Filtrage sémantique et gestion distribuée de flux de données massives

Amadou Fall Dia

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

SPÉCIALITÉ : INFORMATIQUE

« École doctorale Informatique, Télécommunications et Électronique (EDITE) »

réalisée à l'**Institut Supérieur d'Électronique de Paris (ISEP)**

présentée et soutenue publiquement par

Amadou Fall DIA

le 13 décembre 2018

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse

Filtrage sémantique et gestion distribuée de flux de données massives

---

Directeur de thèse :	**Mme Elisabeth Métais**
Co-encadrement de la thèse :	**Mme Zakia Kazi Aoul**
Co-encadrement de la thèse :	**M. Aliou Boly**

devant le jury composé de :

M. Laurent d'Orazio, Professeur Université de Rennes 1	Rapporteur
Mme Myriam Lamolle, Professeur Université Paris 8	Rapporteuse
M. François Goasdoué, Professeur Université de Rennes 1	Examinateur
M. Hubert Naacke, Professeur Université Paris 6	Examinateur
Mme. Maguelonne Teisseire, Professeur Irstea, UMR TETIS	Examinatrice

Résumé

Notre utilisation quotidienne de l'Internet et des technologies connexes génère, de manière continue et à des vitesses rapides et variables, de grandes quantités de données hétérogènes issues des réseaux de capteurs, des *logs* de moteurs de recherches génériques ou spécialisés, des données de sites de contenu multumédia, des données de mesure de stations météorologiques, de la géolocalisation, des applications IoT (l'Internet des objets), etc. Traiter de telles données dans les bases de données conventionnelles (Systèmes de Gestion de Bases de Données Relationnelles) peut être très coûteux en ressources temporelles et mémoires.

Pour répondre efficacement aux besoins et aider à la prise de décision, ces flots d'informations nécessitent des traitements en temps réel. Les Systèmes de Gestion de Flux de Données (SGFDs) posent et évaluent des requêtes sur les données récentes d'un flux dans des structures appelées fenêtre. Les données en entrée des SGFDs sont de différents formats bruts tels que CSV, XML, RSS, ou encore JSON. Ce verrou d'hétérogénéité émane de la nature des flux de données et doit être levé. Pour cela, plusieurs groupes de recherche ont bénéficié des avantages des technologies du web sémantique (RDF et SPARQL) en proposant des systèmes de traitement de flux de données RDF appelés RSPs. Cependant, la volumétrie des données, le débit d'entrée élevé, les requêtes concurrentes, le croisement des flux RDF à de larges volumes de données stockées et les calculs coûteux baissent considérablement les performances de ces systèmes. Il faut prévoir une nouvelle approche de réduction de la charge de traitement des flux de données RDF.

Dans cette thèse, nous proposons plusieurs solutions pour réduire la charge de traitement de flux de données en mode centralisé. Une approche d'échantillonnage à la volée de flux de graphes RDF est proposée afin de réduire la charge de données et du traitement des flux tout en préservant les liens sémantiques. Cette approche est approfondie en adoptant une méthode de résumé orienté graphe pour extraire des graphes RDF les informations les plus pertinentes en utilisant des mesures de centralité issues de l'Analyse des Réseaux Sociaux. Nous adoptons également un format compressé des données RDF et proposons une approche d'interrogation de données RDF compressées sans phase de décompression.

Pour assurer une gestion parallèle et distribuée des flux de données, le travail présenté propose deux solutions supplémentaires de réduction de la charge de traitement en mode distribué : un moteur de traitement parallèle et distribué de flux de graphes RDF et une approche de traitement optimisé des opérations de croisement entre données statiques et dynamiques sont présentés.

Abstract

Our daily use of the Internet and related technologies generates, at a rapid and variable speeds, large volumes of heterogeneous data issued from sensor networks, search engine logs, multimedia content sites, weather forecasting, geolocation, Internet of Things (IoT) applications, etc. Processing such data in conventional databases (Relational Database Management Systems) may be very expensive in terms of time and memory storage resources.

To effectively respond to the needs of rapid decision-making, these streams require real-time processing. Data Stream Management Systems (SGFDs) evaluate queries on the recent data of a stream within structures called windows. The input data are different formats such as CSV, XML, RSS, or JSON. This heterogeneity lock comes from the nature of the data streams and must be resolved. For this, several research groups have benefited from the advantages of semantic web technologies (RDF and SPARQL) by proposing RDF data streams processing systems called RSPs. However, large volumes of RDF data, high input streams, concurrent queries, combination of RDF streams and large volumes of stored RDF data and expensive processing drastically reduce the performance of these systems. A new approach is required to considerably reduce the processing load of RDF data streams.

In this thesis, we propose several complementary solutions to reduce the processing load in centralized environment. An on-the-fly RDF graphs streams sampling approach is proposed to reduce data and processing load while preserving semantic links. This approach is deepened by adopting a graph-oriented summary approach to extract the most relevant information from RDF graphs by using centrality measures issued from the Social Networks Analysis. We also adopt a compressed format of RDF data and propose an approach for querying compressed RDF data without decompression phase.

To ensure parallel and distributed data streams management, the presented work also proposes two solutions for reducing the processing load in distributed environment. An engine and parallel processing approaches and distributed RDF graphs streams. Finally, an optimized processing approach for static and dynamic data combination operations is also integrated into a new distributed RDF graphs streams management system.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

1.1 Motivations

Aujourd'hui, vivre sans la technologie est pratiquement impossible. Avec les innombrables outils technologiques, les utilisateurs produisent à chaque instant des données non quantifiables. D'une part, en considérant uniquement les médias sociaux, selon [5] [1], en une minute, 7 millions de snaps sont envoyés sur Snapchat, 216 millions de photos sont aimées sur Facebook, 2,4 millions de photos sont aimées sur Instagram, 350 000 tweets sont envoyés sur Twitter, 400 heures de vidéos sont téléchargées sur YouTube, 10 000 images sont épinglées sur Pinterest, 18 000 upvotes (vote positif) ou downvotes (vote négatif) sont réalisés sur Reddit, 1 million de vues sur Vine, 110 000 appels sur Skype, 70 millions de mots traduits sur Google Translate, 830 000 fichiers téléchargés sur Dropbox, 570 000 GIF visionnés, etc. D'autre part, partant du simple thermomètre visuel jusqu'au microscope électronique complexe ou même l'Œil d'observation d'un satellite, des avalanches de données sont générées. Ces dispositifs sont le plus souvent appelés capteurs. Par exemple, les stations d'observation météorologique (température, humidité, visibilité, etc.), la gestion du trafic routier (surveillance routière, congestion, assistance STI, "éco-conduites", etc.), le suivi à distance de patients (fréquence cardiaque, pression artérielle, etc.) ainsi que les réseaux de transports et de distribution d'eau (télérelevé de compteurs d'eau, débits, variation de pression, détection de fuites, etc.) génèrent sans cesse

1. Publié par Thomas Coëffé le 29 décembre 2016 sur le site `www.blogdumoderateur.com` connu pour ses statistiques hallucinantes sur l'utilisation des médias sociaux et de l'Internet en général dans le monde.

d'énormes quantités d'informations. Parmi ces informations, nombreuses sont celles qui nous parviennent sous forme de flux de données hétérogènes. La collecte et l'utilisation de ces données ont conduit à la considération de leurs caractéristiques qui sont entre autres la rapidité, la variété du rythme de génération, l'hétérogénéité et la continuité de ces données souvent appelées flux de données (data streams).

Pour répondre efficacement aux besoins des utilisateurs et aider à la prise de décision, ces flots infinis d'informations nécessitent des traitements en temps réel. Ainsi, leur gestion a suscité, ces dernières décennies, un grand intérêt dans la communauté des bases de données. Les Systèmes de Gestion de Bases de Données Relationnelles (SGBDR) ont très vite montré leurs limites pour la gestion des flux de données. En effet, comme montré dans [6, 7, 8, 9], les SGBDR sont inadaptés à la nature continue, à l'hétérogénéité ainsi qu'à la volumétrie des flux. Ainsi, la solution née pour pallier la déficience des SGBDR est l'utilisation de Systèmes de Gestion des Flux de Données (SGFD) ou DSMSs (Data Streams Management Systems) en anglais. Ces SGFD dont les plus connus sont Stream [10], TelegraphCQ [11] et Aurora [12, 13] posent des requêtes continues qui s'évaluent au fur et à mesure sur des sous-ensembles finis des données du flux. Ces sous ensembles du flux delimités par intervalle de temps ou par nombre d'items sont appelés **fenêtres**. Les SGFD, pour leur part, arrivent pleinement à assurer un traitement à la volée des données c'est-à-dire, s'assurent que les requêtes continues des utilisateurs (requêtes CQL[14]) ont une réponse en temps réel. Cependant, la nature hétérogène de ces données ainsi que la différence entre les SGFD devant les traiter et les échanger constituent un problème d'interopérabilité auquel la gestion efficace des flux de données s'est heurtée. Pour lever en partie ce verrou d'hétérogénéité et permettre une interopérabilité des flux de données issues de sources diverses et de formats variés (JSON, XML, CSV, RDF, etc.), la communauté des bases de données a initié l'adoption de technologies du Web Sémantique telles que le modèle de données **RDF**[2] (Ressource Description Framework), le langage de requête **SPARQL**[3] (Simple Protocol and RDF Query Language) et le langage de représentation des connaissances **OWL**[4] (Web Ontologie Langage) sur les flux de données. L'application de ces technologies sur les flux de données a favorisé des initiatives telles que le SSW[5] (Semantic Sensor Web) qui définissent un cadre d'intégration et de description de données de formats différents (données de capteurs, de météo, de réseaux sociaux, etc.) sous un modèle commun (RDF) donnant ainsi naissance aux flux de données sémantiques (ou flux RDF). Ainsi, pour un traitement en temps réel des flux de données sémantiques, un groupe de travail du W3C appelé RSP (RDF Stream Processing) a proposé différentes extensions de SPARQL autour de systèmes de traitement centralisé dont les principaux sont **Streaming SPARQL** [15], **C-SPARQL** [16], **SPARQL**$_{stream}$ [1], **EP-SPARQL** [17], **CQELS** [18] et **Sparkwave** [19]. Ces systèmes RSP sont capables de gérer à la volée des flux RDF mais le passage à l'échelle engendre une croissance exponentielle du temps de calcul des requêtes et une baisse considérable des performances entraînant même de nombreux blocages et des pannes des systèmes. C'est ainsi que dans une logique de passage à l'échelle avec de meilleurs temps de calcul de requêtes, les systèmes **CQELS Cloud** [20], **C-SPARQL on S4** [21], **Strider** [22] et le prototype **DIONYSUS** [23] ont été proposés comme principaux systèmes de traitement continu et distribué de flux RDF. Cependant, l'efficacité des approches utilisées dans ces systèmes distribués laisse à désirer en cas de débit élevé de flux RDF en entrée, de présence de larges volumes de données statiques, de requêtes concurrentes, d'enchaînement de requêtes (une requête ayant besoin de résultats de l'exécution

2. https ://www.w3.org/RDF/
3. https ://www.w3.org/TR/rdf-sparql-query/
4. https ://www.w3.org/OWL/
5. http ://knoesis.org/projects/ssw

d'une précédente pour s'exécuter) et de combinaison entre données statiques et dynamiques.

Le contexte de cette thèse s'inscrit ainsi dans le traitement optimisé de flux de données sémantiques autour de l'étude des systèmes de traitement continu existants. Nous apportons des solutions à la principale problématique soulevée par le traitement centralisé et distribué des flux de données RDF à très large échelle.

1.2 Problématique et défis

Dans [7], un flux de données est défini comme étant une séquence d'items continue, ordonnée, arrivant avec des débits souvent importants. Les systèmes actuels de traitement de flux de données RDF ont levé le verrou de variété en optant pour le modèle RDF.
Cependant, la principale problématique de ces systèmes réside dans la réduction de la charge de traitement des flux RDF tout en garantissant leur cohérence et en autorisant le croisement de données dynamiques et statiques. Cette problématique peut être concrètement associée à la charge trop élevée des données en entrée, à la structure ou au format des données RDF adoptés, au croisement entre données RDF statiques et dynamiques et aux techniques de partitionnement de requêtes adoptées.

1.2.1 Volume de données

Les systèmes informatiques, quels qu'ils soient, dans le domaine d'application des flux de données, sont en permanence confrontés à la rapidité d'arrivée des données potentiellement infinies. Le volume de données en entrée et à traiter par ces systèmes peut très vite exploser compliquant leur gestion. A l'évidence, cette charge de données entraine des baisses considérables des performances allant jusqu'au blocage voire la panne de ces systèmes. Ceci est notamment observé avec les systèmes de traitement centralisé de flux de données RDF (RSP) car disposant de ressources très limitées qui ne peuvent pas être allouées dynamiquement suivant la charge de données RDF. De plus, l'accumulation de données à l'entrée des systèmes RSP cibles, dûe à un temps moyen de traitement des données d'une fenêtre qui dépasse la fréquence d'arrivée des données d'un flux, entraîne la perte "non souhaitée" de l'information car la taille des buffers d'entrée ou des files d'attentes de ces systèmes n'est pas infinie. C'est ainsi que le premier défi de cette thèse réside dans la réduction de la charge de traitement centralisé disposant de ressources limitées ne pouvant être allouées dynamiquement suivant la charge de données RDF. Cependant, cette réduction de la charge nécessite une orientation sémantique, c'est-à-dire, considérant la structure orientée graphe des données RDF afin de préserver la cohérence des données.

1.2.2 Structure de données

La plupart des systèmes de gestion de données RDF (dynamiques ou statiques) modélisent encore les données RDF comme ensemble de triplets (la plus petite unité d'une donnée sémantique : sujet, prédicat et objet). Cette structure orientée triplet entraîne des coûts élevés dans le traitement des requêtes en raison d'un nombre excessif d'opérations de jointure qui produisent des résultats intermédiaires excessifs le plus souvent inutiles pour produire le résultat final de la requête. De plus, pour le traitement des requêtes SPARQL continues dans un environnement dynamique, cette approche ne prend en entrée que des successions de triplets, ignorant ainsi les liens sémantiques qui pourraient exister entre eux. Les triplets sont ensuite réunis jusqu'à l'obtention du résultat de

la requête. Cette procédure réduit les performances de la requête (opérations de jointure élevées à effectuer) et surcharge la mémoire (un nombre élevé de résultats intermédiaires provenant de nombreuses opérations de jointure entre les triplets). Ainsi, la plus petite unité d'une donnée sémantique devrait être orientée graphe afin de regrouper tous les triplets appartenant à un même événement dans un même graphe RDF.

1.2.3 Croisement de données statiques et dynamiques

C-SPARQL [16] et **CQELS Cloud** [20] croisent à la volée (au sein de chaque session de fenêtre) des données RDF dérivant de données statiques et dynamiques. De manière plus concrète, lors de l'opération de jointure entre données statiques et dynamiques, les systèmes chargent les données RDF statiques disponibles dans leur intégralité en mémoire locale ou sur disque ou bien y accèdent directement. Cette méthode est en train de devenir un système labyrinthique dû à la croissance exponentielle des bases de données RDF (appelées triple stores) contenant des millions de triplets RDF [24]. Cette phase importante est jusqu'ici négligée ou parfois oubliée par les systèmes existants mais peut avoir un impact négatif sur les performances de traitement des requêtes (goulot d'étranglement, surcharge, résultats incorrectes, incomplets ou incohérents).

1.2.4 Partitionnement de requête

L'une des approches pour résoudre le problème de la gestion de larges volumes de données RDF statiques et dynamiques consiste à partitionner les données RDF et/ou les requêtes SPARQL sur plusieurs nœuds de traitement. La plupart des méthodes de partitionnement de données RDF existantes se concentrent uniquement sur les données RDF statiques (stockage de manière permanente). Cela implique des opérations de partitionnement à froid ("offline") qui, pour la plupart, nécessitent de charger la totalité des données (des millions de triplets). L'application des mêmes techniques de partitionnement de données RDF statiques sur les flux RDF se traduit principalement par deux problèmes :

1. Le partitionnement des données RDF dans un contexte de croisement de données statiques et dynamiques risque de causer des blocages qui dégraderaient inévitablement la nature temps réel des systèmes. Indépendamment de la méthode de partitionnement utilisée dans les données RDF stockées, les flux de données RDF doivent être partitionnés à la volée à chaque entrée d'un item du flux (graphe RDF).
2. Les méthodes de partitionnement de données RDF existantes ne sont pas scalables dans la mesure où la quantité de résultats intermédiaires peut conduire à un grand nombre d'opérations de jointures qui sont souvent coûteuses et nécessitent des communications et une synchronisation fréquentes entre les différents nœuds de traitement.

En outre, les propositions de partitionnement de données RDF existantes nécessitent de multiples passages ($k-$partitioning avec $\mathcal{O}(n)$) sur l'ensemble des données statiques. Ces approches de partitionnement ne peuvent pas s'adapter à la nature dynamique des flux RDF. Les principales approches de partitionnement de données RDF statiques sont orientées hachage [25, 26, 27, 28, 29, 30, 31, 32] ou graphe [33, 34, 35, 36, 37, 38, 39]. Cependant, ces approches restent peu pratiques pour les flux RDF et presque impossibles à appliquer dans un cadre de croisement de données statiques et dynamiques.

Différentes pistes ont été étudiées et ont donné lieu à plusieurs défis :

Défi 1 : Réduire la charge de traitement des flux RDF avec une approche centralisée via l'échantillonnage.

Défi 2 : Réduire la charge de traitement des flux RDF avec une approche centralisée via le résumé.

Défi 3 : Réduire la charge de traitement des flux RDF avec une approche centralisée via la compression des données.

Défi 4 : Réduire la charge de traitement des flux RDF avec une approche distribuée via le partitionnement de données et de requêtes.

Défi 5 : Réduire la charge de traitement des flux RDF avec une approche distribuée via l'optimisation du croisement entre données.

Ces défis ont donné lieu à 5 contributions que nous décrivons brièvement dans ce qui suit.

1.3 Contributions

Le but de cette thèse est de proposer des solutions pour réduire la charge de traitement de flux de données sémantiques. En particulier, nous nous sommes orientés, dans un premier temps, vers une approche de gestion centralisée avec des solutions de réduction de la charge de traitement via l'échantillonnage, le résumé et la compression de flux RDF. Ces solutions sont destinées aux systèmes RSP centralisés en leur permettant d'optimiser leurs performances tout en garantissant la cohérence des données. Dans un second temps, nous nous sommes focalisés sur la gestion distribuée de flux de données RDF combinés à de larges volumes de données RDF statiques. Cette partie est composée de deux systèmes complémentaires implémentés de bout en bout et renfermant nos approches de traitement continu et distribué de flux de données RDF.

Nos contributions adoptent un format de données RDF orienté graphe et sont résumées en cinq (5) points décrits ci-après.

1.3.1 Échantillonnage et interrogation de flux de graphes RDF

Nous étendons le langage de requête et le système C-SPARQL pour l'échantillonnage à la volée de flux de données RDF. Dans le but de préserver la sémantique des données dans l'échantillon, nous adoptons une approche orientée graphe à la place des successions de triplets RDF utilisés par C-SPARQL. L'objectif de cette extension est d'une part l'ajout d'opérateurs d'échantillonnage dans la syntaxe de requête de C-SPARQL et d'autre part l'extension de ses modules continu et statique avec des algorithmes d'échantillonnage correspondants à ces opérateurs. En guise de validation de cette implémentation, nous choisissons les algorithmes d'échantillonnage **Uniform**, **Reservoir** et **Chain** correspondant respectivement aux opérateurs de requêtes **UNIFORM**, **RESERVOIR** et **CHAIN**.

Cette première contribution constitue un cas particulier de la réduction de la charge par un système RSP via de nouveaux opérateurs de requêtes, un module permettant l'implémentation de n'importe quelle technique de délestage en continu d'une partie des données en entrée tout en maintenant une cohérence entre les données conservées.

Cependant, avec l'échantillonnage orienté graphe, notre module d'extension arrive à réduire la charge de traitement mais ne permet pas d'extraire les informations les plus pertinentes du graphe. Nous avons, dans notre deuxième contribution, approfondi ce module (d'abord au sein d'un système RSP et plus tard dans un système que nous avons implémenté de bout en bout) avec un nouvel algorithme de résumé de flux de graphes RDF qui accorde plus d'importance aux nœuds et liens les plus "pertinents" du graphe RDF.

1.3.2 Résumé orienté graphe de flux de graphes RDF

Dans cette deuxième contribution, nous proposons une approche de résumé de flux de graphes RDF en étendant la mesure de centralité appelée **centralité de proximité** (**betweeness centrality**) issue de l'analyse des réseaux sociaux. L'objectif de cette contribution est de réduire la charge de traitement par l'exploitation de la richesse des graphes RDF afin de générer des résumés assez représentatifs des données. Avec l'extension de la mesure de centralité de proximité, nous concevons une analyse à la volée de graphes RDF en entrée afin d'en extraire, selon un pourcentage de résumé donné, les nœuds ainsi que les liens les plus informatifs avant de se délaisser du reste du graphe. De plus, cette approche de génération continue de résumés de graphes RDF est orientée besoin de l'utilisateur. En effet, elle combine les besoins exprimés par les utilisateurs (à travers les requêtes SPARQL continues) aux algorithmes étendus de mesure de centralité des éléments du graphe RDF.

1.3.3 Interrogation de flux de données RDF compressées au format RDSZ

Afin de pousser la qualité de la réduction de la charge en matière d'espace de stockage, nous adoptons dans cette troisième contribution un format compressé des flux de données RDF et proposons une approche de traitement de requêtes SPARQL continues sur données RDF compressées au format RDSZ[40] (un algorithme de compression de flux de données RDF). Pour les systèmes RSP ayant adopté ce format de compression de données, nous éliminons ainsi les phases assez complexes et coûteuses (en termes de ressource mémoire et de temps de calcul) de décompression suivi de traitement des requêtes sur données RDF. Nous exécutons directement les requêtes SPARQL sur un ensemble de données de taille beaucoup moins importante dans le but de gagner en performance en termes de temps de calcul de requêtes et d'espace de stockage.

Une fois la charge de traitement réduite, les systèmes de traitement de flux de données RDF nécessitent une montée en charge et des temps de latences ultra faibles pour faire face à la fois aux flux massifs de données RDF statiques et dynamiques. Ainsi, notre quatrième contribution initie nos approches de gestion continue et distribuée de flux de graphes RDF autour de notre première architecture distribuée DRSS (Distributed RDF SPARQL Streaming).

1.3.4 Système de traitement distribué de flux de graphes RDF (DRSS)

DRSS constitue à la fois notre premier langage et moteur de traitement de flux de graphes RDF implémenté de bout en bout au dessus de la plateforme de traitement temps réel Apache Storm. Nous avons d'abord proposé DRSS comme système socle avec un nouveau langage de requête continue et une nouvelle approche de distribution des données et du traitement.
Il s'agit d'abord de définir un langage de requête avec trois nouvelles variantes de fenêtrage (intégrant le maximum de contexte des flux de données) mais aussi une architecture distribuée composée d'une partie statique et d'une partie temps réel. La partie statique nous permet d'analyser les requêtes utilisateurs, regrouper celles qui partagent des sous-structures communes avant de les partitionner. L'un des avantages du partitionnement de la requête avant l'arrivée des flux RDF est d'adopter le partitionnement des graphes RDF statiques et dynamiques aux partitions de requêtes obtenues dans le but de distribuer et de dupliquer les partitions de graphes à destination de nœuds de traitement précis. Ce déploiement assez stratégique nous permet de réduire considérablement les communications entre nœuds de traitement pour une exécution locale et parallèle des partitions

de requêtes SPARQL continues.

Nous adoptons dans **DRSS** une approche de partitionnement de graphes de requête SPARQL centrée sur les nœuds de jointure autour desquels nous définissons les notions de pattern complet (full pattern) et pattern léger (light pattern). Les patterns complets constituent les centres des partitions de requête et sont, si besoin, dupliqués entre partitions nécessitant d'échanger des données pour compléter l'exécution d'une requête. Les patterns légers quant à eux, restent figés à leurs partitions et ne sont jamais dupliqués. Cette procédure nous permet de traiter toutes les partitions d'une requête parallèlement en échangeant peu ou pas de données entre les nœuds de traitement.

Les systèmes de traitement distribué de flux de données RDF existants sont moins performants en cas de croisement des flux RDF avec de larges volumes de données RDF statiques (stockées) dans des dépôts (bases de données orientées triplets) locaux ou distants. DRSS a apporté une première solution permettant de gérer l'importation, le croisement avec des données dynamiques et le rafraichissement des données statiques sans blocage dans le traitement. Cependant, dans la dernière contribution de cette thèse, nous avons approfondi l'approche de DRSS en nous intéressant au problème de jointure entre graphes RDF statiques et dynamiques.

1.3.5 Approche de jointure distribuée entre graphes RDF statiques et dynamique (JSS-RDF)

Notre dernière contribution, ajoute entre autres à la solution DRSS un module de gestion des données statiques et un module de jointure rapide entre données statiques et dynamiques. Nous proposons principalement dans **JSS-RDF** une approche de jointures continues et distribuées entre des données statiques et dynamiques en utilisant les filtres de Bloom. De cette approche de jointure, découle une grande réduction des résultats intermédiaires durant le processus de jointure et un pré calcul des partitions de requête en fonction des variables de requête SPARQL sélectionnées entre les deux natures (statiques et dynamiques) des données RDF. Nous réalisons un appariement en mémoire (si nécessaire) de chaque triplet de graphe entrant (côté flux) avec un triplet de graphe stocké. JSS-RDF assure l'opération de jointure continue sans interrogation de dépôts locaux ou distants durant l'exécution de la requête ainsi qu'un rafraichissement périodique (jour, semaine, mois, etc.) des données statiques. JSS-RDF est une solution complète et performante de gestion continue et distribuée de flux de graphes RDF statiques et dynamiques.

1.4 Jeu de données

Les évaluations de performances des différentes approches proposées sont réalisées en utilisation deux jeux de données réelles.

1. **Premier jeux de données** : données issues de capteurs déployés dans un réseau de transport et de distribution d'eau potable mesurant des phénomènes tels que la pression d'eau, le débit d'eau, le taux de chlore, la température de l'eau, etc.
2. **Second jeux de données** : données issues de capteurs déployés dans des stations météorologiques mesurant des phénomènes tels que la température, la visibilité, la vitesse du vent, l'humidité, etc.

