des services pour la manipulation de données RDF, pour les principaux langages d'ontologie (RDFS, OWL). Cela lui permet de supporter des inférences avec différents niveaux d'expressivité. Le projet propose également un support pour SPARQL (nommé ARQ) et SHACL. Enfin, il comporte

9. https ://www.w3.org/TR/2017/REC-shacl-20170720/
10. https ://www.w3.org/TR/2017/NOTE-shacl-af-20170608/
11. https ://jena.apache.org/

plusieurs SGBD RDF, notamment la seconde version de son Triple Database (TDB2).

Eclipse RDF4J[30] est un framework RDF open source développé en Java et maintenu par la fondation Eclipse. Les fonctionnalités supportées sont relativement proches de celles d'Apache Jena. Par exemple, il supporte SPARQL 1.1 et plusieurs formats de données RDF, e.g., RDF/XML, Turtle. Il propose un stockage en mémoire des triplets RDF et un stockage sur disque. Son stockage en mémoire est particulièrement efficace sur de petits jeux de données. Son API pour la manipulation de triplets RDF et de requêtes SPARQL a été adopté pour le développement de plusieurs SGBD RDF que l'on peut qualifier de prêt à la production, e.g. Blazegraph [12] qui est à la base d'AWS Neptune.

RDF stores pour le Edge Computing

Les SGBD RDF spécialement conçus pour l'informatique en périphérie nécessitent souvent des structures de données compressées ainsi que la capacité à gérer un ensemble de données généralement moins volumineux que pour un SGBD classique, une stratégie d'indexation adaptée, un optimiseur de requêtes plus léger. D'autres exigences telles que la stratégie de stockage adaptée aux appareils, le traitement en continu et l'efficacité énergétique sont également prises en compte dans certains systèmes existants.

WaterFowl [16] a été conçu et développé dans le cadre d'un projet académique dans l'objectif de fournir une preuve de concept d'un SGBD RDF compact et incluant des capacités de raisonnement. Inspiré de HDT [35], il a poussé la logique de l'utilisation des SDS en représentant l'ensemble des membres d'un triplet RDF avec des BVs et WTs. Pour ce faire, il exploite un encodage de dictionnaire[34] qui donnera par la suite naissance à l'approche LiteMat.

Bien que possédant certaines des caractérisques d'un SGBD RDF pour le Edge computing, WaterFowl ne peut être considéré comme un concurrent direct des systèmes que nous allons décrire dans la suite de cette section. En effet, il n'est pas équipé d'un système de publication-souscription -pub-sub en anglais) qui lui permettrait de recevoir des flux de messages depuis des capteurs. De même, son moteur d'exécution de requêtes ne lui permet pas de supporter la notion de fenêtrage, e.g., sautante et glissante.

RDF4Led [30] est un SGBD RDF spécifiquement conçu pour l'informatique en périphérie. Il adopte une approche de persistance des données, c'est-à-dire que toutes les données sont stockées sur une carte SD. Cela est partiellement motivé par un composant d'exécution de requêtes basé sur plusieurs index. Cette approche lui permet de minimiser son empreinte sur la mémoire vive mais impose une perte de performance en raison d'opérations d'entrée/sortie. Pour stocker ses triplets sous un format plus compact sur la carte SD, ce système utilise un format hybride de propriétés et d'objets liés à un sujet, dénoté Molécule RDF [50]. Ces molécules sont ensuite stockées en blocs de la taille des blocs de mémoire flash (SSD) sur lesquels ils sont persistés. Cette approche permet un

12. https ://blazegraph.com/

gain de performance lorsque les données doivent être persistées. RDF4Led ne propose pas de solution pour le raisonnement des triplets RDF.

Fed4Edge [43] est une version décentralisée du moteur CQELS (Continuous Query Evaluation over Linked Stream)[28] qui bénéficie des travaux menés sur RDF4Led. CQELS correspond à extension de SPARQL visant à interroger les données RDF en continu en supportant la définition de fenêtres. Fed4Edge adopte le langage de requête continue CQELS-QL [**?**] basé sur SPARQL. Fed4Edge se concentre sur la fédération de requêtes SPARQL [5] et ne supporte pas le raisonnement lors de l'exécution de requêtes.

SuccinctEdge [13] (SE) est un SGBD RDF client-serveur, en mémoire et capable d'inférence, caractérisé par sa conception basée sur les SDS, en particulier des BVs et WTs [39] qui sont basées sur des structures RRR, et en s'appuyant sur un seul index. Ces fonctionnalités permettent d'avoir un système de gestion de bases de données extrêmement compact où le traitement des requêtes est sans décompression, c'est-à-dire que les requêtes SPARQL traduites sont traitées en n'opérant que sur des SDS compressées sans nécessiter de décompression. Ces deux caractéristiques, c'est-à-dire l'empreinte mémoire réduite et le traitement optimisé des requêtes, sont pertinentes pour l'informatique à la périphérie. Des évaluations précédentes ont souligné que SE offre de meilleures performances de requêtage et des services de raisonnement plus efficaces (avec certaines extensions RDFS), une taille de stockage plus réduite et un temps de création plus rapide par rapport à ses concurrents directs. L'un des inconvénients de l'utilisation de structures RRR est que la modification de ces graphes est un processus long car il nécessite une reconstruction complète des WTs et des BVs. La manière dont SE stocke les triplets RDF correspond à un entrelacement de BV et de WT pour chaque terme d'un triplet.

La Figure 2.6 représente l'architecture de SuccinctEdge, on y remarque son approche du stockage de triplets avec trois WTs et deux BVs en ordre PSO. Cet ordre d'entrelacement des structures représente l'index unique de la solution de stockage de SE et est motivé par la pertinence de l'accès pour les requêtes de détection d'anomalies dans un contexte IoT. Autrement dit, les BGP des requêtes exécutées en continu sur un client SE ont tous des constantes, c'est-à-dire des IRI, à la position prédicat, tandis que les sujets et les objets peuvent être des variables ou des constantes.

Une implémentation du système existe, écrite en C++ et dont les SDS sont basées sur la librairie sdsl-lite.

2.4 Edge computing

L'Edge computing, ou informatique en périphérie, est un concept consistant à effectuer des traitements de données et des opérations informatiques directement au niveau du périphérique ou du point le plus proche de l'utilisateur ou du capteur. Ceci est en opposition à l'approche qui centralise toutes les opérations dans un centre de données (data center) distant. En d'autres termes, les

13. https ://github.com/xwq610728213/SuccinctEdge

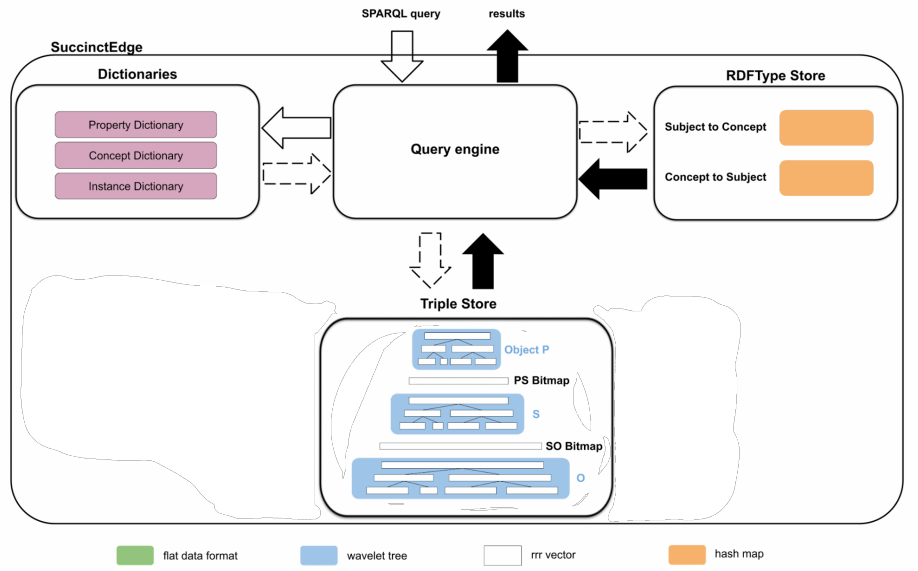

FIGURE 2.6 – Composants de SuccinctEdge

calculs sont effectués localement, à la "périphérie" du réseau. Comme illustré sur la Figure 2.7 l'informatique en périphérie est à distinguer du fog computing et du cloud computing. Les caractéristiques des machines évoluent dans cette pyramide. Ainsi, plus l'on descend sur cette pyramide et moins les machines individuelles ont de capacité de mémoire, stockage, calcul et plus le nombre de machines différentes augmente. Ce modèle est souvent utilisé pour réduire la latence, c'est-à-dire le délai entre l'envoi d'une demande et la réception d'une réponse. Cela est particulièrement important pour les applications en temps réel, telles que les systèmes embarqués, les applications médicales ou la détection d'anomalie.

En distribuant les traitements sur différents points de la périphérie du réseau, cela soulage également la charge des centres de données centraux, permettant ainsi de mieux gérer les pics de trafic et d'améliorer la résilience du réseau. Les capacités des machines utilisées dans le cadre de l'informatique de périphérie peuvent énormément varier, allant de micro-contrôleurs à peine capable de collecter et transmettre les données à des micro-ordinateurs, telles que des Raspberry Pi ou des Arduino capables de traiter des données avec des applications légères. L'informatique en périphérie est rendue possible grâce à l'évolution des technologies de l'IoT, et des réseaux de communication, qui permettent aux périphériques de plus en plus petits d'effectuer des tâches de calcul plus avancées. Cette approche hybride, combinant les capacités de traitement des appareils locaux et les ressources du Cloud computing, permet le développement d'applications plus robustes et efficaces.

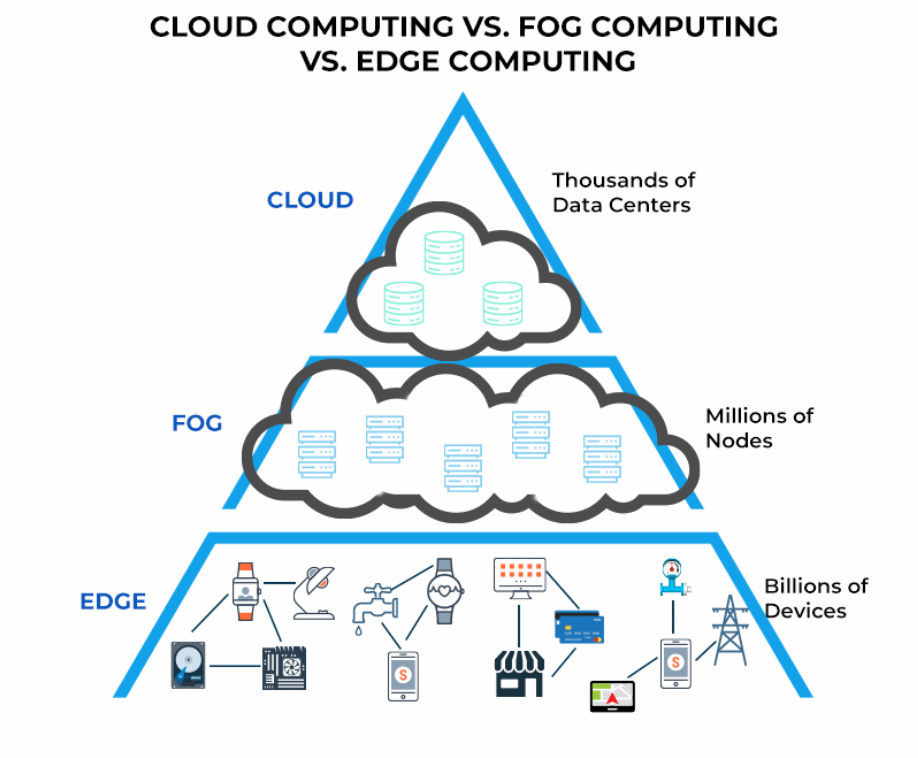

FIGURE 2.7 – Architecture cloud, fog et edge

2.5 Outils légers pour l'échange de flux de données

Nous présentons maintenant deux implémentations open source compatibles avec le Edge computing.

2.5.1 Apache Edgent

Apache Edgent [14] est un modèle de programmation de traitement de flux et un runtime léger pour exécuter des analyses sur des appareils en périphérie. Il se présente sous la forme d'une bibliothèque Java open source. Celle-ci permet d'effectuer des opérations analytiques sur des flux de données provenant de plusieurs appareils tout en laissant une empreinte mémoire réduite. Edgent permet de transformer facilement les données en fournissant des fenêtres, des flux et des opérations standards sur des fonctions d'agrégation telles que compter, calculer la moyenne, la somme, le minimum et le maximum d'un ensemble d'éléments. Des transformations plus complexes sont également possibles via la définition de fonctions par l'utilisateur. Le projet Edgent est rentré dans l'incubateur de

14. https ://edgent.incubator.apache.org/

l'Apache Software Foundation (ASF) fin 2016 et a été abandonné par l'ASF fin 2020. Certaines de nos expérimentations ont été conduites avec Edgent durant cette période d'activité du projet. Dès lors que nous avons appris son abandon par l'ASF, nous avons adopté le système Mosquitto que nous décrivons ci-dessous.

2.5.2 Eclipse Mosquitto

MQTT [15] (Message Queuing Telemetry Transport) est un protocole de messagerie léger et efficace conçu pour envoyer des données entre des appareils ou des applications dans un modèle de publication/souscription ou publish/subscribe en anglais. Il repose sur deux types d'acteurs principaux : les éditeurs (publishers) et les abonnés (subscribers). Les éditeurs sont responsables d'envoyer des données (messages) à un sujet (topic) spécifique, tandis que les abonnés expriment leur intérêt à recevoir des données à partir de sujets spécifiques. Les sujets agissent comme des canaux ou des sujets auxquels les messages sont publiés. Le protocole MQTT est adapté aux environnements où la bande passante et la puissance de calcul sont limitées.

Eclipse Mosquitto[31] est un agent de messages léger qui implémente MQTT (Message Queuing Telemetry Transport), un protocole de messagerie de publication-abonnement qui utilise généralement la suite de protocoles TCP/IP. Mosquitto a été conçu pour fonctionner sur des appareils ayant des ressources limitées, tels que des capteurs, c'est-à-dire gérer plus de 1000 clients avec moins de 3 Mo de RAM. Par conséquent, il convient particulièrement à l'informatique en périphérie (Edge computing).

15. https ://mqtt.org/

Chapitre 3

Informatique à la périphérie et traitement en continu de graphes de connaissances

Dans ce chapitre, nous présentons un système de gestion de graphe de connaissances (KG) conçu pour s'exécuter sur des terminaux que l'on retrouve dans l'informatique à la périphérie. Nous considérons que ceux-ci émettent des flux de données à des fréquences inférieures à la seconde. Lors de la phase de

conception, nous avons pris en compte les limitations inhérentes de ces terminaux, à savoir la puissance de calcul et l'espace de stockage limités, ainsi que les attentes des applications, telles qu'une faible latence, un haut débit et une gestion intelligente des données, i.e., capacité à raisonner. Il en résulte un SGBD RDF en mémoire, compact, sans compression, qui gère des flux de données, prend en charge des requêtes continues et certaines formes de raisonnement. Le système aborde le traitement efficace des requêtes sur des données arrivant en continu à un rythme élevé et est bien adapté aux applications événementielles telles que la détection d'anomalies et de risques. Nous mettons en évidence de manière empirique ses propriétés de robustesse, de latence et de débit dans un environnement IoT réel provenant du domaine de la gestion de l'énergie.

3.1 Introduction

Ces dernières années, les applications informatiques s'exécutant sur le Edge Computing ont suscité un vif intérêt[2]. Ce paradigme de calcul distribué complète l'informatique du Cloud computing, parfois qualifié d'informatique dans les nuages, en rapprochant le traitement, la gestion et le stockage de grands ensembles de données là où ils sont le plus nécessaires. Les principaux avantages de cette approche sont de réduire les temps de réponse et d'économiser la bande passante du réseau en évitant que l'analyse des données ne se produise dans le Cloud. Ces analyses soutiennent généralement des applications événementielles qui incluent la détection d'anomalies et de risques dans des domaines allant, par exemple de la surveillance des bâtiments, d'usines ou des véhicules à la détection de fraudes financières.

Potentiellement, l'informatique "intelligente" à la périphérie, c'est-à-dire impliquant des services de raisonnement s'exécutant sur les terminaux du Edge computing, peut (i) faciliter la configuration d'environnements IoT, (ii) déduire des conséquences implicites depuis des connaissances explicitement représentées, (iii) fournir des informations de contexte et des explications sur les inférences. Pour garantir un tel comportement, un système de gestion de KG est nécessaire sur les appareils rencontrés dans le Edge computing. Les propriétés attendues de tels systèmes sont de garantir (i) une empreinte mémoire réduite du système de gestion des données ainsi que des données et connaissances traitées, et (ii) un traitement efficace des requêtes et des inférences.

Récemment, nous avons conçu SuccinctEdge[51] qui peut être caractérisé comme un système fédéré basé sur les événements [13] car il intègre et interroge les données provenant de multiples capteurs dans un client SuccinctEdge. De plus, ces clients envoient certaines données et métadonnées à un serveur SuccinctEdge lorsque des anomalies ont été localement, c'est-à-dire sur la machine du client, détectées. L'ensemble du système peut être défini comme une base de données en mémoire, compacte, sans compression et ne comportant qu'un unique index pour le modèle de données de graphe RDF. Nous avons satisfait aux exigences susmentionnées en utilisant des SDS et en développant des algorithmes d'optimisation spécialisés. Dans ce chapitre, nous présentons des extensions qui

permettent le traitement de données non bornées émises par plusieurs capteurs. En interrogeant en continu un flux de données RDF, notre système est capable de prendre en charge des applications événementielles nécessitant des inférences.

Pour garantir des performances optimales en termes de faible latence (le temps entre le début et la fin d'un événement) et de haut débit (la quantité totale de travail effectué dans un laps de temps donné), notre extension s'appuie sur le système open source Eclipse Mosquitto. Ce courtier de messages léger convient aux appareils que l'on retrouve dans le Edge computing et offre un ensemble de fonctionnalités essentielles pour une plateforme de traitement de flux. Nous pouvons donc nous concentrer sur des tâches telles que le traitement et l'optimisation des requêtes, en prenant en charge plusieurs modèles de streaming, de stratégies de fenêtrage et de raisonnement. Concrètement, nos contributions consistent en : (i) une solution pour détecter les risques et les anomalies à partir de données temporelles via l'interrogation de graphes RDF non bornés, (ii) la prise en charge de différents modèles de traitement de flux, c'est-à-dire un événement à la fois ou bien par paquet d'événements (approche généralement dénotée micro-batch), et les stratégies de fenêtrage, c'est-à-dire des fenêtres qualifiées de sautantes (tumbling) et glissantes (sliding), (iii) une réécriture des composants d'exécution des requêtes de SuccinctEdge : prise en charge d'une extension SPARQL continue, une nouvelle approche d'optimisation de requêtes activée par le raisonnement basé sur la distinction entre les portions statiques et dynamiques des requêtes continues, et (iv) une évaluation approfondie des dimensions d'exactitude, de robustesse, de latence et de débit dans un scénario réel utilisé auprès de notre partenaire spécialiste du domaine de l'énergie (e.g., gaz, électricité, eau).

Dans la section 3.2, nous introduisons le contexte sémantique d'un cas d'utilisation concret qui nous servira de fil rouge dans ce chapitre. La section 3.3 présente notre système Streaming SuccinctEdge. La section 3.4 donne quelques compléments sur des travaux connexes. Nous évaluons notre système dans la section 3.5, mettons en évidence quelques enseignements et concluons le chapitre dans la section 3.6.

3.2 Contexte sémantique de l'exemple fil rouge

L'exemple fil rouge de ce chapitre est basé sur un cas d'utilisation fréquent dans la détection d'anomalies basée sur des capteurs. Il se base sur un scénario réel de notre partenaire ENGIE et concerne la distribution d'énergie.

La Figure 3.1 présente un extrait d'un graphe traité par notre extension streaming de SuccinctEdge. Il contient quelques mesures liées à la distribution de certaines matières premières, par exemple l'eau et le gaz, dans un bâtiment. Étant donné un tel graphe, notre système exécute des requêtes qui peuvent détecter certains schémas d'anomalies, par exemple des fuites dans le réseau de distribution.

Dans nos expérimentations chez ENGIE, nous avons constaté que plusieurs types et marques de capteurs sont fréquemment utilisés pour observer des me-

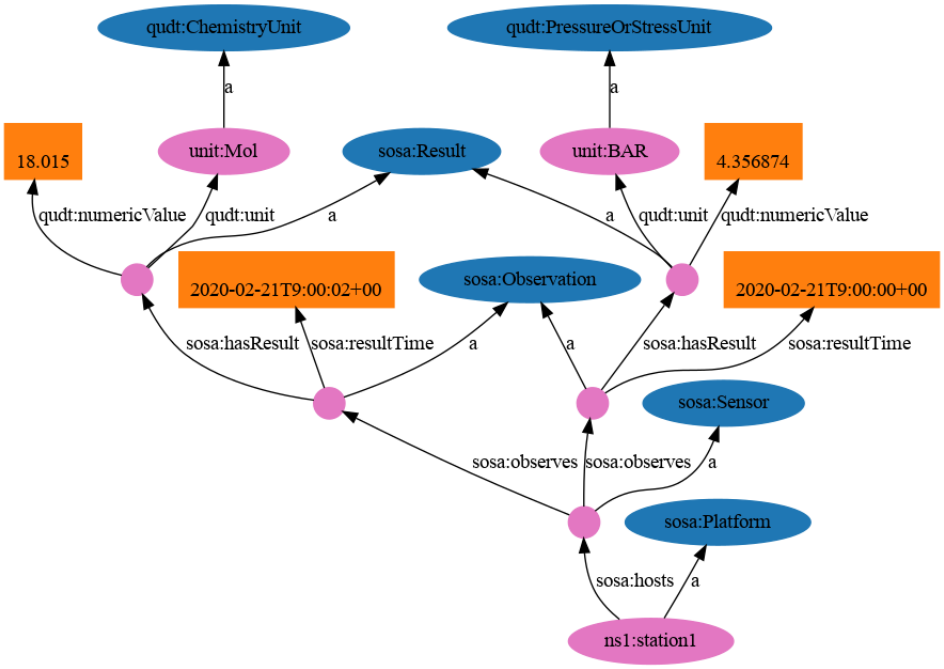

FIGURE 3.1 – Extrait de graphe de notre cas d'utilisation

sures similaires, par exemple la pression et le débit. Ces capteurs peuvent également produire des mesures exprimées dans des unités différentes, par exemple $Bar, Pascal, psi, Torr$ pour les mesures de pression ; $ft^3/min, gal/min, m/min$, m^3/h pour le volume par unité de temps. Dans ce contexte, il est nécessaire d'intégrer toutes les informations récupérées dans un seul système d'information.

La capacité d'associer des concepts et des propriétés de KG aux mesures produites par ces capteurs est la première étape de cette intégration sémantique. De plus, les experts du domaine définissent généralement des concepts de propriétés observables, par exemple $AtmosphericTemperature$, qui peuvent être organisés en hiérarchies et peuvent donc être utilisés à des fins de raisonnement. Les inférences RDFS sont efficacement traitées dans SuccinctEdge, grâce à notre approche d'encodage LiteMat et de réécriture des requêtes.

Une deuxième étape d'intégration sémantique consiste à faciliter l'écriture des requêtes SPARQL en transformant automatiquement les requêtes en fonction des caractéristiques d'un capteur donné, par exemple en fonction des annotations de concepts et des unités utilisées. Pour répondre à ces besoins, nous encourageons les experts du domaine à exprimer les requêtes en termes de concepts relativement élevés. L'expression d'une requête avec des concepts abstraits, c'est-à-dire se situant haut dans la hiérarchie des concepts, permet d'écrire une seule requête qui peut s'appliquer à des capteurs effectuant des mesures similaires mais annotés avec des concepts et des unités de mesure différents. Ainsi,

les experts du domaine n'ont pas à se soucier des inférences puisque celles-ci sont gérées automatiquement par le système. La simplicité de cette approche était attendue chez ENGIE pour des raisons de productivité. En effet, elle permet aux administrateurs de la plateforme et experts du domaine de se concentrer sur leurs tâches spécifiques, e.g. gestion des ressources de calcul et de stockage, sélection/maintenance des ontologies, plutôt que d'adapter une requête donnée à un nombre potentiellement élevé de capteurs dans un environnement industriel.

Considérons deux plates-formes de capteurs. La première correspond à celle décrite dans la Figure 3.1, où la pression est de type $qudt : PressureOrStressUnit$ et est exprimée en unité de Bar. Dans la deuxième plate-forme, une mesure de pression similaire est de type $qudt : PressureUnit$ et est exprimée en unité de HectoPA. Étant donné que l'ontologie QUDT [1] indique que

$qudt : PressureOrStressUnit \sqsubseteq qudt : PressureUnit$,

une seule requête SPARQL (détaillée dans la Section 3.3.3) peut être écrite pour tenir compte de la spécificité de chaque capteur dans ces plates-formes.

3.3 Streaming SuccinctEdge

Dans cette section, nous présentons l'extension de SuccinctEdge pour la gestion de flux de données. Cette évolution du système implique une adaptation des composants relatifs au stockage des données et de traitement des requêtes.

3.3.1 Flux de données au sein de l'application

Le contexte de ce cas d'utilisation correspond à des bâtiments où des capteurs sont installés et émettent en continu diverses mesures. Les données des capteurs sont continuellement analysées pour détecter des anomalies et/ou identifier des situations à risque, par exemple des fuites de gaz et d'eau, une surconsommation d'énergie.

La figure 3.2 présente le flux de données typique de cette configuration IoT. Dans une première étape, dénotée (1), nous considérons l'installation physique d'un nouveau capteur dans le bâtiment. À l'étape 2, les responsables de l'installation, appelés les "IoT Persons", déclarent aux administrateurs de la plateforme le schéma associé aux mesures récupérées à partir de ce dispositif, ainsi que certaines métadonnées, par exemple la marque du capteur, son emplacement dans le bâtiment, la date de sa dernière calibration. Notez que cette approche s'applique également lorsqu'un capteur existant est remplacé. Par conséquent, nous considérons qu'il n'est pas possible qu'un capteur soit remplacé sans que les administrateurs en soient informés.

Les administrateurs demandent ensuite à une équipe d'experts du domaine de définir les correspondances entre les éléments du schéma et une représentation sémantique du domaine. Le schéma peut être décrit dans un format tel que CSV alors que la représentation sémantique suivra le modèle de données RDF avec un langage d'ontologie tel que RDFS ou OWL. Dans cette approche, nous

1. https ://qudt.org/

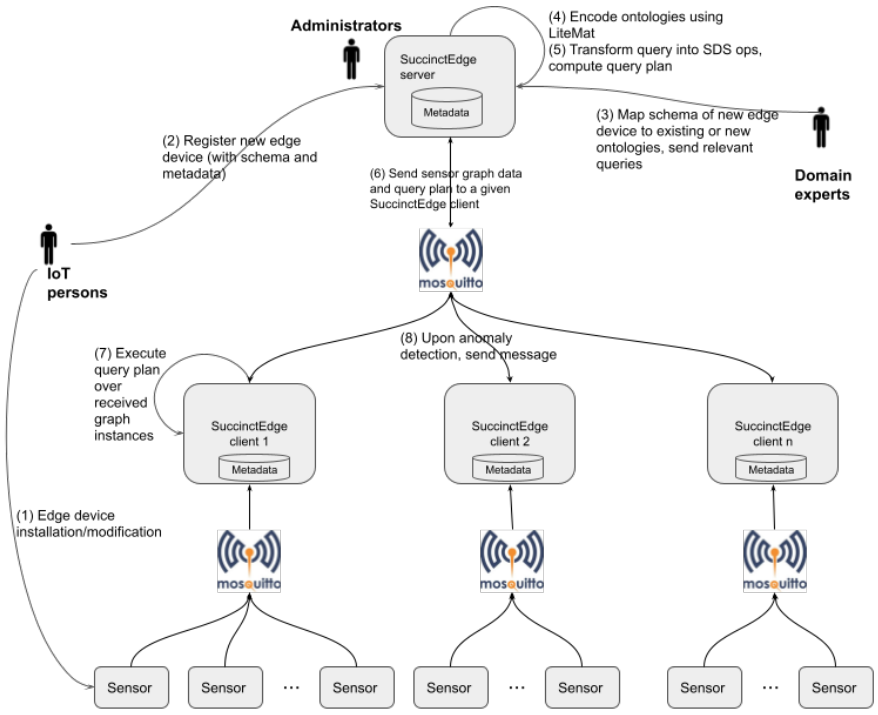

FIGURE 3.2 – Flux de données pour la mise en place d'une plateforme Streaming SuccinctEdge

bénéficions d'un vaste ensemble d'ontologies disponibles pour l'annotation des domaines de l'IoT et de capteurs, par exemple Sensor, Observation, Sample, Actuator (SOSA [2]), Quantities, Units, Dimensions, and Types (QUDT [3]) ou Smart Appliances Reference (SAREF [4]). La représentation sémantique de ces données incite donc à utiliser le modèle de données RDF. Chez ENGIE, l'utilisation de ces ontologies a simplifié la tâche de description, de manipulation et de connexion des capteurs et actionneurs. Les experts du domaine définissent également des requêtes pertinentes (en utilisant SPARQL), c'est-à-dire celles permettant la détection d'anomalies et de risques (étape 3).

En utilisant LiteMat[18], introduit en Section 2.1, les administrateurs peuvent encoder les ontologies requises par ce nouveau graphe (étape 4) et transformer les requêtes SPARQL en requêtes optimisées exprimées en termes d'opérations SDS (étape 5). Ces plans de requêtes physiques sont envoyés au client SuccinctEdge approprié (étape 6). Ces requêtes sont exécutées en continu lors de la réception de messages des capteurs (étape 7). Ces requêtes ne peuvent être modifiées que sur demande des administrateurs, par exemple lors du remplacement du capteur connecté. En cas de détection d'une anomalie par une requête, un message est envoyé du client SuccinctEdge au serveur SuccinctEdge avec des informations du contexte telles que les identifiants du dispositif et de la requête, les données anormales et l'heure de l'événement de l'anomalie (étape 8). Pour effectuer certaines de ces tâches, SuccinctEdge conserve des métadonnées, par exemple les associations requête/capteur, instance du client/capteur, client Mosquitto/instance du client.

3.3.2 Architecture

Comme indiqué dans la Section 2.3.6, le stockage du graphe RDF dans un SE client correspond à une séquence de WT et de BM. SuccinctEdge adopte une approche d'unique indexation basée sur la permutation PSO. Cela signifie qu'une seule copie des triplets RDF est stockée dans le système. Compte tenu de notre approche d'indexation PSO, nous distinguons les propriétés de type de données, c'est-à-dire lorsque l'objet est un littéral, et les propriétés d'objet, c'est-à-dire lorsque l'objet n'est pas un littéral mais une IRI. Dans la plupart des cas d'utilisation que nous avons rencontrés, les relations entre les instances dans les graphes RDF changent rarement car elles représentent les connexions entre des objets physiques, par exemple des plates-formes, des capteurs. Nous pouvons donc représenter tous les triplets contenant une propriété d'objet (sauf *rdf :type* pour lequel un stockage spécial est proposé) avec une combinaison de Wavelet Trees (WT) et de vecteurs de bits (BV). Intuitivement, chaque prédicat, sujet, ensemble d'objets est stocké sous forme de WT (resp. WT_p, WT_s et WT_o) et deux BVs relient respectivement le WT_p à WT_s et WT_s à WT_o.

Pendant ce temps, les données générées par les capteurs, c'est-à-dire les mesures numériques, qui sont des objets de certaines propriétés de type de données,

2. http ://www.w3.org/TR/ns/sosa
3. http ://qudt.org/schema/qudt
4. https ://ontology.tno.nl/saref.ttl

changent continuellement. Un taux de mise à jour élevé n'est pas adapté à un stockage par WT pour deux raisons suivantes : (i) chaque valeur d'objet nécessiterait un identifiant unique, mais cela n'est pas raisonnable car ces valeurs sont principalement numériques et donc potentiellement infinies, (ii) la mise à jour d'un WT ne peut pas être effectuée efficacement.

La Figure 3.3 montre les détails des structures de données utilisées pour les triplets contenant une propriété de type de données. Les propriétés et les sujets sont stockés comme dans la partie des propriétés d'objet, c'est-à-dire avec deux WTs et un BV. Un BV relie le WT du sujet à une couche d'objet, notée O. Dans cette couche, chaque objet représente une donnée dynamique qui est horodatée et fréquemment ajoutée. Le système stocke un pointeur pour chaque objet qui pointe vers une structure de type file. Lorsque de nouvelles données arrivent, nous les ajoutons au début de la file. Ces structures de type file possèdent certaines fonctions auxiliaires pour optimiser les opérations d'agrégation (par exemple MIN, MAX, AVG, SUM, COUNT) présentes dans une requête. La fonction correspondante est activée à la demande du système, par exemple en fonction de la sémantique de flux (voir la prochaine sous-section). Lorsque nous exécutons un motif de triplet (TP) avec une propriété de type de données, nous recherchons l'intervalle d'index des objets en utilisant les WTs et les BVs des deux premières couches, puis pour chaque objet dans cet intervalle, nous prenons la valeur dans sa file de données correspondante et calculons la fonction indiquée dans la requête.

Exemple : transfert de données de deux capteurs différents

Nous considérons le transfert de données de deux capteurs différents (S1 et S2) comme illustré dans la Figure 3.3. Nous supposons que chaque capteur mesure une seule valeur, c'est-à-dire S1V et S2V. S1T (resp. S2T) représente le timestamp reçu de S1 (resp. S2) et S1V (resp. S2V) fait référence à la mesure de S1 (resp. S2). Par conséquent, SuccinctEdge a distribué une structure de type file pour chaque série de données.

Même si les capteurs peuvent envoyer des messages à des fréquences différentes, SuccinctEdge peut gérer cette situation grâce à une implémentation de structure de type map pour distribuer chaque objet de type de données à un ensemble de capteurs correspondants. Grâce à cette approche, nous conservons facilement toutes les séquences de données provenant d'un capteur dans la même structure de données contiguë. De plus, les fenêtres glissantes et les fenêtres temporelles imposent le maintien de curseurs sur ces structures.

Dans notre système Streaming SuccinctEdge, Mosquitto prend en charge l'échange de données entre le serveur et le client SuccinctEdge, ainsi qu'entre le client SuccinctEdge et les capteurs (voir la figure 3.2).

3.3.3 Extension SPARQL continue

Plusieurs projets ont étendu le langage de requête SPARQL pour prendre en charge l'interrogation continue des flux de données RDF. En tant qu'approche

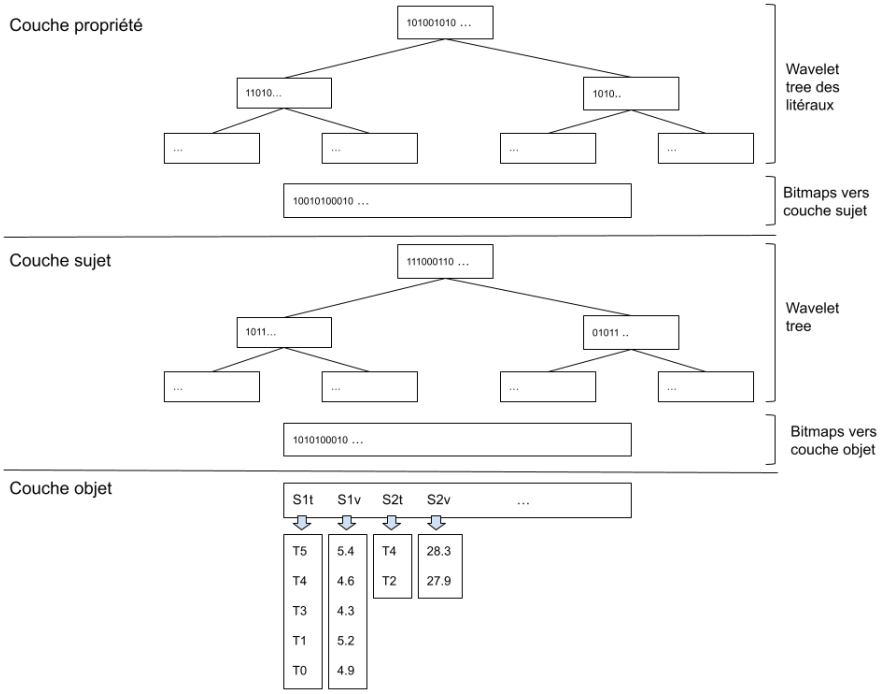

FIGURE 3.3 – Structure des propriétés de type de données avec deux WT et deux BM

bien établie, la syntaxe et la sémantique de C-SPARQL [9] ont influencé notre propre extension SPARQL.

Comparé à la syntaxe de C-SPARQL, nous prenons actuellement uniquement en charge les fenêtres logiques basées sur le temps, et notre bloc de description RANGE, spécifiant la durée de la fenêtre logique, apparaît dans la clause SELECT.

Dans le contexte de nos expériences chez ENGIE, nous n'avons pas rencontré de cas d'utilisation nécessitant une fenêtre physique basée sur le comptage des triplets. De plus, nous n'avons pas encore rencontré de situation où des graphes nommés sont nécessaires. Ainsi, la clause FROM n'est pas prise en charge dans notre extension SPARQL pour le moment. Cependant, nous prévoyons de résoudre cette limitation dans la prochaine version de SuccinctEdge. En plus des fenêtres temporelles, nous prenons également en charge les fenêtres glissantes, voir la Section 3.3.5, en utilisant le mot-clé STEP pour spécifier la fréquence de glissement de la fenêtre. Nous pouvons également associer une opération d'agrégation à chaque variable de requête. Une autre extension que nous avons apportée est la possibilité d'indiquer une fonction d'agrégation à calculer pour chaque variable de type de données, par exemple MAX, MIN et AVG.

La figure 3.4 présente une requête de détection d'anomalie de pression phy-

```
SELECT ?x ?s ?ts ?newV[Max] [RANGE 5000ms TUMBLING] WHERE {
    ?x rdf:type sosa:Platform; sosa:hosts ?s.
    ?s rdf:type sosa:Sensor; sosa:observes ?o.
    ?o rdf:type sosa:Observation;
    sosa:resultTime ?ts;
    sosa:hasResult ?y.
    ?y rdf:type sosa:Result;
    qudt:unit ?u, qudt:numericValue ?v.
    ?u rdf:type qudt:PressureOrStressUnit.
    FILTER (?newV<3.00 || ?newV>4.50)
    BIND(if(regex(str(?u), "http://qudt.org/vocab/unit/BAR"),?v,
    if(regex(str(?u),"http://qudt.org/vocab/unit/HectoPA"),?v/1000,0)) as ?newV)
}
```

FIGURE 3.4 – Physical pressure anomaly detection query

sique. Cette requête détecte les anomalies liées à une valeur de pression incorrecte (exprimée en bar ou en hectopascals) pour les capteurs des stations 1 et 2. Nous observons que dans la clause RANGE, une fenêtre temporelle de 5 secondes est spécifiée. De plus, la variable numérique ?v1 est suivie d'une instruction [MAX] qui indique que, pour chaque liaison de ?v1 dans l'ensemble de résultats, nous prenons la valeur maximale dans la fenêtre de données. La clause FILTER détecte les anomalies, et la clause BIND effectue certaines transformations de données (entre les unités de mesure bar et hectopascal).

De telles requêtes sont exécutées en continu sur certains clients SuccinctEdge. Chaque fois que cette requête renvoie un résultat, ses variables distinctes, c'est-à-dire les liaisons de variables de la clause SELECT, sont envoyées à un serveur SuccinctEdge pour alerter sur une éventuelle anomalie. Notez que le message envoyé au serveur est également enrichi avec des métadonnées client telles que l'identification du client SuccinctEdge et de la requête.

3.3.4 Traitement des requêtes

Le processeur de requêtes décrit dans [51] a été étendu avec une décomposition du motif de graphe de base (BGP) d'une requête. La motivation de ce nouveau processeur de requêtes est double et repose sur une observation des paramètres IoT du monde réel.

Premièrement, les requêtes continues analysant les données en flux sont généralement hautement sélectives, c'est-à-dire qu'elles renvoient des ensembles de réponses dont la taille est relativement réduites, d'environ quelques dizaines de tuples, et elles récupèrent à la fois des informations statiques, telles que les identifiants des capteurs et des entités, et des informations dynamiques, telles que l'analyse des mesures récentes et leurs horodatages. De plus, un capteur peut produire un ensemble de mesures correspondant à différents types d'informations, par exemple la pression, le débit, le pH, etc. Mais la plupart du

temps, une seule valeur est produite par type d'information dans une sortie de capteur donnée. Notez que cela a également été observé dans d'autres contextes industriels, par exemple le projet FUI (Fonds unique Interministériel) Waves [27].

Deuxièmement, le graphe sémantique, c'est-à-dire la TBox, associé à un capteur change rarement, sauf si le capteur est remplacé. Comme indiqué dans la Section 3.2, le remplacement d'un capteur dans notre écosystème IoT est nécessairement notifié à l'équipe d'administration de la plateforme. A la suite de cela, les étapes 3 à 6 de la figure 3.2 s'exécutent normalement. Cette approche de flux des données au sein de l'application empêche notre contexte sémantique de devenir incohérent avec notre configuration IoT.

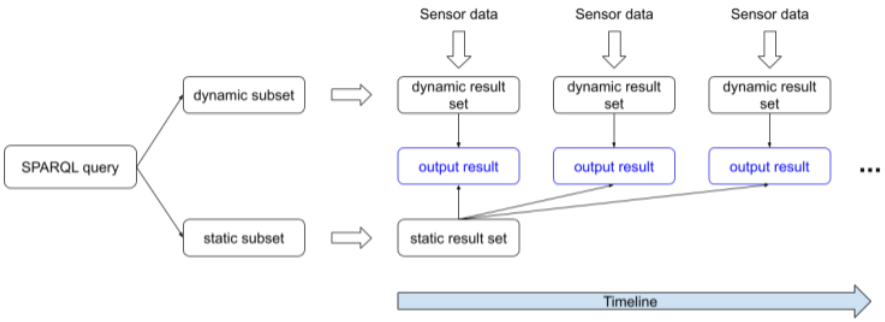

FIGURE 3.5 – Traitement statique et dynamique des requêtes

En tenant compte de ces deux observations, notre nouveau processeur de requêtes distingue un sous-ensemble statique et un sous-ensemble dynamique du BGP. Intuitivement, lorsqu'un capteur produit ses premières mesures, l'intégralité du BGP de la requête continue est exécutée et les liaisons des variables distinguées de la partie statique du BGP sont mises en cache par SuccinctEdge. Notez que cette exécution de requête peut impliquer une forme de raisonnement (RDFS) qui est gérée par la fonctionnalité de réécriture de LiteMat. Cette réécriture traite principalement les inférences de hiérarchie de concepts et de propriétés.

Ensuite, pour les mesures successives produites par ce même capteur, seule la partie dynamique du BGP doit être exécutée et intégrée dans l'ensemble de résultats de la requête (voir Figure 3.5). Les variables dynamiques distinguées correspondent à des objets de propriétés de type de données et n'influencent donc pas la correspondance du motif de graphe d'une requête car elles correspondent aux feuilles du graphe. L'exécution de ce sous-ensemble dynamique du BGP peut nécessiter le calcul de fonctions d'agrégation et certaines transformations de données, par exemple la conversion d'une pression de bar en pascal.

Le principe de conception de notre composant de traitement de requêtes est de tirer parti de cet aspect et de calculer un plan physique une seule fois pour

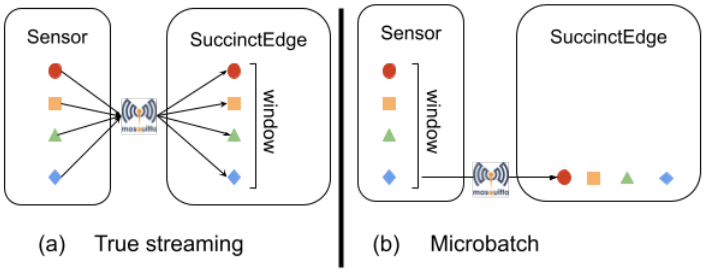

FIGURE 3.6 – Modes d'échange supportés

une requête donnée. Cela améliore considérablement l'exécution de la requête, car dans un contexte de traitement de requêtes continues, une requête peut être calculée un nombre indéfini de fois. Cette approche rappelle le concept de requête paramétrée, c'est-à-dire une requête précompilée qui n'a besoin que de certains paramètres pour compléter son exécution. Dans notre contexte de flux, les paramètres correspondent à la partie dynamique du BGP, c'est-à-dire les objets associés aux propriétés de type de données (y compris les horodatages ou les mesures) ou le résultat de l'application d'une fonction d'agrégation. Étant donné que le sous-ensemble statique du BGP est mis en cache pour optimiser son exécution, nous devons ré-assembler les deux parties lorsqu'une anomalie est trouvée avant de la signaler. Cela est effectué en recherchant à travers une série de structures de données de hachage où la clé correspond à une paire de requête et d'identification de capteur.

Dans la requête de la Figure 3.4, la partie statique mise en cache correspond aux variables `?x` et `?s`, respectivement les IRIs de la plateforme et du capteur, tandis que la partie dynamique correspond aux variables `?ts` et `?v1[MAX]`, respectivement l'horodatage de l'événement et la valeur maximale de pression d'une mesure de pression dont la valeur maximale dépasse un certain seuil.

Nous avons vu dans la Figure 3.2 que le calcul du plan physique est effectué sur un serveur SuccinctEdge qui dispose généralement de plus de ressources, en termes de CPU et de mémoire, qu'un appareil Edge, par exemple un Raspberry Pi, où s'exécute un client SuccinctEdge. L'optimiseur de requêtes de SuccinctEdge combine des heuristiques avec une approche basée sur les coûts. Les statistiques de cette dernière sont stockées dans les dictionnaires de LiteMat qui restent sur la machine exécutant le serveur SuccinctEdge. Le reste de l'évaluation de la requête est effectué sur un client SuccinctEdge.

3.3.5 Modes d'échange de flux de données

Streaming SuccinctEdge prend en charge deux modes d'échange de flux de données. Dans la Figure 3.6, les événements de données sont représentés sous forme de formes. Dans le mode de streaming véritable, chaque capteur est immédiatement envoyé à son client SuccinctEdge, via Eclipse Mosquitto. L'avantage

principal ici est de limiter la latence des données. Cependant, cela se traduit par un débit de données faible et des coûts réseau élevés en raison des échanges fréquents de données entre le capteur et Eclipse Mosquitto.

Dans le mode de micro-batch, le capteur conserve un certain nombre d'événements, correspondant généralement à la durée de la fenêtre temporelle de la requête. Une fois la limite de cette fenêtre atteinte, l'ensemble complet de données est envoyé au client SuccinctEdge, également via Eclipse Mosquitto. Ce mode limite le nombre d'échanges de données sur le réseau, mais augmente la latence des données.

Ces deux modes peuvent fonctionner selon les stratégies de fenêtres glissantes et de fenêtres fixes, qui sont toutes deux prises en charge par SuccinctEdge. Les deux stratégies de fenêtrage ont une durée fixe et un intervalle de glissement. Dans le cas des fenêtres fixes, l'intervalle de glissement est égal à la durée fixe, tandis que dans les fenêtres glissantes, l'intervalle de glissement est différent de la durée fixe. Lorsque l'intervalle de glissement est inférieur à la durée fixe, les données contenues dans une fenêtre se chevauchent avec la fenêtre précédente. SuccinctEdge propose des optimisations pour gérer intelligemment ce chevauchement. Dans le cadre de nos expérimentations chez ENGIE, nous n'avons pas rencontré de situation où un intervalle de glissement supérieur à la durée fixe était pertinent. En fait, cette dernière situation implique que certaines données de capteur ne sont pas traitées.

3.3.6 Apache Edgent

Une implémentation reposant sur Apache Edgent pour les communications entre senseurs et clients a d'abord été testée. Edgent permet d'accéder aux données senseurs avec un modèle un modèle publish/subscribe. La librairie permet aussi de traiter les agrégations de données directement au niveau des capteurs. Cela avait pour avantage de ne transmettre que le résultat de l'opération plutôt que toutes les données de la fenêtre avant d'effectuer l'opération sur le client. Un autre avantage est que si plusieurs clients ont besoin d'effectuer la même agrégation de données sur le même senseur, le résultat pourra être réutilisé plutôt que recalculé.

Cependant plusieurs raisons nous ont poussé à abandonner Edgent au profit de Mosquitto. Bien qu'Edgent ai été développée pour l'informatique en périphérie, le système était implanté sur la machine virtuelle java (JVM) et aurait eu besoin de machines avec plus de ressources que celles normalement utilisées pour des senseurs.

Dans nos expériences nous avons comparé le temps d'exécution de requêtes d'agrégations avec calcul au niveau du senseur avec le temps d'exécution de ces mêmes requêtes avec calcul au niveau du client. Le résultat était qu'avec les tailles de fenêtres de nos cas d'usages, quelques dizaines de secondes à quelques minutes, il n'y avait pas de différences majeures dans les temps de traitement des requêtes entre Edgent et Mosquitto. Nous avons aussi déterminé que le temps de calcul des agrégations était trivial. Il y a également le fait que le projet Edgent n'est plus maintenu par Apache. Ces raisons nous ont poussés à utiliser

Mosquitto, basé sur une librairie plus légère que Edgent et écrite en C, lui aussi avec un modèle publish/subscribe.

3.4 Travaux connexes

RDF4Led [29] est un SGBD RDF qui est détaillé dans la Section 2.3.6. Cela est partiellement motivé par un composant d'exécution de requêtes basé sur plusieurs index. Outre le fait d'être moins efficace en termes d'exécution de requêtes et d'empreinte mémoire par rapport à SuccinctEdge (voir [51] pour plus de détails), RDF4Led ne prend pas en charge les services de raisonnement et certaines clauses SPARQL qui permettraient une réécriture de requête efficace, par exemple la clause UNION. De plus, RDF4Led n'est pas doté de capacités de traitement de flux. Il n'est donc pas capable d'exécuter des requêtes sur un ensemble de graphes RDF entrants.

Fed4Edge, système qui a été détaillé dans la Section 2.3.6, adopte le langage de requête continue CQELS-QL [44] basé sur SPARQL, tandis que nous utilisons C-SPARQL. Néanmoins, en termes de traitement de requêtes et de fonctionnalités de streaming, les deux systèmes sont assez comparables. Actuellement, Fed4Edge se concentre sur la fédération de requêtes SPARQL [5], tandis que Streaming SuccinctEdge s'est principalement axé sur l'optimisation des requêtes et le raisonnement. Néanmoins, en tant que système fédéré basé sur les événements, SuccinctEdge est également capable d'intégrer et de requêter des données émises par différents capteurs.

WaterFowl [16] est un SGBD RDF basé sur SDS détaillé dans la Section 2.3.6. En terme de stockage, WaterFowl ne prend pas en charge le traitement de flux, ne possède pas la conception de stockage d'objets et l'optimisation du traitement de requêtes de Streaming SuccinctEdge. Enfin, WaterFowl n'est pas équipé de fonctionnalités de streaming, telles que les requêtes SPARQL continues.

Les systèmes existants de traitement centralisé des flux RDF tels que C-SPARQL [9] et CQELS [28] existent depuis un certain temps. Chaque moteur propose sa propre extension de langage de requête continue (généralement basée sur la syntaxe SPARQL) pour interroger des triples annotés dans le temps. Ainsi, ils ont influencé notre extension de SPARQL pour le traitement continu des requêtes. Cependant, aucun de ces systèmes n'a été conçu pour s'exécuter sur un appareil Edge computing.

3.5 Évaluation

Dans cette section, nous évaluons notre système Streaming SuccinctEdge selon les dimensions suivantes : précision, robustesse, évolutivité, performances du traitement de requêtes, latence et débit. Cette évaluation est réalisée sur différents paramètres (fenêtres glissantes et fixes, scénarios d'anomalies) dans le cadre de données synthétiques et d'une étude de cas réelle dans l'un des bâtiments d'ENGIE.