1.5 Organisation du manuscrit

Ce manuscrit est structuré en 9 chapitres répartis en quatre parties :

1. Le chapitre 1 présente l'introduction de nos travaux.
2. Le chapitre 2 est un chapitre préliminaire où nous introduisons les concepts essentiels du Web Sémantique, les notions de flux de données sémantique et les différents systèmes de traitement centralisé de flux de données sémantiques existants. Nous clôturons ce chapitre par un bilan dressant les différentes performances et limites de ces systèmes suivant certains critères.
3. Le chapitre 3 décrit les approches de résumé de flux de données existantes ainsi que celles connexes à nos travaux de résumé de flux de données RDF. Plus précisément, nous présentons quelques méthodes de conception simple et complexe d'un résumé de flux de données et les deux principales approches de conception de résumé de flux de graphes RDF.
4. Le chapitre 4 présente notre première proposition de réduction de la charge par l'échantillonnage à la volée dans un système RSP. Dans ce chapitre, nous détaillons notre approche orientée graphes RDF ainsi que l'extension de la syntaxe et de l'architecture du système RSP choisi C-SPARQL pour l'intégration d'un module d'échantillonnage à la volée de flux de graphes RDF.
5. Le chapitre 5 fournit notre deuxième contribution de réduction de la charge de données à savoir le résumé de flux de graphes RDF. Nous y détaillons notre approche de résumé de flux de graphes RDF en étendant une mesure de centralité issue de l'analyse des réseaux sociaux.
6. Le chapitre 6 détaille les principales méthodes de compression de flux de données RDF (RDSZ [40] et ERI [41]) existantes, dresse un bilan comparatif de ces deux méthodes de compression et présente notre approche de traitement de requêtes SPARQL continues sur flux de données RDF compressées au format RDSZ. Nous y détaillons notre algorithme d'exécution de requêtes SPARQL simples et complexes sans phase de décompression.
7. Le chapitre 7 traite nos études et travaux sur la gestion distribuée de flux de données RDF. Nous y présentons les principaux systèmes de traitement distribués de flux de données RDF existants.
8. Le chapitre 8 présente notre système de traitement distribué de flux de graphes RDF (DRSS). Nous y détaillons notre langage de requête continue avec de nouvelles variantes de fenêtre ainsi que l'architecture de notre système.
9. Le chapitre 9 approfondit notre approche de traitement distribué de flux de graphes RDF. Nous y présentons une architecture distribuée incluant notre approche de jointure distribuée entre données RDF statiques et dynamiques en utilisant les filtres de Bloom.
10. Le chapitre 10 présente la conclusion et les perspectives de cette thèse.

Production scientifique

Conférences internationales
[42]. Chabchoub, Yousra, Zakia Kazi-Aoul, Amadou Fall Dia, and Rayane El Sibai. "ON THE DEPENDANCIES OF QUERIES EXECUTION TIME AND MEMORY CONSUMPTION IN C-SPARQL."

[43]. Dia, Amadou Fall, et al."C-SPARQL Extension for Sampling RDF Graphs Streams."

[44]. Dia, Amadou Fall, Zakia Kazi-Aoul, Aliou Boly, and Elisabeth Métais. "DRSS : Distributed RDF SPARQL Streaming."

[45]. Déme, Ndéye Bousso, Amadou Fall Dia, Aliou Boly, Zakia Kazi-Aoul, and Raja Chiky. "An Efficient Approach for Real-Time Processing of RDSZ-Based Compressed RDF Streams."

[46]. Dia, A. , Togbe, M. , Boly, A. , Aoul, Z. , Metais, E.. "Graph-Oriented Summary for Optimized Resource Description Framework Graphs Streams Processing."

[47]. Dia, Amadou Fall, Zakia Kazi Aoul, Aliou Boly, and Elisabeth Métais. "Fast SPARQL join processing between distributed streams and stored RDF graphs using bloom filters."

Conférence locale
[48]. Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly and Yousra Chabchoub, "Extension de C-SPARQL pour l'échantillonnage de flux de graphes RDF."

Journal
(**Soumis**) **[ASTESJ 2018]**. Dia, Amadou Fall, Zakia Kazi Aoul, Aliou Boly, and Elisabeth Métais. "Optimized SPARQL query Joins Processing between Large Distributed Streams and Stored RDF Graphs." Advances in Science, Technology and Engineering Systems Journal, no. 1 (2018).

Article en cours de rédaction
Dia, Amadou Fall , Zakia Kazi Aoul, Aliou Boly and Elisabeth Métais. "G^{ExP} : distributed RDF **G**raphs streams **E**xploration and **P**runing for continuous SPARQL querying."

Chapitre 2

État de l'art sur les systèmes et approches non distribués pour la gestion des flux de données RDF

Sommaire

2.1 Introduction

La tâche de gestion des flux de données RDF est un domaine largement étudié avec de nombreux travaux qui étendent principalement le langage de requête SPARQL. SPARQL est le langage d'interrogation de données RDF (Resource Description Framework). Cependant, il ne prend pas en compte les données arrivant sous forme de flux appelées aussi données dynamiques (c'est-à-dire, sans cesse mises à jour) telles que les flux de données RDF. Ainsi, pour combler ce manque, de nombreux travaux autour de la gestion de flux RDF se sont attelés à étendre le langage de requête SPARQL afin d'apporter les fonctionnalités requises pour l'interrogation de flux RDF. Ces extensions se regroupent autour des langages de requête et systèmes RSP (RDF Stream Processing) qui proposent des approches de traitement centralisé de flux RDF.

Ce chapitre est structuré comme suit. Nous exposons les travaux existants en apportant, tout d'abord quelques définitions sur les concepts essentiels du web sémantique (sous-section 2.2.1), sur les flux de données (sous-section 2.2.3) avant de faire une étude générale des principaux systèmes RSP centralisés (section 2.3) de par leurs spécificités au

niveau langage (syntaxe de requête) et système (architecture et fonctionnement). La section 2.4 conclut ce chapitre.

2.2 Concepts et définitions

2.2.1 Le web sémantique

Le web classique appelé web actuel dans certaines publications, présentait beaucoup d'insuffisances notamment quand à l'objectif principal de l'informatique qui est le traitement automatique des tâches par les machines. En effet, dans le web classique, beaucoup de tâches nécessitent une intervention humaine. L'exemple le plus connu est celui des moteurs de recherche où la recherche par mot-clé fournit des résultats assez imprécis. Il revient à l'utilisateur lui même d'interpréter et de combiner les résultats puis de lancer une nouvelle recherche plus fine ou d'extraire après interprétation des résultats, la réponse qui lui semble la plus juste car les données du web classique ne sont lisibles et compréhensibles que par les humains. L'objectif principal du web sémantique est d'orienter l'évolution du web actuel pour permettre aux utilisateurs de trouver, partager et combiner l'information plus facilement [1]. Ainsi, l'information sera lisible et compréhensible par les humains mais également par les machines. C'est d'ailleurs ce dont a parlé l'inventeur du World Wide Web Tim Berners-Lee :

> « ...I have a dream for the Web... In the second part of the dream, collaborations extend to computers. Machines become capable of analyzing all the data on the Web - the content, links, and transactions between people and computers. A ■Semantic Web", which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize... »[49]

Tim Berners-Lee émet donc l'idée d'un web sémantique (notion dont il fournit une définition beaucoup plus complète dans [50]) qui selon lui, facilitera la tâche à l'humain car les machines se chargeront de toutes les tâches nécessaires pour permettre à l'utilisateur un accès beaucoup plus rapide à l'information.

Le web sémantique a été normalisé par le W3C et regroupe plusieurs technologies et formats. La collecte, la structuration et la récupération des données reliées fonctionnent à l'aide des technologies qui fournissent une description formelle des concepts, termes et relations au sein d'un domaine de connaissances donné. Ces technologies standardisées [2], utilisées dans le web sémantique sont représentées dans un schéma appelé "semantic web layer cake" (figure 2.1). Dans nos travaux, nous exploitons une partie de ces technologies qui sont définies dans la suite.

2.2.1.1 URL/URI/IRI

Cette première couche représente les identifiants. Ces derniers permettent de repérer ou d'indexer une ressource spécifique dans le web. Sur la figure 2.1, la première couche contient uniquement l'URI et l'IRI mais pour des raisons de compréhension, nous définissons également l'URL.

1. https://en.wikipedia.org/wiki/Semantic_Web
2. https://www.w3.org/TR/owl-features/

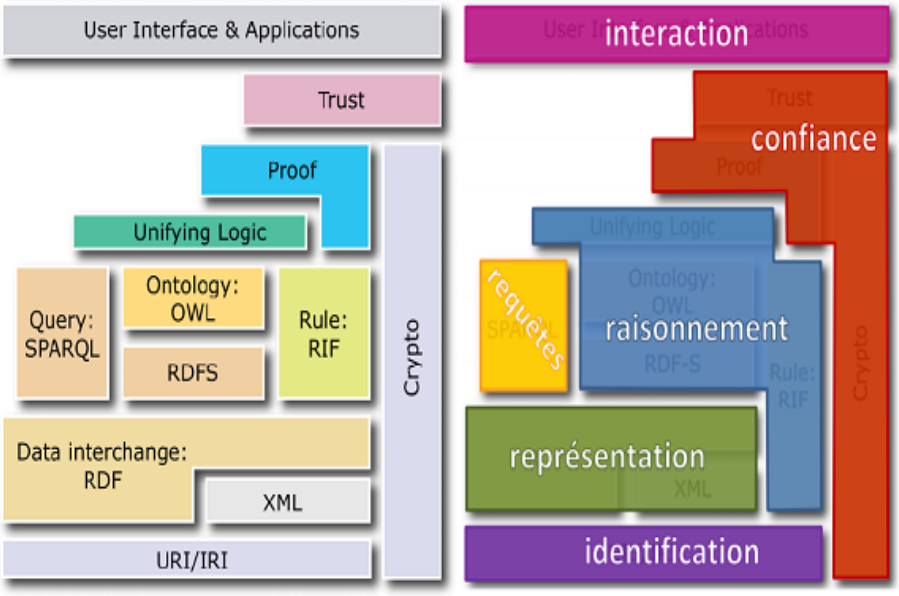


FIGURE 2.1 – Pile des standards du web sémantique.

2.2.1.1.1 URL (Universal Resource Locator)

L'Universal Resource Locator(URL) est une chaîne de caractères utilisée pour identifier de manière unique un document sur le web. Une URL combine les informations nécessaires pour indiquer à un logiciel comment accéder à une ressource Internet. Ces informations peuvent notamment comprendre le protocole de communication, le nom d'utilisateur, le mot de passe, l'adresse IP ou le nom de domaine, le numéro de port TCP/IP, le chemin d'accès et la requête. Les informations nécessaires varient selon la ressource et le contexte d'utilisation de l'URL. En outre un identificateur de fragment peut être ajouté à la fin d'une URL pour identifier un élément à l'intérieur de la ressource. `https://www.lemonde.fr/` est une URL qui pointe vers la page d'accueil du site web du journal Le monde.

2.2.1.1.2 URI (Universal Resource Identifier)

Un URI (Universal Resource Identifier) est une courte chaîne de caractères identifiant une ressource physique ou abstraite sur le web. Un URI doit permettre d'identifier une ressource de manière permanente, même si la ressource est déplacée ou supprimée. L'URI peut être subdivisé en deux sous catégories URL et URN (Universal Resource Name). Cette dernière permet d'identifier un document sur le web à partir de son nom[3] [51].

2.2.1.1.3 IRI (International Resource Identifier)

L'IRI[52] est un type d'adresse informatique prenant en compte les divers alphabets utilisés dans les différentes langues du monde. L'IRI est donc une généralisation et une internationalisation des URI. Contrairement aux adresses URL qui sont limitées au jeu de caractères ASCII (les nombres, les lettres de "a" à "z" sans accent ni autre diacritique mais éventuellement en majuscules, plus quelques signes), les IRI acceptent les milliers de caractères proposés par Unicode (alphabets arabe, latin, asiatiques, etc.), plus précisément en se basant généralement sur le codage UTF-8.

3. `https://www.w3.org/TR/uri-clarification/`

2.2.1.2 Le modèle RDF (Resource Description Framework)

Le Resource Description Framework (RDF) est le standard du W3C pour la représentation de données sémantiques. Il appartient à la couche "représentation" de la pile des standards du web sémantique (figure 2.1).

2.2.1.2.1 Généralités

RDF est le langage de base du Web sémantique. Il s'agit d'un modèle de graphe destiné à décrire de façon formelle les ressources Web et leurs méta-données. Son but est de permettre le traitement automatique de la description des ressources. Une donnée RDF est structurée en un triplet `<sujet, prédicat, objet>`. Chaque triplet permet de représenter une connaissance. Le "`sujet`" représente la ressource à décrire, le "`prédicat`" est la propriété applicable à cette ressource et l'"`objet`" donne la valeur de la propriété. Le sujet peut être identifié par une URI ou un nœud anonyme (défini dans la suite au 2.2.1.2.1.2). Le prédicat est nécessairement identifié par une URI. L'objet peut être un littéral (défini dans la suite au 2.2.1.2.1.1), identifié par une URI ou un nœud anonyme (défini dans la suite au 2.2.1.2.1.2).

2.2.1.2.1.1 Littéral

Un littéral est une valeur de base associée à des types de données comme une chaîne de caractères, une date, un entier, etc.

2.2.1.2.1.2 Ressource Anonyme

Une ressource anonyme ou un nœud anonyme (en anglais blank node ou bnode) est une ressource ou nœud d'un graphe RDF, qui n'est pas identifiée par une URI.

2.2.1.2.2 Syntaxe (Sérialisation)

Le modèle RDF dispose de standards ou formats de représentation (sérialisation) des triplets. Ces formats sont entre autres N-Triples, Turtle, RDF/XML, RDFa, Notation3. Ce dernier encore appelé N3 a été développé pour rendre plus facile la lecture et la compréhension des données RDF par l'humain. La syntaxe N3 requiert que chaque triplet RDF soit écrit obligatoirement dans l'ordre `sujet, prédicat, objet` et point `(.)` obligatoire à la fin de chaque triplet. Cette écriture peut être plus simplifiée en évitant des répétitions d'un même `sujet`. Ainsi, un point virgule `(;)` permet d'introduire un nouveau prédicat et un nouvel objet pour le même `sujet`. Une virgule `(,)` quant à elle permet de lister plusieurs objets pour un même couple (`sujet`, `prédicat`). L'exemple 2.2.1 fournit une représentation sous format N3 de données issues de capteurs déployés dans un réseau de transport et de distribution d'eau potable.

Exemple 2.2.1. Données RDF en format N3.

```
 sensorJ06 measure     resultJ06 ;
           designation AJ06 ;
           observation pressure ;
           inZone zoneId43H ;
           contains factoryAF4D .
resultJ06 type        pressure ;
           uom         mH2O ;
           value       37^^xsd:integer .
sensorJ07 measure     resultJ07 ;
           designation AJ07 ;
           type        pressure ;
```

```
            uom          mH2O ;
            value        29^^xsd:integer .
sensorJ02 measure      resultJ02 ;
            designation AJ02 ;
            type         pressure ;
            uom          mH2O ;
            value        31^^xsd:integer .
```

2.2.1.2.3 Graphe RDF

Les triplets RDF peuvent être représentés sous forme de graphe. Un ensemble de triplets est appelé un graphe RDF visualisé sous la forme de nœuds reliés entre eux par des liens dirigés (on parle de graphe orienté). Dans un graphe RDF, chaque triplet est représenté par un lien `nœud(sujet)-lien(prédicat)-nœud(objet)`. La figure 2.2 fournit le graphe RDF équivalent des données RDF dans l'exemple 2.2.1.

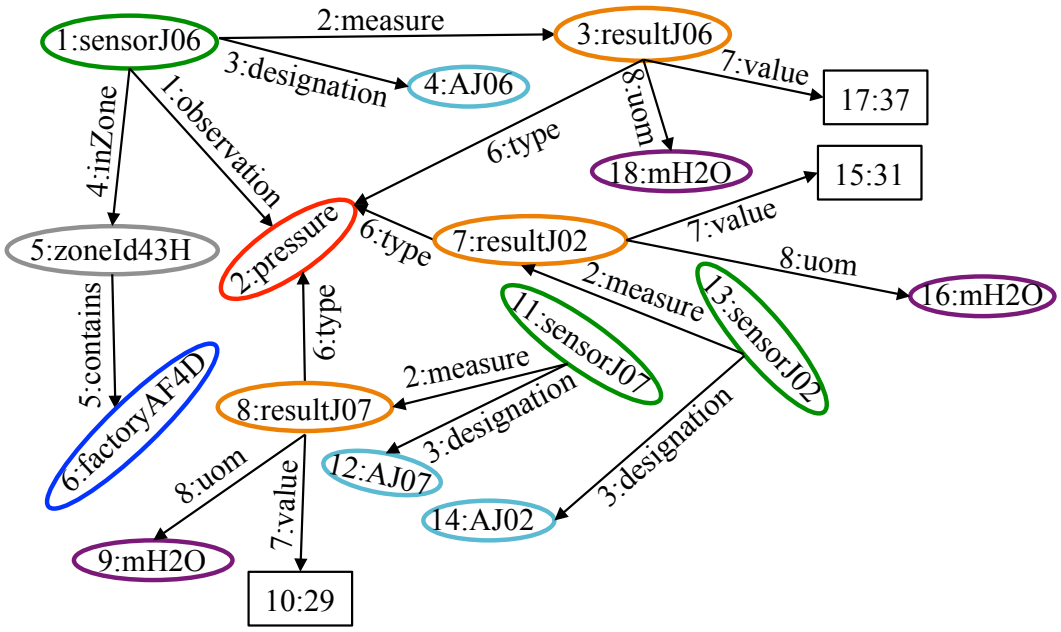


FIGURE 2.2 – Exemple de graphe RDF.

Un graphe RDF comporte trois (3) types de nœuds : les nœuds IRIs, les nœuds anonymes et les nœuds littéraux. Seuls les deux premiers sont susceptibles d'avoir des liens sortants.

Pour les applications des technologies du web sémantique, plusieurs APIs et triple-stores (bases de données orientées triplet) sont proposés. Leur but étant de faciliter la tâche aux développeurs pour la manipulation des données RDF. Les principaux triple-stores et/ou APIs sont RDF4J (ancien Sesame), Jena, Virtuoso, AllegroGraph, Soprano. Une étude comparative de ces triple-store est faite dans [53].

2.2.1.3 RDFS (RDF Schema)

RDF Schema ou RDFS (acronyme de Resource Description Framework) est un langage extensible de représentation des connaissances. Il appartient à la famille des standards du web sémantique publiés par le W3C. RDFS fournit des éléments de base pour la définition

d'ontologies ou vocabulaires destinés à structurer des ressources RDF accessibles grâce au langage de requête SPARQL (détaillé dans la sous-section 2.2.2).

— rdfs :Class permet de déclarer une ressource RDF comme une classe pour d'autres ressources ;

— rdfs :subClassOf permet de définir des hiérarchies de classes ;

— rdfs :domain définit la classe des sujets liée à une propriété ;

— rdfs :range définit la classe ou le type de données des valeurs de la propriété.

2.2.2 Interrogation de données RDF (langage de requête SPARQL)

SPARQL (SPARQL Protocol And RDF Query Language)[54, 55] représente le standard fourni par le W3C pour la couche **requêtes** de la pile des standards dans la figure 2.1. SPARQL joue à la fois le rôle de Protocole (Protocol) et celui de Langage d'interrogation (Query Language) qui ne s'applique qu'aux graphes RDF.
La fonction *Protocol* de SPARQL permet la transmission des requêtes SPARQL et résultats des requêtes SPARQL entre un client web et le endpoint [4] qui fournit des résultats sous divers formats : N3, RDF/XML,etc. Le Protocole SPARQL est une couche au dessus de HTTP où toutes les exigences HTTP pour les demandes et les réponses doivent être suivies par le protocole SPARQL. [5][56].

La fonction *Langage d'interrogation* de SPARQL est approximativement semblable au langage *SQL* des données relationnelles. Elle permet de rechercher, modifier, ajouter ou supprimer des données des graphes RDF. Deux grandes catégories de types de requêtes existent en SPARQL : les requêtes interrogatives et les requêtes constructives. Comme dans SQL, en SPARQL il existe la requête de type **SELECT** avec différentes clauses (**DISTINCT, WHERE, GROUP BY, ORDER BY, etc.**), différentes fonctions d'agrégation (**COUNT, SUM, AVG, MAX,MIN, etc.**). Une requête SELECT permet d'extraire du graphe RDF un sous-graphe correspondant à un ensemble de ressources vérifiant les conditions définies dans une clause WHERE. Une requête CONSTRUCT génère un nouveau graphe qui complète le graphe interrogé. Il y a également les fonctions **ASK** et **DESCRIBE**. Les requêtes SPARQL manipulent des triplets (sujet, prédicat, objet) appelés `pattern de triplet` ou `pattern` semblables aux triplets vus dans l'exemple 2.2.1 avec la seule différence que chaque élément du pattern peut être une `variable` précédée de `?` ou une une valeur précise (URI, litéral ou nœud anonyme). Le `BGP` (Basic Graph Pattern) désigne un ensemble de patterns d'une même clause WHERE. Dans SPARQL, on peut avoir huit (8) différentes compositions de patterns.

1. `sujet, prédicat, objet`
2. `?sujet, prédicat, objet`
3. `?sujet,?prédicat, objet`
4. `sujet, prédicat,?objet`
5. `sujet,?prédicat, objet`
6. `sujet,?prédicat,?objet`
7. `?sujet, prédicat,?objet`
8. `?sujet,?prédicat,?objet`

4. Un service ou point d'accès SPARQL
5. https://www.w3.org/TR/sparql11-protocol/

Le type de pattern 7 retournera tout le graphe interrogé car tous les composants sont des variables. Le type de pattern 1 sera très rare pour ne pas dire inexistant car il suppose que l'utilisateur qui écrit la requête connaît les valeurs de tous les composants : il n'y a donc pas lieu de faire une requête du genre. Ainsi, dans cette thèse, les types de patterns 1 et 7 ne seront pas exploités.

Le traitement des requêtes SPARQL se base le plus souvent, sur la valeur des prédicats. Néanmoins, dans le cadre de cette thèse, une requête avec des prédicats comme variables n'aura aucun impact négatif sur les opérations faites avec la requête SPARQL. L'exemple 2.2.2 et la figure 2.3 fournissent respectivement un exemple de requête SPARQL sur les données RDF de l'exemple 2.2.1 et son graphe de requête correspondant.

Exemple 2.2.2. Requête SPARQL sur les données RDF de l'exemple 2.2.1.

```
SELECT ?sensor ?value WHERE {
?zoneId contains factoryAF4D.
?sensor inZone ?zoneId.
?sensor observation pressure.
?sensor measure ?result.
?result type pressure.
?result value ?value.}
```

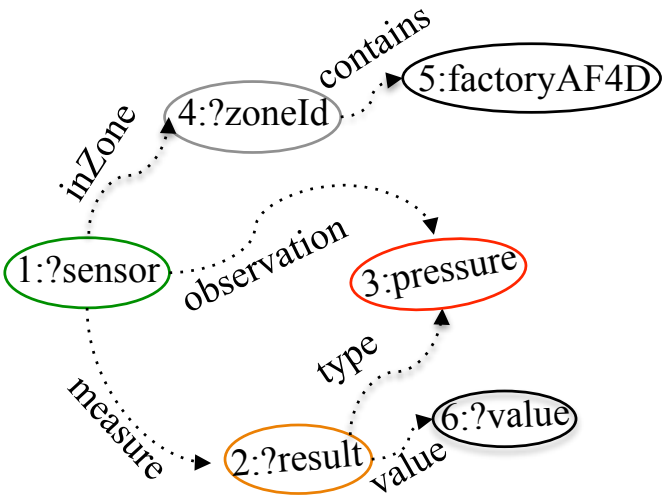


FIGURE 2.3 – Graphe de requête de l'exemple 2.2.2.

Dans la suite, nous présentons la syntaxe et la sémantique de SPARQL dans leur format algébrique.

2.2.2.1 Syntaxe

Soit $\mathbb{V}$ un ensemble $\{?x, ?y, ...\}$ infini de variables disjointes de $\mathbb{U}$, un pattern de triplet t_p peut être défini comme suit : $t_p = (s, p, o) \in (\mathbb{U} \cup \mathbb{B} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{B} \cup \mathbb{V})$. Un pattern de graphe P est alors défini comme une conjonction de patterns de triplets : $P = t_{p_1} \wedge t_{p_2} \wedge ... \wedge t_{p_n}$, avec $n \geq 0$. Les formats de requête SPARQL (SELECT, CONSTRUCT, ASK et DESCRIBE) utilisent des variables du BGP correspondant aux graphes RDF pour fournir des ensembles de résultats. Les expressions de SPARQL sont construites à partir du

BGP en utilisant les opérateurs . ou ; (AND), OPTIONAL (OPT), FILTER, GRAPH et UNION. Si P est un pattern de graphe et que C est une expression de condition construite dans SPARQL, les patterns de graphe de SPARQL sont définis de manière récurrente comme suit :

1. Un pattern de triplet est un pattern de graphe.
2. Les expressions (P_1 AND P_2), (P_1 OPT P_2), (P_1 UNION P_2) sont des patterns de graphe.
3. $\forall E \in (\mathbb{U} \cup \mathbb{V})$, (E GRAPH P) est un pattern de graphe.
4. L'expression (P FILTER C) est un pattern de graphe.

On note par $vars(t_p)$ l'ensemble des variables dans les triplets t_p et par $vars$(P) l'ensemble des variables apparaissant dans le pattern de graphe P. En suivant les définitions données dans [57], nous introduisons des terminologies communes à la syntaxe et à la sémantique du pattern de graphe de SPARQL.

Un appariement de solution μ est une fonction partielle d'un ensemble de variables de requêtes à l'ensemble des termes RDF valides.

$$\mu : \mathbb{V} \rightarrow (\mathbb{U} \cup \mathbb{B} \cup \mathbb{L})$$

$dom(\mu)$ désigne le domaine de μ et est l'ensemble de toutes les variables de $\mathbb{V}$ où μ est défini. $\mu(t_p)$ représente l'ensemble des triplets obtenus en remplaçant les variables de t_p selon μ. Nous affirmons que deux appariements μ_1 et μ_2 sont compatibles ($\mu_1 \sim \mu_2$), si $\forall$ $?x \in dom(\mu_1) \cap dom(\mu_2)$, alors $\mu_1(?x) = \mu_2(x)$. Intuitivement, si μ_1 et μ_2 sont compatibles, alors $\mu_1 \cup \mu_2$ est également un appariement. Rappelons que deux appariements avec des domaines disjoints sont toujours compatibles, et que l'appariement vide $\mu_\emptyset$ (c'est-à-dire l'appariement avec le domaine vide) est compatible avec tout autre appariement.

Étant donné un graphe RDF $\mathbb{G}$ et un pattern de triplet t_p, l'appariement de t_p sur $\mathbb{G}$ est un ensemble d'appariements μ tel que :

— (1) μ a pour domaine les variables de t_p ($dom(\mu) = vars(t)$)
et
— (2) μ t_p pour matcher le graphe $\mathbb{G}$ ($\mu(t_p) \in \mathbb{G}$).

Par exemple, considérons le graphe RDF suivant $\mathbb{G}$ stockant des triplets à propos des capteurs issus d'un réseau de transport et de distribution d'eau potable.

$\mathbb{G}$ ={(SensAJ12, measure, 0.5),
(SensAJ04, observat, pressure),
(pressure, uom, 1000.mH20) }

Ce qui suit fournit une expression de motif de graphe et ses évaluations sur $\mathbb{G}$ conformément aux définitions ci-dessus.