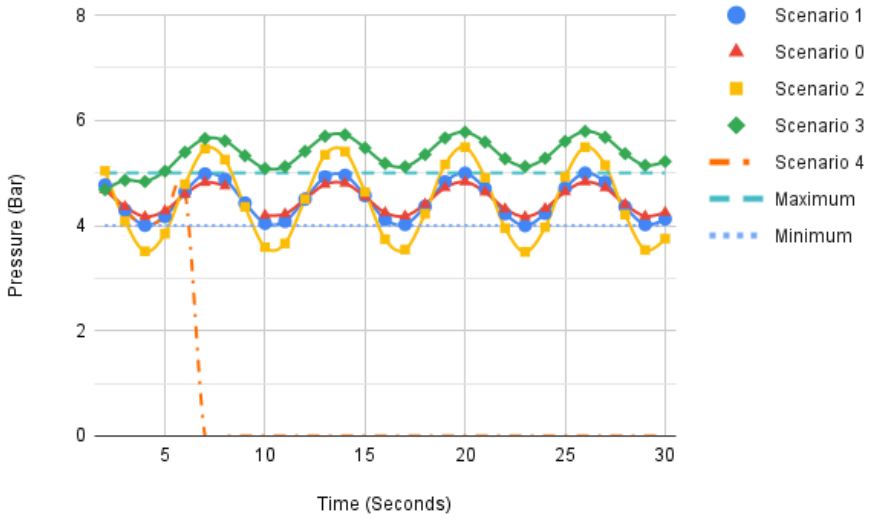

FIGURE 3.7 – Scénarios d'expérimentation sur les mesures de pression

3.5.1 Configuration expérimentale

Le contexte de cette évaluation est ancré dans notre exemple concret (Section 3.2), où des mesures réelles sont analysées dans un bâtiment de notre environnement de recherche. Pour des raisons de confidentialité, nous ne pouvons pas détailler la configuration complète du bâtiment. Néanmoins, nous pouvons indiquer que chaque étage de ce bâtiment de quatre étages est équipé d'un client SuccinctEdge connecté à huit capteurs. Ces capteurs peuvent capturer différentes mesures, telles que la température de la pièce ou la pression physique. L'ensemble des capteurs était hétérogène au moment de l'expérimentation, c'est-à-dire que certains capteurs émettaient des pressions en unité de bars, tandis que d'autres soumettaient des valeurs en hectopascals. Un seul serveur SuccinctEdge reçoit tous les messages des quatre clients SuccinctEdge.

Étant donné que nous ne pouvions pas provoquer d'anomalies dans ce contexte réel, il a principalement été utilisé pour évaluer la robustesse de notre prototype SuccinctEdge en streaming. En ce qui concerne la précision, les performances d'exécution des requêtes, la latence et le débit, nous devions contrôler l'apparition d'anomalies. Pour cette raison, nous avons également traité des données synthétiques caractérisant différentes formes d'anomalies.

Les appareils exécutant les clients SuccinctEdge sont des Raspberry Pi 3B+, équipés d'un processeur Cortex-A53 (ARMv7l) 32 bits à 1,4 GHz et de 1 Go de mémoire LPDDR2 SDRAM. SuccinctEdge est implémenté en C++ (version 14) et utilise la bibliothèque SDS-lite [5]. Eclipse Mosquitto (version 2.0) et le

5. https ://github.com/simongog/sdsl-lite

serveur SuccinctEdge fonctionnent à l'aide d'un JDK (version 8). Le serveur utilise également la bibliothèque Eclipse Paho Java [6] (une bibliothèque cliente MQTT). Les données sont transférées à l'aide du protocole MQTT, qui est un protocole de publication/abonnement basé sur TCP/IP. Dans tous les scénarios, nous utilisions un réseau Wi-Fi. Les détails de l'installation peuvent être trouvés sur GitHub [7].

Dans notre cas d'utilisation, les mesures normales appartiennent à l'intervalle [4,5] (en bars). Un grand ensemble de dimensions évaluées a été expérimenté sur 5 scénarios présentés dans la Figure 3.7. Ils diffèrent par la fréquence d'apparition des anomalies. Intuitivement, il n'y a pas d'anomalie dans le Scénario 0, car toutes les valeurs de pression se situent dans l'intervalle [4,5). Quelques anomalies surviennent régulièrement dans le Scénario 1, des séries d'anomalies sont suivies de mesures correctes dans le Scénario 2. Dans les scénarios 3 et 4, les mesures dérivent vers des états d'anomalie continus (perte totale de pression dans le Scénario 4).

3.5.2 Robustesse du système, précision et évolutivité des clients

En ce qui concerne la robustesse, nous avons réalisé nos tests sur la configuration de bâtiment décrite précédemment. Cette évaluation a été menée pendant une semaine sans aucune défaillance de notre plateforme SuccinctEdge, c'est-à-dire du client, du serveur et des instances Mosquitto.

Pendant cette semaine, nous avons observé cinq anomalies. Nous avons vérifié les données récupérées pour cette semaine avec des experts du domaine. Ils ont confirmé que seules cinq anomalies s'étaient produites pendant cette période. En termes de précision, c'est-à-dire de savoir si le système détecte toutes les anomalies qui se produisaient pendant notre expérimentation et s'il ne détecte pas de fausses anomalies, SuccinctEdge a identifié les cinq anomalies et uniquement ces anomalies. En fait, tant que la requête de détection d'anomalie est correcte, il n'y a aucune raison pour notre système de détecter une fausse anomalie ou d'en manquer une.

Bien sûr, un calibrage incorrect d'un capteur peut fausser notre détection d'anomalie, mais nous considérons que cela ne relève pas de la responsabilité du système SuccinctEdge. Nous prévoyons d'enrichir les fonctionnalités de notre serveur SuccinctEdge pour analyser les mesures historiques afin d'identifier les problèmes de calibrage liés à une dérive des données.

Une autre situation problématique survient si le capteur n'est pas en mesure d'envoyer des mesures à Mosquitto ou si l'instance Mosquitto entre un capteur et un client SuccinctEdge est hors service. Nous sommes en train de mettre en place une tolérance aux pannes dans SuccinctEdge afin d'identifier au moins ces situations. Il existe déjà une solution de surveillance du "heartbeat" dans MQTT qui permet de savoir si le client ou le serveur a perdu la connexion avec le courtier Mosquitto. Cependant, cela ne nous permet pas de surveiller l'état de nos clients et serveurs.

6. https ://www.eclipse.org/paho/index.php ?page=clients/java/index.php
7. https ://github.com/SuccinctEdge/SuccinctEdgePublic

Nous avons évalué notre système avec des données synthétiques pour nous assurer que SuccinctEdge détecte différents motifs d'anomalie, c'est-à-dire ceux de nos 5 scénarios présentés dans la Figure 3.7. Nous avons utilisé différentes stratégies de fenêtrage (fenêtres coulissantes et tumbling), des modes de communication en continu (véritable streaming et micro-batch) et des fenêtres de 1 à 60 secondes. Dans tous les cas, nous avons détecté toutes les anomalies et uniquement les véritables anomalies.

Les scénarios ont également été testés sur une configuration impliquant plusieurs capteurs communiquant avec un seul client Streaming SuccinctEdge, c'est-à-dire jusqu'à 40 capteurs, et avec différentes fréquences, c'est-à-dire avec deux ensembles de 20 capteurs envoyant des messages respectivement toutes les 200 et 300 ms. Le même taux de précision de 100% a été observé.

Enfin, nous avons évalué une plateforme de 40 capteurs avec des fenêtres coulissantes (de 5 secondes à 5 minutes), un streaming réel (de 5 minutes à 1 heure) pendant plus de 3 jours, sans aucune défaillance et avec la même précision.

3.5.3 Débit et utilisation du réseau

Nous commençons par évaluer, sur des données synthétiques, l'impact des échanges de données sur le réseau dans une expérimentation s'étalant sur 24 heures dans des cas relativement extrêmes : une anomalie toutes les demi-heures (Tableau 3.1) et une anomalie toutes les 2 heures (Tableau 3.2).

Lors de l'envoi d'un paquet Mosquitto, nous ne considérons que la taille de l'en-tête et la taille des données utiles (payload). La taille des données utiles est un multiple de 16 octets (8 octets pour la valeur et 8 octets pour le timestamp). Nous comparons différents modes de fonctionnement de SuccinctEdge : le streaming en temps réel et le mode micro-batch (désormais appelé batch) avec la détection d'anomalies pour chaque mode. Certaines détections d'anomalies dans SuccinctEdge sont implémentées à l'aide de la clause FILTER de SPARQL, elles peuvent donc être désactivées s'il n'y a pas de FILTER dans la requête. Ensuite, toutes les mesures des capteurs sont envoyées directement à un serveur SuccinctEdge.

En mode batch, SuccinctEdge envoie uniquement les données pertinentes une seule fois au serveur pour un intervalle de temps donné. Lorsque le mode de détection d'anomalies est activé, le client ne renvoie au serveur que les mesures identifiées comme des anomalies par la requête. Nous testons plusieurs tailles de batch : envoi de batch toutes les heures ou toutes les 2 heures. Chaque batch correspond à un seul paquet Mosquitto, en supposant que la taille des données utiles est inférieure à 256 Mo. Les capteurs produisent également une mesure par seconde.

Dans les tableaux 3.1 et 3.2, nous pouvons observer la charge du réseau pour transférer les données des capteurs sur une période de 24 heures, en utilisant différents modes. Dans le Tableau 3.1, nous détectons une anomalie toutes les demi-heures, tandis que dans le Tableau 3.2, nous avons une anomalie toutes les 2 heures.

Mode	Taille du batch	Taille des données	Mode	Taille du batch	Taille des données
Batch	1 heure	1,38 Mo	Batch	2 heures	1,38 Mo
Batch + détection	1 heure	240 octets	Batch + détection	2 heures	216 octets
Streaming	-	1,46 Mo	Streaming	-	1,46 Mo
Streaming + détection	-	288 octets	Streaming + détection	-	288 octets

TABLE 3.1 – Utilisation du réseau pour 1 capteur, avec 1 anomalie toutes les demi-heures (détection signifie que nous envoyons l'ensemble des résultats de la requête au serveur SuccinctEdge)

Mode	Taille du batch	Taille des données	Mode	Taille du batch	Taille des données
Batch	1 heure	1,38 Mo	Batch	2 heures	1,38 Mo
Batch + détection	1 heure	216 octets	Batch + détection	2 heures	216 octets
Streaming	-	1,46 Mo	Streaming	-	1,46 Mo
Streaming + détection	-	216 octets	Streaming + détection	-	216 octets

TABLE 3.2 – Utilisation du réseau pour 1 capteur, avec 1 anomalie toutes les 2 heures (détection signifie que nous envoyons l'ensemble des résultats de la requête au serveur SuccinctEdge)

De manière évidente, nous obtenons un débit plus élevé lorsque les données sont transférées en mode batch et lorsque nous avons plus d'une anomalie dans la fenêtre temporelle de ce batch. Le mode streaming est légèrement moins efficace, cependant il permet d'obtenir les données instantanément.

Lors de l'examen des propriétés de latence et de débit, nous ne prenons pas en compte l'exécution de la requête qui récupère et met en cache la partie statique de la requête, car cela est amorti par notre approche de traitement des requêtes. Nous ne considérons que l'impact de la réception des mesures successives, c'est-à-dire le calcul des fonctions d'agrégation, la détection des anomalies et l'intégration des parties dynamiques et statiques de la requête.

3.5.4 Temps d'exécution de la requête du client SuccinctEdge

Dans cette section, nous évaluons l'impact de l'ajout de nombreux capteurs sur un seul client SuccinctEdge. Chaque capteur envoie 100 mesures chaque seconde au client, et l'évaluation a été réalisée sur une plage de 1 à 40 capteurs. La Figure 3.8 met en évidence que, pour 100 mesures par seconde, un client SuccinctEdge peut gérer jusqu'à 10 capteurs sans retard dans le traitement de ses requêtes. Elle montre également qu'il pourrait gérer jusqu'à 40 capteurs s'ils n'envoyaient que 10 mesures par seconde. Par conséquent, dans un environnement où chaque capteur envoie une mesure toutes les 10 ms, un nouveau client SuccinctEdge est nécessaire tous les 10 capteurs environ. Compte tenu du coût d'un Raspberry Pi, cela ne constitue pas une limitation de notre solution de streaming globale. De plus, dans le contexte de la détection d'anomalies industrielles attendue dans des entreprises comme ENGIE, une fréquence de mesure de l'ordre des secondes est plus réaliste que des millisecondes.

Ces résultats sont rendus possibles uniquement grâce à l'optimisation du traitement du graphe de motifs de base (BGP) qui s'effectue en deux étapes. Les résultats de la Figure 3.8 représentent le temps d'exécution de la partie dyna-

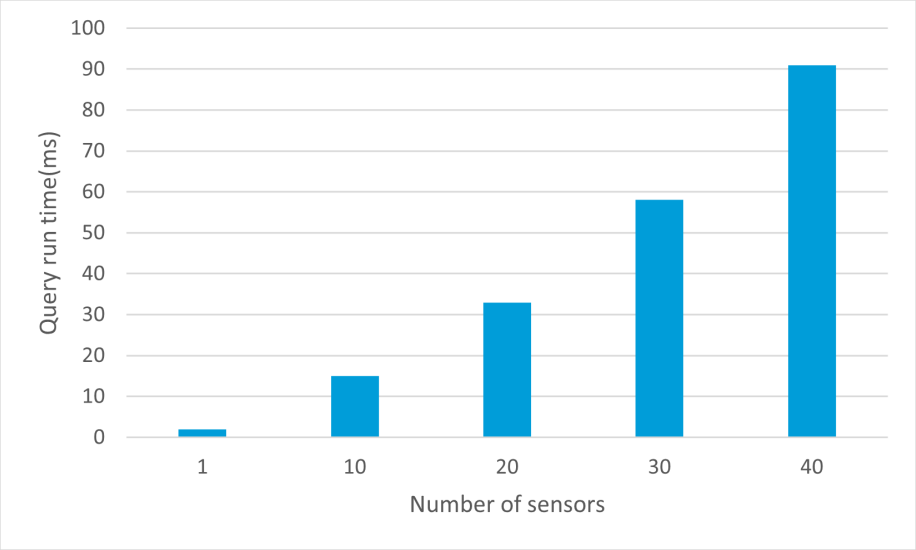

FIGURE 3.8 – Temps d'exécution moyen de la requête (partie dynamique) en streaming réel pour un client avec de 1 à 40 capteurs envoyant 100 mesures par seconde

Nombre de motifs triples	5	8	10	15
Temps (ms)	0,215	0,312	0,414	0,632

TABLE 3.3 – Temps nécessaire pour traiter la partie statique de la requête pour différents nombres de motifs triples

mique du BGP, tandis que le temps d'exécution de la partie statique est présenté dans le Tableau 3.3, variant en fonction du nombre de motifs triples contenus dans le BGP de la requête. On peut constater que la partie dynamique du traitement de la requête (en ms) domine la partie statique (en sous-millisecondes).

3.5.5 Robustesse et latence du serveur SuccinctEdge

Dans cette section, nous évaluons l'impact de l'ajout de nombreux clients sur un seul serveur SuccinctEdge. Comme nous n'avions pas suffisamment de Raspberry Pi pour cette expérimentation basée sur des données synthétiques, nous avons dû créer une image Docker [8] du client SuccinctEdge pour l'exécuter dans plusieurs conteneurs afin de tester les performances du serveur SuccinctEdge. Dans ce scénario, chaque client dispose de 5 capteurs qui envoient des données chaque seconde. Ces valeurs sont toujours considérées comme des anomalies et sont envoyées au serveur.

8. https ://www.docker.com/

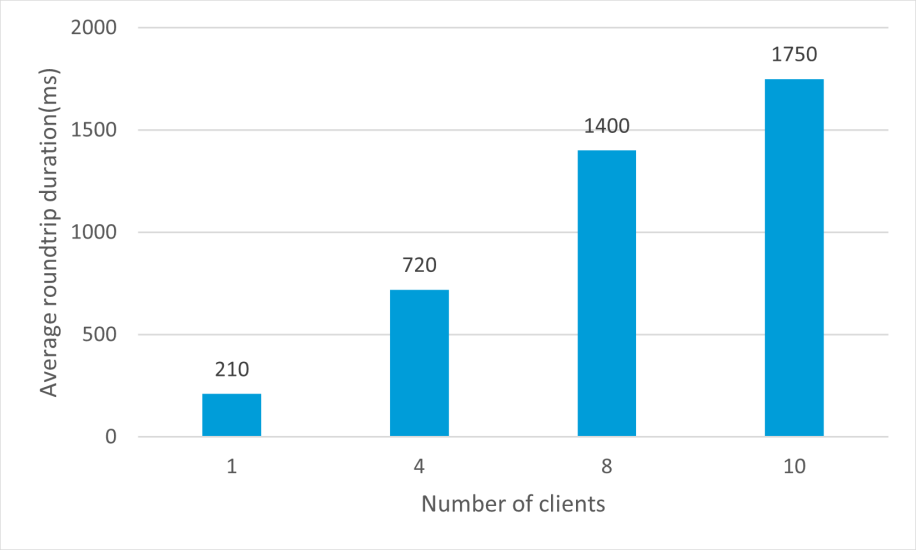

FIGURE 3.9 – Latence moyenne de l'aller-retour entre le client et le serveur SuccinctEdge pour un nombre de clients allant de 1 à 10

Nous mesurons la latence pour un aller-retour entre un client et le serveur SuccinctEdge, le serveur renvoyant un accusé de réception au client pour chaque valeur reçue. La latence augmente de manière linéaire lors de l'ajout de clients à un seul serveur, comme nous pouvons le voir sur la figure 3.9. Dans ce contexte d'expérimentation particulier, au-delà de 10 clients, le serveur est trop lent pour traiter les données à temps, ce qui entraîne une augmentation de la latence au fil du temps. Il convient de noter que ce contexte d'expérimentation est assez extrême, car il suppose que toutes les données reçues par un client sont des anomalies. Par conséquent, dans un environnement réel, il est préférable de consulter les experts du domaine pour connaître les fréquences d'anomalies moyennes et maximales afin de définir le nombre de clients par serveur. Il convient également de noter que, dans notre contexte de bâtiment ENGIE, un serveur SuccinctEdge était connecté à quatre clients.

Enfin, en termes de mesure de latence, nous considérons que la conteneurisation a également un impact négatif non négligeable.

3.6 Conclusion et travaux futurs

Dans ce chapitre, nous avons présenté une tentative originale de traiter les flux RDF à la périphérie d'une infrastructure informatique. Les principales caractéristiques de notre système sont la compacité et la capacité d'inférer des conséquences implicites en temps réel. Notre système, appelé Streaming SuccinctEdge, présente des propriétés pertinentes, telles que la faible latence et le

débit élevé, attendues dans un contexte de Big Data où les données sont produites à une grande vitesse.

L'exécution de nos expérimentations sur un cas d'utilisation réel de détection d'anomalies nous a permis de confirmer plusieurs idées acceptées. Tout d'abord, la latence imposée par une solution de micro-batch est suffisante dans la plupart des cas. En fait, la latence plus faible fournie par une approche du traitement d'un événement à la fois n'est pas vraiment attendue par les applications d'ENGIE : gagner quelques secondes n'est pas aussi important par rapport au temps nécessaire à une intervention humaine pour résoudre un problème, par exemple réparer une fuite dans un réseau de distribution.

Le réglage efficace du nombre de capteurs connectés à un seul client et du nombre de clients connectés à un seul serveur nécessite la connaissance des experts du domaine. Grâce à nos expérimentations, nous avons déjà découvert qu'il est possible de connecter jusqu'à 40 capteurs à un seul client SuccinctEdge et que, dans des cas extrêmes, jusqu'à 10 clients peuvent être connectés à un seul serveur. Ces caractéristiques correspondent totalement à l'attente de notre partenaire industriel.

De plus, sur une période de plus de 24 heures, notre système a trouvé de nombreuses valeurs émises par les capteurs similaires. Ceci peut par exemple correspondre à des valeurs stables de températures ou de pressions. Ce cas de figure peut être considéré comme une limitation de l'envoi de données en mode un événement à la fois car la même donnée sera alors envoyée et traitée plusieurs. Si cette valeur déclenche l'identification d'une anomalie alors nous pouvons considérer ces différentes valeurs comme une confirmation de l'anomalie. Sinon, cela correspond à un envie non optimisé de données. Lorsque nous accumulons ces données dans des micro-batchs, cela signifie qu'elles peuvent être compressées efficacement pour permettre la persistance des données même avec la capacité limitée du disque dur des appareils Edge. Cela pourrait être réalisé dans des travaux futurs par l'intégration d'Apache Parquet [9], un format de stockage en colonnes axé sur l'économie d'espace de stockage, dans Streaming SuccinctEdge.

Nous avons également confirmé que la plupart des requêtes soumises dans un contexte de détection d'anomalies sont assez sélectives et que l'ensemble des réponses a une cardinalité relativement faible. Parmi les applications orientées flux de données d'ENGIE, nous recherchons des cas d'utilisation nécessitant des requêtes avec une sélectivité plus faible.

Enfin, nous avons confirmé que les fenêtres basées sur le temps (par opposition aux fenêtres basées sur le compte ou les sessions) sont plus pertinentes dans la détection des risques/anomalies. De plus, nous n'avons pas encore trouvé de scénario où les fenêtres glissantes seraient plus judicieuses que les fenêtres sautantes lorsqu'elles sont exécutées sur un appareil Edge.

Une expérimentation approfondie sur les modèles de streaming les plus couramment utilisés (un événement à la fois et micro-batch) et les stratégies de fenêtrage (fenêtres glissantes et sautantes) a mis en évidence la précision, la robustesse et le passage à l'échelle du système. Dans un avenir proche, Strea-

9. https ://parquet.apache.org/

ming SuccinctEdge sera utilisé dans plusieurs bâtiments de notre partenaire énergétique. De plus, la communication et la coopération entre les clients SuccinctEdge sera intégrée et nous étudierons la capacité du système à réagir de manière autonome à des anomalies impliquant plusieurs clients SE différents.

Les contributions mentionnées dans ce chapitre ont été présentée sous forme d'un article à la conférence DEBS (Distributed Event-Based Systems) en 2022 [20].

Chapitre 4

Vers une gestion autonome de la détection d'anomalies à l'aide de technologies sémantiques dans un contexte de Edge computing

Dans ce chapitre, nous présentons une approche qui adapte de manière autonome la surveillance des capteurs d'un environnement IoT. Basée sur les technologies du Web sémantique, notre solution permet la génération de requêtes continues pertinentes lorsque certaines anomalies sont identifiées. La génération consiste en une extension du graphe de requête déclenchée lorsque certaines règles sont activées. Ces requêtes sont exécutées sur un système de base de données graphes, respectant le modèle de données RDF, conçu pour l'informatique en périphérie. Nous évaluons l'exactitude des requêtes générées, la robustesse et la latence de notre système dans un cas d'utilisation réel consistant en un contexte de bâtiment intelligent équipé de plusieurs capteurs.

4.1 Introduction

Le nombre d'appareils de l'IoT ne cesse d'augmenter, tout comme la quantité de données générées. Selon un rapport récent de l'International Data Corporation (IDC) [1], cette tendance va se poursuivre et même s'amplifier dans les prochaines années. Ainsi, certaines prédictions de l'IDC estiment qu'en 2025, il y aura un total de 55,7 milliards d'appareils IoT en utilisation, capables de générer environ 80 zettaoctets de données. Les mesures générées par ces capteurs et actionneurs ne sont généralement pas indépendantes les unes des autres, c'est-à-dire que la décision prise à partir de l'analyse des mesures d'un capteur peut avoir un impact sur les actions à effectuer sur ce même capteur ou bien un autre capteur du contexte. Par exemple, dans l'agriculture, le processus par lequel les plantes convertissent la lumière du soleil en énergie chimique est fortement influencé par la température et l'humidité. Les capteurs qui transmettent ces mesures ne doivent pas nécessairement fonctionner 24 heures sur 24 et 7 jours sur 7. Par conséquent, certaines mesures de température peuvent obliger à prendre des mesures d'humidité dans les serres agricoles.

À l'échelle de l'IoT, nous considérons qu'il est essentiel de proposer une gestion autonome et adaptable (GAA) des interactions entre capteurs. Un tel système GAA doit faire partie d'un système de gestion des connaissances générique, c'est-à-dire capable de répondre à des requêtes, effecteur des raisonnements. De plus, pour assurer la réactivité et l'efficacité, une part importante de la GAA doit être réalisée aussi proche que possible des appareils en périphérie.

Ces aspects d'autonomie et d'adaptabilité sont importants pour les futures applications IoT. En effet, ils permettront une gestion efficace des ressources énergétiques (par exemple, les batteries des capteurs), du réseau informatique (c'est-à-dire utilisé pour l'échange de données) et des appareils en périphérie qui interrogent et analysent les données émises. Pour motiver cette position, nous considérons la gestion des capteurs dans un bâtiment moderne. Depuis la pandémie de Covid-19, nous avons vu de plus en plus de capteurs installés dans des bureaux, halls, etc. Ces capteurs sont capables de mesurer la concentration de dioxyde de carbone (CO2) mais peuvent également fournir des informations

1. https ://is.gd/BxpMDb

physiques telles que la température, l'humidité et la pression, pour n'en citer que quelques-unes. Afin de garantir un séjour en toute sécurité à l'intérieur de ces bâtiments, une surveillance et une supervision globales de ces mesures sont essentielles. Par exemple, cela peut sensibiliser à la ventilation des pièces en fonction de la concentration de CO2. Une approche naïve, c'est-à-dire récupérer en continu des mesures de l'ensemble des capteurs, entraînerait une consommation excessive d'énergie. De plus, elle empêcherait les capteurs de fonctionner pendant une longue période, car ils fonctionnent généralement sur batterie ou panneaux solaires. Cela surchargerait également le réseau et la puissance de calcul chargée d'analyser ces informations. Enfin, cela pourrait également augmenter la latence pour identifier une situation à risque.

Dans le cadre que nous venons de décrire, un système GAA dépend de l'interprétation de l'impact qu'une situation physique alarmante peut avoir sur d'autres capteurs. Par exemple, ventiler une pièce pendant un certain temps peut réduire la concentration de CO2, mais cela peut également abaisser la température de la pièce à un niveau inconfortable pour ses occupants ou augmenter la pollution sonore au point que les interactions humaines ne puissent plus avoir lieu dans cette pièce. Dans un tel cas, la GAA doit générer des requêtes pour surveiller la température de l'air et le volume sonore si la concentration de CO2 dépasse un seuil prédéfini. Cette interprétation nécessite une approche basée sur la sémantique qui sera capable de déduire des conséquences implicites à partir de faits et de connaissances explicites. La gestion de cette forme d'autonomie nécessite la génération de la supervision automatique de certaines mesures connexes.