P ={((?S, measure, ?M)
OPT (?S, observat, ?O))
AND (?O, uom, ?U) }

— t_{p1} = (?S, $measure$, ?M), t_{p2} = (?S, $observat$, ?O) and t_{p3} = (?O, uom, ?U)
— $vars(t_{p1})$ = {?S, ?M}, $vars(t_{p2})$ = {?S, ?O} and $vars(t_{p3})$ = {?O, ?U}
— $dom(\mu_1)$ = {?S, ?M} and μ_1(?S) = SensAJ12, μ_1(?M) = 0.5
— $dom(\mu_2)$ = {?S, ?O} and μ_2(?S) = SensAJ04, μ_2(?0) = $pressure$
— $dom(\mu_3)$ = {?O, ?U} and μ_3(?O) = $pressure$, μ_3(?U) = 1000.mH20

	?S	?M	?O	?U
μ_1 :	SensAJ12	0.5		
μ_2 :	SensAJ04		pressure	
μ_3 :			pressure	1000.mH20
$\mu_1 \cup \mu_2$:	$\nsim$	$\nsim$	$\nsim$	$\nsim$
$\mu_1 \cup \mu_3$:	SensAJ12	0.5	pressure	1000.mH20
$\mu_2 \cup \mu_3$:	SensAJ04		pressure	1000.mH20

2.2.2.2 Sémantique

Nous définissons la sémantique des graphes SPARQL avec les conditions de jointure, union, différence, jointure externe gauche, graphe et filtre sur l'ensemble des appariements. Soient M, M_1 et M_2 des ensembles d'appariement. Les opérations binaires algébriques suivantes jointure $\bowtie$, union $\cup$, exception $-$ et jointure externe gauche $\mathbb{M}$ (également appelées Optional) sont définies entre M, M_1 et M_2.

L'évaluation des graphes P, P_1 et P_2 sur un graphe RDF donné $\mathbb{G}$, est une fonction notée $[\![.]\!]_{\mathbb{G}}$. La fonction renvoie également un ensemble d'appariements et est définie récursivement comme suit. Soit R une condition de filtre. La relation $\mu \models R$ indique que μ satisfait à R.

L'opérateur $\mathbb{M}$ est la représentation de l'opérateur OPT et permet d'ajouter des informations si elles sont disponibles (s'il y a une correspondance). Considérons l'expression du modèle de graphe (P_1 OPT P_2) et que P_1 soit un appariement dans $[\![P_1]\!]_{\mathbb{G}}$. S'il existe un appariement $\mu_2 \in [\![P_1]\!]_{\mathbb{G}}$ tel que μ_1 et μ_2 soient compatibles, alors $\mu_1 \cup \mu_2$ appartient à $[\![P_1 \text{ OPT } P_2]\!]_{\mathbb{G}}$. Cependant, s'il n'existe pas d'appariement μ_2, alors μ_1 appartient à $[\![P_1 \text{ OPT } P_2]\!]_{\mathbb{G}}$. L'opérateur OPT est associatif de gauche (et non associatif ou commutatif), réduisant ainsi le nombre de plans de requête que nous pouvons effectuer. Cet opérateur est fréquemment utilisé dans de nombreux systèmes de gestion de données RDF.
Les opérateurs AND et UNION sont à la fois associatifs et commutatifs, c'est-à-dire que les patterns de triplet et les jointures de réordonnancement ne modifient pas les résultats finaux.

2.2.3 Flux de données

Aujourd'hui, beaucoup de systèmes reçoivent de manière continue, de grandes quantités de données générées à grande vitesse. Ces données sont de différentes formes et de différents types.

2.2.3.1 Définition et domaines d'application des flux de données

Dans [7], un flux de données est défini comme étant une séquence d'éléments, ordonnée, générés de façon continue à un rythme rapide et arrivant avec des débits importants. La table 2.1 montre un exemple de flux de données issues d'un capteur d'identifiant `SensorID101` transmettant, toutes les cinq (5) minutes la valeur de la pression d'eau mesurée.

2.2.3.2 Domaine d'application des flux de données

Les flux de données se retrouvent dans plusieurs domaines d'applications extrêmement variés. Comme présenté dans [58], Les flux de données peuvent être divisés en deux grandes catégories.

— Flux de données transactionnelles ;

— Flux de données de mesure.

Estampille temporelle	CapteurID	Pression d'eau
...	...	...
2008 :12 :25 :10 :30 :00	SensorID101	"42.0"
2008 :12 :25 :10 :35 :00	SensorID101	"42.2"
2008 :12 :25 :10 :40 :00	SensorID101	"42.3"
2008 :12 :25 :10 :45 :00	SensorID101	"42.4"
2008 :12 :25 :10 :50 :00	SensorID101	"42.6"
...	...	...

TABLEAU 2.1 – Exemple de flux de données.

2.2.3.2.1 Flux de données transactionnelles

Les flux de données transactionnelles sont utilisés dans divers domaines. Nous pouvons citer entre autres, le domaine financier, le domaine de la télécommunication, et le domaine de l'Internet. Le domaine financier est l'une des plus grandes familles d'applications où le modèle de flux de données est transposable. On assiste de plus en plus, à une incapacité des ressources financières des entreprises à suivre l'explosion des données à gérer. Les entreprises ou plus généralement les organisations disposent de plus en plus d'outils permettant de répondre aux besoins de multiples transactions diverses et variées.

2.2.3.2.2 Flux de données de mesure

Ces flux de données de mesure résident principalement dans deux domaines que sont la surveillance des réseaux IP et les réseaux de capteurs. En raison de nos cas d'usage, nous présentons plus en détail les flux de données de réseaux de capteurs.

Réseaux de capteurs :

Les capteurs sont des dispositifs déployés dans l'environnement allant de la plus petite à la plus grande échelle et transmettent des informations sur divers types d'événements qui surviennent dans leur environnement de déploiement et qui peuvent intéresser diverses applications. On distingue différents types de capteurs : capteurs de température, d'humidité, de pression, de radiation, capteur visuel, etc. Ils sont souvent déployés dans des endroits où la présence humaine permanente est difficile voire impossible. Ces dispositifs souvent en miniature, transmettent des données continues et en flux. Ces données sont appelées des mesures. Ces dernières sont générées et envoyées périodiquement et sont la plupart du temps accompagnées d'autres types de valeurs paramétrées (estampille temporelle, valeur de localisation, fiabilité de la mesure effectuée, etc.). L'ensemble des valeurs transmises par un capteur est appelé flux de données.

2.2.3.3 Systèmes de Gestion des flux de données (SGFD)

Les bases de données relationnelles demeurent omniprésentes dans l'entreprise où, les applications nécessitent un stockage permanent des données et des interrogations sur ces données. Les données y sont stockées de manière statique et les opérations de mises à jour s'y effectuent moins fréquemment que les requêtes de sélection et de jointure de tables. Les requêtes sont exécutées à la demande des utilisateurs et le résultat fourni reflète l'état cohérent de la base. Ces bases de données sont sous l'implémentation des systèmes de gestion de base de données relationnels (SGBDR). Aujourd'hui, plusieurs applications nécessitent des traitements plus rapides des données afin de prendre des décisions ou réagir le plus rapidement possible. Du contrôle du trafic réseau à l'analyse des journaux transactionnels (transactions web, bancaires ou de télécommunication), ou encore la gestion des données de capteurs, la plupart des applications ont besoin de systèmes de gestion de données capables de gérer les flux continus en tenant compte des

critères qui font leurs différences avec les données classiques. Les systèmes de gestion de flux de données (SGFD) ou Data Stream Management Systems (DSMS) en anglais, ont été conçus pour répondre aux besoins de nombreuses applications c'est-à-dire interroger des données générées continuellement. A l'opposé des SGBD, les SGFD fonctionnent avec des données transitoires et des requêtes persistantes. En effet, nous étions habitués aux SGBD avec des données persistantes qui sont stockées à un endroit où elles seront accessibles à la demande de l'utilisateur et par la suite avec des requêtes transitoires qui, une fois exécutées n'existent plus. Ils existe des SGFD généralistes tels que **STREAM**, **TelegraphCQ**, **Aurora** et des SGFD spécifiques tels que **Gigascope et Hancock** [59], **NiagaraCQ** [60], **Cougar** [61] et **TinyDB** [62].

Le but d'un système de gestion de flux de données est de permettre à l'utilisateur de surveiller ou d'interroger un ou plusieurs flux de données via une série de requêtes, tout comme s'il interrogeait une base de données traditionnelle. Cependant, ces systèmes posent plusieurs contraintes dont nous en détaillons quelques unes.

— Le modèle de donnée et la sémantique des requêtes doivent permettre des opérations basées sur le temps (estampille) et l'ordre. Par exemple, obtenir la moyenne des températures observées durant les dix dernières minutes ;

— Dû aux contraintes de stockage, on ne peut revenir en arrière sur un flux datant. Les SGFD ne gèrent pas l'historique des données ;

— Les requêtes faites sur des flux continus n'utilisent pas d'opérateurs qui parcourent les données en entier pour retourner un résultat. Il faut utiliser des techniques de fenêtrage, qui, parfois peuvent omettre des événements de flux importants dans la requête ;

— Les requêtes entrainant un long traitement peuvent rencontrer des changements dans les conditions du système tout au long de leur exécution ;

— La charge générée par le grand nombre de données à traiter en quasi temps-réel devient encore plus importante avec l'augmentation du nombre de requêtes dans le système.

2.2.3.4 Notion de Fenêtre

Un flux de données est par définition continu et illimité dans le temps. Ainsi, les données sont produites en continu et sans limite en taille, en volume et en temps. Pour le traitement des données, il serait très difficile voir impossible de considérer les données dans leur globalité vu la continuité et l'infinité. Une solution est donc d'effectuer des traitements en intervalle de temps régulier ou en fonction d'une quantité fixe de données. Cette manière de procéder est appelé le fenêtrage (windowing en anglais). Il s'agit d'effectuer des traitements en fonction de fenêtres définies préalablement. Il existe deux grandes catégories de fenêtre que sont les fenêtres physiques basées sur le temps et les fenêtres logiques basées sur une quantité de données du flux.

2.2.3.4.1 Fenêtre physique

Les fenêtres physiques sont définies par des intervalles de dates. Il y a une date de début et une date de fin. En fonction des dates de début et de fin, on peut identifier trois (3) types de fenêtres physiques.

— **Fenêtre fixe :** constitue le type de fenêtre le plus simple dans la catégorie des fenêtres physiques. En effet, il s'agit d'une partie du flux de données dont le début et la fin sont précisés. Par exemple, une fenêtre dont la date de début est le 31/03/2018 à 19 :43 :00 et la date de fin est le 12/04/2018 à 10 :43 :00.

— **Point de repère :** cette fenêtre a une date de début bien précise mais la date de fin est la date courante (par exemple, une fenêtre entre le 31/03/2018 à 19 :43 :00 et la date courante). Il est important de noter que pour ce type de fenêtre, la taille augmente avec le déroulement du flux. La figure 2.4 montre un exemple illustrant le principe d'une fenêtre point de repère tout en montrant comment la taille (volume de données) augmente avec le temps.

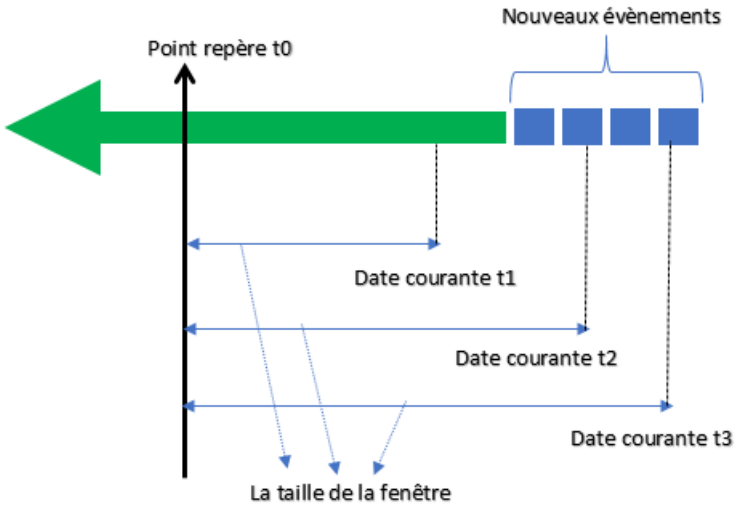


FIGURE 2.4 – Exemple de fenêtre point de repère.

— **Fenêtre glissante :** ce type de fenêtre a pour particularité d'avoir ses bornes (date de début et date de fin) qui sont relatives, c'est-à-dire évoluent avec le temps (par exemple, les cinq (5) derniers jours). Ainsi définie, on peut compter trois variantes de fenêtres glissantes où chaque variable se distingue par sa taille et son pas de décalage :

1. la fenêtre purement glissante (ou sliding window) dont le pas de décalage est inférieur à la taille de la fenêtre ;
2. la fenêtre sautante (ou jumping window) dont le pas de décalage est égal à la taille de la fenêtre ;
3. la fenêtre bondissante (ou hopping window) dont le pas de décalage est supérieur à la taille de la fenêtre.

La figure 2.5 montre les différends types de fenêtres glissantes tout en illustrant la différence au niveau de chacune d'elles.

2.2.3.4.2 Fenêtre logique

Les fenêtres logiques sont définies en termes du nombre d'éléments à considérer dans la fenêtre (par exemple, la fenêtre du 1er au 1000ème élément ou la fenêtre des 1000 derniers éléments). La taille d'une fenêtre logique est connue a priori, alors que celle d'une fenêtre physique ne l'est qu'à posteriori.

2.2.3.5 Les flux de données sémantiques

Les systèmes de gestion de flux de données (SGFD), permettent d'exécuter les requêtes sans interruption et renvoient de nouveaux résultats au fur et à mesure que de

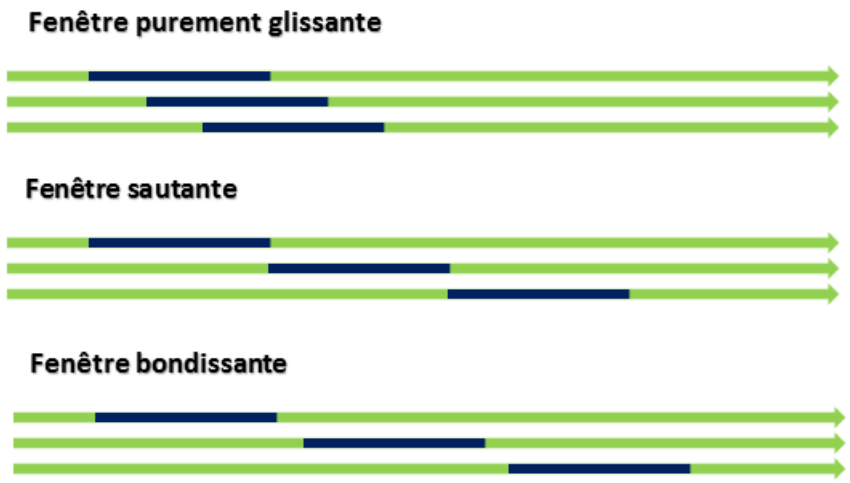


FIGURE 2.5 – Illustration des différents types de fenêtres glissantes.

nouvelles données arrivent. En même temps, le raisonnement sur les données représentées sous le format RDF avance rapidement, et SPARQL a gagné le rôle de langage d'interrogation standard pour ces données représentées en RDF.

Avec le récurrent problème de « grandes masses de données à manipuler et peu de connaissances extraites », les techniques existantes ne sont pas suffisantes pour traiter et extraire les données les plus informatives de la grande quantité de données disponibles. Il urge alors de combiner les techniques de représentation des données statiques au format RDF avec les caractéristiques et l'information contenue dans les flux afin de mener au raisonnement et à l'extraction d'informations pertinentes. C'est pourquoi, un nouveau concept appelé flux RDF (RDF Stream) a été défini. A l'image de RDF pour les données dites « statiques », un flux RDF est un nouveau type de donnée, et est une extension de RDF pour la représentation sémantique des données dynamiques. Comme décrit dans la partie des données classiques, RDF est un standard pour la représentation de données sous un format exprimant plus de sémantiques. Les triplets RDF sont étendus au modèle de flux de données RDF. Un flux RDF est défini comme étant une suite ordonnée de paires, où chaque paire est formée d'un triplé RDF et de son estampille temporelle τ.

$$
\begin{gathered}
(<\text{sujet}_1, \text{predicat}_1, \text{objet}_1>; \tau_1) \\
(<\text{sujet}_2, \text{predicat}_2, \text{objet}_2>; \tau_2) \\
\dots \\
(<\text{sujet}_i, \text{predicat}_i, \text{objet}_i>; \tau_i)
\end{gathered}
$$

L'estampille temporelle peut être considérée comme une annotation des triplets RDF, et est de façon monotone non décroissante entre les flux (τ_i et $\tau_i + 1$). Plus précisément, l'estampille n'est pas strictement croissante car elle n'est pas obligatoirement unique. Deux ou plusieurs triplets consécutifs peuvent avoir la même estampille, signifiant qu'ils se produisent en même temps, bien qu'ordonnancés selon un certain ordre de position. L'estampille apporte plus de performances aux requêtes. Cependant, SPARQL n'est pas assez outillé pour opérer des requêtes sur ce nouveau type de données. C'est ainsi que des extensions de SPARQL ont été proposées pour pallier ce manquement autour des systèmes RSP (RDF Stream Processing).

2.3 Systèmes RSP centralisés

Une classe d'approches proposées a émergé et est principalement orientée sur le concept de triplets estampillés (ou flux de graphes RDF), opérateurs de fenêtrage, requêtes continue, jointure de flux multi sources, jointure entre sources statique et dynamique, etc. Nous détaillons dans la suite les différents systèmes de traitement centralisé de flux de données RDF.

2.3.1 Streaming SPARQL

Steaming SPARQL [15] est l'une des premières propositions d'extension de SPARQL pour l'interrogation de flux de données RDF. L'approche proposée se limite exclusivement à une vue théorique. Aucune implémentation de Streaming SPARQL n'a été rendue disponible. En effet, l'objectif principal de l'approche est d'une part de fournir la sémantique appropriée des nouveaux opérateurs de streaming (fenêtrage, processus continu, etc.), et d'autre part, comment traduire les requête SPARQL en opérateur de streaming en étendant son algèbre. Cependant, cette extension se limite aux opérateurs de SPARQL 1.0 et n'intègre évidement pas les nouveautés observées dans SPARQL 1.1. Ainsi, les auteurs considèrent les flux de données comme de nouveaux types de données et omettent deux aspects importants sur la gestion de flux que sont les fonctions d'agrégation et temporelle. Avec ces limites, le langage proposé n'a pas la puissance requise pour répondre à des requêtes pertinentes dans beaucoup de domaines d'application (agrégation de données issues de réseaux de capteurs, données boursières, etc.).

Le langage proposé n'adopte pas l'approche utilisée dans les langages de requête continues utilisées dans les SGFD. Pour rappel, cette approche consiste à considérer les données d'une portion du flux appelée fenêtre pour y appliquer des opérations algébriques standards comme dans les bases de données relationnelles. En lieu et place, ils ont changé ces opérateurs (standards dans SPARQL) avec les leurs, en introduisant bien évidement une nouvelle sémantique. Du coté syntaxique, les opérateurs de fenêtrage peuvent apparaitre sur plusieurs endroits dans la requête les rendant ainsi plus flexibles et plus complexes. Malheureusement, les auteurs ne prêtent pas une attention particulière à l'impact négatif induit par la flexibilité de ce langage, sur les performances dans le traitement de la requête.

Syntaxe de requête Streaming SPARQL :

```
1.  SelectQuery → ?SELECT? ['DISTINCT' | 'REDUCED'] ?Var | '*'
                        DatasetClause* | DatastreamClause*
                        WhereClause SolutionModifier
2.  NamedGraphClause    → 'NAMED' SourceSelector
3.  DatastreamClause    → 'FROM' (DefaultStreamClause | NamedStreamClause)
4.  DefaultStreamClause → 'STREAM' SourceSelector Window
5.  NamedStreamClause   → 'NAMED' 'STREAM' SourceSelector Window
6.  SourceSelector      → IRIref
7.  GroupGraphPattern   → '{'TriplesBlock? ((GraphPatternNotTriples Filter) '.'?
                            TriplesBlock?)* (Window)? '}'
8.  Window → (SlidingDeltaWindow | SlidingTupelWindow | FixedWindow)
9.  SlidingDeltaWindow → 'WINDOW' 'RANGE' ValSpec 'SLIDE' (ValSpec)?
10. FixedWindow → 'WINDOW' 'RANGE' ValSpec 'FIXED'
11. SlidingTupelWindow → 'WINDOW' 'ELEMS' INTEGER
12. ValSpec → INTEGER Timeunit?
13. INTEGER → [0-9]+
14. Timeunit → ('MS' | 'S' | 'MINUTE' | 'HOUR' | 'DAY' | 'WEEK')
```

Aucun système implémentant Streaming SPARQL n'est à ce jour disponible rendant ainsi difficile son évaluation. Cependant les travaux de ces auteurs constituent un antécédent à un autre système (C-SPARQL) avec lequel il diffère en terme d'approche.

2.3.2 C-SPARQL

Continuous SPARQL ou C-SPARQL [16, 63, 64] figure également parmi les premières proposions d'extension continue de SPARQL et est citée comme référence dans ce domaine. Le système supporte l'exécution continue de requêtes sur flux RDF et est inspiré au niveau syntaxique par les langages de requêtes continues sur flux de données relationnelles comme CQL [14]. Dans ce qui suit, nous détaillons le langage de requête et l'architecture de C-SPARQL.

2.3.2.1 Langage de requête C-SPARQL

Les auteurs ont d'abord étendu la syntaxe de SPARQL 1.0 en introduisant de nouveaux opérateurs permettant de stocker une requête et de l'exécuter sur une portion du flux appelée (fenêtre) selon une fréquence d'exécution. Partant également des limites de Streaming SPARQL [65], les auteurs ont parfaitement introduit des opérateurs permettant la prise en compte de fonctions d'agrégation dans le langage C-SPARQL, d'interrogation à la fois de flux multiples et de donnés statiques. Dans une requête C-SPARQL, une fenêtre extrait les éléments les plus récents d'un flux soit en fonction du temps (portion du flux composée d'un nombre d'éléments se produisant à un intervalle donné) soit en fonction d'un nombre donné d'éléments (portion du flux composée d'un nombre fixé de triplets). Grâce à cet aspect, les données qui satisfont les conditions d'une fenêtre sont considérées comme statiques et interrogées à l'aide de requêtes SPARQL.

C-SPARQL étend la grammaire de SPARQL à travers de nouveaux éléments ajoutés au modèle de graphe de SPARQL. Nous détaillons dans ce qui suit les caractéristiques et fonctionnalités de chaque opérateur introduit dans [16].

Syntaxe de requête C-SPARQL :

```
1. FromStrClause → 'FROM' ['NAMED'] 'STREAM' StreamIRI '[RANGE' Window ']'
2. Window → LogicalWindow | PhysicalWindow
         LogicalWindow → Number TimeUnit WindowOverlap
              TimeUnit → 'd' | 'h' | 'm' | 's' | 'ms'
              WindowOverlap → 'STEP' Number TimeUnit | 'TUMBLING'
         PhysicalWindow → 'TRIPLES' Number
3. Registration→ 'REGISTER' ('QUERY' | 'STREAM') ['COMPUTED
                    EVERY' Number TimeUnit] 'AS' Query
4. AggregateClause → ('AGGREGATE {'('var', 'Function', 'Group') '[Filter]}')*
         Function → 'COUNT' | 'SUM' | 'AVG' | 'MIN' | 'MAX'
         Group → var | '{' var (',' var )* '}'
```

Notons que les clauses entre crochets sont optionnelles.

1. **FromStrClause** : cette fonctionnalité est identifiée par la clause FROM NAMED STREAM plus un URI (StreamIRI) qui identifie la source d'un flux. De plus, la configuration du fenêtrage d'un flux spécifique est aussi possible par la clause RANGE.

2. **Window** : étant donné que les flux RDF sont considérés comme infinis, C-SPARQL introduit la notion de fenêtre inspirée des langages de requêtes continues tels que CQL [14] . Une fenêtre peut être logique (LogicalWindow) ou physique (PhysicalWindow). La première est identifiée par une valeur temporelle (jours, heures, minutes, secondes ou millisecondes) sur la largeur de la fenêtre spécifiée par RANGE et sur la fréquence d'exécution du contenu de la fenêtre spécifiée par STEP ou TUMBLING. La seconde est identifiée par la clause TRIPLES suivie du nombre de triplets RDF autorisés dans la fenêtre.

3. **Registration** : cette nouvelle fonctionnalité s'inspire des SGFDs où la requête est stockée et sa fréquence d'exécution déterminée. Le résultat de l'exécution d'une requête est alors mis à jour par une nouvelle exécution et ainsi de suite. Cette fonctionnalité est applicable sur deux clauses différentes : la clause **QUERY** est utilisée pour les requêtes de type SELECT et ASK, et la clause **STREAM** est utilisée pour les requêtes de type CONSTRUCT et DESCRIBE produisant en sortie des triplets RDF qui, une fois estampillés, deviennent de nouveaux flux RDF. On peut enfin déterminer la fréquence d'exécution de la requête stockée avec la clause **COMPUTED EVERY**. Cette dernière option constitue une limite dans le modèle de traitement de C-SPARQL. En effet, la spécification de la fréquence d'exécution de la requête par l'utilisateur ne permet pas un rafraîchissement automatique des résultats stockés (c'est-à-dire une réexécution de la requête déclenchée seulement par l'arrivée de nouveaux flux).
4. **AggregateClause** : l'agrégation est un besoin primordial dans beaucoup de domaines d'application des flux de données. N'étant pas prise en compte dans SPARQL 1.0, les auteurs incluent dans [16] leurs propres opérateurs permettant l'utilisation de fonctions d'agrégation. C-SPARQL permet plusieurs agrégats indépendants au sein d'une même requête où chacun est composé des parties suivantes : une clause d'agrégation qui débute toujours par une variable ne figurant pas dans la clause WHERE, suivie par une fonction d'agrégation (COUNT, MAX, MIN, SUM, AVG) et bouclée par une ou plusieurs variables figurant dans la clause WHERE servant ainsi de critère de groupement. Notons que ces fonctions d'agrégation furent abandonnées après l'introduction de SPARQL 1.1 dans [63]. Le système a ainsi commuté pour

s'adapter au mieux aux fonctions d'agrégation fournies par défaut dans SPARQL 1.1. En plus des fonctions d'agrégation, C-SPARQL ajoute la fonction `timestamp()`. Cette fonction permet de manipuler des données temporelles contenues dans les flux. Elle est d'une grande utilité dans les requêtes complexes (filtre, comparaison, etc.).

La requête 2.1 présente un exemple simple de requête C-SPARQL pour obtenir l'identifiant de capteur et la valeur de pression de l'eau mesurée sur des données issues de deux flux différents et d'un dépôt (repository) stockant des données statiques (comme les données de localisation (`zoneId`)).

```
SELECT ?sensorId  ?waterPressureValue
FROM STREAM <http://ssn.ex/streamsrc1>  [RANGE 10s STEP 1s]
FROM STREAM <http://ssn.ex/streamsrc2>  [RANGE 5s STEP 1s]
FROM <http://ssn.ex/staticsrc>
WHERE {
        ?zoneId  contains factoryAF4D.
        ?sensorId  inZone  ?zoneId.
        ?sensorId  observation pressure.
        ?sensorId  measure ?result.
        ?result   type  pressure.
        ?result   value  ?waterPressureValue.}
```

REQUÊTE 2.1 – Exemple de requête C-SPARQL.

2.3.2.2 Système C-SPARQL

L'implémentation d'un système permettant d'exécuter des requêtes C-SPARQL sur flux RDF repose sur trois (3) clés principales :

— prise en compte de flux RDF estampillés (triplets RDF plus timestamp) ;

— requêtes continues sur flux RDF ;

— et introduction d'opérateurs explicites pour des fonctions d'agrégation et temporelles.

C-SPARQL adopte le concept de fenêtrage utilisé dans les SGFDs. En évaluant une requête, C-SPARQL est capable de traiter (et combiner) à la fois des données statiques (triplets RDF sans estampille) et des flux RDF.