Les solutions proposées précédemment envoient en continu les données des capteurs vers un serveur sur site (on-premise) ou bien dans le cloud. C'est approche présente certaines limites car transmettre les mesures 24 heures sur 24 et 7 jours sur 7 à une fréquence potentiellement élevée peut ne pas être nécessaire pour certaines surveillances du système. De plus, les systèmes existants sont généralement le résultat d'une implémentation informatique classique, c'est-à-dire non déclarative, et ne peuvent pas s'adapter à des situations nouvelles sans modifier à nouveau le code source. La conception de notre système sur un KG du domaine d'application favorise l'adaptabilité et la capacité à raisonner sur le contexte des mesures. Notre approche se base sur notre RDF Store SuccinctEdge[51] que nous avons déjà décrit précédemment. En fait, nous avons étendu les composants client et serveur de SE pour les adapter aux situations déduites en exécutant des requêtes continues générées automatiquement qui seront exécutées sur les capteurs pertinents de l'infrastructure IoT. Notre extension est autonome, c'est-à-dire qu'elle ne nécessite pas l'intervention d'un être humain puisqu'elle repose sur l'interprétation des connaissances contenues dans les ontologies et les règles SHACL.

Ce chapitre est organisé comme suit. Dans la Section 4.2, nous présentons un exemple concret. Dans la Section 4.3, nous donnons un aperçu de l'extension de notre système. Dans la Section 4.4, nous comparons notre système avec des travaux connexes. Dans la Section 4.5, une expérimentation est réalisée sur notre nouveau système. Nous tirons quelques enseignements de la conception

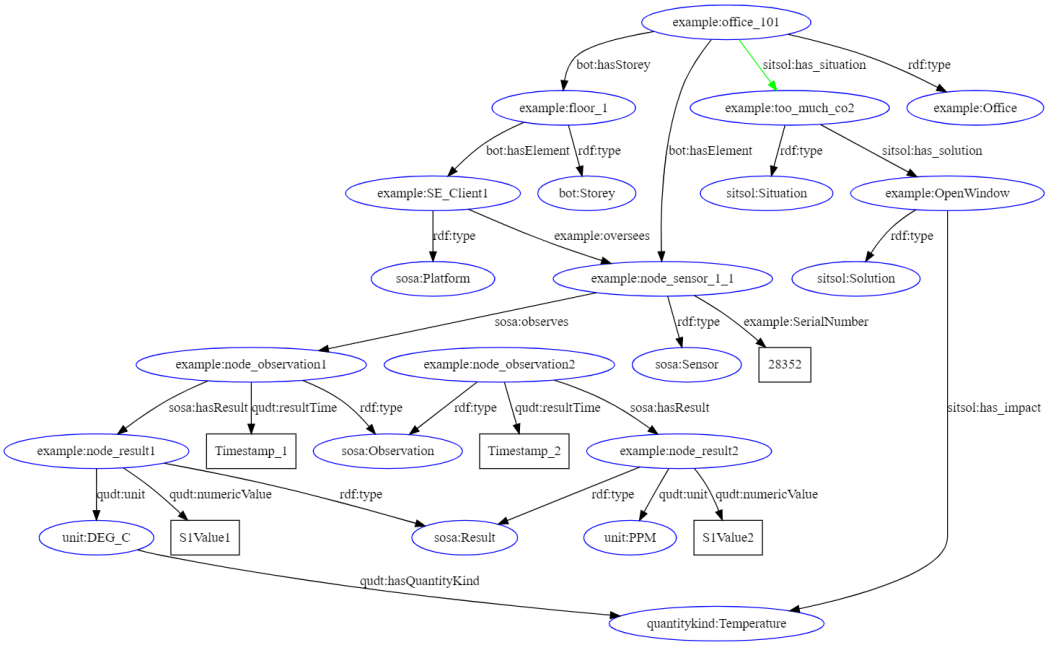

FIGURE 4.1 – Extraction de la base de connaissances relative à un bureau

de l'extension de notre SE dans la Section 4.6. Enfin, nous concluons dans la Section 5.6.

4.2 Exemple illustratif

Notre exemple illustratif est basé sur un cas d'utilisation fréquent dans la détection d'anomalies des capteurs. En fait, il est issu d'un scénario rencontré par ENGIE, notre partenaire sur ce projet. Différents capteurs captent en continu des mesures qui sont analysées pour détecter les anomalies et identifier les situations indésirables ou risquées. Pour ce faire, des capteurs sont installés dans des pièces (bureaux, salles de réunion, halls, etc.) et émettent des mesures avec une certaine fréquence. Ces capteurs sont reliés à un client SE qui est chargé d'identifier, sur la base de certaines mesures émises en continu, différentes situations anormales.

Avec cet exemple concret, nous nous concentrons sur le cas de la surveillance du CO2 car il était particulièrement important pendant la pandémie de Covid-19. La concentration de CO2 est un indicateur clé de la qualité de l'air et son contrôle peut contribuer à réduire le risque de transmission du Covid-19 à l'intérieur de bâtiments [10]. Ces mesures peuvent évoluer rapidement et les réponses doivent être tout aussi rapides pour être efficaces. Ainsi, pour disposer d'un système efficace, il doit avoir un degré d'autonomie lors de la réponse à

une anomalie. En cas de dépassement de la concentration de CO_2, les réponses peuvent consister à intensifier la ventilation de la pièce, par exemple en ouvrant une fenêtre, ou, s'il n'y a pas d'autres solutions, à demander aux occupants de quitter la pièce. Cependant, les réponses peuvent avoir des conséquences imprévues. Par exemple, ouvrir une fenêtre peut être une solution efficace pour un problème de qualité de l'air, mais cela peut également avoir un impact négatif sur la température de la pièce. Par conséquent, nous devons être en mesure de contrôler de manière autonome de nouvelles variables et d'interrompre des solutions si elles s'avèrent contre-productives. Lorsque les variables reviennent à la normale, le système devrait revenir à son état initial et cesser de contrôler inutilement certains capteurs.

4.3 Vue d'ensemble du système

Notre système est basé sur les technologies sémantiques du W3C, en particulier SHACL que nous n'avons pas utilisé jusqu'à maintenant dans le projet SuccinctEdge. Dans cette section, nous motivons et présentons les principaux composants de notre extension SE.

4.3.1 Composant Ontologie

Pour traiter de manière autonome les situations rencontrées dans n'importe quel domaine, un système doit être capable d'interpréter les données et connaissances qu'il manipule. Pour effectuer ce type de traitement, nous utilisons la TBox et la ABox d'une base de connaissances. Dans un contexte de détection d'anomalies basé sur l'analyse de données émises par des capteurs, un large ensemble de TBoxes de haute qualité a déjà été conçu par la communauté du Web sémantique et peut facilement être réutilisé. La Figure 4.1 met l'accent sur l'utilisation des différentes TBoxes que nous utilisons. Cela est facilement identifiable par les espaces de noms de certains concepts et propriétés utilisés dans cet extrait, à savoir SOSA et QDT (ce dernier est exploité en particulier pour ses unités de mesure et les Quantity Kinds (QK) qu'il regroupe), que nous avons déjà utilisées dans le chapitre précédent (Cf. 3.1, ainsi que BOT [2](Building Topology Ontology). Les experts du domaine que nous avons déjà évoqués dans la Figure 3.2 sont responsables de la spécification de la modélisation des bâtiments et de ses différentes pièces, des capteurs et de leurs mesures. La plupart des nœuds de cette Figure 4.1 correspondent à la ABox qui caractérise un exemple donné de l'exemple illustratif.

En ce qui concerne l'aspect GAA, nous avons conçu une ontologie destinée à décrire les anomalies et les solutions pour y faire face à un niveau d'abstraction élevé. L'ontologie, nommée Sitsol pour Situation Solution, repose sur plusieurs classes RDFS dont les principales sont : `Situation`, une classe utilisée pour définir un événement anormal qui doit être résolu ou qui attend une réponse, `Solution`, une classe utilisée pour définir des réponses aux situations,

2. https ://w3id.org/bot

qu'il s'agisse de résoudre la situation ou non, et `Impact`, une classe utilisée pour indiquer qu'une certaine métrique est affectée par une situation ou une solution et doit être vérifiée. Ces classes apparaissent comme le domaine ou la plage des propriétés suivantes de l'ontologie Sitsol : `has_Impact`, `has_Solution`, et `has_Situation`. Respectivement, ces propriétés indiquent qu'une instance de `Situation` ou de `Solution` a un impact sur quelque chose, qu'une `Situation` peut avoir une solution, et que toute entité peut être impliquée dans une situation donnée, c'est-à-dire une anomalie. La figure 4.1 illustre l'utilisation de cette ontologie dans le cas de notre exemple concret.

Les experts du domaine sont également responsables de l'expression de représentations sémantiques des réponses attendues du système avec les concepts `Situation`, `Solution` et `Impact` (qui seront expliqués dans la section suivante).

4.3.2 Composant SHACL

Dans ce composant du système nous utilisons SHACL [3], un langage de règle dont le fonctionnement est détaillé dans la Section 2.3.5, d'une manière originale, car il est utilisé pour interpréter les anomalies dans nos mesures de capteurs et pour déclarer les réponses à utiliser en conséquence.

Les experts du domaine spécifient un certain nombre de formes SHACL utilisées pour détecter les anomalies dans les mesures des capteurs. Nous utilisons ces formes pour déduire les solutions à appliquer et l'impact que ces solutions peuvent avoir. Lorsqu'une anomalie est détectée par un client SE, la mesure est envoyée au serveur SE où une situation peut être déduite et temporairement stockée dans le KG à l'aide d'un graphe nommé, qui ont été détaillés dans la sous-section 2.3.1 Dans ce cas, le système appliquera une solution et générera de manière autonome de nouvelles requêtes pour surveiller les métriques nouvellement impactées. Lorsqu'une situation est résolue, son graphe nommé peut être efficacement supprimé et l'exécution des requêtes générées sera arrêtée.

Dans le scénario illustré dans la figure 4.1, nous considérons un bureau équipé de deux capteurs, un capteur de CO_2 mesurant les PPM (parties par million) et un capteur de température mesurant les degrés Celsius. Une propriété `has_situation` a été déduite et marquée en vert dans la figure 4.1, reliant le bureau à la situation "TooMuchCO2" et nous permettant de savoir quelles solutions utiliser.

Nous commençons par rechercher les formes ayant des cibles existantes dans le graphe. Ces formes sont ensuite explorées pour trouver des contraintes de plage de valeurs qui peuvent être présentes dans ces formes. Ces types de contraintes sont utilisés pour déclarer les limites valides pour les données numériques, les quatre types de contraintes possibles sont :

http ://www.w3.org/ns/shacl#minInclusive
http ://www.w3.org/ns/shacl#maxInclusive
http ://www.w3.org/ns/shacl#minExclusive
http ://www.w3.org/ns/shacl#maxExclusive

3. https ://www.w3.org/TR/2017/REC-shacl-20170720/

```
example:TooMuchCO2Shape
 a sh:shape;
 sh:targetClass bot:Zone;
  sh:property [
   sh:path bot:hasElement;
   sh:property [
    sh:path sosa:observes;
    sh:property [
     sh:path sosa:hasResult;
     sh:property [
      sh:path qdt:numericValue;
      sh:maxInclusive 15;
      sh:rule [
       rdf:type sh:TripleRule;
       sh:object sh:focusNode;
       sh:predicate sitsol:has_situation;
       sh:subject example:tooMuchCO2;
];];];];]; .
```

Listing 4.1 – Office CO2 Shape

Nous utilisons le type des contraintes ainsi que leurs valeurs pour générer des filtres de requête SPARQL continus (à détailler dans la sous-section 4.3.3). Ces requêtes se voient attribuer un signal et sont envoyées aux clients SE pour être exécutées sur les mesures des capteurs connectés au client. La liste 4.1 présente la forme utilisée pour déduire la situation "tooMuchCO2" dans la figure 4.1. La `sh:rule` de cette forme a conduit à la déduction de la situation.

Il était techniquement possible de créer un système qui fonctionne exclusivement sur la base de l'ontologie et de son langage, sans jamais utiliser SHACL. Cependant, nous avons constaté que cela obligerait les utilisateurs à modifier le graphe chaque fois qu'ils souhaiteraient apporter une modification au comportement du système. Si cela ne concernait que l'extension du graphe, par exemple lorsqu'un nouveau capteur est ajouté à une pièce, cela ne poserait pas de problème. Cependant, cela signifierait également que toute modification de la température souhaitée dans une pièce devrait être représentée dans le graphe. Au lieu de cela, en utilisant SHACL pour déclarer le comportement du système, nous n'avons besoin que de lire à partir du graphe et d'ajouter les triplets que nous pouvons déduire à partir des formes. À l'exécution, le système peut traiter des centaines de ces triplets déduits et doit les ajouter ou les supprimer en continu. Cela rend l'efficacité de l'ajout et de la suppression de ces triplets déduits critiques. Étant donné que les triplets que nous déduisons à partir d'une anomalie et d'une forme sont toujours ajoutés et supprimés ensemble, nous utilisons des graphes nommés pour les regrouper et les gérer de manière plus efficace. Cela permet d'exécuter le système et de modifier son comportement sans jamais affecter le graphe d'origine.

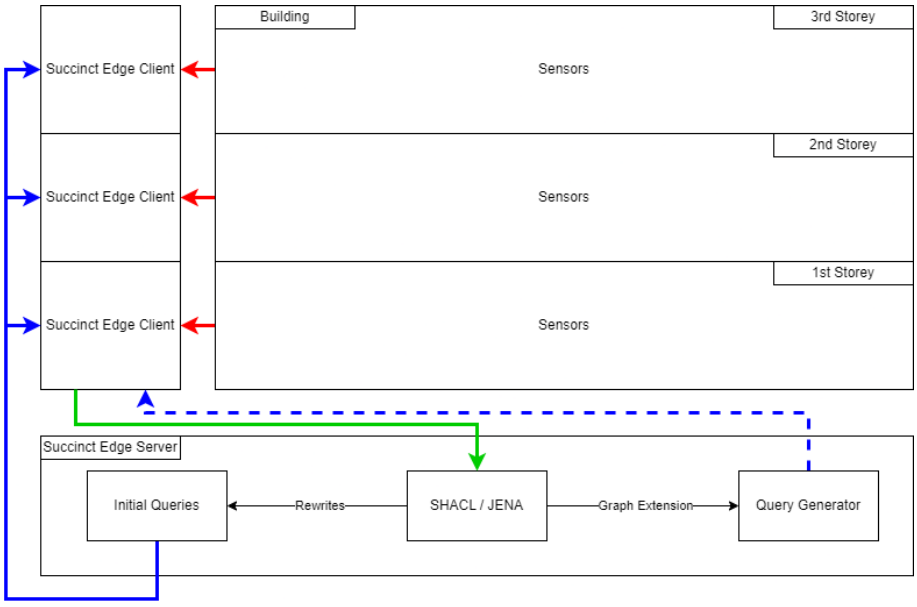

FIGURE 4.2 – Building running example, simplified Workflow

4.3.3 Évolution du client/serveur SuccinctEdge vers GAA

La version précédente de SE était principalement utilisée pour traiter des requêtes sur des périphériques Edge. Ces clients exécutent des requêtes qui, dans certains cas, détectent des anomalies. Ces anomalies sont ensuite envoyées à un serveur SE[20]. Avec notre nouvelle approche, un nouveau serveur a été écrit en Java en utilisant Jena. Le serveur distribue toujours une requête initiale, illustrée par la flèche bleue pleine dans la figure 4.2, fournie par l'utilisateur, aux clients SE. Cependant, lorsqu'une anomalie est détectée, le client envoie un signal au serveur qui déclenche une validation SHACL, illustrée par la flèche verte dans la figure 4.2. La validation SHACL génère une extension de graphe utilisée pour générer une nouvelle requête pour les clients, illustrée par la flèche bleue en pointillés dans la figure 4.2. Cette nouvelle requête peut également déclencher un signal de sorte que les requêtes peuvent être enchaînées pour revenir à une requête initiale ou pour avoir plusieurs requêtes s'adaptant à la situation.

Chaque fois qu'une requête est distribuée aux clients SE et pour chaque senseur qu'elle cible, la requête est associée à un signal unique. Un signal est un entier de 64 bits qui est envoyé avec la requête, pour le client ce signal sert alors à représenter un couple requête/senseur, tandis qu'il est stocké dans un dictionnaire signal requête/senseur sur le serveur. Lorsqu'une anomalie est détectée du côté du client, la valeur de l'anomalie et la valeur du signal sont renvoyées au serveur SE. Le serveur possède les métadonnées lui permettant également de savoir où les les mesures du senseur se trouvent dans le graphe.

Grâce à ces deux seules informations il est capable de recréer le triplet anomalie à son niveau et le faire valider par une Shape SHACL. Le serveur peut aussi récupérer la requête initiale et l'utiliser pour en générer une nouvelle.

Une fois qu'un signal a été utilisé pour étendre le graphe, il est oublié par le système, car toutes les requêtes s'exécutent en continu sur les clients et n'ont pas besoin d'être générées une seconde fois. Un signal est également associé à chaque requête générée, ce qui nous permet d'enchaîner automatiquement et d'adapter les requêtes en cours à la situation actuelle du système.

4.3.4 Génération de requêtes

Une fois que le graphe a été étendu à l'aide du composant SHACL, nous devons maintenant générer de nouvelles requêtes. Nous générons de nouvelles requêtes en nous basant sur les situations décrites dans l'extension du graphe, la même requête peut être envoyée à plusieurs clients en fonction de l'impact associé à la situation. Le processus de génération d'une seule requête se décompose en trois étapes : la génération de la clause SELECT, la génération des différents TPs du BGP utilisé dans la clause WHERE, et enfin la génération de la clause FILTER utilisée par le client pour détecter l'anomalie. La clause STREAMING spécifiant l'aspect continu de la requête est actuellement définie ad hoc en fonction de la situation.

Pour obtenir des données des capteurs situés dans la même pièce que la requête précédente adressait, nous devons générer des TPs qui interrogent le même contexte. Pour ce faire, pour chaque motif triple dans la requête précédente qui ne sélectionne pas un capteur et ses mesures, nous le récupérons et l'utilisons pour la nouvelle requête, par exemple, en surveillant uniquement les bureaux occupés.

La génération de la clause SELECT implique la création de variables pour les mesures des capteurs. Pour une pièce donnée d'un bâtiment, nous récupérons le QK pointé par la propriété `has_impact` de la solution concernée. Ensuite, nous récupérons l'identifiant du capteur qui surveille ce QK pour lier les variables de mesure aux valeurs récupérées par ce capteur. Par exemple, si la solution a un impact sur la température, nous récupérons l'identifiant du capteur qui se trouve dans la même pièce que la requête précédente et qui surveille la température pour obtenir ses mesures.

Pour la clause FILTER, nous utilisons des formes SHACL pour obtenir les valeurs qui définissent une anomalie. Par exemple, si une forme SHACL définit les températures acceptables dans un hall d'immeuble dans un intervalle de 18°C à 24°C, nous générons une clause de filtre (?Value $<$ 18 $||$?Value $>$ 24) pour filtrer les valeurs à l'intérieur de l'intervalle.

4.3.5 Précision de la requête générée

La précision des requêtes générées, c'est-à-dire le fait qu'elles s'adressent au bon capteur, client, pièce et unité, dépend uniquement de la correction de l'ABox. Les experts du domaine doivent définir l'emplacement des capteurs et

quel client gère quels capteurs. Cela est dû au fait que SuccinctEdge associe les clients et les capteurs dans la même pièce que la requête qui a déclenché la première anomalie. Le QK mesuré par la requête générée est défini par l'extension du graphe.

4.4 État de l'art

Le travail présenté dans [36] soutient que certaines applications détectent les anomalies en utilisant un raisonnement basé sur des règles pour vérifier que les exigences fonctionnelles de ces applications sont satisfaites par le bâtiment et ses dispositifs. Les inférences basées sur les règles reposent sur Semantic Web Rule Language (SWRL), qui est connu pour être indécidable dans le cas général. Comparé à notre approche, les règles SHACL sont moins expressives que SWRL, mais notre système bénéficie de sa facilité d'utilisation.

Dans [33], un cadre est présenté. Il repose sur des graphiques pour modéliser les relations entre les capteurs et les systèmes. Une ontologie de diagnostic est utilisée pour expliquer les anomalies lorsqu'elles sont détectées. Cette ontologie de diagnostic remplit principalement le même rôle que notre ontologie du système, mais nous utilisons également la nôtre pour générer de manière autonome des requêtes afin de gérer les impacts potentiels.

RDF4Led [29] est un SGBD RDF qui est détaillé dans la Section 2.3.6. À notre connaissance, aucune forme d'autonomie n'a été intégrée dans ce système. Son comportement dépend donc entièrement du code source de l'application.

4.5 Expérimentation

Dans cette section, nous évaluons les dimensions suivantes : calcul des règles SHACL, performances de génération de requêtes, évolutivité et latence du système SuccinctEdge étendu.

4.5.1 Paramètres expérimentaux

Le contexte de l'expérimentation est défini dans le cadre d'un bâtiment intelligent que nous avons décrit dans la Section 4.2. Les capteurs situés dans les pièces envoient des mesures aux clients SE. Chaque client est connecté à un seul serveur SE, et un serveur SE écoute plusieurs clients SE.

Dans chaque pièce gérée par le système, il y a au moins un capteur. Celui qui nous intéresse, à savoir Class'Air [4], émet notamment des concentrations de CO_2 en ppm et des températures de l'air en degrés Celsius.

Le serveur SuccinctEdge ne valide que les règles SHACL et génère des requêtes lorsqu'il détecte une anomalie. Nous avons dû simuler les capteurs pour contrôler leur fréquence et les valeurs qu'ils envoyaient aux clients afin de pouvoir déclencher des anomalies à volonté et évaluer les performances du système

4. https ://pyres.com/solutions/classair/

pour plusieurs anomalies reçues simultanément. La Shape SHACL la plus utilisé lors des évaluations est la Shape du listing 4.1. Un client SuccinctEdge s'exécute sur un Raspberry Pi 3B+, équipé d'un processeur Cortex-A53 (ARMv7l) 32 bits SoC 1,4 GHz et de 1 Go de LPDDR2 SDRAM.

Les clients de SuccinctEdge sont implémentés en C++ et utilisent la bibliothèque SDS-Lite [5], tandis que le serveur est implémenté en Java et utilise Apache Jena [6] pour traiter les formes et règles SHACL et gérer les KG. Le client et le serveur SE échangent des données à l'aide d'Eclipse Mosquitto [7], dont le fonctionnement a été détaillé dans la Section 2.5.2. Le code source de SE est disponible sur Github [8] [9]. Pour les raisons mentionnées dans le chapitre précédent, nous n'avons pas considéré Apache Edgent dans cette expérimentation.

Pour chaque expérience menée, chaque client envoie exactement une anomalie en même temps que les autres pour déclencher une validation SHACL par client sur le serveur. Nous considérons que les anomalies sont toujours vraies et que les capteurs sont toujours correctement calibrés, de sorte que nous devons traiter les données pour générer automatiquement une requête pour chaque anomalie.

4.5.2 Validation SHACL et extension du graphe

Tout d'abord, nous évaluons le temps de traitement de la validation SHACL avec les données reçues des capteurs. Pour ce faire, nous validons un nombre différent de règles en même temps pour une anomalie, et nous mesurons le temps nécessaire pour valider ces règles et étendre le KG avec des annotations de graphes nommés. Dans la Figure 4.3, nous nous concentrons sur le temps nécessaire pour valider des ensembles de règles SHACL de tailles différentes. Nous pouvons constater que l'évolution est sous-linéaire et que, pour un ensemble de quarante règles, la validation globale s'effectue dans une plage de 450 ms. Ces performances sont mises en contexte avec le processus de génération de requêtes dans la Figure 4.4. Nous pouvons voir que le temps de validation SHACL correspond seulement à une fraction du temps total de génération de requêtes, c'est-à-dire environ 15% du processus global de génération de requêtes (génération de requêtes + validation SHACL).