C-SPARQL est un moteur fondé sur une approche logique stipulant que toute information peut être divisée en connaissance et donnée. Chaque requête C-SPARQL peut donc être divisée en deux parties : une partie statique traitant la connaissance (la partie sémantique) et une partie dynamique traitant les données (la partie flux). Cette nouvelle considération trouve une parfaite adéquation à la réutilisation des techniques de traitement de flux de données existantes dans le système C-SPARQL. L'implémentation du moteur se base donc sur une architecture composée de deux principaux sous-systèmes : STREAM[10] ou Esper [6] comme SGFD et RDF4J [7](ex Sesame) ou Jena [8] comme moteur de requêtes SPARQL. La figure 2.6 montre l'architecture du moteur C-SPARQL et met en

6. http ://www.espertech.com/esper/
7. http ://rdf4j.org/
8. https ://jena.apache.org/

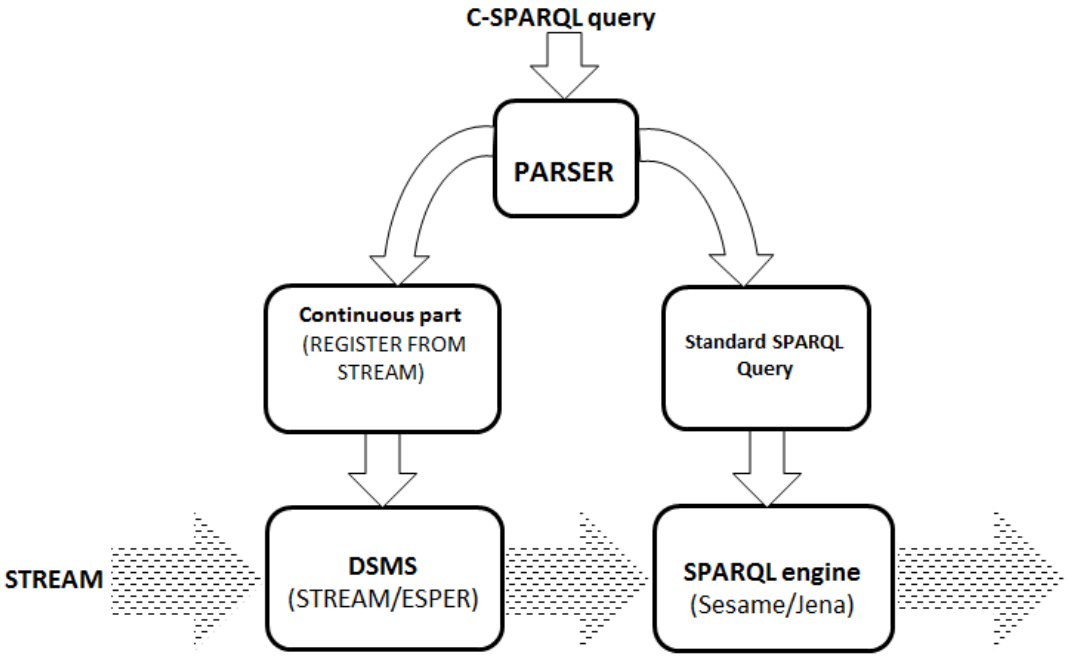


FIGURE 2.6 – Architecture de C-SPARQL.

évidence un module important qu'est le Parser. Ce dernier reçoit en entrée une requête C-SPARQL, l'analyse, la transforme et produit en sortie l'information dont a besoin la couche SGFD et moteur SPARQL (parties CQL et SPARQL). Le SGFD se charge d'abord de stocker temporairement les données du flux spécifié dans la requête en y appliquant une fenêtre logique ou physique. Une fois le graphe résultant est généré, la partie SPARQL de la requête est exécutée par le moteur SPARQL. Cette procédure est répétée autant de fois qu'il sera nécessaire (c'est-à-dire suivant la valeur indiquée de la fréquence d'exécution de la requête C-SPARQL. Finalement le résultat retourné est annoté et transmis.

C-SPARQL est considéré comme une "boîte noire" (black box) et délègue l'exécution de la requête à d'autres systèmes. Les parties "statique" et "dynamique" de la requête C-SPARQL ont besoin d'être transformées respectivement en SPARQL standard et en CQL. La puissance de C-SPARQL réside dans le fait qu'il réutilise des systèmes existants développés, testés et même exploités. Cependant cette délégation de tâches ainsi que la transformation des flux RDF et les requêtes C-SPARQL constituent une véritable limite de ce système. En effet, cette approche ne fournit guère un contrôle absolu ni sur le plan d'exécution de la requête ni sur l'implémentation du système ou de ses composants. Ceci rend difficile l'optimisation des requêtes. Dans l'architecture de C-SPARQL, la couche SGFD et la couche moteur SPARQL sont considérées comme des plugins indépendants et donc peuvent être remplacés par d'autres SGFDs ou moteurs SPARQL existants. C'est ainsi que dans [66], les auteurs ont implémenté un prototype de C-SPARQL sur les requêtes continues et une analyse en temps réel des données RDF issues de médias sociaux. Leur architecture utilise Esper comme SGFD et Jena comme moteur de requête SPARQL. Les auteurs évaluent les temps d'exécution d'une requête C-SPARQL sur les deux systèmes (un système utilisant le couple Stream-Sesame à une autre utilisant le couple Esper-Jena) dans deux (2) environnements identiques. Cette évaluation fournit des résultats plus probants en terme de temps d'exécution et d'output avec le couple Esper-Jena. Cependant, l'utilisation de ce couple n'influe guère sur les optimisations attendues.

2.3.3 CQELS

CQELS (Continuous Query Evaluation over Linked Streams) [18] représente également une autre proposition d'extension de SPARQL pour le traitement de flux de données sémantiques. Sa grammaire est définie à partir de celle de SPARQL 1.1. Le système utilise une approche native complètement différente de celles déjà vues (notamment avec C-SPARQL). Nous présentons dans la suite la syntaxe de requête et l'architecture de CQELS.

2.3.3.1 Langage CQELS

La syntaxe du langage présente plusieurs fonctionnalités dont nous détaillons chaque partie dans ce qui suit.

Syntaxe de requête CQELS :

1. **GraphPatternNotTriples** → GroupOrUnionGraphPattern | MinusGraphPattern
| GraphGraphPattern | StreamGraphPattern
| ServiceGraphPattern | Filter | Bind

2. **StreamGraphPattern** → STREAM [Window] VarOrIRIref {Triples Template}
Window → Range | Triple | NOW | ALL
Range → RANGE Duration (SLIDE Duration)
Triple → TRIPLES INTEGER
Duration → (INTEGER 'd' | 'h' | 'm' | 's' | 'ms' | 'ns')+

CQELS ajoute un nouvel élément à la définition du modèle de graphe de SPARQL. Le **StreamGraphPattern** est donné dans la requête CQELS par le terme clé STREAM et contient la configuration de la fenêtre employée dans la requête, l'URI identifiant la source du flux RDF (VarOrIRIref) et un modèle de triplet standard (TriplesTemplate). Il y a deux types de configuration dans le fenêtrage avec CQELS : le premier est temporel et est spécifié en largeur par le terme RANGE et en fréquence par STEP. Le second est donné par TRIPLE suivi par le nombre de triplets donnés ou autorisés dans la fenêtre.

La requête 2.2 décrit le même cas d'usage de la requête donnée précédemment sur deux sources de flux différentes.

```
SELECT ?sensorId1  ?waterPressureValue1
                          ?sensorId2 ?waterPressureValue2
WHERE {
  STREAM <http://ssn.ex/streamsrc1> [RANGE 10s STEP 1s]
         { ?sensorId1 observation pressure.
           ?sensorId1 measure ?result1.
           ?result1 type pressure.
          ?result1 value ?waterPressureValue1}
 STREAM <http://ssn.ex/streamsrc2> [RANGE 5s STEP 1s]
         { ?sensorId2 observation pressure.
           ?sensorId2 measure ?result.
           ?result 2 type pressure.
           ?result2 value ?waterPressureValue2.} }
```

REQUÊTE 2.2 – Exemple de requête CQELS.

CQELS réutilise toutes les clauses dans une requête SPARQL. L'exploitation de cette capacité lui permet d'exprimer des requêtes relatives à des besoins plus complexes.

2.3.3.2 Système CQELS

CQELS est le seul système à utiliser une approche native. Il fournit ses propres opérateurs et son propre modèle de traitement. C'est une architecture logicielle combinant le traitement de données statiques et de flux de données liées (flux RDF). En commun avec C-SPARQL, il s'inspire du modèle de traitement des DSMSs en utilisant un ensemble d'opérateurs algébriques relationnels sur une portion du flux (fenêtre). Mais, contrairement à C-SPARQL, CQELS fournit un modèle de traitement où la fréquence d'exécution de la requête n'est pas périodique mais déclenchée par l'arrivée de nouveaux triplets. La figure 2.7 montre une vue d'ensemble de l'enchaînement dans le traitement d'une requête par le système CQELS.

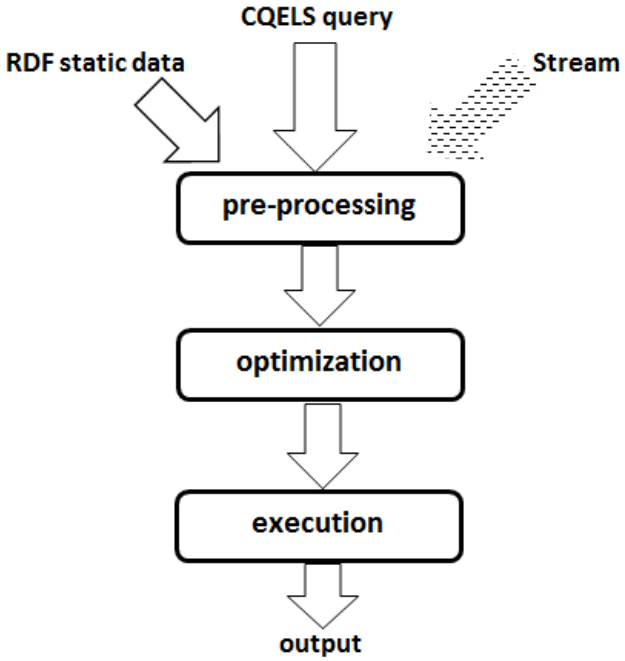


FIGURE 2.7 – Modèle de traitement de CQELS.

Le système prend en input trois types de données : données statiques (triplets RDF sans annotation temporelle), flux de données et requêtes CQELS. Ces entrées sont ensuite soumises au modèle de traitement en trois principales étapes (cf. figure 2.7) :

— **pre-processing** : il s'occupe de l'optimisation du stockage et de la représentation des données. En effet, il transforme l'information dans un format compact rendant facile le chargement de grandes quantités de données lors de l'exécution de la requête. Il se charge également du stockage des résultats intermédiaires des requêtes.

— **optimization** : son rôle est de choisir le plan d'exécution de la requête le moins couteux. L'optimiseur choisit entre un ensemble de plans d'exécution pré-calculés.

— **execution** : venant en dernière position, il se charge d'exécuter la requête et retourner le résultat.

Pour l'exécution d'une requête sur flux de données, le système CQELS utilise trois groupes d'opérateurs que nous détaillons comme suit :

— **opérateurs de fenêtrage** : considère les éléments contenus dans une portion du flux entrant.

— **opérateurs relationnels** : ces opérateurs opèrent sur des données finies (contenues dans une fenêtre) afin d'y appliquer des traitements relationnels à l'image des bases de données relationnelles

— **opérateurs de streaming** : similaires aux opérateurs R2S de CQL, leur rôle est de convertir le résultat d'une requête en flux.

Le système CQELS est donc un puissant Framework qui se focalise plus sur l'optimisation de l'exécution de la requête.
CQELS adopte le modèle de traitement utilisé dans les DSMSs en combinant les opérateurs de fenêtrage et relationnels pour générer de nouveaux flux à partir des résultats obtenus. Une des différences notables entre CQELS et C-SPARQL réside dans le moteur de traitement qui, gère le traitement des données statiques et flux sans le déléguer à un quelconque composant externe.

2.3.4 SPARQL$_{stream}$

SPARQL$_{\text{stream}}$ [1] est une extension de SPARQL qui s'inspire d'une part, des premiers travaux d'extension, principalement C-SPARQL, d'autre part, des langages de requête continues tels que CQL ou SNEEql [67]. Ces langages possèdent trois (3) grandes familles d'opérateurs : opérateurs de fenêtrage (window operators), opérateurs relationnels (relational operators) et les opérateurs de flux (stream operators).

— **Opérateurs de fenêtrage :** appelés très souvent S2R (Stream-to-Relation), ils sont utilisés pour extraire des données (destinées à être traitées) de flux d'entrée. L'opérateur utilisé est spécifié avec la clause FROM et peut être : **ROWS** (spécifie le nombre de tuples à extraire du flux en commençant par le tuple le plus récent), **RANGE** (spécifie la période dans laquelle on extrait les tuples du flux en commençant par celui le plus récent).

— **Opérateurs relationnels :** souvent notés R2R (Relation-to-Relation), ces opérateurs utilisent les données extraites par les opérateurs S2R afin d'y appliquer les fameux opérateurs arithmétiques, de comparaison, de logique et des fonctions d'agrégation. Ils sont spécifiés dans les clauses SELECT et WHERE.

— **Opérateurs de flux :** appelés R2S (Relation-to-Stream), ces opérateurs prennent en entrée les données retournées par les opérateurs R2R, les convertissent en flux avant de retourner le résultat. Il existe trois (3) types d'opérateurs de flux :

1. **ISTREAM** (Insert STREAM) : il retourne les tuples les plus récemment ajoutés dans la relation (retournée par le Relation operator). Pour ce faire, l'opérateur regarde le contenu de la relation avant et après sa mise à jour.
2. **RSTREAM** (Relation STREAM) : il retourne tous les tuples présents dans la relation retournée en considérant l'intervalle de temps (spécifié avec l'opérateur).
3. **DSTREAM** (Delete STREAM) : similaire à l'opérateur ISTREAM, cet opérateur compare la relation retournée par R2R avant et après son rafraîchissement pour retourner cette fois-ci les tuples qui ont été supprimés. Par rapport aux

deux précédents opérateurs, DSTREAM est le moins utilisé.

Nous détaillons dans la suite la syntaxe de requête et l'architecture du système $\text{SPARQL}_{\text{stream}}$.

2.3.4.1 Langage SPARQL_{stream}

$\text{SPARQL}_{\text{stream}}$ étend également SPARQL 1.1 et donc prend en compte les fonctions d'agrégations, ce qui du reste est indispensable dans le traitement de flux de données massives. Contrairement à C-SPARQL, $\text{SPARQL}_{\text{stream}}$ ne dispose pas de fonction temporelle permettant d'effectuer des opérations basiques de comparaison et de filtre temporelles sur les éléments (triplets) des flux. Dans ce qui suit, nous détaillons sa syntaxe.

Syntaxe de requête $\text{SPARQL}_{\text{stream}}$:

NamedStream → FROM [NAMED] STREAM StreamIRI Window
Window → NOW- Integer TimeUnit [UpperBound] [Slide]
UpperBound → TO NOW- Integer TimeUnit
Slide → SLIDE Integer TimeUnit
TimeUnit → 'MS' | 'S' | 'MINUTES' | 'HOURS' | 'DAY'

La syntaxe de $\text{SPARQL}_{\text{stream}}$ suit de près celle de SPARQL 1.1.
Dans la clause FROM de la requête $\text{SPARQL}_{\text{stream}}$, l'IRI streamIRI précédé des mots clés NAMED STREAM identifie un flux RDF virtuel. On peut ajouter les clauses FROM NAMED STREAM dont l'ensemble forme l'union des graphes des flux RDF manipulés par la requête.

Dans la définition de la fenêtre temporelle, les expressions NOW− et TO NOW− désignent respectivement les limites supérieure et inférieure de la fenêtre. Le terme clé NOW désigne l'instant courant. Par exemple, les expressions [NOW− 5 minutes], [NOW− 5 TO NOW− 2 minutes] signifient respectivement les 5 dernières minutes, les 5 dernières minutes en excluant les 2 dernières. Notons que la limite inférieure reste optionnelle. Cette dernière peut être le plus souvent omise car dans de nombreux domaines d'application des flux de données, les exécutions sont appelées à être continues et non fragmentées en intervalles.

L'expression optionnelle SLIDE représente l'écart entre deux évaluations successives de la requête dans cette fenêtre. Sa valeur met le traitement dans trois (3) cas de figure :

— si la valeur est inférieure à la largeur de la fenêtre alors nous sommes dans le cas de fenêtres glissantes (c'est-à-dire que des triplets vont figurer dans plusieurs fenêtres) ;

— si elle coïncide avec la largeur de la fenêtre alors nous avons des fenêtres dites TUMBLING (c'est-à-dire que chaque triplet ne figurera que dans une et une seule fenêtre) ;

— enfin, si elle est supérieure, alors nous avons des fenêtres sautantes (c'est-à-dire, l'exécution sera échantillonnée et va donc accuser des sauts de triplets).

$\text{SPARQL}_{\text{stream}}$ ajoute dans sa clause SELECT ou CONSTRUCT les opérateurs
R2S (Relational−to−Stream) que sont Rstream, Istream et Dsream. Ces opérateurs prennent en entrée une ou plusieurs relations et retournent des flux continus qui sont par la suite réutilisés ou évalués à nouveau. La différence entre ces trois opérateurs réside dans le type

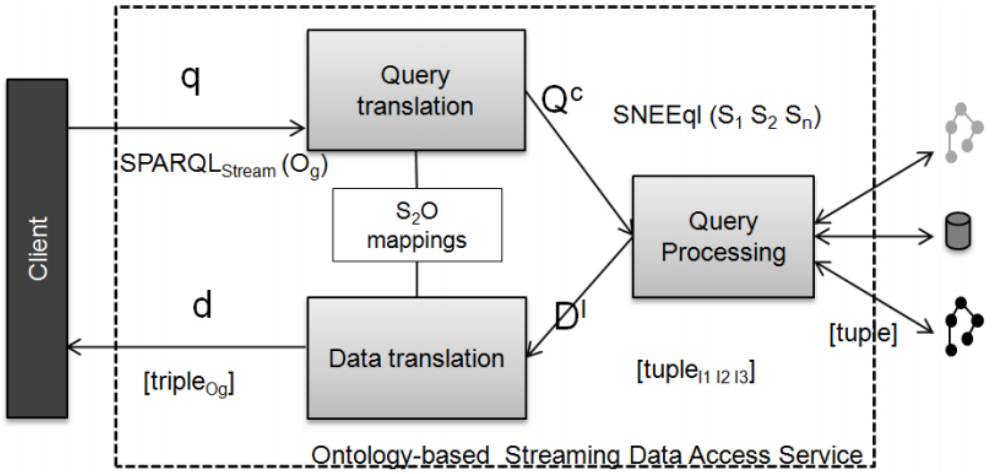


FIGURE 2.8 – Architecture de SPARQL$_{\text{stream}}$ [1].

de données retournées : l'opérateur RSTREAM retourne tous les triplets figurant dans la relation selon une fréquence précisée avec l'opérateur ; l'opérateur Istream retourne uniquement les triplets dernièrement insérés dans la relation ; enfin l'opérateur Dstream retourne les triplets dernièrement supprimés de la relation.

2.3.4.2 Système SPARQL$_{stream}$: Morph−streams

Le système SPARQL$_{stream}$ [1] actuellement connu sous le nom de Morph-streams est un moteur de requêtes sur flux RDF implémenté en langage Scala. Il utilise des techniques d'accès aux données basées sur une ontologie. La solution implémentée permet aux utilisateurs de soumettre leurs requêtes en se basant sur une ontologie existante et d'obtenir le résultat sous un format RDF. Dans ce cas, le moteur a besoin de transformer les requêtes SPARQL$_{stream}$ en requêtes qui pourront être exécutées en correspondance avec les sources des flux de données. Ainsi la requête est transformée en SNEEql [67] qui est un langage de requête continue comme CQL. La figure 2.8 donne une vue détaillée de l'architecture du système SPARQL$_{stream}$. Dans l'optique de transformer les requêtes SPARQL$_{stream}$ en requêtes adaptées aux sources des flux de données, un ensemble de transformations et d'appariements sont effectués. La couche S2O (Stream-to-Ontology) qui est une extension de R2O (Relational-to-Ontology) est utilisée pour définir l'appariement entre le modèle d'ontologie et le schéma relationnel des flux de données. Cet appariement est fait dans le but de traduire la requête en langage de requête continue SNEEql. Une fois les requêtes continues générées, la phase d'évaluation proprement dite de la requête commence avec le Query Processing. Cette couche utilise une technique distribuée pour extraire les données appropriées depuis les sources et effectue les opérations requises pour l'évaluation de la requête c'est-à-dire la projection, la sélection et la jointure. En dernier lieu, la couche Data translation traduit un ensemble de tuples résultats en instances de l'ontologie exprimée au format RDF.

SPARQL$_{\text{stream}}$ partage plusieurs caractéristiques avec ses concurrents et apporte bien entendu certaines améliorations. Il supporte tous les opérateurs R2S (Relation-to-Stream) là où C-SPARQL et CQELS ne prennent en compte respectivement que les opérateurs Rstream et Istream. Il est basé sur le concept de flux de triplets RDF virtuels qui peuvent être constamment interrogés en utilisant des fenêtres glissantes comme dans C-SPARQL, et en adoptant la technique d'évaluation sémantique de flux de SNEEql [67], adapté au modèle RDF.

2.3.5 Event Processing SPARQL (EP-SPARQL)

Similaire à C-SPARQL, le langage EP-SPARQL [17] définit une extension du langage sémantique SPARQL pour le traitement de flux d'informations sémantiques. Par contre, EP-SPARQL utilise une approche différente. En effet, les auteurs spécifient la notion d'événement (complexe). Cette dernière est utilisée dans le but d'identifier et de traiter les situations pertinentes et significatives en utilisant un ensemble d'opérateurs temporels et sémantiques. Dans le domaine des flux de données, les événements peuvent être représentés par le changement rapide des données. dans la suite, nous détaillons la syntaxe de requête et l'architecture de EP-SPARQL.

2.3.5.1 Langage EP-SPARQL

Tout d'abord, le langage EP-SPARQL permet la spécification d'événements complexes en le combinant avec la base de connaissance (sémantique).
Pour rendre leur concept plus formel, les auteurs d'EP-SPARQL définissent un flux RDF comme un ensemble $\mathscr{S}$ constitué de trois principaux éléments :

$<< s, p, o >, \tau_\alpha, \tau_\beta >$ où $< s, p, o >$ est un triplet RDF et τ_α et τ_β dénotent respectivement les bornes inférieure et supérieure de l'intervalle.

En plus des opérateurs purs de SPARQL, la syntaxe d'EP-SPARQL se caractérise par l'ajout d'un ensemble d'opérateurs permettant le traitement d'événements complexes. Ces opérateurs se basent principalement sur les informations temporelles des différents triplets RDF.

1. **FILTER** → filtre les variables dont l'expression de filtrage sur un $< spo, \tau_\alpha, \tau_\beta >$ est vérifiée (true).
2. **AND** → joint les éléments $< spo, \tau_\alpha, \tau_\beta > et < spo', \tau'_\alpha, \tau'_\beta >$. Le résultat de la jointure a respectivement pour timestamp de début et de fin $\tau''_\alpha = min(\tau_\alpha, \tau'_\alpha)$ et $\tau''_\beta = min(\tau_\beta, \tau''_\beta)$.
3. **UNION** → forme la disjonction entre $< spo, \tau_\alpha, \tau_\beta >$ et $< spo', \tau'_\alpha, \tau'_\beta >$.
4. **OPTIONAL** fait correspondre aux variables à retourner les éléments $< spo, \tau_\alpha, \tau_\beta >$ ou $< spo', \tau'_\alpha, \tau'_\beta >$ dont l'expression de filtre est vérifiée.
5. **SEQ** → effectue une jointure (left join, right join ou full join) sélective des éléments $< spo, \tau_\alpha, \tau_\beta >$ et $< spo', \tau'_\alpha, \tau'_\beta >$. c'est-à-dire, il joint $< spo, \tau_\alpha, \tau_\beta >$ et $< spo', \tau'_\alpha, \tau'_\beta >$ seulement si $< spo', \tau'_\alpha, \tau'_\beta >$ se produit seulement après $< spo, \tau_\alpha, \tau_\beta >$.
6. **EQUALS** → fait une jointure entre $< spo, \tau_\alpha, \tau_\beta >$ et $< spo', \tau'_\alpha, \tau'_\beta >$ s'ils se produisent simultanément.
7. **OPTIONALSEQ et EQUALSOPTIONAL** → sont des variantes temporelles de **OPTIONAL**
8. **getDURATION()** → cette fonction est placée dans la condition **FILTER** et retourne un littéral donnant la longueur de l'intervalle de temps associé
9. **getSTARTTIME() et getENDTIME()** → ces fonctions extraient respectivement les timestamps aux bornes inférieure et supérieure de l'intervalle courant.

La requête 2.3 présente un exemple de requête EP-SPARQL avec l'opérateur de séquence SEQ dans lequel sont fournies les valeurs de pression de l'eau élevées.

```
SELECT ?road ?speed WHERE
```

```
{ ?sensorId  observation pressure. }
SEQ { ?sensorId  measure ?result }
AND { ?result   value  ?waterPressureValue }
FILTER ( waterPressureValue > ‘‘27’’^^xsd:float)
```

REQUÊTE 2.3 – Exemple de requête EP-SPARQL.

2.3.5.2 Système EP-SPARQL : ETALIS

ETALIS, introduit dans [68] , est l'implémentation d'un prototype qui permet la spécification d'événements et leur caractérisation dans le temps. ETALIS supporte deux (2) langages : EP-SPARQL [17] et ELE (ETALIS Langage for Events) [69]. Ces deux langages permettent la définition de la notion d'événement complexe ainsi que l'un des aspects intéressants du système ETALIS qui est le raisonnement. ETALIS [9] traduit les requêtes en programme logique Prolog et les exécute dans un unique thread. L'exécution parallèle et distribuée est à ce jour non supportée.

La figure 2.9 illustre comment les requêtes EP-SPARQL sont traitées via ETALIS. Ces requêtes sont traduites en règles Prolog et les événements RDF du flux sont également mappés sur des triplets Prolog. Il y a également une option d'utilisation de la base de connaissances sous la forme de règles Prolog où les événements RDF mappés sont continuellement comparés aux règles traduites à l'aide du moteur ETALIS. EP-SPARQL n'implémente pas et ne fournit pas explicitement d'optimisations basées sur RDF, mais s'appuie sur ETALIS pour l'exécution des jointures.

Le système ETATLIS utilisé pour traiter les requêtes EP-SPARQL suit une approche adaptée aux domaines d'application des flux de données. En effet, à l'instar du système CQELS, ETALIS génère de nouveaux résultats lorsque de nouvelles données arrivent (push). Malgré ce critère important affiché par ce système, EP-SPARQL n'est pas une solution utilisée pour une interrogation continue de requêtes SPARQL une fois de nouvelles informations arrivent. Il reste plutôt une solution orientée vers le traitement d'événements complexes.

Des expérimentations incluant une comparaison d'EP-SPARQL avec C-SPARQL et CQELS est donnée dans [70]. Dans cette expérience, EP-SPARQL présente de meilleures performances en terme de temps de traitement et de montée en charge que C-SPARQL dans différentes conditions. Par contre EP-PARQL ne réitère pas ces performances devant CQELS.