4.5.3 Traitement des anomalies côté serveur

Une fois que le graphe est étendu en utilisant le résultat de la validation SHACL, le système génère des requêtes. Ensuite, nous évaluons le temps nécessaire pour générer une requête pour chaque anomalie reçue. Dans cette expérimentation, chaque client envoie une seule anomalie au serveur, qui renvoie une nouvelle requête adaptée à chaque anomalie. La validation étend toujours

5. https ://github.com/simongog/sdsl-lite
6. https ://jena.apache.org/
7. https ://mosquitto.org/
8. https ://github.com/SuccinctEdge/AutonomousSEServer
9. https ://github.com/SuccinctEdge/AutonomousSEClient

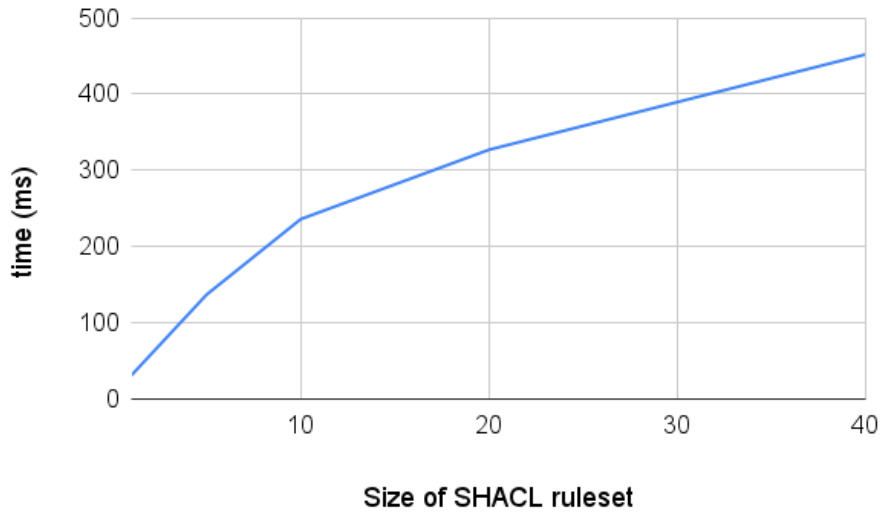

FIGURE 4.3 – Validation SHACL pour différentes tailles d'ensembles de règles

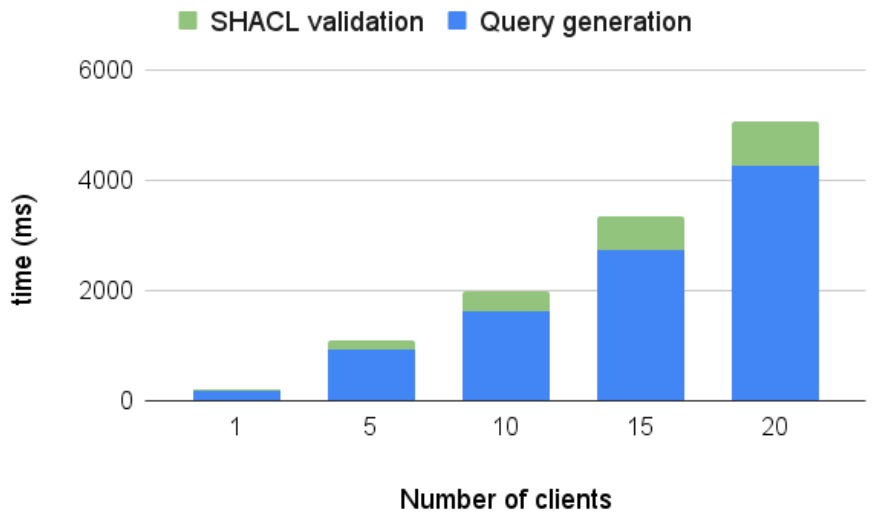

FIGURE 4.4 – Temps de validation SHACL et de génération de requêtes en fonction du nombre de clients et de 40 formes

```
PREFIX qudts: <http://qudt.org/schema/qudt/>
PREFIX qk: <http://qudt.org/vocab/quantitykind/>
PREFIX sosa: <http://www.w3.org/ns/sosa#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?Value WHERE {
  ?X rdf:type sosa:Observation
     ; sosa:hasResult ?Y .
  ?Y qudts:numericValue ?Value
     ; qudts:unit ?Unit .
  ?Unit qudts:hasQuantityKind
     qk:Temperature .
}
```

Listing 4.2 – Example de génération de requête

le graphe avec la même situation, c'est-à-dire que la concentration de CO_2 est trop élevée dans la pièce (supérieure à 800 ppm), et la requête générée est alors adaptée. Seuls deux motifs de triplets changent pour cibler le bon client et le bon capteur. Dans le Listing 4.2, nous affichons une requête générée de base pour cette expérimentation, sans les motifs de triplets pour se concentrer sur une pièce du bâtiment et le capteur.

Nous évaluons le temps total nécessaire pour générer une seule requête pour plusieurs clients et nous montrons les résultats dans la Figure 4.4. Comme vu précédemment, cette figure inclut également le temps nécessaire pour étendre le graphe en utilisant SHACL. Nous avons mesuré cette durée pour un ensemble de clients allant d'un à vingt et validé leurs mesures par rapport à quarante formes différentes. L'objectif est de vérifier la robustesse de notre valideur SHACL, c'est-à-dire sa capacité à fonctionner à une fréquence d'anomalie relativement élevée. La figure montre qu'avec quarante formes à valider, l'ensemble du processus de génération de requêtes est effectué en moins de 2 et 5 secondes respectivement pour dix et vingt clients. De plus, le valideur SHACL ne prend qu'une petite partie (environ 10%) du temps de génération de nos requêtes.

4.5.4 Nombre de requêtes à générer

Dans cette section, nous augmentons le nombre d'impacts déclenchés par situation. Pour chaque impact, nous devons générer une requête différente. Pour évaluer cela, nous n'utilisons qu'un seul client avec un seul serveur, mais nous dupliquons les impacts dans l'extension du graphe, de sorte que nous devons générer une requête par impact. Nous évaluons le temps nécessaire pour générer un nombre différent de requêtes. Nous pouvons constater que cela a un impact sous-linéaire, car le système peut réutiliser des parties de la requête précédente pour chaque nouvelle requête qu'il doit générer. La Figure 4.5 démontre l'efficacité d'avoir plusieurs impacts sur une solution.

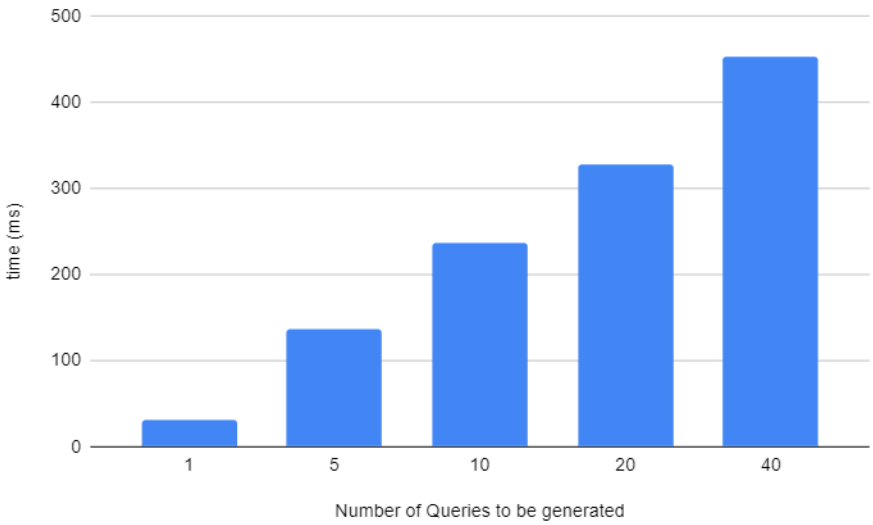

FIGURE 4.5 – Temps de génération de requêtes pour différents nombres d'impacts

4.5.5 Taille du BGP des requêtes générées

Nous évaluons maintenant les performances du système lorsqu'il doit s'adresser à plus d'une pièce du bâtiment et donc à plusieurs capteurs spécifiques. Nous ajoutons plus de motifs de triplets au BGP pour évaluer le temps nécessaire pour générer la requête. Nous utilisons ces différentes propriétés pour augmenter progressivement la taille du BGP :

- `Example:lastMaintenance`
- `Example:installationDate`
- `Example:SerialNumber`
- `Example:Batterylife`
- `Example:Model`
- `Example:belongsTo`
- `Example:Vacant`
- `Example:Domain`

Le temps nécessaire pour générer une requête est grandement impacté par la taille de son BGP, comme nous pouvons le voir dans la Figure 4.6. C'est dû au fait que nous devons récupérer davantage de parties de la requête précédente pour cibler les capteurs qui se trouvent dans le contexte de la requête initiale, comme expliqué dans la sous-section 4.3.4.

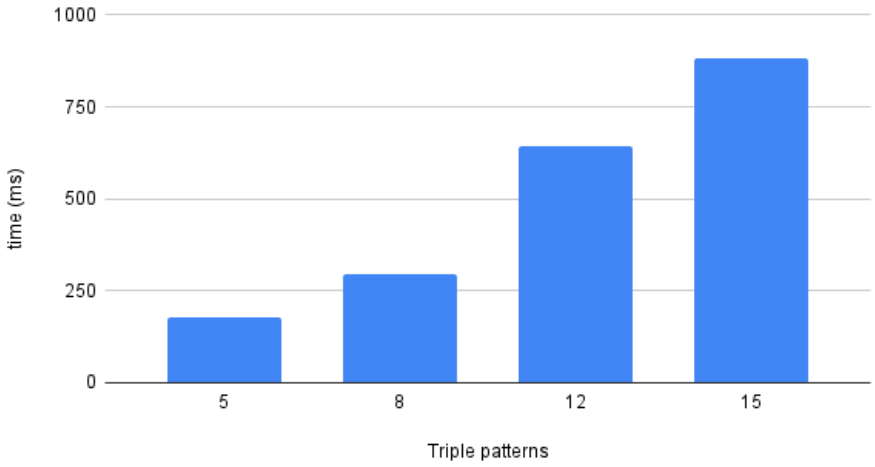

FIGURE 4.6 – Temps de génération de requêtes en fonction de la taille du BGP

4.5.6 Latence du système

Dans cette section, nous mesurons le temps aller-retour entre le serveur et le client. Le client est modifié pour détecter une anomalie dès qu'il commence à exécuter une requête qu'il reçoit. Nous lançons l'échange en envoyant la requête indiquée dans 4.3.3. Cela est fait pour mesurer le temps nécessaire au client pour recevoir une requête, traiter l'anomalie, l'envoyer au serveur, plus le temps pris par le serveur pour générer une requête en réponse à l'anomalie, comme indiqué dans le Listing 4.2, et envoyer cette requête au client. Nous pouvons voir dans la figure 4.7 que le temps pour un aller-retour est fortement impacté par le nombre de clients qu'un seul serveur doit gérer.

4.5.7 Utilisation du réseau

L'utilisation du réseau est fortement réduite, car le serveur n'a besoin que d'une anomalie pour générer une requête, le client envoie uniquement la valeur de l'anomalie et le signal correspondant à la requête déclenchante, donc la surcharge du message est faible. Lorsqu'une valeur a déclenché une extension du graphe, la règle SHACL associée ne peut pas être déclenchée par le même signal. Nous n'envoyons donc la requête générée qu'une seule fois. L'utilisation du réseau dans SuccinctEdge avait déjà été identifiée comme une métrique importante pour le système dans [20], ces modifications nous permettent de continuer à transmettre autant d'informations utiles qu'auparavant, tout en limitant l'utilisation du réseau des clients SuccinctEdge.

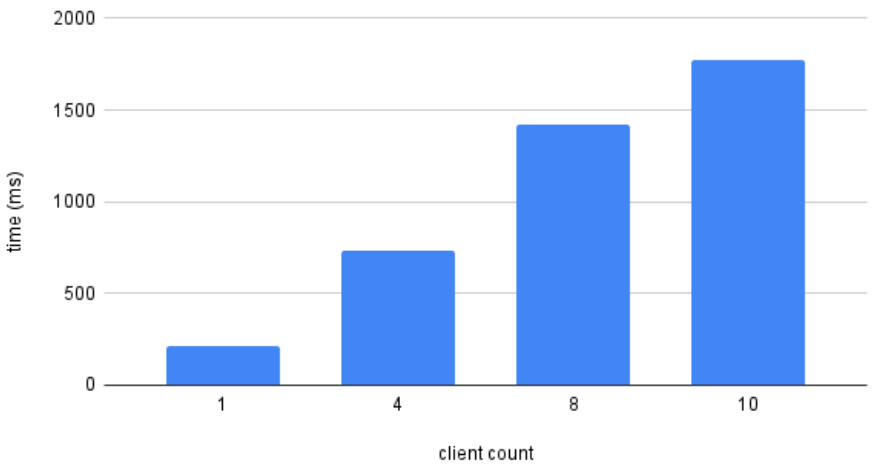

FIGURE 4.7 – Temps d'aller-retour entre le client et le serveur en fonction du nombre de clients

4.6 Retour d'expérience

Bien que nous utilisions SHACL de manière non conventionnelle, il est suffisamment expressif pour les cas que nous voulons représenter. De plus, nous le considérons plus simple d'utilisation pour les experts du domaine par rapport à la conception de contraintes similaires en logique de descriptions [8]. Enfin, notre évaluation met en évidence que le traitement de la validation SHACL est suffisamment efficace dans la plupart des cas d'utilisation que nous avons rencontrés dans le cadre industriel chez ENGIE.

Nous avons constaté que le système est aussi réactif que les capteurs qu'il utilise. Si un capteur ne transmet ses mesures qu'une fois par jour, il est impossible de répondre à une anomalie avant qu'elle ne soit transmise à un client. Ainsi, dans les zones sensibles, nous pourrions vouloir que les capteurs transmettent des données en continu, mais dans les cas où les capteurs utilisent leur batterie comme source d'énergie, cela serait totalement impraticable. Un équilibre doit être trouvé entre la consommation d'énergie des capteurs et la réactivité du système autonome, au cas par cas. De plus, le système doit être en mode streaming réel pour envoyer les anomalies le plus tôt possible au serveur.

En ce qui concerne la consommation d'énergie du client SE, lorsque nous générons de nouvelles requêtes pour contrôler les valeurs des variables qui pourraient être impactées par des solutions, le nombre de requêtes générées peut rapidement augmenter. Si le client SE fonctionne sur batterie, l'exécution potentiellement de dizaines de nouvelles requêtes peut submerger le client ou augmenter considérablement sa consommation d'énergie. Normalement, ces requêtes sont arrêtées lorsque la situation qui les a déclenchées est résolue, mais dans les cas où les situations persistent pendant plusieurs jours, il pourrait être utile de

stopper certaines de ces requêtes de manière autonome avant la résolution de la situation. Cela pourrait être déterminé en fonction de l'importance de l'impact ou en étendant l'ontologie pour permettre aux utilisateurs d'indiquer une sorte de "durée de vie" pour ces requêtes. Une question similaire est la fréquence à laquelle ces requêtes générées doivent être exécutées sur les clients SE, et ici aussi nous pourrions bénéficier du fait de laisser l'utilisateur définir une fréquence de contrôle pour chaque impact.

Actuellement, notre ontologie nous permet seulement de représenter des cas simples, nous pourrions bénéficier d'une extension de l'ontologie. Par exemple, en permettant aux utilisateurs de définir des contextes pour les situations, les solutions et les impacts, en précisant où et quand ils peuvent s'appliquer.

4.7 Conclusion

Nous avons présenté un processus permettant d'adapter de manière autonome un système à une situation basée sur la détection d'anomalies dans un contexte IoT, à partir d'un cas d'utilisation réel avec notre partenaire ENGIE. L'expérimentation a démontré que l'extension du système SuccinctEdge est compacte et efficace sur les appareils rencontrés dans l'Edge computing, et que l'utilisation du réseau est très faible entre les clients et le serveur. Le système est capable de générer efficacement des requêtes basées sur une extension de graphe en utilisant SHACL pour gérer les anomalies côté serveur. Il est capable de produire des solutions et d'adapter les mesures surveillées par les requêtes à la nouvelle situation du système.

Dans les travaux futurs, nous souhaitons utiliser notre système avec plus de capteurs afin de voir comment SuccinctEdge se comporte dans un scénario réel plus étendu, par exemple dans des bâtiments comportant plus de capteurs diversifiés ou dans des environnements industriels. Nous envisageons également de travailler sur la dérive des capteurs au fil du temps. Avec le temps et une utilisation intensive, les capteurs peuvent subir des variations et des erreurs dans leurs mesures provenant d'une altération du calibrage. Un impact direct d'une telle dérive des capteurs est que les valeurs mesurées deviennent de moins en moins précises ou exactes. Ceci peut entraîner des erreurs dans les données collectées et donc des résultats incorrects ou imprécis dans les applications qui dépendent de ces mesures. L'identification de motifs de dérives de capteurs permettrait d'anticiper des recalibrages de ceux-ci. Ainsi, la correction des anomalies identifiées par les requêtes serait plus pertinente.

Jusqu'à présent nous avons réservé nos extensions de graphe au serveur SE. Dans le prochain chapitre, nous considérons la possibilité d'étendre dynamiquement le graphe RDF des clients. De même, nous proposons une solution de mise à jour de la TBox du graphe qui intègre notre utilisation des SDS. Les contributions mentionnées dans ce chapitre ont présentées sous forme d'un article à la conférence IEEE International Conference on Edge Computing and Communications (EDGE), en 2023 [19].

Chapitre 5

Adaptation des graphes de connaissances aux terminaux du Edge computing

Nous considérons que les graphes de connaissances peuvent contribuer à concevoir une informatique en périphérie intelligente. Aboutir à cet objectif nécessite la capacité de répondre efficacement à des requêtes nécessitant un

raisonnement efficace effectué avec un minimum de connaissances accessibles sur un appareil en périphérie du réseau. Dans ce chapitre, nous déterminons la taille minimale du graphe de connaissances dont un appareil en périphérie a besoin en fonction de l'analyse des requêtes qu'il exécute. Nous présentons également une mise à jour progressive de ce graphe de connaissances lorsque de nouvelles requêtes sont introduites dans l'environnement. Nous mettons en évidence l'efficacité de notre solution sur des cas d'utilisation réels rencontrés par notre partenaire industriel.

5.1 Introduction

Le développement d'ordinateurs à carte unique robustes, puissants et abordables, tels que les Raspberry Pis, a inspiré de nombreuses recherches dans le domaine de l'informatique en périphérie. Ce paradigme informatique déplace le stockage, la gestion et le traitement de volumes massifs de données à proximité de l'endroit où les informations sont obtenues et le résultats des traitements sont nécessaires, c'est-à-dire près des capteurs et des actionneurs, réduisant ainsi le transfert de données sur le réseau. Cette approche permet de prendre plus rapidement des décisions concernant d'éventuelles anomalies et réduit la charge sur les réseaux de communication.

Nous estimons que les KG joueront un rôle essentiel dans l'adoption d'un comportement intelligent dans ce nouveau paradigme de traitement de données. Cela inclut la capacité de répondre rapidement aux requêtes de raisonnement en utilisant les connaissances disponibles sur un appareil en périphérie d'un réseau. Plusieurs systèmes de stockage RDF, tels que SuccinctEdge (SE) [51] ou RDF4Led [29], ont été spécifiquement conçus pour l'informatique en périphérie, répondant à des exigences strictes en termes de consommation de mémoire, d'utilisation du processeur et de bande passante réseau.

Cependant, cela peut ne pas être suffisant. En effet, charger l'intégralité du KG d'un environnement industriel sur un terminal du Edge computing peut ne pas être possible car la taille de ce graphe pourrait dépasser la capacité mémoire du terminal. Et même si cela était possible, ce ne serait peut-être pas nécessaire pour un appareil qui doit répondre à un ensemble prédéfini de requêtes et qui n'a donc besoin que d'un sous-ensemble des connaissances du domaine. Une approche d'adaptation du KG stocké sur chaque RDF store d'un terminal à la périphérie peut potentiellement réduire l'empreinte mémoire de la machine et accélérer l'exécution de requêtes. Cela correspond à notre premier énoncé de problème, que nous définissons comme suit :

Problème 1. *Dans le contexte d'un environnement Internet des objets (IoT), comprenant une collection de capteurs (S_i), un groupe de clients de SGBD RDF et un serveur de SGBD RDF qui détient un KG représentant les connaissances du domaine, l'objectif est de fournir à chaque client un sous-graphe spécifique du KG qui est nécessaire et suffisant pour ses besoins.*

Dans ce chapitre, nous proposons une approche qui identifie le sous-ensemble

d'un KG dont chaque appareil en périphérie a besoin pour répondre à son ensemble particulier de requêtes. Cet ensemble de requêtes regroupe deux types de requêtes : i) celles définies par un expert du domaine et ii) des requêtes supplémentaires qui peuvent être nécessaires en cas de suspicion d'anomalie, typiquement, celles générées par les traitements décrits dans le Chapitre 4. Les deux types de requêtes peuvent nécessiter une forme de raisonnement, par exemple la subsumption de classes, et nécessitent donc l'identification et le stockage des connaissances requises pour ces traitements.

Enfin, nous abordons l'aspect dynamique d'un environnement IoT. Cela concerne l'évolution du KG de domaine sous-jacent ainsi que la mise à jour de l'ensemble de requêtes pour un appareil en périphérie donné. Notez que la mise à jour de l'ensemble de requêtes peut impliquer le chargement d'une nouvelle version d'un KG. Cela peut être le cas lorsque des requêtes qui interrogent de nouveaux capteurs sont ajoutées à l'environnement IoT. Le KG spécifique au client qui exécute ces requêtes devra alors ajouter le nouveau capteur en question.

Notre implémentation prototype est basée sur le SGBD RDF SE. Ceci est motivé par certaines propriétés de ce système : une empreinte mémoire de données réduite, des capacités de raisonnement et un traitement efficace des requêtes SPARQL. Ces propriétés sont en partie dues à l'utilisation de structures de données succinctes (SDS), qui sont particulièrement efficaces pour les opérations de lecture, mais notoirement inefficaces lorsqu'une mise à jour est nécessaire. Par conséquent, afin de prendre en charge la mise à jour dans notre système, nous avons mis en œuvre une approche de mise à jour au-dessus des SDS qui convient particulièrement à notre contexte. Cela nous permet de présenter notre deuxième énoncé de problème.

Problème 2. *Dans un contexte IoT utilisant des clients SE et ayant des besoins de connaissances dynamiques, le problème consiste à proposer une solution de maintenance incrémentale efficace pour chaque KG client qui garantit les caractéristiques de performance originales de SE.*

Ce chapitre est organisé comme suit. Dans la section suivante, nous présentons des SDS supportant des ajouts de données et leur application dans SE. Dans la section 5.2, nous proposons une solution pour la mise à jour des BVs et WTs au travers d'une extension de la représentation RRR. Dans la section 5.3, nous présentons différentes stratégies de réduction de graphe de connaissance et leurs avantages. Nous présentons quelques travaux connexes dans la section 5.4 avant d'évaluer notre système dans la section 5.5 sur des ensembles de données synthétiques. Nous concluons le chapitre dans la Section 5.6.

5.2 Mise à jour incrémentale du KG avec un vecteur de bits RRR extensible

Notre objectif est de créer des BVs qui permettent d'ajouter des éléments sans avoir à reconstruire l'ensemble du BV. Cette approche de reconstruction,

est actuellement l'opération la plus coûteuse en temps et calculs sur les clients SE. Si nous reconstruisons un BV de taille M (bits) pour ajouter des données de taille N (bits), la complexité de la construction est alors de l'ordre de O(M+N). On cherche à trouver une solution plus efficace de l'ordre de O(N). L'une des solutions pour cela est de stocker séparément des parties du BV. Cela évite d'avoir besoin de réallouer de l'espace pour les données déjà présente dans le BV, mais signifie que la structure sera potentiellement plus lente à parcourir à cause des séparations entre les espaces mémoires stockant la structure. Nous avons observé dans nos expérimentations que ce coût supplémentaire est négligeable par rapport au gain apporté par notre approche de reconstruction.

Notre solution se base sur le fait que pour effectuer des opérations de `rank` dans les BVs RRR, il n'est jamais nécessaire d'accéder aux valeurs contenues dans plus d'un SB. Cependant, dans le pire des cas, lorsque le `rank` demandé se trouve à la fin d'un SB, le système doit pouvoir parcourir tous les blocs de ce SB. Cela signifie que nous devons maintenir la contiguïté mémoire au sein des SBs pour éviter les défauts de cache lors de l'exécution d'un `rank`. Cependant, il n'est pas nécessaire de maintenir la contiguïté mémoire entre les SB car il n'est pas nécessaire de récupérer des valeurs de plusieurs SB, l'accès à un seul SB est toujours suffisant.

Par exemple, considérons un BV dont les tailles de bloc et SB sont respectivement 5 et 3. On peut alors calculer que le résultat de l'opération `rank(23)` se trouve dans le superbloc 1, car 23/5=4 et 4/3=1. Pour trouver le résultat dans le pire des cas il n'y aura qu'à parcourir le SB 1 depuis son début, mais dans aucun cas il n'y aura besoin d'accéder au SB suivant ou précédent. Cela signifie que puisque nous voulons exécuter nos `rank` sans risquer de défaut de cache (cache miss), il est nécessaire de stocker toutes les données d'un SB sur un même espace mémoire mais pas nécessairement tous les SBs ensemble.

Bien que l'article original sur les BVs RRR présente une structure à part pour supporter les opérations `select`, la plupart des solutions existantes utilisent d'autres approches en raison de la complexité de l'implémentation. Par exemple l'implémentation de la librairie SDSL-lite utilise une recherche dichotomique sur les blocs du BV pour répondre aux `select`. Cette approche ne serait pas efficace si l'on stockait les SBs séparément car cela demanderait d'accéder à des SBs différents pour la plupart des étapes, multipliant les potentielles situations de défaut de cache. Par contre, nous estimons qu'il est nécessaire de créer une structure capable de répondre aux `select` en temps constant qui supporte aussi l'ajout de données.

Il n'y a pas de contraintes particulières à prendre en compte pour les opérations `access`.

Dans la suite de ce document, nous décrivons une structure permettant d'ajouter des éléments à un BV, et donc à un WT, comme une **extension de structure**, et les données qui ont été ajoutées comme une **extension de BV**.

5.2.1 Structure RRR^{EX}

Les deux structures d'un BV RRR sont les blocs (B) qui stockent une version compressée des donnés du BV et les superblocs (SB) qui stockent des valeurs précalculées pour les opérations de `rank`. Pour étendre notre structure que l'on nommera RRR^{EX}, nous créons une structure pour stocker les B et SB à ajouter de manière incrémentale. Nous utilisons un dictionnaire stockant nos B et SB en valeur avec leur numéro de SB pour clé.

Les BVs RRR utilisent principalement deux paramètres pour représenter leurs données : la taille des B (B$^{\#}$), comptant un certain nombre de bits, et la taille des SB (SB$^{\#}$), comptant un nombre de B. Les tailles de B permettent de contrôler le niveau de compression des données et les tailles de SB permettent de contrôler le nombre de valeurs de `rank` précalculées des SB. Nous avons expliqué plus en détail le fonctionnement des BV RRR dans la Section 2.2.1. Nous réutilisons ces paramètres dans les RRR^{EX} pour les B et SB que nous ajoutons. Par exemple, la Figure 5.1.a présente un BV contenant 18 bits représentés sous forme de 9 blocs de taille B$^{\#} = 2$ bits et 3 super blocs de taille SB$^{\#} = 3$ blocs. Grâce à ces deux éléments, RRR peut répondre aux opérations de `rank` en temps constant pour un B# et un SB# donnés.

En reprenant le BV de la Figure 5.1.a, comme BV de base, nous avons dans la Figure 5.1.b, 9 bits de données à ajouter en tant qu'extension. La partie inférieure de la Figure 5.1.b présente les données ajoutées dans deux SB nouvellement créés et la valeur précalculée de leurs rangs, toutes deux stockées dans un dictionnaire avec 0 et 1 comme clés, c'est-à-dire les identifiants de SB. Dans la Figure 5.1.b, le premier SB, donc identifiant 0, dont les données correspondent à la séquence de bits 000110, sont stockées dans le tuple (SB_{rank},B_{SB}) dont les valeurs sont respectivement 9 et '000110'. La valeur SB_{rank} de 9 est calculée à partir de la dernière valeur stockée dans la séquence de SB de la structure RRR, soit 5, plus le nombre de bits à 1 du dernier bloc du B de RRR, soit 4. Pour les entrées suivantes de RRR^{EX}, la valeur de SB_{rank} correspond à la valeur du SB_{rank} précédente plus le nombre de bits à 1 du B_{SB} précédent. Dans le cas de la seconde entrée de notre structure RRR^{EX}, soit à l'index 1, SB_{rank} de 11 correspond à 9+2=11.

Extension de RRR pour Rank et Access

Lorsque nous ajoutons des données à un RRR^{EX}, nous devons donc pouvoir calculer le minimum d'espace nécessaire pour stocker les nouvelles données, ce qui peut être compliqué lorsque nous n'avons pas assez de données à ajouter pour remplir un SB complet, e.g., c'est le cas de l'index 1 dans la Figure 5.1.b. Dans un BV RRR classique, l'espace utilisé pour stocker les valeurs de `rank` précalculées des SB peut être déterminé en utilisant le bit de poids fort de la dernière valeur de `rank`. Ce même principe est utilisé pour stocker les valeurs compressées des blocs que nous connaissons à la création du BV.

Par exemple pour un BV dont la dernière valeur de `rank` est 118, nous aurons besoin de stocker chaque valeur de `rank` sur 7 bits, car c'est le bit de poids fort

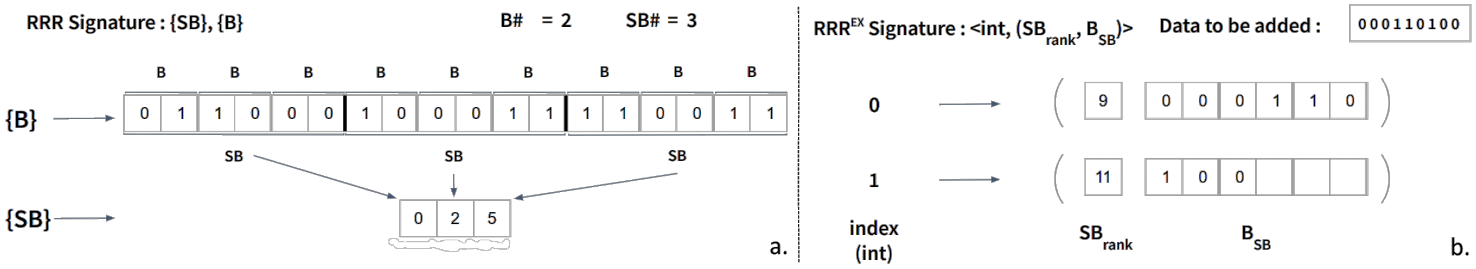

FIGURE 5.1 – structures RRR et RRR^{EX}

de 118 (1110110 en binaire). Cela sert à minimiser l'espace utilisé pour les B et SB.

Dans notre cas cependant, si les données que nous devons ajouter ne suffisent pas à remplir un SB, il est tout de même nécessaire d'allouer un espace capable de stocker les B d'un SB entier ainsi que sa valeur de `rank` sans connaître les données qui seront potentiellement ajoutées plus tard pour le compléter. Pour résoudre ce problème, nous considérons que le reste du SB sera rempli de "1" et nous calculons la valeur de `rank` possible la plus élevé pour cette structure. Cette méthode nous permet de calculer l'espace minimal nécessaire pour stocker toute donnée dans l'espace restant du SB.

Si le dernier SB est incomplet, nous réutilisons ce SB pour les prochains ajouts de données jusqu'à ce qu'il soit complet et que nous devions en commencer un nouveau. Si un B de l'extension est incomplet, ses données seront réécrites lors de la prochaine extension du RRR^{EX}.

La Figure 5.1.b présente un exemple concret de cette situation. On observe bien qu'à l'index 1 de notre structure, le tuple (SB_{rank},B_{SB}) présente un B_{SB} incomplet puisqu'il ne renseigne que 3 bits sur les 6 possibles ($\text{SB}^{\#}$*$\text{B}^{\#}$ = 3*2).

Dans le pire des cas en termes d'utilisation d'espace, si nous étendons un RRR^{EX} d'un seul bit, nous avons besoin d'allouer l'espace pour un SB entier. Cela signifie également que dans tous les cas, nous pouvons toujours ajouter des éléments à la fin d'un RRR^{EX} sans avoir besoin d'allouer plus d'espace que la taille de l'extension et d'un SB. Avec un BV RRR classique, il n'est pas rare que le dernier SB soit incomplet, ne comprenant pas le nombre maximal de B ou bien comprenant un B incomplet. Lorsque le dernier SB est incomplet, nous le représentons dans la partie étendue du RRR^{EX} au moment de la création.

Nouvelle structure pour Select

Notre structure `Select` est composée de deux BVs RRR^{EX}. Lors de l'exécution d'un $\texttt{select}_{\text{i}}(\text{n})$, nous les utilisons pour déterminer le BV contenant le $\text{n}^{\text{ième}}$ bit i. Notre mise en œuvre nécessite l'utilisation d'une opération `rank` sur ces BVs. Un BV est utilisé pour supporter les opérations $\texttt{select}_1$, *i.e.*, pour les bits mis à 1, et l'autre est utilisé pour les opérations $\texttt{select}_0$, *i.e.*, pour les bits mis à 0.

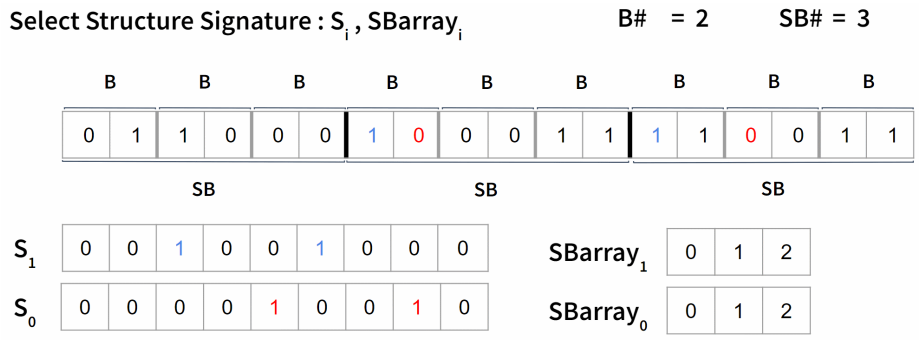

FIGURE 5.2 – Example de structure de Select

L'algorithme Algo. 1 détaille la construction de ces deux BV RRR^{EX} dénommés respectivement S_1 et S_0, leur utilisation est décrite dans Algo. 2. Le premier BV, *i.e.*, S_1, est construit en utilisant la position des bits mis à 1 dans les données d'origine. Si un bit mis à 1 est situé dans un SB différent du bit précédent mis à 1, nous mettons un 1 dans le BV S_1. Le second BV, S_0, est construit de la même manière que le précédent en utilisant la position des bits mis à 0. Dans les deux structures, le premier bit est mis à 0.

Nous créons également deux tableaux contenant la position des SB dans lesquels apparaissent respectivement les bits mis à 1 et à 0. Le tableau pour les SB contenant des 1 est appelé SBarray_1 et le tableau pour les SB contenant des 0 est appelé SBarray_0. Ces positions apparaissent dans l'ordre des SB et sont nécessaires pour couvrir les cas où certains SB ne contiennent que des 1 ou des 0. La mémoire utilisée pour cette structure pourrait être réduite en ne conservant pas la position de chaque SB où les bits apparaissent, mais en écrivant plutôt le décalage entre la position du SB et la position dans le tableau. Dans l'algorithme Algo. 1, les lignes 4 à 11 couvrent la construction du BV $\texttt{S}_0$, et les lignes 12 à 18 couvrent la construction du BV $\texttt{S}_1$, en fonction du $i^{\text{ième}}$ bit des données. La structure résultante est illustrée à la Figure 5.2, où les bits mis à 1 dans les deux BV sont mis en évidence en couleur. Algo. 1 a été simplifié en excluant la construction des tableaux SBarray_1 qui apparaissent dans Algo. 2. Ils sont construits en ajoutant le nouveau SB trouvé dans les lignes 8 et 15 de Algo. 1, mais avec un cas particulier pour le SB avec l'identifiant 0, car nous ne le trouvons jamais en tant que nouveau SB et il ne contient pas nécessairement de valeur définie à 1 ou 0.

Pour pouvoir ajouter des données au RRR_{EX} et conserver notre structure `Select`, il est nécessaire de modifier deux parties de la structure, les RRR_{EX} S_0 et S_1 ainsi que les SBarray_0 et SBarray_1. Pour le RRR_{EX}, les S_0 et S_1 du BV doivent être étendus eux-mêmes en utilisant le même processus que celui que nous avons utilisé pour les construire initialement. Pour SBarray_0 et SBarray_1, nous les remplaçons par des dictionnaires afin de pouvoir y ajouter des données sans réécrire les données existantes. Nous verrons dans la section

Algorithm 1 Construction de la structure de Select

```
1:  i ← 0
2:  last_sb_0 ← 0
3:  last_sb_1 ← 0
4:  while i < data.size() do
5:      if data[i] = 0 then
6:          if last_sb_0 ≠ i/(B^# × SB^#) then
7:              S_0.add(1)
8:              last_sb_0 ← i/(B^# × SB^#)
9:          else
10:             S_0.add(0)
11:         end if
12:     else
13:         if last_sb_1 ≠ i/(B^# × SB^#) then
14:             S_1.add(1)
15:             last_sb_1 ← i/(B^# × SB^#)
16:         else
17:             S_1.add(0)
18:         end if
19:     end if
20:     i ← i + 1
21: end while
```

suivante comment ces structures sont utilisées pour répondre aux opérations SDS.

5.2.2 Operations SDS sur RRR^{EX}

Rank

Pour répondre aux opérations `rank` un algorithme unique est suffisant, le résultat d'un $\texttt{rank}_1(x)$ pouvant être utilisé pour déduire le résultat de $\texttt{rank}_0(x)$, soit $\texttt{rank}_0(x) = x - \texttt{rank}_1(x)$. Une fois que le SB qui contient le $x_{\text{ième}}$ bit est identifié, les opérations au sein du SB sont les mêmes que pour un BV RRR classique.

Select

Pour répondre aux opérations `select`, il est nécessaire de traiter les cas de $\texttt{select}_0$ et $\texttt{select}_1$ séparément.

L'algorithme Algo. 2 détaille l'opération de $\texttt{select}_1$. L'algorithme de $\texttt{select}_0$ est détaillé dans l'algorithme Algo. 3. Les différences entre ces algorithmes se trouvent dans les structures qui sont utilisées à la ligne 4, et dans la façon d'interpréter la valeur dans les structures de `rank` communes, aux lignes 8, 9 et 14. Pour sélectionner la position d'une certaine valeur dans un BV, nous utilisons

Algorithm 2 $\text{Select}_1(\text{i})$

Require: $i \geq 0$
1: **if** $i = 0$ **then**
2: return 0
3: **end if**
4: $SBpos \leftarrow rank_1(S_1, i)$
5: $SBid \leftarrow SBarray_1[SBpos]$
6: $rank\_at\_pos \leftarrow SBrank[SBid]$
7: $Block \leftarrow 0$
8: **while** $rank\_at\_pos + BV[SBid][Block] \leq i$ **do**
9: $rank\_at\_pos \leftarrow rank\_at\_pos + BV[SBid][Block]$
10: $Block \leftarrow Block + 1$
11: **end while**
12: $pos \leftarrow (SBid \times SB^{\#} + Block) \times B^{\#}$
13: **while** $rank\_at\_pos < i$ **do**
14: **if** $BVdata[pos] = 1$ **then**
15: $rank\_at\_pos \leftarrow rank\_at\_pos + 1$
16: **end if**
17: $pos \leftarrow pos + 1$
18: **end while**
19: return pos

une opération de `rank` sur le BV $\mathtt{S}_1$ ou le BV $\mathtt{S}_0$, en fonction de la valeur, comme indiqué à la ligne 4 avec S_1 faisant référence au BV $\mathtt{S}_1$.

Le résultat de l'opération de `rank` est utilisé pour déterminer le SB dans lequel la valeur sélectionnée est située en recherchant le résultat du `rank` dans le tableau associé. Ce tableau est illustré dans la Figure 5.2 en tant que SBarray_1, contenant la position de chaque SB contenant des valeurs définies à 1, et est utilisé à la ligne 5 de l'algorithme Algo. 2. Une fois que nous avons trouvé le SB où se trouve la position, nous calculons la position pour la valeur au début du SB à partir de sa valeur de `rank` précalculée, puis nous recherchons à l'intérieur du SB le B contenant la position, et enfin la position elle-même que nous pouvons retourner. Ce processus se fait en deux parties, de la ligne 8 à 12 où nous itérons à travers les valeurs de `rank` des B de notre SB, et de la ligne 13 à 18 où nous itérons à travers les bits individuels du B contenant la position que nous recherchons. Cela signifie que la complexité d'une opération de `select` peut être réduite à celle d'une opération de `rank` suivie d'une recherche dans un BV de la taille d'un SB.

5.2.3 Wavelet tree extensibles

Il existe peu de travaux sur des WT capables de supporter des mises à jour de leurs données[14][32]. Cela pourrait être dû au manque de travaux sur les BV mutables et, même dans ce cas, les implémentations de WT ont tendance

Algorithm 3 $\text{Select}_0(\text{i})$

Require: $i \geq 0$
1: **if** $i = 0$ **then**
2: return 0
3: **end if**
4: $SBpos \leftarrow rank_1(S_0, i)$
5: $SBid \leftarrow SBarray_0[SBpos]$
6: $rank\_at\_pos \leftarrow (SB\_sizexSBid) - SBrank[SBid]$
7: $Block \leftarrow 0$
8: **while** $rank\_at\_pos + Block\_size - BV[SBid][Block] \leq i$ **do**
9: $rank\_at\_pos \leftarrow rank\_at\_pos + Block\_size - BV[SBid][Block]$
10: $Block \leftarrow Block + 1$
11: **end while**
12: $pos \leftarrow (SBid \times SB^{\#} + Block) \times B^{\#}$
13: **while** $rank\_at\_pos < i$ **do**
14: **if** $BVdata[pos] = 0$ **then**
15: $rank\_at\_pos \leftarrow rank\_at\_pos + 1$
16: **end if**
17: $pos \leftarrow pos + 1$
18: **end while**
19: return pos

à se concentrer sur l'efficacité d'espace, qui est l'un des principaux défis de ces structures mutables. Dans notre cas, nous n'avons a priori pas besoin de pouvoir modifier un triplet déjà présent dans le WT, mais nous devons pouvoir ajouter de nouveaux triplets au fur et à mesure de leur ajout dans le graphe.

Cela nous amène à concevoir un WT et un BV qui peuvent être étendus à volonté tout en préservant les temps de `rank`, `select` et `access` des structures d'origine. En utilisant des BV prenant en charge l'opération d'ajout à la fin, nous pouvons insérer des valeurs à la fin de chaque noeud présent dans un WT. Cela est illustré dans la Figure 5.3, où trois WTs sont représentés et nommés a, b et c. Nous utilisons le WT b comme une extension du WT a, ce qui produit le WT c avec son extension marquée en rouge. Ce WT doit représenter chaque nœud qu'il contient sous la forme d'un BV RRR^{EX} afin de pouvoir étendre chaque nœud. Cela nous empêche de charger l'ensemble de l'arbre dans le cache lorsque nous y travaillons, mais nous permet de repositionner l'arbre sur une nouvelle racine lorsque nous ajoutons de nouveaux éléments qui étendent le dictionnaire sans avoir besoin de le reconstruire, comme illustré dans la Figure 5.4. Lorsque nous devons repositionner l'arbre pour ajouter de nouveaux éléments, des nœuds internes sont créés, préservant ainsi le WT en tant qu'arbre équilibré en hauteur.

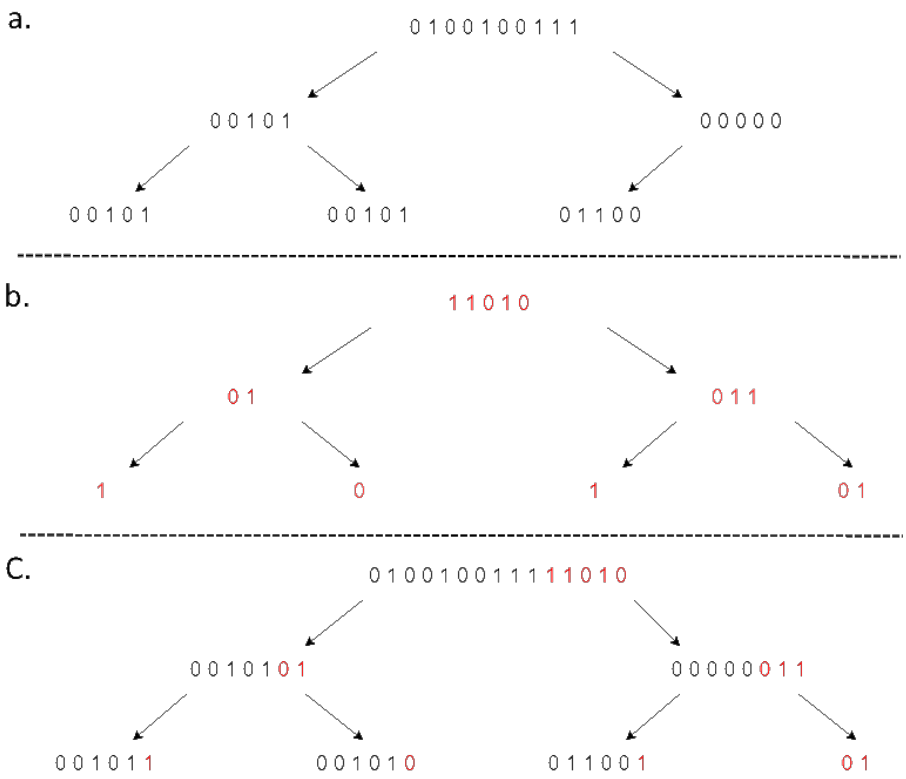

FIGURE 5.3 – Extension de Wavelet tree

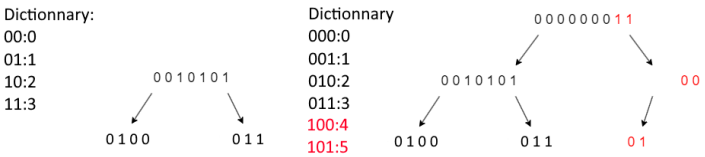

FIGURE 5.4 – Extension de wavelet tree qui provoque un changement de racine

5.3 Stratégies de décomposition du graphe de connaissances

Dans cette section, nous présentons nos solutions au Problème 1 défini dans la Section 5.1. Pour justifier nos solutions, nous utilisons un cas d'usage basé sur le benchmark LUBM. Nous utilisons le KG de LUBM pour des contraintes de confidentialité concernant les données d'ENGIE. De même LUBM est un benchmark permettant un paramétrage lors de la création de graphes RDF, ce qui nous permet de générer des jeux de différentes tailles.

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?M ?C WHERE {
        ?P <lubm:worksFor> %m .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
}
```

Listing 5.1 – Requête SPARQL Générique

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?M ?C WHERE {
        ?P <lubm:worksFor> <http://www.Department0.University0.edu> .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
}
```

Listing 5.2 – Requête SPARQL originale

Le Listing 5.2 est une requête SPARQL recherchant les adresses email des personnes enseignant des cours ainsi que ces cours. Notons que cette requête présente une légère modification de la syntaxe SPARQL puisqu'elle introduit une nouvelle forme à la position de l'objet du 1er TP du BGP : l'objet est préfixé par le symbole '%'. Cette extension est motivée par notre besoin d'instancier facilement une requête avec des paramètres fournis par notre serveur SE qui stocke les métadonnées de l'application. Ceci est réalisé en associant des constantes à chaque paramètre de la requête '%' en utilisant l'approche utilisée dans les requêtes SQL paramétrées. Par exemple, dans la figure 5.2, le paramètre '%m' pourrait être associé à l'IRI suivante :
$< http://www.Department0.University0.edu >$.

Dans la suite du chapitre, lorsque nous construirons de nouvelles requêtes à partir de la requête de la Figure 5.2, nous utiliserons la constante au lieu du paramètre '%m'. Nous utiliserons cette requête pour illustrer la manière dont les différentes stratégies de réduction des graphes transforment les requêtes pour générer leur sous-graphe.

5.3.1 Réduction dirigée par les prédicats

La décomposition du KG est guidée par les prédicats présents dans les requêtes exécutées par chaque client SE. Notre première étape consiste à analyser les requêtes SPARQL soumises à tous les clients SE de notre domaine. Pour un client SE que l'on dénotera SE_k, ses requêtes correspondent à (1) celles définies à l'origine par un expert du domaine et exécutées en continu sur les mesures des capteurs de SE_k, notées Q_o, (2) les requêtes liées aux anomalies inférées à partir de Q_o, notées Q_i. L'exécution des requêtes Q_i n'est pas continue, mais dure seulement un certain temps lorsqu'une anomalie a été déclenchée par une

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
CONSTRUCT {
        ?P <lubm:worksFor> ?D .
} WHERE {
        ?P <lubm:worksFor> ?D .
}

CONSTRUCT {
        ?P <lubm:emailAddress> ?M .
} WHERE {
        ?P <lubm:emailAddress> ?M .
}

CONSTRUCT {
        ?P <lubm:teacherOf> ?C .
} WHERE {
        ?P <lubm:teacherOf> ?C .
}
```

Listing 5.3 – Construct par prédicat

requête Q_o provenant de SE_k. La génération autonome des requêtes suit le procédé décrit dans le Chapitre 4.3.4.

Étant donné l'ensemble de requêtes $Q = Q_o \cup Q_i$, pour chaque $q \in Q$, notre solution extrait tous les prédicats du BGP de q et les stocke dans un ensemble Σ. Ensuite, pour chaque $\sigma \in \Sigma$, nous générons une requête SPARQL de type `CONSTRUCT` permettant de récupérer tous les triplets du KG contenant le prédicat σ. Le TP de ces requêtes `CONSTRUCT` possède des variables distinctes pour le sujet et l'objet du motif de triplets. Notez que puisque nous récupérons des triplets à partir d'un KG unique, nous pouvons supposer que les nœuds anonymes récupérés par nos requêtes `CONSTRUCT` sont conservés tels quels. Cette approche plutôt naïve présente plusieurs propriétés intéressantes dans le contexte de notre stockage basé sur des SDS et supportant des modifications. Nous mettons en évidence ces propriétés dans la section 5.3.3.

Les résultats de l'exécution de toutes les requêtes `CONSTRUCT` sont fusionnés en un sous-graphe qui est chargé dans un client SE_k. La même décomposition du KG peut être stockée dans différents clients SE s'ils exécutent des requêtes similaires. C'est généralement le cas dans un environnement IoT où de nombreux capteurs fournissent des mesures similaires qui sont analysées par certains clients SE. La rapidité de cette méthode dépend directement du nombre de prédicats différents utilisés dans les requêtes des clients.

En appliquant cette stratégie à la requête du Listing 5.2, nous obtenons le résultat présenté dans le Listing 5.3. Étant donné que trois prédicats différents sont utilisés dans les TPs de la requête, cette stratégie produit trois différentes requêtes CONSTRUCT, reprenant chacun des TPs en renommant leurs variables.

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
CONSTRUCT {
        ?P <lubm:worksFor> ?D .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
} WHERE {
        ?P <lubm:worksFor> ?D .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
}
```