De même que C-SPARQL, EP-SPARQL utilise le fenêtrage logique et physique. Mais, contrairement à C-SPARQL, les caractéristiques principales du système sont représentées par l'ensemble de ses opérateurs logiques et temporels cités précédemment. Ces opérateurs combinés permettent au système d'exprimer le modèle complexe des éléments des flux. L'autre grande différence entre les deux réside dans l'annotation temporelle des triplets RDF. En effet, C-SPARQL associe une estampille unique dans le temps à chaque triplet tandis qu'EP-SPARQL adopte deux estampilles représentant les bornes inférieure et supérieure du déroulement de l'événement dans le temps. L'idée derrière cette modélisation temporelle est de faciliter l'écriture des modèles complexes impliquant des contraintes de temps et de contenu des triplets RDF en entrée.

9. ETALIS est une solution open source
http ://code.google.com/p/etalis
Demo en ligne
http ://etalis.fzi.de

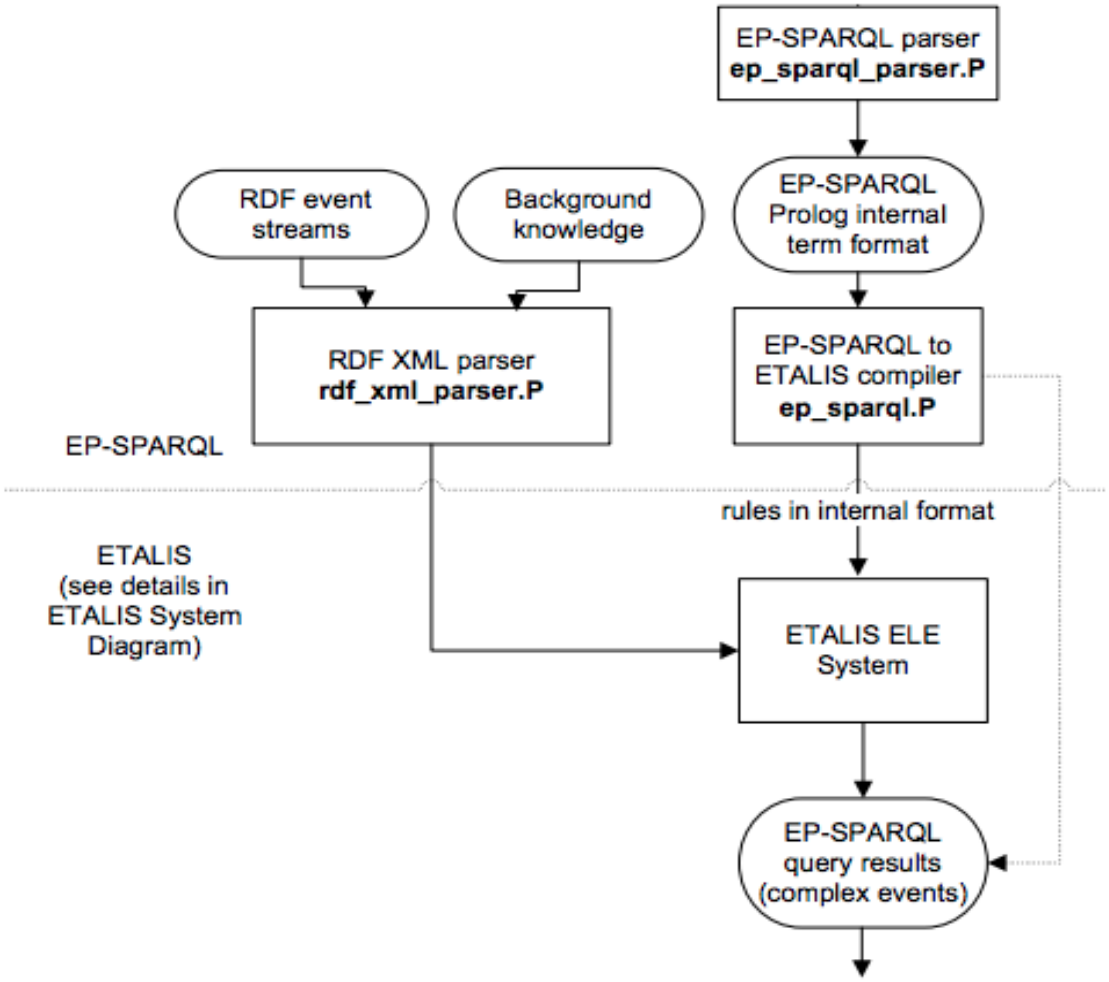


FIGURE 2.9 – Diagramme du système ETALIS.

2.3.6 Sparkwave

Sparkwave [19] est également une autre extension de SPARQL. Son modèle de langage suit de près la syntaxe du langage SPARQL. Son objectif est de fournir des fonctionnalités efficaces permettant des traitements continus sur flux de données RDF, l'expression de contraintes temporelles sous forme de fenêtre et la prise en compte du schéma des données RDF pour le raisonnement. La seule déviation de la syntaxe de Sparkwave par rapport à celle de SPARQL est l'ajout de nouveaux éléments appelés handlers . Cette section est ajoutée au prologue de la requête et permet de répondre au matching des variables. Il est déclenché lorsque Sparkwave identifie la survenue d'un événement spécifique associé à la définition d'un handler donné.

2.3.6.1 Système Sparkwave

Sparkwave est aussi un système de traitement à la volée de flux de données RDF très complexe dans sa structure. Le système implémente l'algorithme Rete [71]. Ce dernier, à la base conçu pour les systèmes de production, représente aujourd'hui une approche générale pour faire face à de nombreux modèles de données. L'algorithme se focalise plus

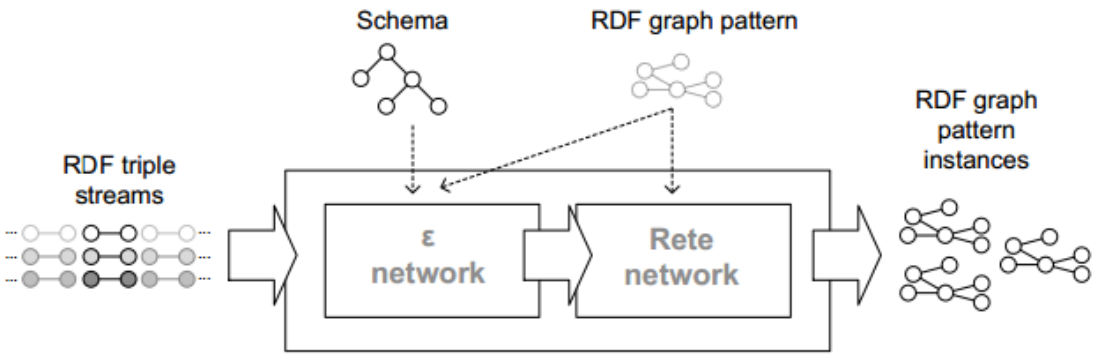


FIGURE 2.10 – Architecture de Sparkwave.

sur une gestion de la mémoire pour la performance en construisant des structures de mémoire compréhensives appelées α− et β−network destinés à vérifier respectivement les conditions interne et externe des modèles à travers un ensemble d'objets. Le caractère intrinsèque des flux observés dans Rete favorise son utilisation sur les flux de données.

Pour atteindre les buts qu'ils se sont fixés, les auteurs de Sparqkwave ont d'abord étendu l'algorithme basique de Rete pour la prise en compte du schéma RDF et les contraintes temporelles des flux de données. Comme le montre la figure 2.10, le composant ϵ−network se charge de générer les triplets suivant le schéma des données RDF. Les nœuds ϵ−network sont reliés au nœud α−network approprié dans un style de flots de données. Le module suivant permet la prise en compte des fenêtres temporelles. Ce module est une extension des fonctionnalités des nœuds β−network qui vérifient si un modèle partiel ou complet est inclus ou non dans la fenêtre. Plus de détails sur les fonctionnalités et la construction des ϵ−network et β−networks sont donnés dans [19]. La portion de données qui sera considérée durant le traitement de la requête sera isolée en utilisant un mécanisme traditionnel de fenêtrage similaire à ceux utilisés dans les DSMSs et dans C-SPARQL. Les auteurs fournissent une première évaluation de leur système. Ils considèrent jusqu'à 100 000 triplets et quatre (4) requêtes de complexités allant d'une simple sélection à la jointure multiple. Après évaluation, Sparkwave fournit plus de résultats que C-SPARQL et CQELS (jusqu'à 100 000 triplets par seconde) et consomme moins de ressource mémoire (au plus 1 GB, c'est-à-dire moins de la moitié par rapport à C-SPARQL).

La version actuelle de Sparkwave présente plusieurs limites dans la mesure où elle ne supporte ni les opérateurs arithmétiques, ni les opérateurs logiques (disjonction et négation) ni les opérateurs temporels.

2.3.7 Autres systèmes

Il existe plusieurs autres approches et ou systèmes, antérieures ou non à ceux étudiés précédemment. Ils emploient pour la plupart des propositions similaires avec des constructions temporelles.

— **SPARQL-ST** [72]. L'extension adopte une donnée de modélisation temporelle et spatiale dans RDF. **SPARQL-ST** définit une syntaxe et une sémantique formelles de SPARQL prenant en charge les requêtes spatiales et temporelles. Les auteurs intègrent d'une part des informations dans des graphes RDF temporels (ensemble de triplets temporels) et d'autre part, une ontologie pour modéliser des caractéristiques spatiales. Les auteurs fournissent la formalisation de **SPARQL-ST** basées sur la syntaxe et la

sémantique de SPARQL définies dans [57]. La sémantique de **SPARQL-ST** étend le concept de cartographie vu dans [73] pour mapper des variables spatiales à un ensemble de triplets RDF et une variable temporelle à un intervalle de temps. Les variables spatiales représentent des objets spatiaux complexes. Les variables temporelles spatiales **SPARQL-ST** sont respectivement préfixées dans les syntaxes par "#" et "%". Les auteurs ont mis en place un prototype construit sur un SGBD commercial. Un inconvénient de l'approche proposée est que **SPARQL-ST** ne prend pas en charge l'actualisation fréquente et que l'exécution doit être déclenchée.

— **TA-SPARQL** [74]. Il est introduit pour interroger le modèle de données RDF annoté dans le temps (TA-RDF) qui est un vocabulaire RDF spécial fournissant des fonctionnalités de base pour la représentation et l'interrogation des flux de données. **TA-RDF** est une extension du modèle RDF où les ressources sont facultativement annotées avec une ressource de valeur temporelle. Ensuite, **TA-SPARQL** est conçu pour prendre en charge les requêtes sur les données dans la syntaxe **TA-RDF**. Les auteurs proposent globalement un système de stockage, d'indexation et d'interrogation des données de séries temporelles dans le spectre des technologies du web sémantique. Le système proposé se concentre uniquement sur les requêtes. En effet, les auteurs reconnaissent que le système actuel ne permet pas de requête continue (exécution continue, fenêtre temporelle, fenêtre triple, opérateurs R2S [10], de séquence [11], co-occurrence [12], fonction temporelle) et supporte des agrégats limités. Le système est implémenté en plus du projet Tupelo [13] qui est un référentiel de contenu sémantique extensible pour le stockage et la récupération de données RDF. Cependant, les auteurs ne fournissent pas d'évaluation expérimentale sur les performances de leur système et le code n'est pas disponible pour pouvoir l'évaluer.

— **stSPARQL** [73]. Les auteurs proposent d'abord le modèle de données **stRDF** puis le langage de requête **stSPARQL**. **stRDF** est également un modèle de données de contraintes qui étend RDF avec la représentation des données spatiales et temporelles. **stSPARQL** étend SPARQL en utilisant une méthode déclarative et simple pour interroger des données spatiales et temporelles. **stSPARQL** et **SPARQL-ST** sont similaires dans l'idée principale et les formalismes syntaxiques et sémantiques utilisés. Cependant, sur la base du comparatif entre les deux propositions répertoriées dans [74], nous notons une notion plus expressive du temps valide dans **stSPARQL**. Les auteurs fournissent aussi des sous-ensembles linéaires de Q^k, en utilisant des notions d'algèbre linéaire. De plus, **stRDF** et **stSPARQL** semblent être moins imposants dans les exigences de sémantique aux utilisateurs des deux approches.

— τ−**SPARQL**. [75]. Les auteurs présentent d'abord une approche pour l'annotation temporelle des triplets RDF (Temporal-RDF) avec des intervalles de validité temporelle. τ−SPARQL est une requête temporelle avec une syntaxe de traduction principale vers la requête SPARQL. L'approche présente une structure d'index spécifique pour les intervalles temporels. Ce qui améliore le temps de récupération des requêtes de point de temps. Les auteurs définissent deux formats d'utilisation des requêtes τ−SPARQL : les requêtes ponctuelles et les requêtes temporelles. La première citée vise à récupérer des informations valables à un moment précis, tandis que le dernier interroge les données en autorisant l'utilisation d'intervalles génériques et de points temporels. Cependant, ce système ainsi que les approches similaires proposées dans [72] (SPARQL-ST), [74] (stSPARQL), [76] (T-SPARQL), [77] et

10. R2S
11. sequence
12. co-occurrence
13. http ://www.christian-faure.net/2008/11/16/vers-les-semantic-ria/

[76] (T-SPARQL) manquent de support pour les requêtes continues et ne disposent pas non plus de raisonneurs temporels.

2.3.8 Bilan

En général, les systèmes de traitement de flux RDF adoptent des approches similaires. Tout d'abord la notion d'estampille temporelle est ajoutée au modèle de donnée RDF. Nous passons ainsi des triplets $< S, P, O >$ aux quadruplets $(< S, P, O >, \tau)$ où τ indique la datation implicite (attribuée par le système à l'arrivée) ou explicite (attribut de type date dans la structure du flux correspondant au moment où la donnée a été générée) des éléments dans un flux.

Ensuite, la grammaire du langage de requête SPARQL est étendue avec une nouvelle syntaxe et/ou d'opérateurs pour l'interrogation de ces flux RDF. Toutes ces extensions se sont parfaitement inspirées des langages de requêtes continues (notamment CQL et SNEEql) utilisés par les DSMSs. La principale différence entre ces systèmes proposés réside principalement dans l'implémentation du système où certains choisissent de déléguer leur traitement à des DSMSs existants accompagnés d'un moteur SPARQL (sesame, Jena, Virtuoso, etc). D'autres par contre utilisent une approche native (CQELS et Sparkwave).

Tous les travaux proposés sur l'extension de SPARQL pour le traitement sémantique de flux de données ajoutent l'aspect flux de données (timestamp) aux données RDF/SPARQL et supportent également le fenêtrage glissant. On distingue néanmoins quelques différences qui sont parfois très importantes figurant aussi bien au niveau caractéristiques du langage qu'au niveau implémentation et fonctionnalités de ces systèmes.
Streaming SPARQL (non implémenté) est la seule extension ne supportant pas l'agrégation. $\text{SPARQL}_{\text{stream}}$ ne prend pas en compte les fenêtres basées sur le nombre de triplets RDF (fenêtrage physique). Par contre, il supporte le fenêtrage temporel (Fenêtre logique) et tous les opérateurs R2S (ou window-to-stream) de CQL. C-SPARQL est le seul à supporter le stockage des requêtes ainsi que les flux résultants pour qu'ils puissent être mis à jour après chaque exécution. C-SPARQL, CQELS et SPARQLstream étendent SPARQL 1.1 alors que Streaming SPARQL est basé sur SPARQL 1.0.

Les différences majeures entre ces systèmes peuvent être listées avec les principaux critères suivants :

— **Requêtes continues** : ce critère est fondamental pour le traitement de flux de données continus. En se basant sur cette propriété, nous distinguons deux types de systèmes : les systèmes avec fréquence d'exécution périodique par exemple toutes les 5 minutes (concernent C-SPARQL seulement) et ceux dont l'exécution est synchronisée avec l'arrivée de nouvelles informations.

— **Annotation des données** : cette propriété concerne l'ordonnancement des flux. Hormis EP-PSARQL qui utilise une annotation dite multiple, c'est-à-dire composée d'un intervalle de temps, les autres systèmes appliquent une annotation définie par un temps physique de type date (année, mois, jour, heure, minute, seconde). Ce critère relève du type de gestion des données imposé par la complexité de l'intégration et des besoins.

— **Opérateurs temporels** : C-SPARQL et EP-SPARQL sont les seuls systèmes à offrir des opérateurs temporels permettant de manipuler les données de type date contenues dans les flux. Ces opérateurs permettent de capturer l'évolution des informations au cours du temps ainsi que l'application de filtres temporels.

— **Opérateurs d'agrégation** : les fonctions d'agrégation, indispensables dans la gestion de flux de données dans bon nombre de domaines d'application, sont supportées par l'ensemble des systèmes implémentés jusqu'ici.

— **Types de fenêtres** : seuls Streaming SPARQL, C-SPARQL et CQELS utilisent à la fois les fenêtres physiques (nombre de triplets) et les fenêtres logiques (temporelle). SPARQL$_{stream}$ ne supporte que les fenêtres logiques là où EP-SPARQL n'utilise ni l'une ni l'autre.

— **Opérateurs relationnels et de stream** : mis à part SPARQL$_{\text{stream}}$ qui supporte tous les trois opérateurs R2S, C-SPARQL prend en charge Rstream et CQELS utilise Istream. Notons que Streaming SPARQL et EP-SPARQL n'ont implémenté aucun des trois opérateurs.

— **Union, Join, Filter et Optional** : les opérateurs basiques d'union, de jointure, de filtre et Optional de SPARQL sont implémentés par tous les systèmes.

— **Traitement parallèle/distribué** : enfin, plusieurs systèmes et approches de traitement distribué de flux de données RDF sont proposés. Nous détaillons leurs approches dans le chapitre 7.

Le tableau 2.2 fait un récapitulatif (non exhaustif) des propriétés propres et communes aux différents systèmes étudiés précédemment.

En **résumé**, nous pouvons sortir les points suivants :

— De multiples notions de flux RDF retenues

— annotation temporelle implicite (ordre d'arrivée des éléments du flux dans le système)

— annotation *point in time* (une annotation par triplet)

— annotation *interval* (deux annotations par triplets)

— les sémantiques des langages de requêtes sont similaires mais ne sont pas identiques [78]

— un grand défaut d'opérateurs R2S ou temporels dans plusieurs systèmes

Les systèmes existants adoptent des approches différentes dans leur architecture et il n'est pas toujours évident de dire de façon générale quelle approche demeure la meilleure. A notre avis, SPARQL$_{\textbf{stream}}$ et EP-SPARQL sont les plus fiables. Étant donné que CQELS et Sparkwave utilisent des approches natives, ils peuvent apporter des optimisations logiques et adaptables contrairement aux autres systèmes. Ces deux systèmes natifs sont ainsi plus optimisés et donc plus performants en terme de temps d'exécution.

Cependant, du point de vue des fonctionnalités, nous pouvons noter que C-SPARQL est le seul système qui fournit à la fois les opérateurs Union, Join, Optional et Filter, les fenêtres logiques et physiques, les opérateurs d'agrégation, l'exécution continue, les flux multiples, l'interrogation de données statiques et données de flux, les fonctions temporelles, etc. De plus, en termes de développement, le système C-SPARQL est entièrement basé sur un Framework JAVA où les principales librairies sont Jena pour la manipulation sémantique et Esper pour le traitement continu. De plus, nous notons une plus grande facilité dans l'installation, le développement, une documentation des librairies bien fournie ainsi qu'une extensibilité de l'architecture.

	Streaming SPARQL	C-SPARQL	CQELS	$SPARQL_{stream}$	EP-SPAQRL	Sparkwave
Entrée	Flux RDF	Flux RDF & RDF	Flux RDF & RDF	Flux relationnel	Flux RDF & RDF	Flux RDF & RDF
Union, Join, Optional, Filter	✓	✓	✓	✓	✓	✓
Modèle de fenêtre	Point temporel	Point temporel	Point temporel	Point temporel	Intervalle temporel	Point temporel
Fenêtre de triplet	✓	✓	✓	×	×	×
Agrégation	×	✓	✓	✓	✓	×
Opérateur Temporel	×	✓	×	×	✓	×
Fréquence d'exécution	Périodique (pull)	Périodique (pull)	À la demande (push)	Périodique (pull)	À la demande (push)	À la demande (push)
Opérateur R2R	SPARQL 1.0	SPARQL 1.1	SPARQL 1.1	SPARQL 1.1	SPARQL 1.0	SPARQL 1.0
Opérateur R2S	×	Rstream	Istream	Rstram, Istream & Dstream	×	Istream
Architecture	Boite grise	Boite noire	Boite grise	Boite noire	Boite noire	Boite noire
Raisonnement	×	Non implémenté	×	×	✓	✓
Parallèle/ Distribué	×	×	✓ (CQELS Cloud)	×	×	×

TABLEAU 2.2 – Comparaison des principaux systèmes RSP centralisés.

2.4 Conclusion

Les systèmes de traitement de flux de données sémantiques suscitent de plus en plus l'intérêt de plusieurs groupes de recherche. Jusqu'ici, plusieurs systèmes dont C-SPARQL, CQELS, $SPARQL_{stream}$, EP-SPARQL et Sparkwave ont été proposés. Dans ce chapitre, nous avons défini les concepts fondamentaux du web sémantique et des flux de données sémantiques. Nous avons également présenté les différents systèmes de traitement (non distribué) existants en définissant les différents langages de requête ainsi que les architectures et fonctionnalités des systèmes dont une implémentation est fournie. Tous les systèmes étendent SPARQL et utilisent des approches différentes. C-SPARQL est le système qui regroupe le plus d'opérateurs de requêtes, d'opérateurs de fenêtrage, les flux multiples, le croisement de flux et des données statiques. De plus, son architecture est modulaire et permet l'ajout de fonctionnalités dans les modules de traitement statique et continu. Dans l'optique de prendre en compte le traitement parallèle et distribué des flux

de données sémantiques, nous nous focalisons, dans le chapitre 7 sur les systèmes existants utilisant différentes approches de traitement distribué de flux de données RDF.

Cependant, le large volume de flux données RDF généré et leur fréquence d'arrivée rapide réduit considérablement les performances des systèmes RSP centralisés. Cette charge de traitement pourrait être réduite grâce à l'extension des systèmes RSP par l'ajout de mécanismes de résumé et de compression en temps réel de flux de données RDF. Les chapitres 3, 4 et 5 et 6 présentent les techniques existantes de résumé de flux de données et nos contributions sur la réduction de la charge de traitement par l'échantillonnage, le résumé et la compression.

Chapitre 3

État de l'art sur les résumés de flux de données

Sommaire

3.1 Introduction

Les systèmes de gestion de flux de données (SGFD) permettent de répondre aux requêtes continues via l'utilisation de fenêtres temporelles définies sur la mémoire du système. L'expiration des données rend impossible l'évaluation de toute requête non prévue a priori. Ceci suppose de définir à l'avance sur un flux l'ensemble des requêtes dont on pourra avoir besoin par la suite. Pour l'évaluation des requêtes a posteriori, il est nécessaire de conserver une trace de l'historique du flux car l'utilisation exclusive d'un SGFD ne permet pas de satisfaire un tel besoin. Il en est de même pour les systèmes de gestion de base de données (SGBD) qui se trouvent inadaptés pour répondre à ce type de besoin. En effet, ces systèmes sont capables de conserver des historiques, cependant ils font face à deux contraintes qui les rendent inadaptés dans un contexte de flux de données.

— Taille des données : un flux de données est théoriquement infini. Il est par conséquent impossible de le conserver en totalité dans une structure de données ;
— Débit d'écriture : les opérations d'insertion dans les SGBD sont suivis de plusieurs opérations qui prennent un temps d'écriture considérable (opération de verrouillage, de journalisation, d'indexation, etc.). Ceci ne peut s'appliquer à des données arrivant avec des débits importants.

Pour pallier les insuffisances de ces systèmes, une solution permettant un traitement *a posteriori* des données historisées du flux consiste à conserver des représentations compactes appelées résumés. L'objectif est de conserver un historique du flux sans dépasser l'espace de stockage et le temps de traitement accordés pour l'application. Il existe une

variété de techniques qui peuvent être utilisées pour la construction des résumés de flux de données(échantillonnage).

Ce chapitre est structuré comme suit. Dans la section 3.2, nous apportons la définition d'un résumé de flux de données. La section 3.3 est dédiée aux concepts généraux de résumé de flux de données et la section 3.4 détaille les techniques existantes de résumé de flux de données RDF. La section 3.5 conclut ce chapitre.

3.2 Définition d'un résumé de flux de données

Selon Csernel dans [79], la notion de résumé est duale. En effet, l'information doit d'une part être réduite par un procédé d'abstraction à ses idées les plus pertinentes et d'autre part la taille prise par le résumé doit être nettement plus faible que celle du ou des objets qu'il résume. La première partie relève donc plus d'opérations comme la fouille de données visant à mettre en relief les grandes tendances tandis que la seconde tient plus de la compression dans le sens où l'information doit être concentrée. Par définition, un résumé de flux permet de conserver une trace des événements apparus dans un flux de données. Cette notion est apparue avec le besoin de traiter l'ensemble des analyses qui ne sont pas prévues à l'avance pour un flux particulier. Par ailleurs, la conservation d'un résumé permet de réaliser des avancées dans des domaines tels que la gestion des données en retard, les techniques de réduction de charge (load shedding) [80, 81] ou encore l'approximation de certaines requêtes comprenant des opérateurs bloquants. Dans [82], est défini un résumé de flux de données comme suit : "*un résumé de flux de données consiste à conserver, avec un minimum d'événements, les représentations statistiques de toutes les périodes temporelles du flux.*"

3.3 Conception de résumés de flux de données

La façon la plus naturelle de concevoir un résumé des données consiste à conserver en mémoire une version allégée de l'information que ce soit en réduisant sa dimensionnalité, son nombre d'éléments ou l'espace de représentation de ces derniers. Les sous-sections suivantes fournissent une description des différentes approches envisagées à ce jour dans ce domaine.
Il faut noter aussi que la conception d'un résumé est basée sur une dimension spatiale et/ou temporelle. La dimension spatiale définit la structure utilisée pour disposer en mémoire les données conservées. Par contre, la dimension temporelle définit la gestion de l'historique conservé en fonction du temps.
Toutes les approches décrites ici sont des extensions de techniques qui existaient déjà dans le cadre du traitement des données statiques et permettaient de faire face à des jeux de données particulièrement larges. Dans les cas étudiés ici, ces approches ont en général été adaptées pour faire face au modèle de données particulier que sont les flux de données et aux contraintes techniques qui leur sont associées. Les contraintes liées au temps de traitement et à l'espace de stockage étant classiques à l'élaboration d'un résumé et plus généralement dans le traitement de données massives.

3.3.1 Conception simple d'un résumé de flux de données

Nous illustrons dans ce qui suit une liste non exhaustive des principales techniques utilisées dans la littérature pour la construction d'un résumé de flux de données.

3.3.1.1 Échantillonnage

L'approche par échantillonnage aléatoire est une approche particulièrement adaptée aux flux de données. Cette technique permet de fournir, à partir d'un échantillon, des informations sur une large population. En général, ces techniques nécessitent une seule passe pour échantillonner les données ; ce qui facilite leur adaptation dans le cas des flux de données. Cependant, les algorithmes d'échantillonnage exigent généralement l'intégralité des données afin de sélectionner un échantillon représentatif. Dans la suite, nous désignons par k la taille de l'échantillon, n la taille du flux ou de la fenêtre et c les constantes utilisées dans les formules de complexité.

3.3.1.1.1 Échantillonnage de type réservoir

Un algorithme classique proposé par Vitter est largement utilisé dans le cadre des flux de données : l'échantillonnage réservoir [83]. Il permet d'obtenir de façon incrémentale un échantillon uniforme aléatoire de l'ensemble d'un flux sans nécessiter une connaissance a priori de la taille du flux. La taille maximale du réservoir est fixée à l'avance et la mise à jour du réservoir se fait en temps constant. Le principe de l'approche consiste à maintenir un réservoir de taille k à partir des données du flux. Lors de l'étape d'initialisation, les k premiers événements du flux sont insérés dans le réservoir. Lorsqu'un nouvel événement i ($i > k$) arrive, il remplace de manière aléatoire l'un des éléments du réservoir avec une probabilité d'inclusion de $\frac{k}{i}$. Cette approche a l'avantage d'être simple à implémenter et à mettre en œuvre. En revanche, plus on avance dans le flux, plus la probabilité d'inclusion est réduite (car $\lim_{i\to\infty} \frac{k}{i} = 0$).
Si la taille de la fenêtre est connue d'avance, une variété de l'échantillonnage réservoir peut être utilisée nommée échantillonnage séquentiel [84]. Il s'agit de générer des sauts à effectuer entre deux inclusions. Cette méthode permet d'obtenir efficacement un échantillon aléatoire uniforme avec une complexité inférieure à la méthode standard.
Dans [85], Aggarwal propose une méthode d'échantillonnage par réservoir biaisé. Il s'intéresse à la faisabilité de la construction d'un échantillon biaisé sur des flux de données pour privilégier les éléments récents du flux par rapport aux éléments anciens. Ces comportements sont définis par les contraintes de l'application.
Fonction de biais : soit $p(r,t)$ la probabilité que l'événement i du flux introduit à l'instant r soit toujours présent dans le réservoir à l'instant t. La fonction de biais $f(r,t)$ est proportionnelle à cette fonction de probabilité. Dans son approche, Aggarwal propose une fonction de biais dite exponentielle définie comme suit : $f(r,t) = e^{-\lambda(t-r)}$ avec λ le taux de biais, $\lambda \in [0,1]$.
Cette approche a pour avantage l'inclusion des méthodes de remplacement simples par l'utilisation de la fonction de biais. Cependant, la taille de l'échantillon n'est pas fixée au départ comme c'est le cas dans l'échantillonnage réservoir de Vitter. Ici, une borne supérieure sur la taille du réservoir est contrôlée par les fonctions exponentielles utilisées. Cette technique est biaisée étant donné qu'elle assure une importante probabilité d'inclusion associée aux événements récents.

3.3.1.1.2 Échantillonnage sur des fenêtres glissantes

L'échantillonnage réservoir présenté à la sous-section précédente est très efficace dans le cas d'insertion ou de mises à jour. Cependant, il trouve des limites dans l'expiration des données dans une fenêtre glissante. En effet, dans ce type de fenêtre, les éléments ne faisant plus partie de la fenêtre courante deviennent invalides et doivent être remplacés dans l'échantillon. Des algorithmes permettant de maintenir à jour un échantillon sur une fenêtre glissante sont alors nécessaires. Dans [86], les auteurs introduisent plusieurs

algorithmes permettant de maintenir un échantillon de taille k sur une fenêtre glissante de taille n.

3.3.1.1.2.1 Échantillonnage périodique

Le principe consiste à maintenir un échantillonnage réservoir sur les k premiers événements du flux. Lorsqu'un événement de l'échantillon expire, il est remplacé par le nouvel événement qui arrive. Cette technique est efficace en terme de mémoire puisqu'elle n'a besoin que de la place pour stocker les k événements à chaque étape. En revanche, elle souffre d'un défaut de périodicité. En effet, si l'événement i est inclus dans l'échantillon, l'événement $i+cn$ sera aussi inclus pour tout entier $c > 0$; ce qui n'est pas acceptable pour un certain nombre d'applications. Pour pallier ce défaut, une autre technique utilisée est celle de l'échantillonnage avec réserve.

3.3.1.1.2.2 Échantillonnage avec réserve

Dans cet algorithme, on dispose d'un échantillon dit de «*réserve*» (ou «*backing sample*»). Chaque événement du flux est ajouté à cet échantillon avec une probabilité *(2ck log n)/n*. On génère ensuite un échantillon aléatoire à partir de l'échantillon de réserve. Lorsqu'un événement expire, on le supprime de la réserve.

3.3.1.1.2.3 Échantillonnage chaîné

Cette approche permet de maintenir une chaîne de remplaçants pour chaque événement i dans l'échantillon. Au total, on maintient k chaînes de remplaçants pour un échantillon de taille k. Chaque événement i est ajouté à l'échantillon avec une probabilité $\frac{1}{min(i;n)}$ avec i l'indice de l'événement courant et n la taille de la fenêtre. Comme l'événement d'indice i est ajouté à l'échantillon, il faut choisir l'indice de son remplaçant. Cet indice est sélectionné aléatoirement dans l'intervalle $[i+1, i+n]$ selon une loi uniforme. Lorsque l'événement de cet indice arrive, il est ajouté à l'échantillon et on sélectionne son remplaçant de la même façon que précédemment. On construit ainsi une chaîne de remplaçants potentiels. Quand un événement est éliminé de l'échantillon, la chaîne de ses remplaçants est aussi supprimée. Cette méthode a l'avantage de garantir une taille mémoire en $O(k)$.

3.3.1.1.2.4 Échantillonnage stratifié

Gemulla et Lehner présentent dans [87] une technique permettant de maintenir un échantillon à partir d'une fenêtre glissante physique. Il s'agit de partitionner une fenêtre W en l strates temporelles ($l > 1$). Un échantillon uniforme E_i est maintenu sur chacune des strates ($avec 1 < i < l$). Chaque échantillon possède une taille fixe n. En plus de ces échantillons, l'algorithme maintient, pour chaque strate, sa taille N_i ainsi que l'estampille temporelle de son événement le plus récent. Ces deux mesures sont importantes pour le maintien des échantillons.
Le principal défi de cet échantillonnage est le partitionnement des frontières des strates temporelles. Ce partitionnement a un impact important sur la qualité de l'échantillon généré. Dans le cas le plus simple, la fenêtre W est divisée en strates selon le partitionnement temporel équivalent (*Equi-width stratification*). Une seconde méthode revient à diviser la fenêtre en strates de tailles égales (en terme de nombre d'événements). Cette approche est appelée stratification équidistante (*Equi-depth stratification*).
Une parfaite stratification équidistante est impossible à réaliser. Dans [87], les auteurs présentent également un algorithme basé sur la fusion des strates pour se rapprocher le plus possible d'une stratification équidistante (*Merge-based stratification*). L'idée revient à fusionner les deux strates voisines ; ce qui réduit l'information mémorisée dans l'échantillon représentatif de ces deux strates. Il restera donc un espace libre à la fin de l'échantillon.

Cet espace peut être utilisé pour d'autres événements.
De nombreux autres algorithmes d'échantillonnage ont été développés tels que : l'échantillonnage par priorité [86], l'échantillonnage par couplage aléatoire [88], l'échantillonnage de jointure [89], [90], [91], [92], etc.
Quelle que soit la technique d'échantillonnage utilisée, elles présentent toutes les mêmes avantages. Les échantillons peuvent être ajoutés l'un à l'autre et la qualité d'un échantillon n'est pas affectée par le nombre d'éléments qu'on y insère. Par ailleurs, toutes ces techniques sont extrêmement rapides à exécuter.

3.3.1.2 Agrégats

Parmi les techniques de résumé les plus simples et les plus basiques, la constitution d'agrégats reste une des méthodes les plus employées en raison de sa facilité de mise en place et de son efficacité en terme de vitesse de traitement et de réduction de la place occupée.

3.3.1.2.1 Agrégats simples

Ces agrégats se reposent sur une seule variable, que celle-ci soit quantitative ou qualitative. Sur les données de type numérique, on peut globalement distinguer deux types d'agrégats simples : les moments et les quantiles ; chacun d'entre eux s'attachant à représenter un aspect différent de l'information. Sur les données de type qualitatif, l'équivalent serait le tableau d'effectifs. Dans les deux cas, les agrégats simples [79] tolèrent bien l'insertion d'un grand nombre d'éléments. Par ailleurs, leur calcul repose sur peu d'opérations, ce qui les rend rapides.

— **Moments.** Les moments constituent la première grande famille d'agrégats simples et le moment d'ordre 1 ($\overline{x}(1)$) est sans doute le plus connu de tous puisqu'il correspond à la moyenne arithmétique. De façon plus générale, on peut définir pour un ensemble de n valeurs $x1, x2, ..., xn$, correspondant par exemple à n tirages indépendants d'une variable aléatoire, son moment d'ordre m comme suit :

$$\overline{x}(m) = \sqrt[m]{\frac{1}{n}\sum_{i=1}^{n} x_i^m}$$

On retrouve bien grâce à cette formule que la moyenne arithmétique correspond au moment d'ordre 1 et la moyenne quadratique au moment d'ordre 2. Deux autres cas sont aussi à distinguer : lorsque m tend vers 0, on retrouve la moyenne géométrique et lorsque m tend vers l'infini on retrouve le maximum de l'ensemble de valeurs.
Cette famille d'agrégats est couramment utilisée depuis longtemps pour résumer des informations en raison de sa simplicité de calcul et la moyenne en raison de sa simplicité d'interprétation. C'est tout aussi vrai pour les flux de données car les moments sont très facilement calculables de façon incrémentale puisqu'il s'agit de mettre à jour une somme et un effectif. On peut par ailleurs noter qu'il est souvent possible de calculer des approximations des moments à partir de résumés plus complexes.

— **Quantiles.** Les quantiles représentent l'autre grande famille d'agrégats simples mais il s'agit cette fois-ci d'une fonction définie sur $]0,1[$. Le quantile d'ordre $0,5$ ($\hat{Q}(0,5)$) est lui aussi d'usage courant puisqu'il s'agit de la médiane. Si on considère toujours un ensemble de n valeurs $x1, x2, ..., xn$ ordonné, alors on définit le quantile $\hat{Q}(u)$ par :

$$\forall u \in]\frac{i-1}{n}, \frac{i}{n}[, \hat{Q}(u) = x_i$$

On retrouve donc bien la médiane pour $u = 0,5$. On peut également retrouver plusieurs quantiles classiques comme les déciles pour $u = 0.1, 0.2, ..., 0.9$ ou les quartiles pour $u = 0.25, 0.5, 0.75$. Bien que plus difficiles à calculer que la moyenne, les quantiles restent une façon simple, rapide et pratique de donner beaucoup d'informations sur une distribution. Cependant, il est très difficile de les calculer de façon précise sur des flux de données. En effet, un quantile est l'exemple type d'une valeur simple à calculer mais pour laquelle il faut connaître l'intégralité des valeurs du flux pour obtenir une valeur exacte. C'est pourquoi on cherche souvent à pouvoir calculer certains quantiles de façon approchée ([93] et [94]) ou à partir d'agrégats plus complexes.

— **Tableaux d'effectifs.** En ce qui concerne les données de type qualitatif, l'équivalent des agrégats simples est l'activité de comptage qui consiste à compter combien de modalités de chaque type sont présentes dans l'ensemble des observations. Cette solution est efficace si le nombre de modalités distinctes est faible mais peut rapidement devenir problématique si ce dernier est élevé.

3.3.1.2.2 Agrégats complexes

Les familles des quantiles et des moments représentent les deux familles d'agrégats simples qu'on peut considérer comme classiques. À ceux-là s'ajoute un grand nombre d'agrégats plus complexes pour une seule tâche ou un seul type de données. Ces derniers se différencient des agrégats précédents par le fait qu'ils sont conçus non pas pour une seule variable numérique mais plutôt pour un vecteur constitué de plusieurs variables. Traiter l'ensemble de ces variables en même temps permet de faire des économies d'espace mémoire.

Agrégats hiérarchiques. Une famille d'agrégats complexes concerne les agrégats prenant en compte les hiérarchies au sein des données à résumer. Ils permettent en général d'agréger ces données en fonction de cette hiérarchie que ce soit directement notamment dans le cas des données qualitatives ou comme partie intégrante d'un processus plus complexe. Un exemple simple est l'agrégation d'éléments en fonction du temps en regroupant tous les éléments par année, mois, jour, heure, etc. D'autres types de données comme les adresses IP par exemple sont aussi particulièrement adaptés à ce genre d'agrégation.

Cluster Feature Vector (CFV). Le CFV ou vecteur des caractéristiques de classe en français est un agrégat complexe introduit par Zhang qui l'a utilisé pour son algorithme de classification automatique BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [95]. Adaptée à des données volumineuses, l'approche BIRCH est basée sur une classification effectuée sur un résumé compact des données. C'est pourquoi il peut traiter un grand volume de données en utilisant une mémoire limitée. Cette méthode est incrémentale et n'effectue qu'une seule passe sur les données. Les informations nécessaires au clustering sont organisées sous forme d'arbre équilibré avec une taille limitée. Chaque entrée de l'arbre décrit une micro-classe et les nouveaux nœuds sont insérés dans l'entrée la plus proche. Son principe est simple : pour un ensemble d'éléments à résumer, on conserve l'effectif de l'ensemble et pour chaque variable la somme de toutes les valeurs prises par cette variable dans l'ensemble des éléments ainsi que la somme de leurs carrés. A partir de ces valeurs, on peut reconstituer les moments d'ordre 1 et 2 facilement et fournir des estimations sur certains quantiles.

Plus récemment, Aggarwal a adapté les CFV dans [96] pour résumer des ensembles d'éléments issus des flux de données. On peut noter à cette occasion une adaptation intéressante pour traiter l'étiquette de temps pour chaque élément d'un flux de données comme n'importe quelle autre variable. Ainsi, le CFV conserve donc la somme de toutes les étiquettes de temps mises sous une forme numérique ainsi que la somme de leurs carrés.
Par ailleurs, on peut noter que si aucun argument théorique n'empêche les CFV de tolérer un très grand nombre d'insertions, des problèmes liés au stockage de la somme des carrés de chaque variable peuvent survenir lors de l'implémentation. En ce qui concerne la vitesse, les CFV sont des structures rapides pour des éléments de faibles dimensionnalités. Lorsqu'il y a beaucoup de dimensions, le nombre d'opérations à effectuer pour chaque insertion les rend beaucoup plus lents.

3.3.1.3 Histogrammes

Une autre façon permettant de résumer un flux de données consiste à utiliser des histogrammes. Cette structure est souvent utilisée pour résumer des données qualitatives ou quantitatives.
Un histogramme sépare la population en un ensemble de groupes ou classes suivant des attributs particuliers. Pour chaque groupe, on conserve ses fréquences. Il est relativement facile d'utiliser les histogrammes pour des variables quantitatives afin de répondre à différents types de requêtes telles que les requêtes portant sur un intervalle. Des stratégies peuvent être élaborées pour améliorer la résolution des requêtes à partir de l'histogramme [97, 98]. Il suffit de déterminer l'ensemble des segments inclus dans l'intervalle spécifié par l'utilisateur.
Dans le contexte des flux de données, cette approche devrait permettre une meilleure représentation d'un échantillon issu des flux. Plusieurs types d'histogramme sont proposés dans la littérature dont les principaux sont détaillés dans la suite.

3.3.1.3.1 Histogramme simple

L'histogramme simple est la version de base de l'histogramme ; il ne concerne qu'une variable quantitative et possède un seul paramètre, à savoir le nombre k de cases dans l'histogramme. Il existe un équivalent pour les variables qualitatives appelé diagramme en bâtons. Pour les données quantitatives, k donne le nombre d'intervalles dans lequel on peut diviser le domaine de définition de la variable. Pour les données qualitatives, k indique en combien de groupes les modalités doivent être regroupées. A l'arrivée de chaque nouvel élément, l'effectif de la case qui lui correspond est incrémenté de 1, les autres ne changent pas. La construction en ligne de ce type d'histogramme ne pose aucune difficulté puisqu'il s'agit uniquement d'attribuer chaque élément à la case qui lui correspond. Cela en fait des structures rapides en terme d'exécution.

3.3.1.3.2 Histogramme à profondeur uniforme (équidistribué)

L'histogramme à profondeur uniforme est une forme améliorée de l'histogramme simple. Le principe de base est le même mais cette fois les frontières des cases ne sont pas choisies arbitrairement mais de façon à ce que chaque case de l'histogramme ait un effectif aussi semblable que possible. Cela revient à calculer les quantiles de la distribution, les valeurs déterminant la limite entre les cases. Cette méthode permet de mieux équilibrer l'effectif de l'histogramme.

3.3.1.3.3 Histogramme V Optimal (équivariant)

Pour équilibrer l'histogramme, l'histogramme équivariant prend le problème de l'approche opposée à celle de l'histogramme équidistribué. En effet, il cherche non pas à

équilibrer l'effectif des différentes cases mais à minimiser leur variance de façon à produire des cases aussi homogènes que possible même si leurs effectifs sont variables. Ces histogrammes sont étudiés depuis longtemps dans le cadre de l'optimisation des SGBD. Au cours de ces études, des méthodes incrémentales ont été développées pour traiter des flux de données [99].

3.3.1.3.4 Histogramme fin-biaisé et requêtes Iceberg

Plusieurs applications maintiennent des agrégats simples (tel que le COUNT) dans un attribut afin de détecter des valeurs d'agrégats au-dessus d'un certain seuil. Ces requêtes sont dénommées requêtes Iceberg [100]. Elles sont utilisées dans différents domaines tels que l'extraction de données, la fouille de données, la détection de copies et le regroupement. Par exemple, un moteur de recherche pourrait être intéressé par la collecte de termes de recherche qui figurent dans plus de 1% de requêtes. D'autre part, les événements fréquents sont largement utilisés pour l'analyse des tendances. L'histogramme fin-biaisé permet donc de répondre à des requêtes Iceberg. Dans [100], les auteurs présentent un algorithme efficace pour calculer les résultats des requêtes Iceberg sur des données stockées sur disque. Leur algorithme nécessite plusieurs passes ; ce qui est incompatible avec les flux de données. Manku et Motwani ont présenté dans [101] un algorithme aléatoire et un autre déterministe afin de calculer la fréquence dans les requêtes Iceberg sur les flux de données. L'algorithme aléatoire utilise l'échantillonnage adaptatif dont le principe est que tout événement qui représente une fraction ϵ des événements fera probablement partie de l'échantillon uniforme de taille $\frac{1}{\epsilon}$. En revanche, l'algorithme déterministe maintient un échantillon des événements distincts avec leur fréquence.

3.3.1.4 Sketchs

Les sketchs sont des structures de données compactes et de petites tailles. Ils sont utilisés pour des tâches bien spécifiques telles que l'estimation du nombre d'événements distincts dans une liste. Ils utilisent des techniques probabilistes qui visent à projeter aléatoirement chaque élément dans un espace particulier en utilisant des fonctions de hachage et à ne garder que les composantes les plus pertinentes gagnant ainsi de l'espace tout en préservant l'essentiel de l'information. Plusieurs types de sketchs sont utilisés dans le cadre des flux de données.

3.3.1.4.1 Flajolet Martin (FM)

FM [102] permet d'estimer le nombre de valeurs distinctes dans une série d'événements. Cette méthode a été développée en 1985 pour résumer une masse importante de données dans les grandes bases de données. En raison de ces principales caractéristiques (incrémentale et rapide), il est parfaitement adapté au contexte des flux de données. L'inconvénient de cette approche est qu'elle ne présente de bons résultats que pour un seul agrégat (DISTINCT COUNT). De plus, ses performances se dégradent avec le nombre d'éléments différents insérés.

3.3.1.4.2 Filtre de Bloom

Les Filtres de Bloom représentent une structure compacte introduite par Burton H. Bloom dans [103]. Cette structure permet de mémoriser, dans un espace fini, un ensemble d'événements. L'intérêt d'une telle structure est de vérifier rapidement si un événement a déjà été inséré ou non dans le filtre. Plusieurs extensions ont été développées autour des filtres de Bloom. *Almeida et al.* ont introduit dans [104] le "scalable Bloom filter" qui permet de construire des filtres de taille variable. De plus, dans [105], *Fan et al.* présentent le "counting Bloom filter" afin de gérer la suppression dans ces filtres. Cependant, avec

toutes ces extensions, la construction du filtre n'est plus aussi rapide que la version standard. Nous utilisons les filtres de Bloom dans la cinquième contribution de cette thèse (chapitre 9).

3.3.2 Conception complexe d'un résumé de flux de données

Les sous-sections précédentes ont décrit des méthodes destinées à résumer l'information en prenant en compte les contraintes de traitement liées aux flux de données mais sans tirer avantage de leur structure temporelle.
Or, ce qui distingue un flux de données d'une vaste quantité de données, c'est avant tout son rapport au temps. En effet, chaque élément d'un flux dispose d'une étiquette issue d'un ensemble ordonné quelconque ; une des représentations classiques du temps étant le format jj/mm/aaaa ou le nombre de secondes.
Dans ces approches, des techniques de fenêtrage donnant la possibilité d'archiver des résumés couvrant l'intégralité du flux avec un espace borné sont utilisées afin de produire des résumés dits généralistes.
Dans [79], un résumé généraliste doit remplir 4 conditions : traiter un large champ de requête, permettre des analyses supervisées des données, autoriser des tâches d'analyses exploratoires et prendre en compte la dimension temporelle. Ce qui revient à définir ce résumé selon 2 aspects : l'aspect fonctionnel (réaliser des analyses variées) et l'aspect conceptuel (combiner organisation spatiale et organisation temporelle).
Dans [79], est introduit la notion de "résumé généraliste" en donnant une définition réaliste de celle-ci. Pour ce faire, les auteurs se basent sur une proposition : restreindre la classe des requêtes concernées par le résumé aux seules requêtes de type SELECT et COUNT où la sélection, prise ici au sens de filtrage, peut se faire sur toutes les variables à l'exclusion des identifiants ; les requêtes portant sur un objet spécifique sont donc exclues du champ du résumé généraliste.
Cette restriction laisse une très grande expressivité au résumé généraliste. En effet, la capacité à répondre exactement à toute requête de type SELECT AND COUNT sur le temps et les variables descriptives revient à connaitre exactement la densité jointe sur l'espace TxD (T : espace des estampilles temporelles, D : espace de description des événements du flux[organisation spatiale]). En d'autres termes, le résumé généraliste permet une estimation approchée de la densité jointe. Alternativement, une estimation approchée de la densité jointe permet de répondre approximativement à toute requête de type SELECT et COUNT. Un résumé généraliste est donc une structure de données permettant une approximation optimale de la densité jointe sur TxD sous les contraintes d'espace-mémoire, de temps de mise à jour / de construction et d'exécution de requêtes. Partant de cette définition, un résumé est dit généraliste s'il respecte les 4 conditions suivantes :

— Respect des contraintes d'espace mémoire et de puissance de calcul ;
— Expression sous forme de variables appartenant à TxD ;
— Approximation de toute requête SELECT et COUNT sur TxD ;
— Calcul de l'erreur d'approximation en fonction des ressources disponibles (mémoire, CPU).

3.3.2.1 Organisation temporelle complexe

Il s'agit d'une organisation particulière qui gère l'historique en pondérant le niveau de détail par l'âge des données. Ces fenêtres ont une longueur variable qui croît avec le temps de sorte que les événements les plus récents sont observés dans des fenêtres plus petites et les événements les plus anciens dans des fenêtres plus grandes. L'idée ici est de décrire

de façon précise les événements les plus récents. Dans [106] sont définie deux techniques permettant l'implémentation de ces structures. On distingue ainsi les fenêtres inclinées et le système de clichés.

3.3.2.1.1 Système des fenêtres inclinées

Une technique permettant une meilleure gestion de la dimension temporelle lors du résumé d'un flux de données est la technique des fenêtres inclinées. Encore appelées *Titled Time Frame* en anglais, les fenêtres inclinées sont des structures qui permettent de modéliser et compresser la dimension temporelle. Elles permettent aussi la reconstitution des informations sur une vaste période temporelle. Au lieu de s'appuyer sur des propriétés soustractives des résumés, elles s'appuient sur leurs propriétés additives. L'idée ici est d'avoir des résumés couvrant des périodes temporelles de taille variable, plus fines pour le présent et plus granuleuses pour le passé lointain et de pouvoir ensuite assembler les fragments en fonction des besoins.
Dans [107, 2] sont présentées deux implémentations du système de fenêtres inclinées : le modèle des fenêtres temporelles naturelles et le modèle logarithmique. Ces deux modèles se distinguent par leur découpage du temps. Les niveaux de granularité sont précisés par les applications.

3.3.2.1.1.1 Modèle naturel des fenêtres inclinées Dans ce modèle, la période est structurée en plusieurs granularités basées sur une échelle naturelle du temps.
Comme le montre la figure 3.1, on stocke pendant une heure les informations à la minute. Ensuite, pendant une journée, les informations sont stockées à l'heure, etc. Ce modèle permet une réduction importante du nombre de valeurs conservées. Par exemple, dans ce modèle, pour conserver les données d'une année, on a besoin de 126 valeurs (60 + 24 + 30 + 12) au lieu de 525726 valeurs (365 * 24 * 60).

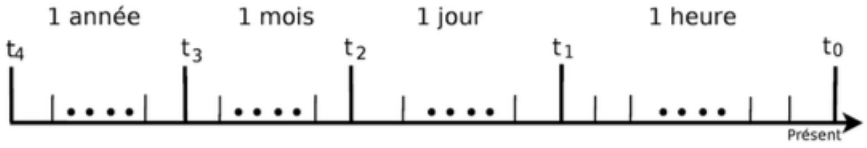


FIGURE 3.1 – Modèle naturel des fenêtres inclinées [2].

3.3.2.1.1.2 Modèle logarithmique des fenêtres inclinées

Dans ce modèle, la période est structurée en plusieurs granularités basées sur une échelle logarithmique du temps. Dans le cas d'un log en base 2 (cf. Figure 3.2) pour les données d'une année avec une précision de 15 minutes, nous avons besoin de : $\log_2(365 * 24 * 4) + 1 \simeq 16$ unités de temps au lieu de $(365 * 24 * 4) = 35040$ unités.

FIGURE 3.2 – Modèle logarithmique des fenêtres inclinées [2].

3.3.2.1.2 Système des clichés

Une autre technique permettant de traiter efficacement la dimension du temps dans le

cadre des flux de données et plus particulièrement celui de la construction de résumés de flux de données est le système dit des clichés [96]. Pour ce modèle, il faut d'abord définir une structure appelée cliché (snapshot). Un cliché est une image qui décrit l'état du système à un instant donné. Par exemple, l'ensemble des tuples d'une table à un instant t est un cliché de cette table. Dans ce modèle, on mémorise des clichés à des intervalles de temps variables. Ces clichés sont représentés par leurs estampilles temporelles.

3.3.2.2 Algorithmes de résumés généralistes

Les avantages des algorithmes de résumés généralistes sont multiples. Ils permettent entre autres de prendre en compte l'évolution des données ainsi que l'exploration des données sur différentes parties du flux. Plusieurs algorithmes de résumés généralistes sont apparus. Certains se basent sur les techniques d'échantillonnage (StreamSamp [108]) ou sur un système de clustering (CluStream [96]) pour résumer les données d'un flux, d'autres utilisent une structure d'automate pour l'indexation des motifs fréquents du flux de données ([109, 110]). Nous présentons ci-dessous quelques algorithmes de résumés basés sur l'échantillonnage et le clustering des données.