Listing 5.4 – Construct par BGP

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
CONSTRUCT {
        ?P <lubm:worksFor> <http://www.Department0.University0.edu> .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
} WHERE {
        ?P <lubm:worksFor> <http://www.Department0.University0.edu> .
        ?P <lubm:emailAddress> ?M .
        ?P <lubm:teacherOf> ?C .
}
```

Listing 5.5 – Construct par BGP avec constantes

5.3.2 Réduction dirigée par les BGPs

Cette méthode consiste à directement convertir chaque requête d'un client en une requête CONSTRUCT reprenant les mêmes TP, puis de réaliser l'union de ces CONSTRUCT pour obtenir le sous graphe final. L'opération de conversion est très simple et le sous graphe produit sera quasiment toujours plus petit que celui produit par la réduction par prédicat. Cependant cela signifie que cette opération est a réaliser pour chaque requête. Sa rapidité dépend donc directement du nombre de requêtes différentes. Quand plusieurs requêtes reçues par un client sont identiques, à l'exception des constantes, il peut être efficace de remplacer toutes les constantes par des variables afin de ne créer qu'une requête CONSTRUCT pour plusieurs requêtes. Si ce n'est pas le cas ou que l'on préfère minimiser la taille du sous-graphe, il est préférable de préserver les constantes des requêtes originales et de créer une requête CONSTRUCT pour chaque requête du client.

En appliquant cette stratégie à la requête du Listing 5.2, nous obtenons le résultat présenté dans le Listing 5.4. La requête CONSTRUCT produite est quasiment identique à la requête originale à la différence de la clause SELECT et des constantes qui ont été remplacées par des variables. Lorsque l'on décide de préserver les variables de la requête originale, produisant la requête du Listing 5.5.

L'une des limites de cette méthode lorsque l'on charge ces sous graphes dans SE, est que pour un prédicat qui apparait dans le sous graphe, tous les triplets qui comportent ce prédicat dans le graphe ne sont pas nécéssairement présents dans le sous-graphe. Cela signifie que si à l'avenir une requête est ajoutée et nécessite d'étendre le sous graphe avec de nouveaux triplets contenant ce prédicat, les performances de toutes les requêtes qui le recherchent seront impactées. C'est le cas par exemple lorsque l'on a un client qui exécute une requête sur un capteur, puis qu'il reçoit une nouvelle requête portant sur un autre capteur.

5.3.3 Justification de notre stratégie `CONSTRUCT`

Comme mentionné précédemment, le client SE stocke une paire de BV et WT à chaque niveau de son index PSO unique. Bien que nous puissions insérer de nouvelles valeurs dans un WT en les ajoutant à la fin, il n'en va pas de même pour le stockage de SE en raison de sa structure en trois niveaux. L'une des méthodes les plus efficaces pour minimiser le graphe stocké au niveau du client est de convertir directement le BGP en requêtes SPARQL `CONSTRUCT`. Cependant, cette approche trouve ses limites quand une nouvelle requête nécessite d'ajouter des triplets contenant des prédicats déjà présents sur le client. Lorsque l'on stocke un triplet sur un client SE, ce triplet peut avoir été stocké ou non sur d'autres clients SE plus tôt, mais pour garantir que les ids dans les WTs du client SE sont optimaux, nous définissons et maintenons les ids des sujets, objets et prédicats localement. Cela signifie qu'un même sujet pourrait avoir plusieurs ids différents sur différents clients SE. Cela ne nous empêche donc pas d'ajouter un prédicat qui existe sur un autre client SE, mais lorsque l'on ajoute un triplet contenant un prédicat qui existe déjà sur le client où on l'ajoute, l'existence de son id nous contraint de l'ajouter sur une feuille au milieu du WT P. Il est alors nécessaire d'ajouter des données au milieu des BVs et WTs inférieurs. Ce qui, comme nous l'avons constaté, est très inefficace car cela nous forcerait à reconstruire complètement ces structures.

Si nous voulons insérer un triplet contenant un nouveau prédicat dans le WT de SE, nous pouvons simplement créer une nouvelle feuille à sa droite pour y insérer le nouveau prédicat. Cela signifie que les sujets, les objets et toutes les autres valeurs que nous devons insérer dans le système peuvent également être insérés à droite de ces structures. Cependant, si nous voulions insérer un triplet contenant de nouveau ce prédicat, toute requête recherchant des sujets ou objets en rapport avec ce prédicat devra parcourir le WT des prédicats une fois de plus. Cela signifie que, en construisant nos sous-graphes basés sur la stratégie d'extraction de prédicats expliquée dans la section 5.3.1, nous pouvons garantir que toute nouvelle requête peut être exécutée sur le sous-graphe déjà existant, sans perte de performances.

La Figure 5.5 représente l'application d'une extension de graphe sur les SDS d'un client SE, la partie étendue du graphe est représentée en rouge, soit 2 triplets comportant le même prédicat, qui n'était pas encore présent dans le graphe initial, représenté en noir. Bien que nous ayons remarqué qu'il était moins efficace d'exécuter des requêtes lorsqu'un prédicat est présent plus d'une

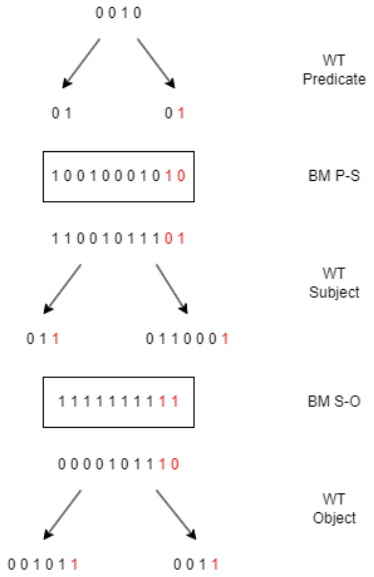

FIGURE 5.5 – Extension des SDS d'un client SE

fois dans le WT P, cela peut aussi être vrai dans le second WT, le WT S, si le sujet est de nouveau ajouté au graphe dans un triplet comportant le même prédicat mais un objet différent, en particulier pour des requêtes où ce sujet serait présent en constante. Cela est dû au fait que SE n'utilise pas les mêmes opérations pour résoudre les TPs dans une requête. Si un sujet ou un prédicat est présent en tant que constante dans un TP, nous le recherchons dans un WT en sélectionnant chaque occurrence. Le fait d'ajouter plusieurs fois le même prédicat ou sujet augmente le nombre de ces occurrences et multiplie donc ces opérations.

5.3.4 Discussion sur les strategies

Si nous comparons les avantages des différentes stratégies, les stratégies par BGP ont l'avantage de toujours produire des sous-graphes de plus petite taille ou de taille égale au sous-graphes de la stratégie par prédicat. Cette stratégie en revanche, appliquée naïvement, garantie de toujours pouvoir étendre le sous-graphe produit sur un client SE sans perte de performances pour le requêtage. Plus il y a de requêtes différentes, plus il faut de temps pour exécuter la stratégie par BGP par rapport à la stratégie par prédicat. C'est pourquoi nous nous concentrerons sur la stratégie par prédicat dans le reste de ce chapitre.

Sur la Figure 5.6 est représentée l'application des trois différentes stratégies de réduction de graphe, les nœuds et les arcs qui sont entourés sont inclus dans le sous-graphe produit par chaque stratégie. Nous remarquons que la stratégie par

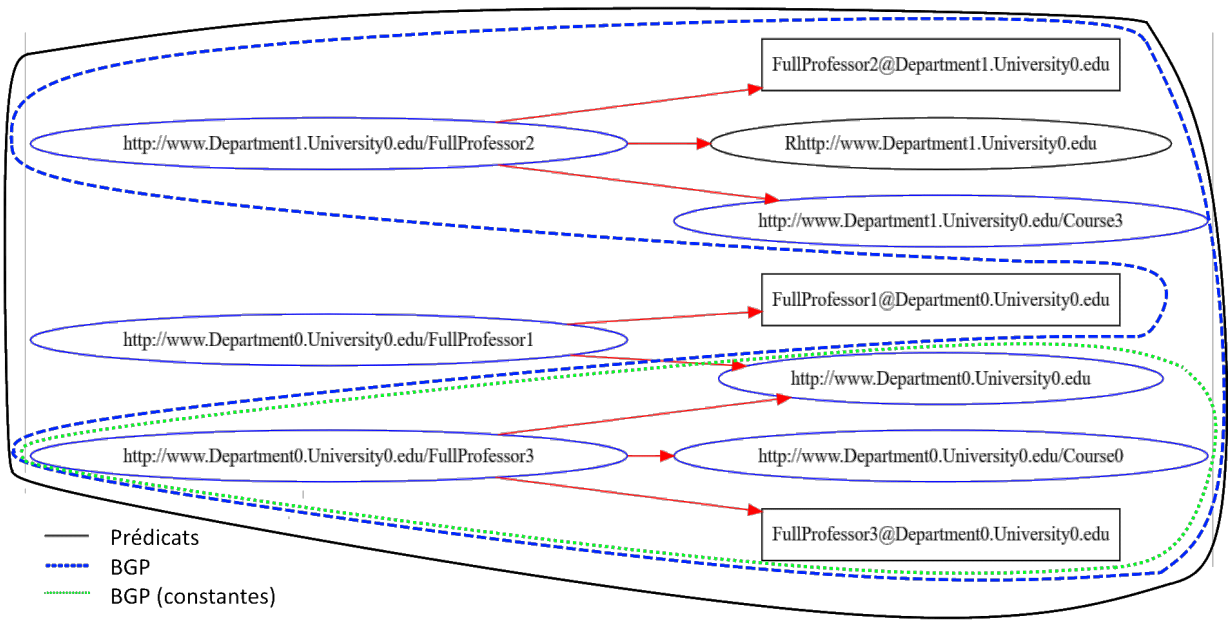

FIGURE 5.6 – Application des trois stratégies de réduction de graphe sur un extrait de jeu de donnée LUBM

prédicat tend à créer des sous-graphes disjoints qui pourraient être facilement réduits tout en restant capable de répondre aux requêtes. Nous nous pencherons ensuite sur différentes options pour optimiser les sous-graphes de la stratégie par prédicats.

5.3.5 Optimisations de la stratégie par prédicats

Comme nous l'avons vu précédemment, la stratégie par prédicat tend à créer des sous-graphes disjoints, la plupart de ces sous-graphes ne contiennent pas les nœuds et relations nécessaires aux requêtes et peuvent donc clairement être supprimés afin de minimiser la taille du graphe. Avec un ensemble composé des prédicats présents dans les TPs d'une requête, nous pouvons vérifier pour chaque sous-graphe s'il contient au moins chaque prédicat des TPs de la requête. Cela ne garantit pas que le sous-graphe contienne un résultat à la requête, mais cela permet d'éliminer la plupart des sous-graphes superflus.

Lorsque cette optimisation est appliquée à différentes requêtes, en fonction de leurs TP, il est possible de regrouper plusieurs requêtes. Si A et B sont des ensembles contenant les prédicats de deux requêtes, si A est un sous-ensemble de B, alors il n'est nécessaire que d'exécuter l'opération avec A afin d'avoir un graphe satisfaisant les requêtes de A et B. Par exemple, avec trois requêtes dont les prédicats sont [A,B,C,D], [A,B], [A,B,E], il n'est nécessaire que d'exécuter

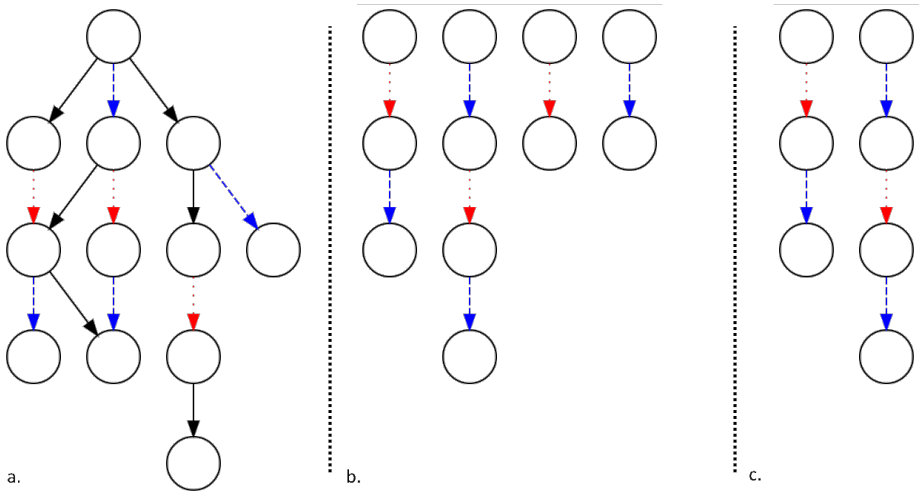

FIGURE 5.7 – Graphe simplifié

l'opération avec l'ensemble [A,B].

La Figure 5.7.a représente une version simplifiée d'un graphe où les arcs en tirets et pointillés, respectivement bleus et rouges, représentent des prédicats utilisés dans les TPs d'un ensemble de requêtes pour un client. La Figure 5.7.b représente le sous graphe produit par l'application naïve de la stratégie par prédicat, retirant les arcs absents des TPs des requêtes. La Figure 5.7.c représente le résultat final, le sous-graphe ne contenant que des graphes dont les nœuds connexes contiennent tous les prédicats nécessaires à l'exécution d'au moins une requête.

5.4 Travaux connexes

SuccinctEdge[51] et RDF4Led [29] sont tous deux des SGBDs RDF conçus pour l'informatique en périphérie. Les grandes lignes de ces systèmes ont été décrites dans la section 2.3.6. Néanmoins, nous pouvons ajouter que bien qu'ils utilisent une représentation de données et de connaissances hautement compacte, ils n'adaptent pas les connaissances chargées sur un client en fonction de sa charge de requêtes.

Les auteurs de [45] présentent une bibliothèque de SDS dynamiques axée sur le traitement de chaînes. Comparées à nos structures, elles permettent une plus grande mutabilité, comme l'insertion de valeurs n'importe où dans la structure, mais elles ne sont pas basées sur RRR et ne proposent pas d'opérations de sélection ou de rang en temps constant.

Le travail présenté dans [41] propose une structure de rang et de sélection basée sur RRR, tout comme la nôtre, mais bien qu'il offre d'excellentes performances en termes de temps de rang et de sélection, il reste une structure

immuable.

5.5 Évaluation

5.5.1 Paramètres expérimentaux et ensembles de données

Le serveur SE est utilisé pour générer le sous-graphique basé sur un KG et les requêtes SPARQL soumises. Ces graphes sont chargés sur certains clients SE et des requêtes continues sont exécutées sur eux. Le matériel du serveur pour l'expérience comprenait un processeur AMD Ryzen 5 3600 avec 6 cœurs et 3,6 GHz. Un client SE s'exécute sur un Raspberry Pi 3B+, équipé d'un processeur Cortex-A53 (ARMv7l) 32 bits SoC 1,4 GHz et 1 Go de LPDDR2 SDRAM. Les clients SE sont implémentés en C++ et utilisent notre extension de la bibliothèque SDS-Lite, la bibliothèque SDSL-lite [22], qui intègre le BV décrit dans la section 5.2 et un WT adapté à ce BV. Le serveur est implémenté en Java et utilise Apache Jena [1] pour gérer ses KG. Le code source du client et du serveur sont disponibles sur Github [2, 3].

Pour évaluer les performances de notre système avec des données disponibles publiquement, nous avons utilisé le benchmark synthétique de l'Université Lehigh (LUBM)[24]. Il peut facilement être utilisé pour créer des ensembles de données contenant plusieurs centaines de milliers de triplets représentant des universités. Nous avons décidé de créer quatre ensembles de données contenant plus de 100 000 triplets (LUBM1), 200 000 triplets (LUBM2), 300 000 triplets (LUBM3) et 400 000 triplets (LUBM4).

Number of operations	RRR Select	RRR^{EX} Select	RRR^{EX} extended Select	RRR^{EX} to RRR select percentage
10.000	0,686	0,586	3,816	85%
20.000	1,453	1,079	7,602	74%
40.000	5,41	2,221	15,748	41%
80.000	11,245	4,398	33,136	39%

TABLE 5.1 – Temps d'exécution d'opérations de select sur BV avec RRR et RRR^{EX}, tous les temps sont en ms

Number of operations	RRR Rank(ms)	RRR^{EX} Rank(ms)	RRR^{EX} extended Rank(ms)
10.000	0,002	0,002	0,763
20.000	0,002	0,002	1,476
40.000	0,002	0,002	2,997
80.000	0,002	0,002	5,79

TABLE 5.2 – Temps d'exécution d'opérations de rank sur BV avec RRR et RRR^{EX}

1. https ://jena.apache.org/
2. https ://github.com/SuccinctEdge/ExtendableSEServer
3. https ://github.com/SuccinctEdge/ExtendableSEClient

5.5.2 Efficacité de la sélection dans le vecteur de bits

Les BV que nous avons utilisés dans l'expérience utilisaient les paramètres RRR suivants : BS=8 bits et SBS=3 blocs. Notre objectif est d'évaluer l'efficacité de la sélection d'un grand nombre de valeurs dans un BV avec la structure `select` décrite dans la section 5.2, par rapport à un BV RRR qui repose sur une recherche binaire pour sélectionner ses valeurs. Dans le tableau 5.1, nous représentons la méthode de recherche binaire sous le nom "RRR Select", notre RRR^{EX} sous le nom de "RRR^{EX} `Select`", et une structure RRR^{EX} dont les données sont entièrement stockées en tant qu'extension sous le nom de "RRR^{EX} extended Select". Nous pouvons constater que notre structure `select` est environ 15% plus rapide que la recherche binaire pour 10.000 opérations de `select`. Cet écart se creuse rapidement à mesure que le nombre de `select` augmente, et nous pouvons également constater que le temps nécessaire pour exécuter les opérations de sélection sur les BV RRR^{EX} augmente de manière linéaire. On remarque aussi que les `select` sur les parties étendues d'un RRR^{EX} sont les moins efficaces, prenant plus de temps que les `select` par recherche binaire et RRR^{EX} réunis.

Nous souhaitons également évaluer l'impact des extensions de notre BV sur les opérations de `select`. Ainsi, nous stockons le même BV entièrement en tant qu'extension, comme expliqué dans la section 5.2. Toujours dans le tableau 5.1, nous pouvons constater que la sélection de valeurs dans la partie étendue d'un BV est significativement plus lente que dans le BV d'origine. Nous avons pu vérifier que l'existence d'une extension dans un BV n'affecte pas les performances des opérations de `select` sur la partie d'origine du BV. La différence de performance peut s'expliquer par les cache miss, car nous accédons à des portions non contiguës de la mémoire lors de la sélection d'autant de valeurs consécutives. Cependant, cela n'est pas souvent le cas lors de l'exécution d'une requête dans SuccinctEdge, car la seule opération comparable consisterait à interroger plusieurs fois tous les triplets contenus dans une extension de graphe.

En ce qui concerne les opérations de `rank`, dans le tableau 5.2, nous ne voyons pratiquement aucune différence entre le BV RRR d'origine et un RRR^{EX}. Cependant, lorsque l'intégralité du BV est stockée en tant qu'extension, nous constatons un ralentissement considérable. Cela est en partie dû aux mêmes raisons que pour l'opération de `select`, mais aussi à certaines inefficacités dans notre implémentation.

5.5.3 Réduction de graphes

Nous construisons nos sous-graphes comme décrit dans la section 5.3.1, avec des requêtes qui peuvent être trouvées sur le GitHub du serveur et l'ensemble des jeux de données LUBM 1, 2, 3 et 4. Chaque ensemble de données a été utilisé pour créer quatre sous-graphes contenant environ 4%, 9%, 13% et 17% des triplets présents dans le graphe d'origine. Nous pouvons voir à partir de la Figure 5.8 et de la Figure 5.10 que même avec des tailles de graphe relativement importantes pour nos cas d'utilisation, c'est-à-dire dans le cadre du Edge com-

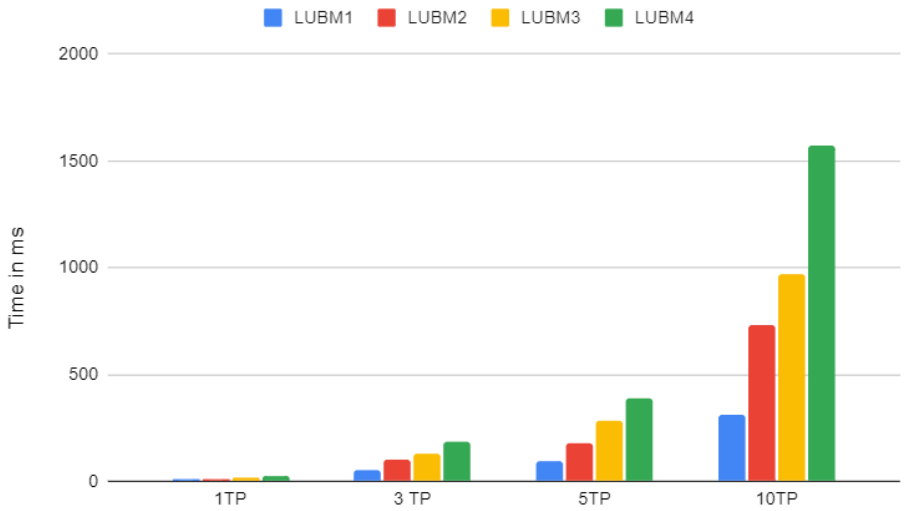

FIGURE 5.8 – Temps de construction du graphe par prédicats en fonction de requêtes de différentes tailles

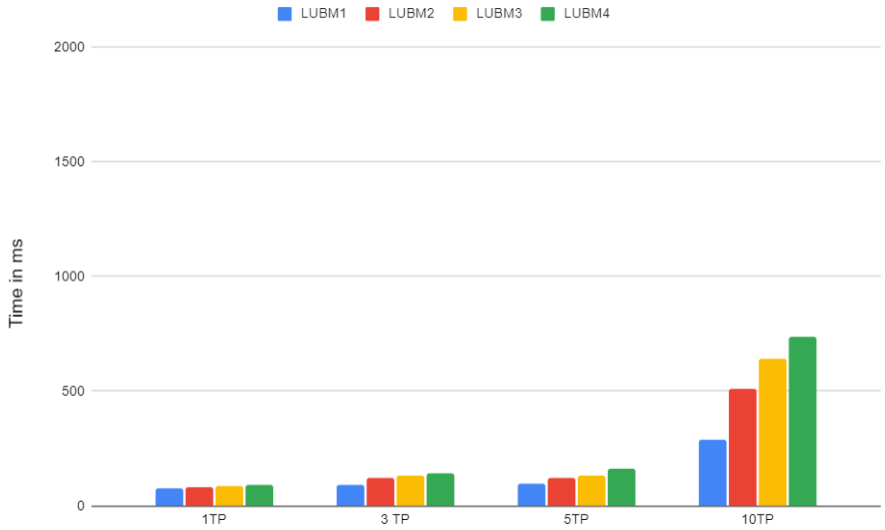

FIGURE 5.9 – Temps de construction du graphe par BGP en fonction de requêtes de différentes tailles

puting, le temps nécessaire pour construire notre sous-graphe est seulement une fraction du temps qu'il faudrait pour charger l'intégralité du graphe sur notre client SE. Cela est encore plus évident lorsque l'on demande seulement une pe-

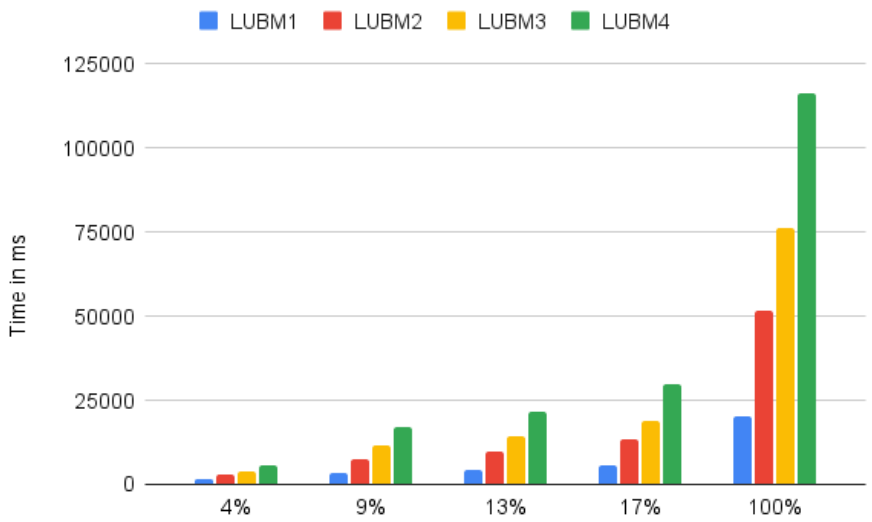

FIGURE 5.10 – Temps de chargement des sous-graphes au niveau du client en fonction du pourcentage du graphe complet

tite partie du graphe d'origine. Lorsque l'on demande pratiquement l'intégralité du graphe pour construire le sous-graphe, cela peut ne pas sembler utile, mais même une petite réduction de la taille du graphe à ce niveau aura un impact positif sur les clients dans les sections ultérieures en termes de consommation de mémoire et de CPU.

La Figure 5.9 est obtenue en utilisant la stratégie de réduction par BGP décrite dans la section 5.3.2 en utilisant une seule requête comportant le nombre de TP indiqué sur la figure. On remarque qu'elle est plus rapide à exécuter que la stratégie utilisant les prédicats pour le même nombre de TP et une seule requête, elle produit aussi des sous-graphes contenant moins de triplets et donc plus rapides à charger.

5.5.4 Impact de la réduction de graphe sur le requêtage

Dans cette section, nous mesurons l'efficacité des requêtes sur les sous-graphes que nous avons construits précédemment dans la section 5.5.3. Nous avons utilisé des requêtes différentes de celles utilisées pour créer l'ensemble de données d'origine, celles-ci peuvent également être trouvées sur le GitHub. Dans la Figure 5.11, nous pouvons voir que la réduction de la taille du graphe a un impact significatif sur le temps de requête, le réduisant d'environ 20%. Nous avons également remarqué que la taille de la réduction du graphe n'est pas la métrique la plus importante. Il semble que la réduction du nombre d'entités différentes présentes dans le graphe soit plus importante que la taille globale

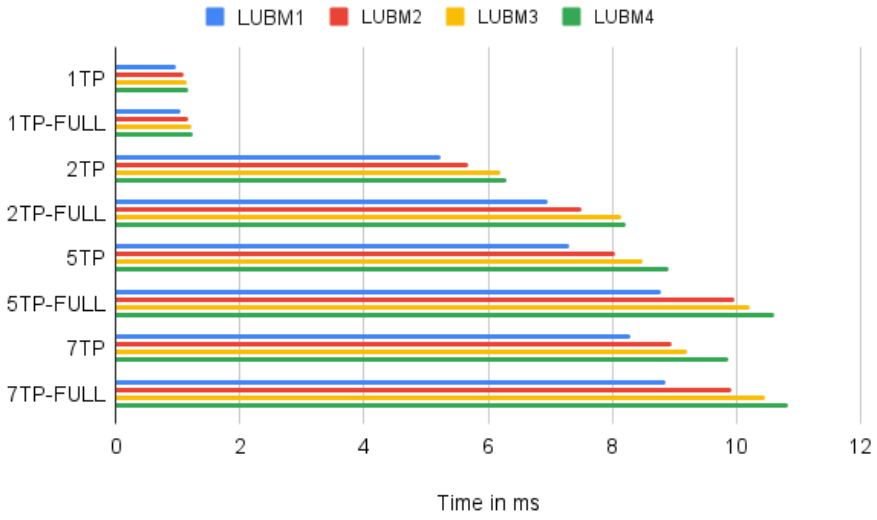

FIGURE 5.11 – Temps d'exécution des requêtes pour le sous-graphe et le graphe complet en fonction du nombre de motifs triples dans la requête

du graphe. Cela est dû au fait que la profondeur des WTs dans le système SE dépend du nombre d'entités uniques. Nous pouvons supposer qu'un sous-graphe englobant l'intégralité du graphe d'origine prendrait autant de temps à exécuter une requête que sur le graphe d'origine.

5.5.5 Mise à jour du graphe

TABLE 5.3 – Temps d'extension du graphe en fonction de la taille du sous-graphe déjà existant

Original dataset	Server time(ms)	Size	4% (ms)	9% (ms)	13% (ms)	17% (ms)
LUBM1	6	2414	843	820	847	898
LUBM2	12	5561	2093	2041	2148	2235
LUBM3	14	8202	3038	3090	3240	3217
LUBM4	18	11677	4597	4682	4945	4987

Dans cette section, nous évaluons l'efficacité de l'extension des graphes stockés sur les clients SE en fonction des extensions calculées par le serveur. Nous avons construit nos extensions en utilisant le même processus que dans la section 5.5.3, mais en utilisant une requête contenant des prédicats qui n'ont pas été utilisés dans les requêtes précédentes. Cela nous a donné quatre sous-graphes que nous avons utilisés pour étendre nos sous-graphes existants. Le tableau 5.3