3.3.2.2.1 StreamSamp

Il s'agit d'un algorithme de résumé généraliste basé sur l'échantillonnage des flux de données. Développé par *Csernel et al.* dans [108], StreamSamp combine à la fois une approche spatiale qui est l'échantillonnage aléatoire et une approche temporelle qui est le système des fenêtres inclinées en particulier le modèle logarithmique. Cet algorithme, à la fois généraliste et rapide dans son élaboration, est basé sur 2 étapes :

1. **Traitement en ligne** : cette étape permet de constituer les échantillons à partir du flux de données. Dès leur arrivée, les événements du flux sont échantillonnés à un taux fixe α et placés dans des échantillons de taille T. Lorsque T est atteinte, StreamSamp mémorise sur le disque l'échantillon ainsi que sa date de début et de fin de constitution. L'ordre 0 est associé à cet échantillon. On procède par la suite à un second échantillonnage sur le flux. Plus on avance dans le temps, plus le nombre d'échantillons d'ordre 0 constitués augmente. Comme il est impossible de conserver à l'infini ces échantillons, StreamSamp procède à la fusion des ses échantillons suivant le principe des fenêtres inclinées. Le nouvel échantillon construit couvre la même période temporelle que ses deux échantillons parents mis bout à bout. Il est conçu par rééchantillonnage aléatoire (sélection de $\frac{T}{2}$ événements de chacun de ses échantillons parents).
2. **Traitement hors ligne** : cette étape permet l'exploitation du résumé conservé pour une période donnée P. On commence par constituer l'échantillon final E_f sur lequel va porter l'analyse. Pour cela, on concatène les échantillons d'ordres différents inclus dans P afin de reconstituer le flux de cette période. On associe par la suite un poids à chaque événement appartenant à E_f de façon à conserver la même représentativité pour tous les événements de cet échantillon. Pour le calcul du poids, on fait intervenir l'ordre auquel est associé l'élément : $2^{ordre} x \frac{1}{\alpha}$.

Grâce à son échantillonnage aléatoire, StreamSamp ne dépend pas du taux d'arrivée des flux. Ainsi, plus le taux d'échantillonnage augmente, plus les événements sont pris dans les échantillons. En revanche, les performances de StreamSamp se dégradent avec l'ancienneté. En effet, les anciens éléments ont un poids croissant pour une taille fixe de l'échantillon. Par conséquent, si un échantillon contient des événements récents (poids beaucoup plus faible) et des événements anciens, ces derniers vont considérablement augmenter la variance dans le résultat des requêtes.

3.3.2.2.2 CluStream

Aggarwal [96] a adapté les CFV (Cluster Feature Vector) pour résumer des ensembles d'éléments issus de flux de données. La plupart des algorithmes présentés jusque là permettaient de traiter des flux de données avec un nombre très réduit de passes sur les données et de façon incrémentale. Cependant, ces algorithmes ne prennent pas en compte l'évolution des données et par conséquent la qualité des classes produites devient mauvaise si les données évoluent de façon importante dans le temps.
CluStream est un algorithme de résumé généraliste dont l'objectif est la classification automatique des données numériques. Il fournit une structure particulièrement adaptée au résumé des flux de données. Cette méthode se base sur le croisement d'une approche spatiale permettant de conserver des statistiques sur des classes et d'une approche temporelle qui permet de gérer le résumé au cours du temps.
L'objectif de CluStream est de construire un résumé sous forme d'un ensemble de microclasses évolutives, régulièrement mémorisées dans des clichés. Pour cela, Aggarwal s'est inspiré de l'algorithme BIRCH [95] en utilisant la structure des CFV pour représenter une classe ou une sous-classe. Cette structure conserve des données statistiques qui résument l'ensemble des événements d'une micro-classe sans pour autant conserver les événements eux-mêmes. Afin de résumer des ensembles d'événements issus du flux de données, CluStream opère en 3 étapes :

1. **Initialisation** : il s'agit d'initialiser k micro-classes de départ. Pour cela, il utilise l'algorithme de classification $k-means$ sur les événements du début du flux. Ces microclasses vont contenir les informations statistiques des CFV.
2. **Etape de traitement en ligne** : il s'agit de la mise à jour des micro-classes. En effet, lorsqu'un événement arrive, l'algorithme détermine la classe qui lui est proche. L'événement est alors absorbé par cette micro-classe. Le CFV associé à la microclasse est ainsi mis à jour. Si aucune micro-classe proche n'est trouvée, une nouvelle micro-classe est créée ne contenant que cet élément. Étant donné que la taille de la mémoire est bornée, on ne peut pas créer un nombre infini de micro-classes. Deux stratégies sont alors possibles : soit l'algorithme fusionne deux micro-classes, soit il supprime la micro-classe la plus ancienne. Un critère d'ancienneté est alors donné par Aggarwal, basé sur l'hypothèse que les étiquettes temporelles des événements au sein de chaque micro-classe suivent une distribution gaussienne. Par ailleurs, un système de cliché est mis en place pour mémoriser l'état de toutes les micro-classes. Ces clichés sont ensuite conservés selon une structure temporelle.
3. **Etape de traitement hors ligne** : dans cette étape, on utilise les propriétés mathématiques (additives et soustractives) des CFV afin de suivre l'évolution des microclasses. On effectue des opérations de soustractions des clichés afin d'obtenir le contenu du flux entre deux prises de clichés. Pour ce faire, le système se réfère aux historiques des micro-classes qui ont été conservés (historiques de fusion, d'insertion et de suppression). Cette technique permet de fournir, dans la limite du nombre de clichés pris, une version du résumé pour n'importe quelle section du flux entre la prise de deux clichés.

En se basant sur une technique de clustering couplée avec une structure de système de clichés, CluStream permet de conserver des clichés consécutifs pour n'importe quel horizon temporel. Ceci permet de suivre l'évolution des flux de données au cours du temps. Cependant, son inconvénient est le traitement lourd des calculs de distance. Ce qui ne convient pas pour des flux de données très rapides. Une deuxième limite de cet algorithme concerne le nombre élevé de paramètres et dont certains dépendent de la nature même du flux de données et de la vitesse d'arrivée des événements (exemple le critère d'ancienneté d'une micro-classe).

3.3.3 Bilan

Les SGFD permettent d'exprimer des requêtes continues qui s'évaluent au fur et à mesure sur un flux ou sur des fenêtres. Ces requêtes doivent être spécifiées avant l'arrivée du flux. De nouveaux besoins peuvent parfois apparaître après le passage de l'information. Dans ce cas, le système ne pourra plus répondre aux requêtes posées car toutes les données n'appelant aucun traitement sont définitivement perdues. Il est donc nécessaire dans certains cas de conserver un résumé du flux de données.
Dans cette section, nous avons présenté les différentes approches permettant de résumer un flux de données individuel. Les résumés créés peuvent être classés selon leur fonctionnalité : certains sont destinés à une tâche particulière (sketch par exemple), d'autres permettent de réaliser des analyses variées sur les données. Le choix d'une approche n'est pas arbitraire mais doit respecter les exigences et besoins attendus (exemple : précision des résultats, structure du flux, domaine d'application) et les ressources disponibles.
Toutes ces approches de résumé se concentrent sur les flux de données. En effet, les approches de résumé et algorithmes présentés dans cette section permettent de résumer des données afin de les rendre disponibles pour une exploitation ultérieure. Notons qu'il existe d'autres approches permettant de résumer les données et flux de données telles que la Réduction des données, l'Échantillonnage stratifié, l'Agrégation temporelle des flux de données, etc. Cependant, les flux de données sont issus de sources diverses et variées (réseaux de capteurs, réseaux sociaux, Internet des Objets, etc.). Ainsi, dû à l'hétérogénéité et la multi-modalité des flux de données, les techniques de croisement, d'intégration, d'échantillonnage et de résumé nécessitent d'être adaptées suivant le modèle RDF qui est le format standard de représentation des données hétérogènes sous un modèle commun.

3.4 Conception de résumés de flux de graphes RDF

A l'image des flux de données classiques, les flux de données RDF ne peuvent pas être stockés dans leur totalité. En effet, ces données sont le plus souvent volumineux et verbeux et leur interrogation en temps-réel (requête SPARQL) est très couteuse.
Plusieurs travaux récents se sont intéressés au résumé de flux de graphes RDF classés en deux principales catégories d'approches : la catégorie des approches d'agrégation et de regroupement [111, 112, 113] et celle des approches d'extraction de structures [3, 114]. Dans la suite de ce chapitre, nous exposons les techniques proposées dans quelques travaux au niveau de chaque catégorie d'approches.

3.4.1 Agrégation et regroupement

L'approche d'agrégation et de regroupement consiste à identifier et créer des groupes de nœuds dans le graphe d'entrée (soit G), en se basant sur la similitude des attributs de ces nœuds et également sur les relations de voisinage qui existent entre ces nœuds. Certains travaux de recherche se sont basés sur cette approche pour proposer des solutions de résumé de graphes RDF. Dans [111, 3] est proposée une approche dont l'objectif primaire est de faciliter l'utilisation de SPARQL aux utilisateurs. En effet, vu la diversité des données, il est difficile de construire des requêtes SPARQL sans connaître ni la structure derrière les données ni le vocabulaire des données à interroger. Ceci constitue un vrai problème pour les utilisateurs. La méthode proposée dans cet article permet aux utilisateurs de formuler des requêtes SPARQL rapidement et aisément sur plusieurs sources de données hétérogènes même si la structure et le vocabulaire de ces données leur sont inconnus. Cette méthode consiste à aider l'utilisateur lors de la formulation de la requête en lui

recommandant des éléments structurels de requête dès que possible. Pour atteindre cet objectif, la méthode se base sur le résumé de graphes de données. Dans la suite, nous allons définir, quelques notions essentielles pour la compréhension de la technique utilisée dans [3] avant de présenter la technique de résumé.

3.4.1.1 Définition de quelques notions essentielles

— **[Dataset]** est un sous-graphe du graphe G à résumer.

— **[NodeCollection]** est l'ensemble de nœuds qui partagent des propriétés particulières. Cet ensemble est divisé en deux autres ensembles non chevauchants. Il s'agit de l'ensemble des entités et l'ensemble des nœuds vierges.

— **[EntityCollection]** est l'ensemble des nœuds entité (les nœuds qui possèdent des liens dont ils sont le nœud départ) du même Dataset qui ont des propriétés similaires.

— **[BlankCollection]** est l'ensemble des nœuds appelés terminaux (nœuds feuille) du même Dataset qui sont des cibles des liens ayant le même libellé et ayant la même source de EntityCollection.

3.4.1.2 Approche de résumé utilisée

Dans cette approche, le principe utilisé pour résumer le graphe est la modification de la structure du graphe pour permettre son parcours rapide par des indicateurs définis. Pour cela, il existe un modèle (figure 3.3) qui représente la forme sous laquelle le résumé de chaque graphe sera obtenu à la fin des opérations de résumé. Ce modèle définit

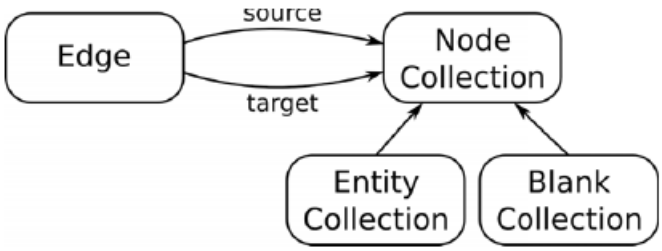


FIGURE 3.3 – Vocabulaire RDF pour le résumé du graphe de données dans [3].

deux classes principales : Edge (Lien) et NodeCollection (Noeud). Un NodeCollection peut être subdivisé en un EntityCollection ou un BlankCollection. Les Edges et les NodeCollections sont associés par l'intermédiaire des propriétés source et cible (target). La propriété source est le nœud où débute le lien et la propriété cible étant le nœud arrivée du lien. Les Edges et les NodeCollections ont également les propriétés :

— *origine* qui indique le Dataset dans lequel ils ont été définis

— *cardinalité* qui indique le nombre de fois qu'ils apparaissent dans le dataset ;

— *étiquette* qui définit le nom ou libellé associé à un nœud ou à un lien.

Avec cette méthode, le résumé se fait sous deux angles : en fonction de la classe et en fonction de l'attribut. La caractéristique de classe regroupe les nœuds d'entité qui partagent des classes identiques, comme la classe Article dans la figure 3.5 qui regroupe les nœuds A1 et A2. La collection d'entités résultante est marquée avec l'ensemble d'étiquettes de classe. Le regroupement par attributs est nécessaire lorsqu'un nœud d'entité n'a pas de

définition de classe. L'hypothèse est que les entités ayant des attributs similaires sont susceptibles d'appartenir à la même classe. Le résultat qu'ils ont obtenu en appliquant leur méthode de résumé est présenté comme suit.

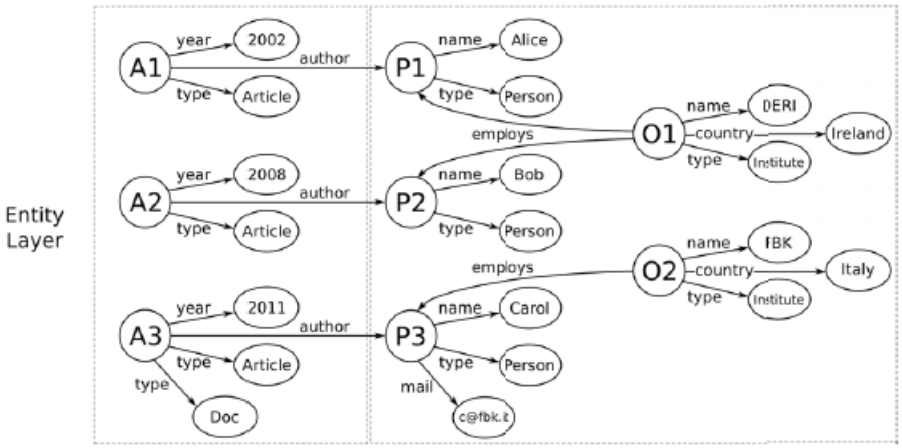


FIGURE 3.4 – Graphe G à résumer dans [3].

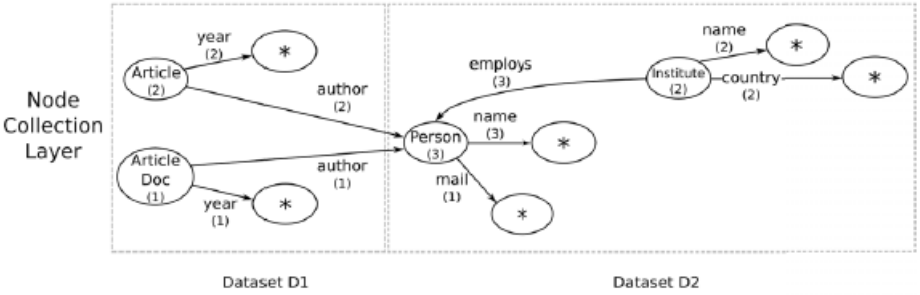


FIGURE 3.5 – Regroupement des noeuds identiques dans [3].

FIGURE 3.6 – Regroupement en Dataset dans [3].

Il existe d'autres travaux rangés dans cette même catégorie. L'objectif de tous ces travaux est le même : créer des groupes de nœuds et de liens suivant la similitude des attributs des noeuds et les relations de voisinage. Contrairement à l'approche dans [112] qui propose de supprimer les nœuds littéraux du graphe afin de réduire la taille du graphe RDF, celle dans [113] propose d'exploiter les nœuds littéraux dans le but de calculer la similarité entre les entités du graphe.

3.4.2 Extraction structurelle

Cette catégorie d'approche, quant à elle, est basée sur l'extraction d'un schéma où le graphe récapitulatif est obtenu sur la base d'une relation d'équivalence entre le graphe RDF G et une classe d'équivalence sur les nœuds de G.
Dans [112] est proposée une suppression des nœuds littéraux du graphe afin de réduire

la taille du graphe RDF. Les travaux dans [3] se sont basés sur cette catégorie d'approches pour proposer des résumés de graphes RDF. L'objectif de tous ces travaux est le même : créer des groupes de nœuds et de liens suivant la similitude des attributs des nœuds et les relations de voisinage.

3.4.3 Bilan

La notion de résumé de graphe RDF a été abordée au niveau de plusieurs travaux de recherche. Dans les deux sections précédentes de ce chapitre, nous avons présenté les deux grandes catégories d'approches exploitées. Vu le nombre important de publications existantes dans ces deux catégories, il nous serait difficile d'exposer toutes les techniques de résumé de graphes publiées dans chaque catégorie d'approches. En fonction du domaine d'étude, du domaine d'application et des capacités dont il dispose, l'utilisateur fera un choix entre les deux catégories d'approches. Ensuite il choisira également la technique qui lui convient le mieux selon ses critères définis et ses objectifs. Un bon résumé de graphe doit être lui même un graphe, avoir une taille plus petite que le graphe RDF source et son évaluation doit fournir un résultat assez approximatif des résultats d'évaluation sur les données brutes. Nos travaux dans cette thèse respectent ces critères et en rajoutent d'autres tout en s'outillant de techniques et d'algorithmes additionnels.

3.5 Conclusion

Le fonctionnement des SGFD permet l'interrogation à la volée et en temps réel des données du flux par des requêtes continues. Ces requêtes doivent être spécifiées avant l'arrivée du flux. Cependant, de nouveaux besoins peuvent parfois apparaître après le passage de l'information. Dans ce cas, le système ne pourra plus répondre aux requêtes posées car toutes les données n'appelant aucun traitement sont définitivement perdues. Il est donc nécessaire dans certains cas de conserver un résumé du flux de données. De nombreuses techniques de résumés ont été développées comme les sketchs, l'échantillonnage, les agrégats, les histogrammes, etc. Ces techniques recouvrent des approches très variées en fonction de la nature du résumé à construire. Cependant nous pouvons distinguer quelques grandes catégories. Une première catégorie concerne les techniques déterministes où leur application répétitive sur le même jeu de données fournira toujours le même résultat. Il s'agit, entre autres, des histogrammes. Une autre catégorie correspond aux techniques probabilistes où le résumé a une probabilité donnée d'être sélectionnée. Ceci signifie que si on applique ces techniques à plusieurs reprises sur le même jeu de données, le résultat à l'issue du résumé pourra varier. C'est le cas de l'échantillonnage et des sketchs par exemple.

Nous pouvons ensuite distinguer les résumés spécialisés des résumés à visée généraliste. Toutes ces techniques fonctionnent correctement sur les flux de données. Leur application sur des flux de graphes RDF a fait l'objet de travaux de recherche ayant pour objectif principal la réduction de la taille du graphe RDF. Il est à noter qu'aucune des techniques proposées ne tient compte du graphe RDF dans sa globalité. En effet, ces techniques se focalisent uniquement sur les besoins exprimés par les utilisateurs pour résumer le graphe : auquel cas, il y a une perte importante d'informations car les utilisateurs ne sont pas forcément susceptibles de connaître toutes les données contenues dans le graphe. On note aussi qu'elles suppriment les données figurant aux niveaux des nœuds feuille (constitués le plus souvent de littéraux) du graphe, auquel cas, elles ne sont pas adaptées au domaine des réseaux de capteurs. Notre approche se détache des techniques utilisées jusqu'ici car cherchant à tenir compte à la fois des besoins des utilisateurs mais également des nœuds feuille du graphe car ils sont importants dans certains domaines d'application. En effet,

pour prédire par exemple, la survenance d'une tempête de neige, il faudra évaluer, la température de l'air, la force du vent, les chutes de neige, etc. qui sont des informations situées au niveau des nœuds littéraux (nœuds feuilles).

Chapitre 4

1ère contribution : échantillonnage et interrogation de flux de graphes RDF

Sommaire

4.1 Introduction

Aujourd'hui, nous produisons plus de données que de ressources pour les traiter. Notre utilisation quotidienne des réseaux sociaux (Facebook, Twitter, Linkedin, etc.), des plateformes de diffusion de contenus et services multimédias (YouTube, Flickr, iTunes, etc.), des réseaux de capteurs (observation, télérelevé, etc.), de l'Internet des objets (géolocalisation, déclenchement d'alarmes en temps-réel, etc.), etc. génère en continu d'énormes quantités de flux de données hétérogènes. Ces données étant souvent issues de sources diverses se présentent sous des formats hétérogènes (JSON, XML, CSV, RDF, etc.). Les systèmes devant les traiter et les échanger sont également différents et présentent ainsi un problème d'interopérabilité. Pour lever ce verrou d'interopérabilité, la communauté des bases de données s'est ouverte aux technologies du web sémantique à travers les trois principaux standards que sont RDF (Ressource Description Framework), SPARQL (Simple Protocol and RDF Query Language) et OWL (Web Ontologie Langage). Des initatives visant à appliquer des technologies du web sémantique au traitement en temps réel de flux de données ont favorisé l'émergence d'un nouvel axe de recherche organisé autour du groupe de travail du W3C RSP [1] (RDF Stream Processing). A l'instar des SGFD (Systèmes de Gestion de Flux de Données), ce groupe de travail a proposé plusieurs extensions de

1. https ://www.w3.org/community/rsp/

SPARQL pour la gestion des flux de données RDF. Les principales propositions de langages et/ou de systèmes RSP (Streaming SPARQL [15], SPARQL continu (C-SPARQL) [64], CQELS [18], SPARQL $_{stream}$ [1], EP-SPARQL [17] et Sparkwave [19]) étendent tous SPARQL mais adoptent des approches différentes pour traiter en temps-réel les flux RDF.

Étant donné que le volume et la vitesse souvent variable des données entrantes augmentent, il est difficile de traiter tout le contenu d'un flux. Les systèmes ont donc besoin de techniques pour l'allocation de ressources dynamiques ([115] et [116]) ou la réduction de la charge de données en entrée. Pour ce qui concerne la réduction de la charge de données en entrée, lorsque le débit du flux d'entrée est élevé (dépasse les capacités de traitement des ressources des moteurs RSP), les systèmes sont surchargés. Par exemple, un temps d'exécution de requête élevé comparé au débit du flux d'entrée entraînera une surcharge et donc une perte de données importante et une latence dans le traitement. Pour maintenir le rythme d'arrivée des données, le système permettrait de réduire une partie de la charge en fonction d'une méthode donnée. Aucune des extensions de SPARQL existantes n'implémente des mécanismes d'échantillonnage ou de résumé des données d'un flux.

Dans ce chapitre, nous proposons d'étendre le langage et le moteur C-SPARQL en ajoutant des opérateurs d'échantillonnage pour nous permettre de réduire, à la volée, les données RDF d'entrée, tout en préservant les liens sémantiques. Nous considérons d'abord un nouveau format d'entrée (orienté graphe) des données. En effet, C-SPARQL prend en entrée une suite de paires, où chacune est formée d'un triplet RDF et de son estampille temporelle. Cette forme de représentation ne garantit pas une préservation des liens sémantiques entre les différents triplets de l'échantillon (c'est-à-dire, après la phase d'échantillonnage des données contenues dans une fenêtre logique ou physique). Dans le contexte des flux de données RDF, les événements ou faits sont souvent représentés par un ensemble de triplets RDF, et non par un seul. Ainsi, au lieu du format triplet RDF (<*sujet*, *prédicat*, *objet*>, *estampille*), nous adoptons une approche orientée graphe où chaque graphe représente un événement ou un fait formé par un ensemble de triplets RDF estampillés. Nous étendons ensuite la syntaxe de requête de C-SPARQL et le module d'exécution continue de son architecture (Esper) en ajoutant des opérateurs d'échantillonnage.

Ce chapitre est structuré comme suit. La section 4.1 introduit ce chapitre. La notion de flux de graphes est introduite dans la section 4.2. Nous fournissons, dans la section 4.3, les trois algorithmes d'échantillonnage utilisés pour l'implémentation de nos opérateurs d'échantillonnage. Notre proposition d'extension de la syntaxe de requête et de l'architecture de C-SPARQL est détaillée dans la section 4.4. La section 4.5 présente l'évaluation des performances ainsi que les résultats obtenus. Enfin, la section 4.6 conclut ce chapitre.

4.2 Flux de graphes RDF

Dans cette section, nous présentons la notion de flux de graphes RDF adoptée pour l'approche d'échantillonnage et de traitement de requêtes dans C-SPARQL.

4.2.1 Flux RDF orienté graphe

La plupart des extensions existantes de SPARQL pour la gestion de flux RDF prennent en entrée une succession de triplets RDF. Ce modèle de représentation permet un traitement en temps-réel plus ou moins large des flux mais ne consomme que des triplets, en ignorant totalement la structure orientée graphe des données RDF. Dans ce cas, chaque

événement est distribué dans un ensemble de triplets. En considérant les flux de triplets successifs, il devient difficile voire quasi impossible de capturer les limites ou liens potentiels entre triplets d'un même événement (c'est-à-dire, des triplets faisant partie ou non d'un même événement). Par conséquent, cette succession de triplets RDF appartenant à un même événement devrait être regroupée sous un format orienté graphe RDF et annotée avec la même estampille.

Nous considérons un format orienté graphe des flux de données RDF en étendant la définition du format quadruplet (⟨<sujet, prédicat, objet>, estampille⟩) des flux RDF. Un graphe RDF peut être construit à partir d'un ou d'un ensemble de triplets RDF. Plus formellement, étant donné un ensemble de triplets st_r, nous définissons $G\langle st_r \rangle$ comme un graphe dirigé orienté où chaque nœud représente un sujet s ou objet o d'un triplet et chaque lien entre deux nœuds correspond au prédicat p du même triplet. La notion de connectivité dans la définition d'un graphe RDF est importante pour tous les triplets RDF qui forment un événement. En effet, les triplets dans un événement partagent entre eux un chemin unique ou multiple. Une connectivité dans le graphe RDF $G\langle st_r \rangle$ est une séquence de liens (prédicats) tel que $\forall 1 \leq i < n$, p_i partage un sommet (sujet ou objet) avec p_{i+1}.

Avec la définition ci-dessus, le flux de graphes RDF peut être simplement défini comme une séquence de paires $\langle G(st_r)_i, \tau_i \rangle$, où $G\langle st_r \rangle_i$ est un graphe RDF représentant un événement et τ_i l'instant où cet événement se produit (instant de génération ou d'arrivée au système de traitement).

$$(G\langle st_r \rangle_i, \tau_i)$$
$$(G\langle st_r \rangle_{i+1}, \tau_{i+1})$$
$$\ldots$$

Deux graphes RDF peuvent partager une même valeur d'estampille, ce qui signifie que leurs événements se produisent en même temps. Il faut également noter que deux triplets avec la même estampille ne figurent pas nécessairement dans le même événement. La figure 4.1 montre un exemple de flux de graphes RDF à partir de capteurs déployés dans un réseau de distribution d'eau potable. Les données collectées concernent la pression, le débit, la teneur en chlore, la température, etc. L'échantillonnage des flux RDF, lorsque les événements sont décomposés en une liste de triplets RDF individuels, peut rompre les liens sémantiques entre eux dans l'échantillon. Dans ce cas, les échantillons peuvent se retrouver avec des données sans signification et les requêtes sur ces derniers concerneront seulement un graphe RDF partiel et retournent évidemment des résultats faux ou incomplets. Les deux exemples dans les figures 4.2 et 4.3 illustrent bien une simulation d'échantillonnage de graphes RDF où on observe, dans l'approche orientée triplet (figure 4.2), des éléments des triplets qui disparaissent avec les liens rompus durant la phase d'échantillonnage tandis que celle orientée graphe (figure 4.3), maintient tous les liens d'origine entre triplets présents dans l'échantillon obtenu. Par conséquent, comme initié dans [117], nous adoptons une approche orientée graphe qui assure la préservation de la sémantique des données (c'est-à-dire des liens entre les sujets et les objets) après une opération d'échantillonnage.