```
PREFIX lubm:  <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?T ?N ?U ?D ?C WHERE {
1.    ?T <lubm:doctoralDegreeFrom> ?N .
2.    ?T <lubm:mastersDegreeFrom> ?U .
3.    ?T <lubm:worksFor> ?D .
4.    ?T <lubm:teacherOf> ?C .
}
```

FIGURE 5.12 – Requête Q1

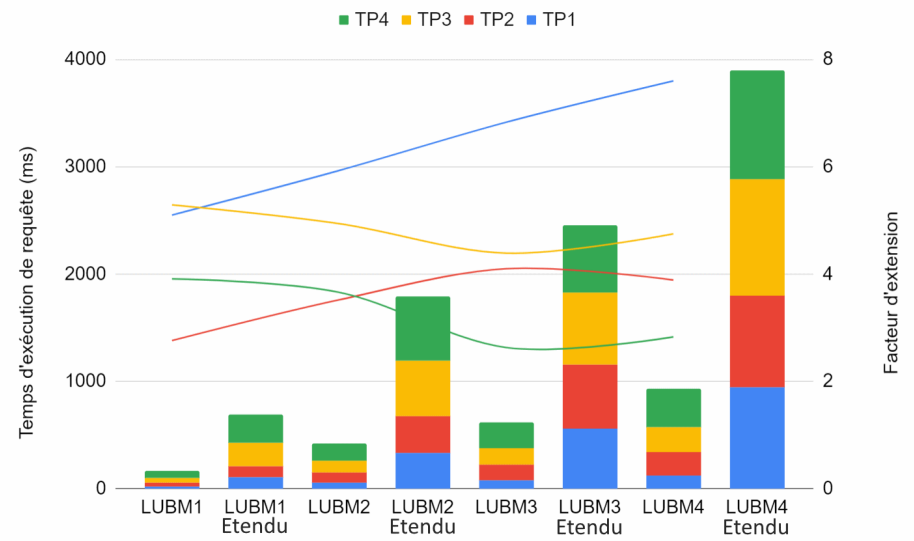

FIGURE 5.13 – Temps d'exécution des TPs de la requête 5.12 sur graphes de base et étendu

montre que le temps nécessaire pour étendre un graphe d'une certaine taille est comparable au temps qu'il aurait fallu si l'extension avait été incluse dans le graphe d'origine. Cela est confirmé par l'expérimentation présentée dans la figure 5.10. De plus, cette même figure met également en évidence que le temps nécessaire pour construire l'extension est également une fraction du temps nécessaire pour la charger sur le client. Enfin, nous pouvons observer que plus le graphe sur le client SE est grand, plus il faut de temps pour charger les données supplémentaires. Par exemple, en comparant les sous-graphes de 4% et de 17% en utilisant l'extension LUBM 1, il faut environ 6,5% de temps supplémentaire pour charger les mêmes données sur le sous-graphe plus grand.

5.5.6 Impact de l'extension de graphe sur le requêtage

Dans cette section, nous évaluons l'efficacité du requêtage des données contenues dans les parties étendues des graphes stockés sur les clients SE, pour cela nous comparons le temps de requêtage des quatre TPs de la requête de la Figure 5.12, lorsque toutes les données sont chargées dans le graphe à l'initialisation, avec le temps de requêtage de ces mêmes TPs lorsque les données sont chargées dans le graphe en extension d'un graphe original vide. On utilise pour jeux de donnée les LUBM 1, 2, 3 et 4 à partir desquels on a construit des sous-graphes par les prédicats contenus dans la requête de la Figure 5.12. Les extensions sont réalisées en ajoutant chaque triplet un à un pour simuler le pire cas pour les WTs et BVs de SE. En effet, l'ajout de grands volumes de triplets permettant généralement d'optimiser la création des BVs des feuilles et des noeuds internes des WTs.

Dans la Figure 5.13, les courbes représentent le facteur d'extension qui est le facteur entre le temps d'exécution du TP lorsque le graphe est étendu comparé à l'original. Ces courbes sont mesurées sur l'axe vertical droit. Les histogrammes de la figure représentent le temps d'exécution des TPs de la requête 5.12 avec les TP1, TP2, TP3 et TP4 correspondant aux 1, 2, 3 et 4 de la requête respectivement. Ces histogrammes sont mesurés avec l'axe vertical gauche. Les histogrammes nommés comme 'Étendu' représentent les temps pour les TPs sur portion étendue du graphe tandis que les autres représentent les temps sur portion originale. On remarque que le coût d'exécution des extensions varie selon les TPs entre 2 et 8 fois plus que dans le graphe original. Pour deux d'entre eux, les TPs 3 et 4, ce facteur baisse avec l'augmentation de la taille des jeux de données tandis qu'il augmente avec les TPs 1 et 2. Cela peut s'expliquer par le fait que lors de l'extension des WT de SE, certaines données sont principalement stockées en extension des noeuds internes du WT, tandis que d'autres sont stockées dans de nouveaux BVs comme décrits dans la section 5.2.3.

Puisque la portion étendue des WTs n'impacte pas les temps de requêtage sur la portion originale des WTs, nous pouvons calculer le temps d'exécution de la requête en fonction de quels TPs utilisent des données stockées en extension. Si l'on considère seulement un TP étendu au sein d'une requête, avec le pire des cas, en remplaçant le TP1 sur le LUBM4, nous ne faisons que doubler le temps d'exécution de la requête. Enfin, plus il y a de TP au sein d'une requête, moins le coût de ces extensions impacte le temps d'exécution des requêtes.

Le temps d'exécution d'un même TP sur la partie originale du graphe peut donc varier entre 2 et 8 fois plus selon comment il a été ajouté au WTs. C'est beaucoup moins que la différence de performance entre les opérations de rank et select sur les portions étendues ou non des BVs RRR constatée dans la section 5.5.2 qui allait jusqu'à plusieurs milliers de fois plus lentes pour les ranks. Ces performances correspondent à ce qui s'approche des pires conditions possibles. Cependant, elles restent acceptables pour l'utilisation que l'on peut prévoir pour notre système. Néanmoins, la large différence de rapidité d'exécution de requête en fonction des extensions de graphe implique que plus la part de requêtes du client contenant des TPs avec des portions étendues du graphe augmente, plus

il devient réaliste de reconstruire complètement les WTs et BVs contenus sur le clients, que ce soit pour optimiser l'utilisation de mémoire ou de CPU.

5.6 Conclusion

Dans ce chapitre, nous avons présenté un processus pour minimiser le coût mémoire de stockage des graphes de connaissances sur les appareils rencontrés dans un contexte de Edge computing en proposant une manière simple de décomposer le graphe et en utilisant de nouvelles représentations de structures de données succinctes, notamment des BV et des WTs, qui prennent en charge les opérations d'ajout.

À l'avenir, nous développerons de nouvelles façons de décomposer les graphes de connaissances en utilisant différentes stratégies pour réduire davantage la consommation de mémoire et/ou l'utilisation du CPU des systèmes de gestion de données sur les appareils Edge. En particulier, nous pourrions utiliser des stratégies qui permettent d'insérer des triplets en utilisant des prédicats déjà présents dans nos graphes. Le fait que l'interrogation des parties étendues du graphe et des extensions consomme plus de mémoire que les données BV d'origine soulève la question de savoir quand il est plus efficace de reconstruire complètement le graphe. Lorsqu'il devient nécessaire de reconstruire les WTs et BVs étendus, il serait possible de réutiliser les valeurs calculées dans la partie étendue du BV pour construire le nouveau BV. Cela concerne aussi bien les valeurs de `rank` précalculées à chaque SB que les valeurs représentant chaque bloc compressé du BV ainsi que les structures de `select`. Cela réduirait la construction du nouveau BV à une simple réécriture des données.

Les contributions mentionnées dans ce chapitre ont présentées sous forme d'un article à la conférence IEEE Big Data 2023.

Chapitre 6

Conclusion

6.1 Synthèse des contributions

6.1.1 Succinct Edge Continu

Gestion de flux de données

Les données collectées à partir des capteurs sont généralement considérées comme des flux de données continus. Ce constat nous a amené à définir une structure de données différente pour stocker et traiter les données des capteurs. Le stockage des triplets de SuccinctEdge représentant les données des capteurs a donc été séparé des autres données et connaissances, i.e., le graphe de connaissances. Nous y avons remplacé le WT de la couche objet car elle supporte difficilement la fréquence à laquelle les données peuvent être reçues. Nous associons chaque objet de ces triplets à une structure de type file d'attente pour stocker et regrouper les données du flux dans l'ordre où elles arrivent. Cela permet à SuccinctEdge de prendre en charge les requêtes sur le graphe dynamique et de traiter efficacement les requêtes sur les flux de données senseurs.

Étant donné qu'une même requête continue va être exécutée plusieurs fois avec pour seule différence la valeur des données senseurs dans la requête, nous

avons choisi de diviser chaque requête en parties dynamiques et statiques. Le résultat temporaire de la partie statique est maintenu, caché pendant l'exécution continue de la requête, tandis que la partie dynamique est recalculée avec chaque nouvelle valeur de donnée émise par un senseur. Les deux parties de la requête sont ensuite regroupées pour former la requête finale. Cette approche a un effet considérable sur le traitement des requêtes, car elle évite la répétition du calcul de la partie statique à chaque nouvelle valeur.

Extension SPARQL Continue

Pour interroger les flux de données en continu que représentent les mesures des capteurs sur les clients SE, une extension de SPARQL est nécessaire. Nous avons conçu notre extension SPARQL en nous inspirant de la syntaxe de C-SPARQL, qui est l'une des approches bien établies pour le requêtage continu basé sur le langage SPARQL. Cette extension prend en charge plusieurs fonctions d'agrégation sur des fenêtres de données en continu, les fenêtres sont définies par une durée et un mode de fenêtre. L'extension comporte deux modes de fenêtres, glissante et sautante, qui peuvent être utilisés pour différents cas d'usages. Notre solution supporte également les modes micro-batch et événement par événement. Ces extensions nous permettent de traiter la plupart de nos cas d'utilisation pour l'interrogation des flux de données RDF.

6.1.2 Succinct Edge Autonome

Generation de requête autonome

Afin de répondre aux anomalies sans intervention humaine après chaque détection, nous avions besoin d'un système capable de prendre en charge des anomalies de manière autonome. Nous avons proposé une méthode de génération de requête à partir de requêtes précédentes et d'une ontologie de gestion d'anomalie, capable de répondre rapidement aux anomalies tout en ciblant précisément les clients affectés. Durant son évaluation le système était capable de générer des requêtes en réponses aux anomalies dans la seconde où elles étaient détectées.

Extension de graphe et inférence par SHACL

Afin de créer un système capable de répondre à des anomalies en autonomie il est nécessaire de définir des règles pour encadrer le comportement du système. Nous avons donc proposé une utilisation du langage de règle SHACL pour permettre l'inférence de situations d'anomalies et de solutions à appliquer. Des enchaînements de règles permettent de représenter des situations plus complexes que l'on représente dans le graphe au sein de graphes nommés. Le fait que ces derniers soient associés à l'anomalie permet de facilement rétablir le contexte d'origine dès lors que l'anomalie n'est plus d'actualité.

6.1.3 Succinct Edge Adaptable

SDS extensibles

Les SDS offrent de très bonnes performances en termes de mémoire et de vitesse de requêtage mais leur immutabilité est très contraignante une fois appliquée dans SE, il est nécessaire de créer des structures mutables pour permettre l'ajout dynamique de données RDF à SE. Pour ce faire nous avons donc modifié les deux structures utilisées dans SE, les BV et les WT pour supporter des ajouts en fin de structure, le minimum nécessaire pour ajouter des données au système. Pour le WT, nous avons représenté chaque nœud du WT par un BV extensible, cela nous permet de réaliser un ajout sur n'importe quelle feuille de l'arbre, ainsi que d'ajouter de nouveaux nœuds à l'arbre sans jamais réécrire l'arbre. Pour le BV, nous nous sommes basés sur les BVs RRR [46] qui permettent d'effectuer des opérations de `rank` et `select` en temps constant pour une taille de bloc et superbloc données. Nous avons étendu la structure avec plusieurs dictionnaires servant à représenter les données étendues ainsi que les structures de `rank` et `select` tout en préservant les temps d'opérations constants sans réécrire tout le BV. Pour accompagner cette structure, nous avons aussi proposé une structure alternative à celle de RRR pour les opérations `select`, la nouvelle structure repose sur l'opération de `rank` RRR qui supporte bien la répartition de sa structure sur plusieurs dictionnaires.

Stratégies de réduction de graphe

Afin de réduire l'espace mémoire occupé par le graphe RDF sur les SE clients et d'optimiser l'exécution des opérations sur leurs SDS il est nécessaire de réduire la taille de ces graphes. Nous avons donc proposé deux stratégies de réduction de graphe basées sur les requêtes à exécuter afin de construire un sous-graphe capable de supporter l'exécution de ces requêtes. La première est basée sur les prédicats présents dans les requêtes adressées à un client, en créant une requête CONSTRUCT pour chaque prédicat présent dans les requêtes. Il est possible de créer un graphe capable de répondre à toutes requêtes contenant ces prédicats, facile à créer et qui sera toujours plus petit que le graphe original, à moins qu'il n'utilise tous les prédicats de la base de connaissances. L'autre méthode convertit directement le BGP des requêtes d'un client en requêtes CONSTRUCT, ce qui peut être répétitif si plusieurs requêtes sont similaires, mais produit un graphe généralement plus précis que l'autre stratégie.

6.2 Pistes de recherche pour l'avenir

6.2.1 Stratégies de réduction de graphe et reconstruction SE

La section 5.5.3 propose deux stratégies de réduction de graphe, une stratégie dirigée par prédicats qui produit des sous-graphes plus grands et une stratégie dirigée par BGP qui produit des sous-graphes plus petits mais plus lents à requêter. La stratégie la plus appropriée dépend du cas d'utilisation. Par exemple,

dans le cas où les machines utilisées n'ont que très peu d'espace mémoire, la stratégie BGP devrait être préférée. En revanche, dans le cas où il est plus important de répondre rapidement aux requêtes, il serait plus intéressant d'utiliser la stratégie des prédicats. Ce choix va aussi dépendre de l'ontologie utilisée dans les KG stockés, la stratégie par prédicat produisant des résultats très variés, *e.g.*, en termes de taille, selon les prédicats utilisés dans les requêtes comme dans le graphe. Quelle que soit la stratégie utilisée, aucun graphe qui a subi des extensions n'est aussi léger ou simple à requêter qu'un graphe que l'on aurait reconstruit pour intégrer ces extensions. La reconstruction du graphe dans SE étant l'action la plus coûteuse, en temps comme en calculs, il est important de déterminer à partir de quand il est plus efficace de reconstruire le graphe de SE, plutôt que de l'étendre.

6.2.2 Résumés de mesures

L'empreinte de mémoire limitée des machines en périphérie et l'exécution continue des requêtes font que la mémoire des hôtes des clients SuccinctEdge finie toujours par être saturée. Actuellement nous gérons ce problème en vidant périodiquement la mémoire pour toujours avoir de l'espace pour les nouvelles mesures. Cette suppression régulière de données nous limite dans notre étude de données n'ayant pas provoqué l'identification d'anomalies. Ces données pourraient pourtant être analysées sur des machines plus puissantes sur le Cloud ou le Fog, par exemple, pour découvrir des motifs dans les mesures qui précèdent les anomalies. Il serait aussi intéressant de pouvoir déterminer, à partir des données et requêtes connues, quelle stratégie devrait être utilisée pour la réduction du graphe de chaque client, toujours dans le but de minimiser la taille du graphe et les temps de calcul à leur niveau.

Pour éviter de transmettre toutes les mesures directement aux serveurs, il est nécessaire de stocker ces données au niveau du client temporairement, ou d'en réduire la quantité, par exemple en créant des résumés à partir de plusieurs mesures. Il est nécessaire de trouver une approche capable de compresser ces données à moindres coûts de calculs et d'espace de stockage pour les clients, tout en préservant un maximum d'informations sur ces données afin qu'elles puissent être traitées.

6.2.3 Requêtage analytique de graphe

Même si de nombreuses optimisations ont été développées dans SuccinctEdge autour de l'utilisation des SDS, nous pensons que dans une approche de requêtage analytique, de nouvelles optimisations, basées sur les SDS, sont possibles, en particulier dans le cadre de deux différents domaines, les requêtes cubes et les algorithmes de graphe.

Les requêtes cubes sont des requêtes à multiples dimensions et ont des applications sur des secteurs variés en fonction des dimensions utilisées, pouvant être aussi bien temporelles que géographiques, et sont pertinentes au cadre de

l'Edge Computing. Leur efficacité au sein d'un système comme SE est hautement dépendant des structures et indexes utilisés. L'utilisation des SDS pourrait se montrer efficace mais nous sommes ouverts à l'utilisation de nouvelles structures de stockage pour les parties statiques comme dynamiques de nos graphes, telles que les structures de données compactes[6].

En particulier nous avons remarqué qu'il serait nécessaire de maintenir des indexes PSO et POS pour répondre à certaines requêtes efficacement, les GROUP BY en SPARQL par exemple. Dans [40], l'auteur propose une solution dénommée Ring qui est une structure de données compacte dont la particularité est de maintenir simultanément six indexes (PSO, POS, OSP, OPS, SOP, SPO) en se basant sur des WTs mais également d'autres SDS, e.g., FM-index. Cette structure serait un remplacement potentiel à la structure de SE ou au moins un point de comparaison intéressant en termes d'espace et d'efficacité de requêtage pour ces types de requêtes.

Les SDS nous permettent de maintenir des graphes appropriés à l'Edge computing mais leur efficacité dans l'exécution d'algorithmes de graphes reste à évaluer. Ces algorithmes ont déjà été utilisés dans le cadre de SE dans le Section 5.3.5 avec l'utilisation des composantes connexes mais il existe d'autres cas d'utilisations potentielles avec les algorithmes du type page rank et et énumération de triangles.

Bibliographie

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In Proceedings of the 33rd International Conference on Very Large Data Bases 2007, pages 411–422, 2007.

[2] Yuan Ai, Mugen Peng, and Kecheng Zhang. Edge computing technologies for internet of things : a primer. Digital Communications and Networks, 4(2) :77 – 86, 2018.

[3] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J., 31(3) :1–26, 2022.

[4] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. Stream reasoning and complex event processing in ETALIS. Semantic Web, 3(4) :397–407, 2012.

[5] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. Federating queries in SPARQL 1.1 : Syntax, semantics and evaluation. J. Web Semant., 18(1) :1–17, 2013.

[6] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, SIGMOD '21 : International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pages 102–114. ACM, 2021.

[7] Kevin Asthon. That 'internet of things' thing. RFID, 22(7) :97 – 114, 2009.

[8] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. The Description Logic Handbook : Theory, Implementation and Applications. Cambridge University Press, USA, 2nd edition, 2010.

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying DDF streams with C-SPARQL. SIGMOD Rec., 39(1) :20–26, September 2010.

[10] Martin Z. Bazant, Ousmane Kodio, Alexander E. Cohen, Kasim Khan, Zongyu Gu, and John W.M. Bush. Monitoring carbon dioxide to quantify the risk of indoor airborne transmission of covid-19. Flow, 1 :E10, 2021.

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

[12] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. Journal of Web Semantics, 3(4) :247–267, 2005. World Wide Web Conference 2005 - Semantic Web Track.

[13] K. Mani Chandy and Michael Olson. Specifications and architectures of federated event-driven systems. In Intelligent Event Processing, Papers from the 2009 AAAI Spring Symposium, Technical Report SS-09-05, Stanford, California, USA, March 23-25, 2009, pages 21–26. AAAI, 2009.

[14] Olivier Curé and Guillaume Blin. An update strategy for the waterfowl RDF data store. In Matthew Horridge, Marco Rospocher, and Jacco van Ossenbruggen, editors, Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014, volume 1272 of CEUR Workshop Proceedings, pages 377–380. CEUR-WS.org, 2014.

[15] Olivier Curé and Guillaume Blin. RDF Database Systems : Triples Storage and SPARQL Query Processing. Morgan Kaufmann, 2015.

[16] Olivier Curé, Guillaume Blin, Dominique Revuz, and David Célestin Faye. Waterfowl : A compact, self-indexed and inference-enabled immutable RDF store. In ESWC, pages 302–316, 2014.

[17] Olivier Curé, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. Litemat : A scalable, cost-efficient inference encoding scheme for large RDF graphs. In 2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015, pages 1823–1830. IEEE Computer Society, 2015.

[18] Olivier Curé, Weiqin Xu, Hubert Naacke, and P. Calvez. Litemat, an encoding scheme with RDFS++ and multiple inheritance support. pages 269–284, 10 2019.

[19] Joffrey de Oliveira, Christophe Calle, Philippe Calvez, and Olivier Curé. Towards autonomous anomaly management using semantic technologies at the edge. In 2023 IEEE International Conference on Edge Computing and Communications (EDGE), pages 159–165, 2023.

[20] Joffrey de Oliveira, Christophe Callé, Weiqin Xu, Philippe Calvez, and Olivier Curé. Knowledge graph stream processing at the edge. In Yongluan Zhou, Panos K. Chrysanthis, Vincenzo Gulisano, and Eleni Tzirita Zacharatou, editors, 16th ACM International Conference on Distributed and Event-based Systems, DEBS 2022, Copenhagen, Denmark, June 27 - 30, 2022, pages 115–125. ACM, 2022.

[21] Joffrey de Oliveira, Hammad Aslam Khan, Olivier Curé, and Philippe Calvez. Semantic data transformation. In DeepOntoNLP/X-SENTIMENT@ ESWC, pages 39–46, 2021.

[22] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice : Plug and play with succinct data structures. In 13th International Symposium on Experimental Algorithms, (SEA 2014), pages 326–337, 2014.

[23] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2) :199–221, 1993.

[24] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM : A benchmark for OWL knowledge base systems. J. Web Semant., 3(2-3) :158–182, 2005.

[25] Stephen Harris and Nicholas Gibbins. 3store : Efficient bulk RDF storage. In Raphael Volz, Stefan Decker, and Isabel F. Cruz, editors, PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003, volume 89 of CEUR Workshop Proceedings. CEUR-WS.org, 2003.

[26] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. ACM Comput. Surv., 54(4) :71 :1–71 :37, 2022.

[27] Houda Khrouf, Badre Belabbess, Laurent Bihanic, Gabriel Képéklian, and Olivier Curé. WAVES : big data platform for real-time RDF stream processing. In Daniele Dell'Aglio, Emanuele Della Valle, Thomas Eiter, Markus Krötzsch, Maria Maleshkova, Ruben Verborgh, Federico M. Facca, and Michael Mrissa, editors, Joint Proceedings of the 3rd Stream Reasoning (SR 2016) and the 1st Semantic Web Technologies for the Internet of Things (SWIT 2016) workshops co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 17th - to - 18th, 2016, volume 1783 of CEUR Workshop Proceedings, pages 37–48. CEUR-WS.org, 2016.

[28] Danh Le-phuoc, Minh Dao-tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In ISWC 2011, pages 370–388. Springer, 2011.

[29] Anh Le-Tuan, Conor Hayes, Marcin Wylot, and Danh Le-Phuoc. RDF4Led : An RDF engine for lightweight edge devices. In Proceedings of the 8th International Conference on the Internet of Things, IOT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Anh Le-Tuan, Conor Hayes , Manfred Hauswirth, and Danh Le-Phuoc. Pushing the scalability of RDF engines on IoT edge devices. Sensors, 20(10) :2 :1–2 :8, 2020.

[31] Roger A Light. Mosquitto : server and client implementation of the MQTT protocol. The Journal of Open Source Software, 2(13) :265, 2017.

[32] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms (TALG), 4(3) :1–38, 2008.

[33] Ahlam Mallak, Ali Behravan, Christian Weber, Madjid Fathi, and Roman Obermaisser. A graph-based sensor fault detection and diagnosis for demand-controlled ventilation systems extracted from a semantic ontology. In 2018 IEEE 22nd International Conference on Intelligent Engineering Systems (INES), pages 000377–000382, 2018.

[34] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. Compression of RDF dictionaries. In Sascha Ossowski and Paola Lecca, editors, Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012, pages 340–347. ACM, 2012.

[35] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. Exchange and consumption of huge RDF data. In Elena Simperl, Philipp Cimiano, Axel Polleres, Óscar Corcho, and Valentina Presutti, editors, The Semantic Web : Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings, volume 7295 of Lecture Notes in Computer Science, pages 437–452. Springer, 2012.

[36] Iori Mitzutani, Ganesh Ramanathan, and Simon Mayer. Semantic data integration with devops to support engineering process of intelligent building automation systems. In Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, BuildSys '21, page 294–297, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and efficient minimal RDFS. Web Semant., 7(3) :220–234, September 2009.

[38] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal deductive systems for RDF. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, The Semantic Web : Research and Applications, pages 53–67, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[39] Gonzalo Navarro. Wavelet trees for all. J. Discrete Algorithms, 25 :2–20, 2014.

[40] Gonzalo Navarro. Compact data structures meet databases (invited talk). In Floris Geerts and Brecht Vandevoort, editors, 26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece, volume 255 of LIPIcs, pages 2 :1–2 :16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[41] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In Ralf Klasing, editor, Experimental Algorithms, pages 295–306, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[42] Thomas Neumann and Gerhard Weikum. RDF-3X : A risc-style engine for rdf. Proc. VLDB Endow., 1(1) :647–659, August 2008.

[43] Manh Nguyen Duc, Anh Lê Tuán, Jean-Paul Calbimonte, Manfred Hauswirth, and Danh Le Phuoc. Autonomous RDF stream processing for iot edge devices. In Semantic Technology - 9th Joint International Conference, JIST 2019, pages 304–319. Springer, 2019.

[44] Danh Le Phuoc, Minh Dao-Tran, Anh Lê Tuán, Manh Nguyen Duc, and Manfred Hauswirth. RDF stream processing with CQELS framework for real-time analysis. In Frank Eliassen and Roman Vitenberg, editors, Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015, pages 285–292. ACM, 2015.

[45] Nicola Prezza. A framework of dynamic data structures for string processing. ArXiv, abs/1701.07238, 2017.

[46] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, page 233–242, USA, 2002. Society for Industrial and Applied Mathematics.

[47] Mahadev Satyanarayanan. The emergence of edge computing. Computer, 50(1) :30–39, 2017.

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

[49] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. RDF data storage and query processing schemes : A survey. ACM Comput. Surv., 51(4) :84 :1–84 :36, 2018.

[50] Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudre-Mauroux. dipLODocus[RDF]—short and long-tail rdf analytics for massive webs of data. volume 7031, pages 778–793, 10 2011.

[51] Weiqin Xu, Olivier Curé, and Philippe Calvez. Knowledge graph management on the edge. In Proc. of the 24th International Conference on Extending Database Technology, EDBT 2021, pages 229–240, 2021.