Pour réduire la volumétrie des données, les approches orientées triplet ou graphe nécessitent des algorithmes d'échantillonnage adaptés à un environnement de flux de données en ne conservant qu'un échantillon du flux.

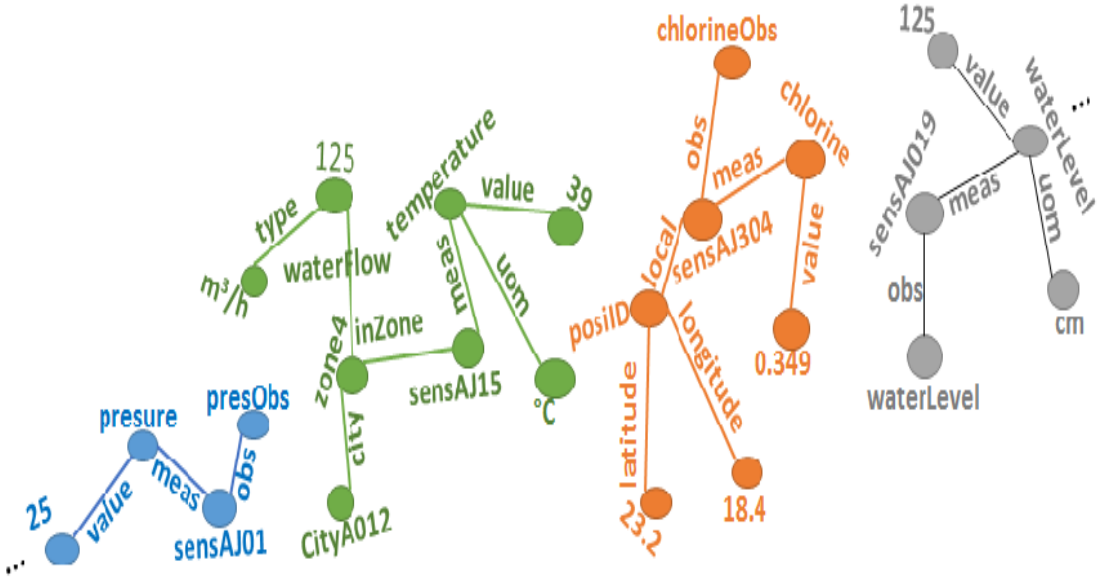


FIGURE 4.1 – Exemple de flux de graphes RDF issus d'un réseau de transport et de distribution d'eau potable.

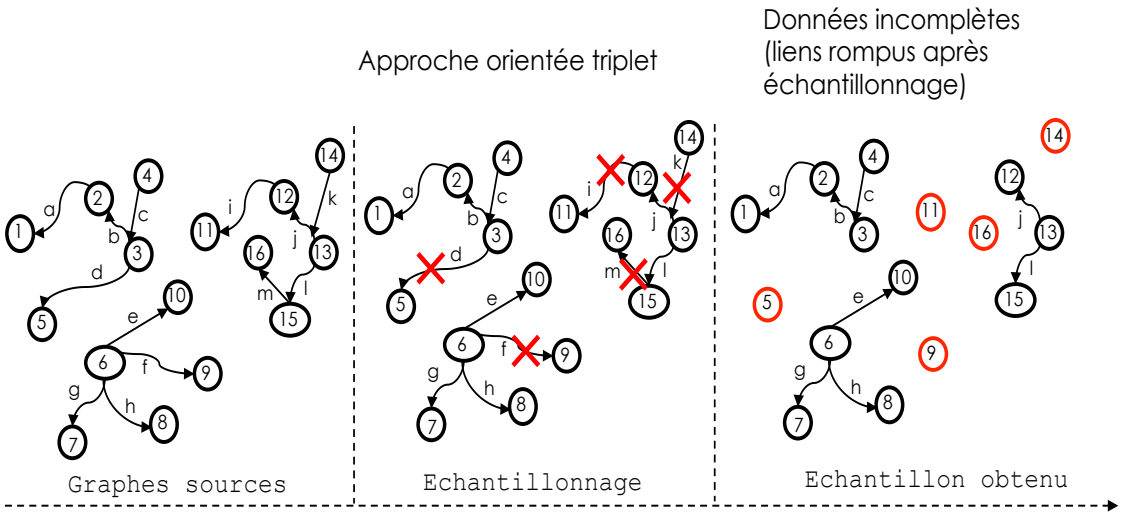


FIGURE 4.2 – Echantillonnage orienté triplet.

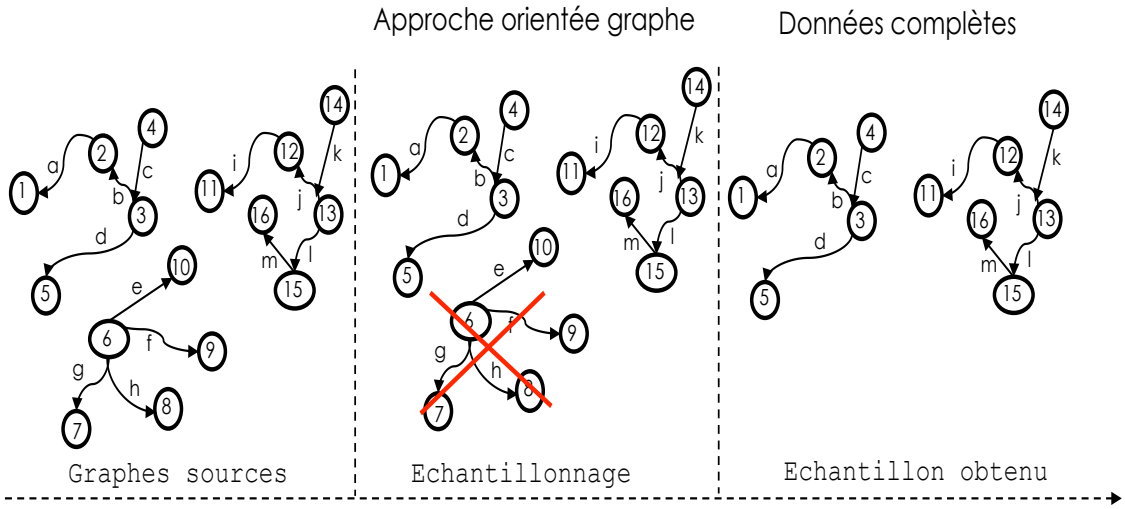


FIGURE 4.3 – Echantillonnage orienté graphe.

4.3 Approches d'échantillonnage

Les systèmes de traitement de flux RDF nécessitent un traitement continu, rapide et intelligent des données continues dans les flux. Du fait du volume et de la vitesse de génération des données, il est nécessaire d'extraire des échantillons de données à partir des flux en entrée. Le processus d'échantillonnage permet de réduire la charge de données à traiter et le coût de transmission des données dans le système.

Il existe plusieurs solutions de résumé et d'échantillonnage de flux de données. Nous en avons présenté quelques unes dans le chapitre précédent. Cependant, dans ce chapitre, nous choisissons d'utiliser trois (3) approches (échantillonnages aléatoire, réservoir et chaîne) non pas en matière de performances par rapport aux autres mais en guise de validation de notre extension du langage de requête présenté à la sous-section 4.4.1 et du module de traitement continu de C-SPARQL présenté à la sous-section 4.4.2. En effet, l'extension que nous proposons est assez modulaire et particulièrement adaptable aux algorithmes d'échantillonnage présentés dans le chapitre 3. Nous détaillons dans ce qui suit ces trois algorithmes.

4.3.1 Échantillonnage uniforme sans remplacement

L'échantillonnage aléatoire simple [118] peut être avec ou sans remplacement. Il sélectionne sans remplacement et avec la même probabilité p un échantillon aléatoire d'éléments (triplets ou graphes RDF) de taille n à partir d'un ensemble d'indices d'une fenêtre W d'éléments. L'indice d'un élément dans W ne peut être sélectionné qu'une seule fois.

Cette méthode d'échantillonnage reste très basique et a l'avantage d'être simple et facile à mettre en œuvre. Cependant, cette technique donne à tous les éléments de la fenêtre la même chance d'être inclus dans l'échantillon. Cela peut être considéré comme un inconvénient car dans le contexte des flux, nous sommes souvent intéressés par les données les plus récentes. Dans la constitution de l'échantillon, il faut donner plus de chance aux données relativement récentes.

4.3.2 Échantillonnage réservoir

L'idée principale de tout type d'algorithme d'échantillonnage réservoir [119] est de maintenir un échantillon aléatoire avec une taille fixe n dans un "réservoir". Après chaque processus de fenêtrage sur les flux RDF, un échantillon aléatoire de taille n est extrait. Initialement, nous avons placé les n premiers éléments du flux RDF reçus dans le "réservoir" R. Ensuite, chaque nouvel élément dans la fenêtre a la probabilité $\frac{n}{i}$ de remplacer l'élément d'indice r_i dans notre réservoir R. Cette méthode favorise nettement les triplets ou graphes RDF les moins récents du "réservoir" R (car $(\lim_{i\to\infty} \frac{n}{i} = 0)$). Par conséquent, avec cet algorithme, les triplets ou graphes les plus anciens sont plus susceptibles d'être inclus dans l'échantillon. Cet algorithme trouve ses limites à l'expiration des données dans une fenêtre glissante.

4.3.3 Échantillonnage chaîne

Les auteurs dans [120] ont décrit la technique d'échantillonnage chaîne qui consiste à construire un échantillon de triplets ou graphe RDF de taille n sur une fenêtre glissante de taille $\omega > n$. Pour les fenêtres glissantes, l'algorithme d'échantillonnage en chaîne génère aléatoirement des remplaçants parmi les éléments qui ont expiré avant de les stocker. En effet, dans ce type de fenêtre, les éléments ne faisant plus partie de la fenêtre courante

deviennent invalides, et doivent être remplacés s'ils appartiennent à l'échantillon.

L'algorithme 1 décrit notre extension de la méthode d'échantillonnage chaîne présentée dans [120]. Dans la première fenêtre, nous ajoutons des indices i dans l'échantillon avec la probabilité $p = \frac{min(\omega,i)}{\omega}$. L'indice successeur r_i de l'indice i est aléatoirement choisi dans l'intervalle $[i+1, i+\omega]$ et le remplace dans l'échantillon après son expiration (i sort de la fenêtre). Le successeur de r est aléatoirement choisi de la même manière dans l'intervalle $[r+1, r+\omega]$ et ce processus est ainsi répété indépendamment des précédents. Cette méthode est particulièrement adaptée aux fenêtres glissantes car elle permet de tenir à jour un échantillon mais tout en préservant sa représentativité. Cependant, l'algorithme chaîne nécessite des ressources mémoire importantes du fait du critère de sélection non redondant des indices successeurs.
Ces techniques d'échantillonnage nous permettent de faire face à la volumétrie et au taux

Algorithme 1 Algorithme d'échantillonnage chaîne

Require: chainSamp (ω, p)
R$_{epl} \leftarrow \varnothing$
S← put indexes (i) from the first window (with size ω) with probability $\frac{\mathrm{M}in(\omega,i)}{\omega}$
1: $i \leftarrow 0$ index
2: **while** i in S **do**
3: Select a random successor r_i with probability p between $i+1$ and $j+\omega$
4: R$_{epl} \leftarrow r_i$
5: **end while**
6: **while** each new index is added **do**
7: $i \leftarrow i+1$ //*move window index by on step*
8: Replace each expired index j in S by its successor r_j in R without redondancy
9: Choose a random successor for r_j between r_j+1 and $r_j+\omega$ without redondancy
10: **end while**
Return S

d'arrivée souvent élevé des flux RDF en réduisant la charge de données à traiter sans perdre trop d'information. Cependant, dans un environnement de flux, ces approches doivent être adaptées aux différents types de fenêtres logiques et physiques pour pouvoir être incluses dans un système de traitement "à la volée" des flux RDF. Nous pouvons ainsi étendre le module de fenêtrage d'un système afin d'y implémenter les opérations d'échantillonnage continu dans l'enchaînement des flux RDF.

4.4 Notre extension de C-SPARQL

Le système C-SPARQL [64] fournit une architecture modulaire pour le traitement des requêtes C-SPARQL sur les flux RDF. Cette architecture est composée de deux parties : un système de gestion de flux de données (STREAM ou Esper) et un moteur de requêtes SPARQL (Jena ou Sésame).

La figure 4.4 présente l'architecture de C-SPARQL. Le module *Query Translator* configure, initialise et distribue les parties continues et statiques d'une requête C-SPARQL. Cette tâche de répartition traite une requête C-SPARQL correcte et crée deux (2) instances :

1. ***ContinuousEngine*** se compose d'un système de gestion de flux de données (Esper dans la dernière version) qui traite à la volée les triplets RDF en leur appliquant le concept de fenêtre via une requête CQL. Ce module génère un ensemble de quadruplets ($sujet$, $prédicat$, $objet$, $estampille$) destinés au moteur SPARQL

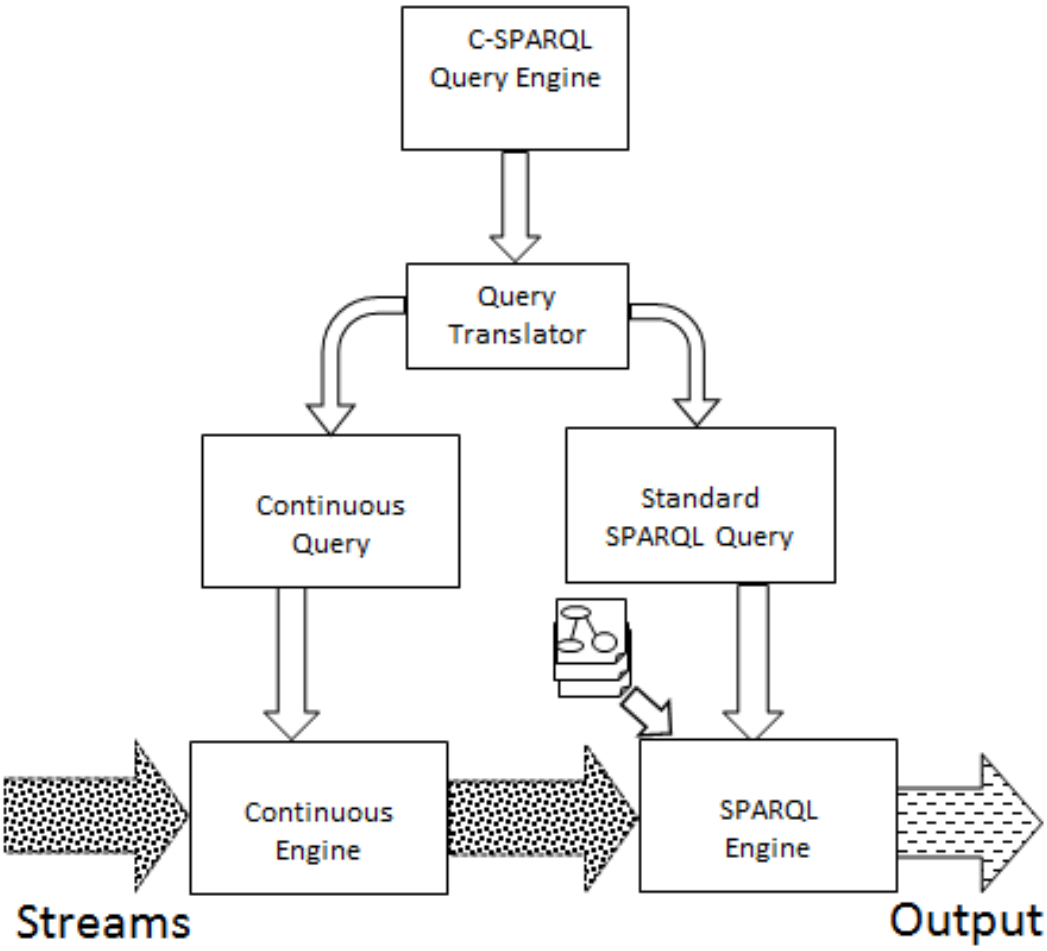


FIGURE 4.4 – Architecture de C-SPARQL .

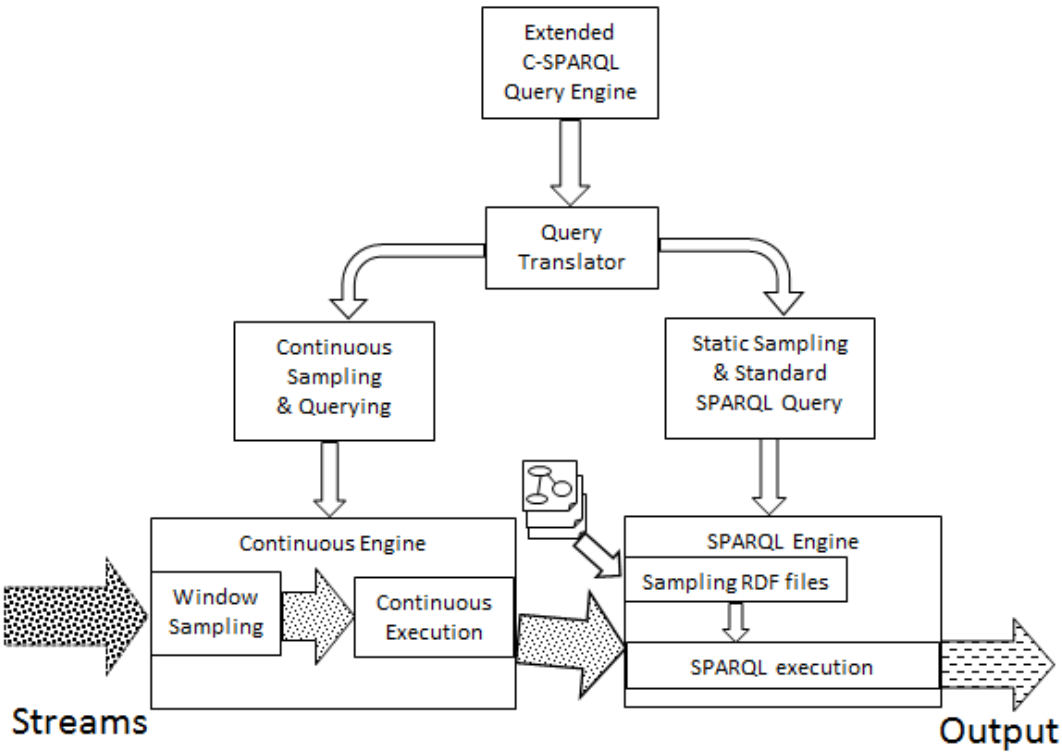


FIGURE 4.5 – Architecture étendue de C-SPARQL .

2. ***SparqlEngine*** est le composant dit “sémantique" dans cette architecture. Sa tâche principale est l’exécution de la requête SPARQL déduite de celle de C-SPARQL à chaque fois que le **ContinuousEngine** produit des quadruplets.

La contribution principale de ce chapitre est l’extension de l’architecture du moteur C-SPARQL afin d’y intégrer des opérateurs d’échantillonnage de flux RDF. Afin de réduire la charge des flux de données RDF et de ne traiter et stocker qu’un échantillon représentatif pour de futurs traitements tels que le raisonnement, notre contribution peut se traduire par :

1. La considération des flux de données des graphes en entrée au lieu d’une séquence ordonnée de triplets RDF pour la préservation des liens sémantiques.
2. L’ajout de trois (3) opérateurs d’échantillonnage dans la syntaxe de requête de C-SPARQL : `UNIFORM`, `RESERVOIR` et `CHAIN`.
3. La mise en œuvre des méthodes d’échantillonnage continu dans Esper.

4.4.1 Opérateurs d’échantillonnage

Comme indiqué ci-dessous, nous étendons la syntaxe de la requête C-SPARQL en ajoutant de nouveaux opérateurs dans les clauses **FROM STREAM** (source(s) des flux de données) et **FROM** (source(s) des données statiques).

1. *PREFIX $prefixName$: < IRI >
2. SELECT ‘?$variables'$
3. *FROM STREAM < S$tream$IRI > [W$indow$] **[SAMPLING Token]**
4. *FROM < S$tatic$IRI > **[SAMPLING Token]**
5. WHERE { ‘M$apping\ variables'$;|.
6. *FILTER (‘$condition'$)}
7. GROUP BY ‘?$variables'|expression$
8. HAVING ‘$aggregation\ condition'$

9. ORDER BY '?*variables*'

SAMPLING → UNIFORM | RESERVOIR | CHAIN
Token → [W*indow*] %P|*Size*
[W*indow*] → '*chain window size*'
%P → '*sampling percentage*'
Size → '*reservoir size*'

La syntaxe étendue contient de nouveaux opérateurs pour les méthodes d'échantillonnage appliquées en continu sur les flux de graphes RDF en entrée et les données statiques (les repository RDF). A titre d'exemple, la requête suivante échantillonne aléatoirement, avec un taux de 60%, les graphes en entrée et fournit toutes les minutes, tous les capteurs dont la valeur de pression détectée dépasse 27 (unité de mesure) durant les 10 dernières minutes.

```
REGISTER QUERY exceedsPressures AS
SELECT ?sensorID ?WaterPressure
FROM STREAM <http://waterdist.org/sens> [RANGE 10m STEP 1m] [UNIFORM %60]
FROM <http://waterdistrib.org/staticdata.rdf>
WHERE {?sensorID ex:hasPressure ?WaterPressure .
        FILTER (?WaterPressure > 27)
            }
```

4.4.2 Extension de l'architecture de C-SPARQL

Notre approche d'extension est basée sur la mise en œuvre de méthodes d'échantillonnage au sein d'Esper. La figure 4.5 montre l'extension proposée de l'architecture de C-SPARQL où ses modules traditionnels restent des plugins indépendants. Nous étendons les trois (3) modules *Query Translator*, *ContinuousEngine* et *SparqlEngine*.

1. Le module ***QueryTranslator*** analyse la requête reçue en vérifiant d'abord les opérateurs d'échantillonnage contenus dans les clauses FROM et FROM STREAM. Si la requête ne comprend pas d'opérateurs d'échantillonnage, les flux RDF en entrée sont traités en continu sans aucune phase d'échantillonnage. Dans le cas contraire, après validation des opérateurs d'échantillonnage, nous créons deux (2) instances *ContinuousSampQuery* et *SparqlSampQuery* qui seront respectivement traitées par *ContinuousEngine* et *SparqlEngine*.
2. Le module ***ContinuousEngine*** reçoit une requête continue avec les opérateurs d'échantillonnage associés. Chaque méthode d'échantillonnage est appliquée en continu sur une fenêtre de flux de graphes en entrée (*WindowSampling*). Nous traitons ensuite la partie CQL de la requête C-SPARQL sur des graphes de notre échantillon dans le buffer selon la syntaxe d'Esper. Les résultats sont transmis au troisième module *SPARQLEngine*.
3. Le module ***SPARQLEngine*** reçoit également des graphes de l'échantillon statique des repository (*SampStaticRDF*). Il exécute, en dernière étape, la requête SPARQL sur des graphes continus et statiques déjà échantillonnés.

Dans l'environnement des flux de données RDF, la volumétrie, le coût des calculs excessifs et le débit élevé représentent la principale baisse de performance constatée au niveau des systèmes RSP comme C-SPARQL. L'extension des modules **fenêtrage** et **moteur de requête** du système C-SPARQL a permis de prélever en continu des échantillons de graphes RDF suffisamment représentatifs pour permettre de répondre aux différentes requêtes C-SPARQL prévues. Cependant, les performances attendues (en termes de temps de calcul et de préservation de la sémantique) dans le processus d'exécution des requêtes C-SPARQL "étendues" à partir de l'échantillon peuvent être évaluées et comparées aux

scénarios d'exécution de requêtes C-SPARQL sources sur des données brutes (non échantillonnées).

4.5 Évaluation

Cette section évalue la qualité et la pertinence de notre extension de C-SPARQL. Pour ce faire, nous nous concentrons sur les performances réalisées en termes de temps d'exécution (sous-section 4.5.1) et de préservation de liens sémantiques (sous-section 4.5.2). Nous considérons, comme étude de cas, un ensemble de données provenant de capteurs déployés dans un grand réseau de transport et de distribution d'eau potable. Les données observées peuvent être la pression, le débit, le chlore ou la température. La gestion du réseau de distribution d'eau nécessite des capacités de traitement et de raisonnement en temps réel sur les cours d'eau afin de détecter rapidement les anomalies telles que les fuites d'eau, les pannes dans le réseau, etc. Nous considérons le traitement de 80000 graphes, où chaque graphe contient 10 triplets. Cet ensemble de données est envoyé sous forme de graphes et de flux de triplets respectivement à des débits de 500 graphes/seconde et 5000 triplets/seconde. Les évaluations sont réalisées sur un ordinateur avec un processeur à 2,66 GHz, Core 2 Duo et 4 Go de RAM.

4.5.1 Évaluation du temps d'exécution

Pour l'évaluation du temps d'exécution des requêtes, nous considérons la requête simple donnée ci-dessous. Le but de la requête est de fournir les valeurs moyennes de pression de l'eau mesurées par chaque capteur. La requête est effectuée sur 1000 graphes échantillonnés avec respectivement les opérateurs d'échantillonnage UNIFORM, RESERVOIR et CHAIN.

```
REGISTER QUERY AvgWaterPressure AS
PREFIX ex : <http ://waterdist.org/>
SELECT    ?sensorID    (AVG(?pressureValue)  AS  ?AvgPressure)
FROM STREAM <http ://waterdist.org/stream> [RANGE TRIPLES 1000] [SAMPLING [window] percent|size]
WHERE {?sensorID ex :hasPressure ?pressureValue . }
GROUP BY    ?sensorID
```

Pour les opérateurs d'échantillonnage (UNIFORM) et (RESERVOIR), nous observons sur les figures 4.6 et 4.7, l'évolution du temps de traitement des requêtes en faisant varier respectivement le pourcentage d'échantillonnage (*pourcentage*) et la taille du réservoir (*taille*). Enfin, avec l'opérateur CHAIN, nous évaluons sur la figure 4.8, le temps d'exécution de la requête sous deux conditions : le pourcentage d'échantillonnage variable et la taille de la fenêtre de la chaîne (*Window*).

Avec un échantillonnage aléatoire uniforme sans remplacement (figure 4.6), le temps d'exécution augmente en fonction du pourcentage d'échantillonnage. Nous notons également une tendance similaire avec l'échantillonnage de type réservoir (figure 4.7) en fonction de la taille du réservoir maintenue en mémoire. L'évolution du temps d'exécution avec l'échantillonnage en chaîne (figure 4.8) dépend de la taille de la fenêtre et du ratio. Quel que soit le pourcentage d'échantillonnage, le temps d'exécution des requêtes suit une tendance croissante. Cela peut s'expliquer par la sélection d'un successeur aléatoire sans redondance. Ainsi, les observations confirment les performances obtenues en réduisant à la volée la charge des flux en entrée.

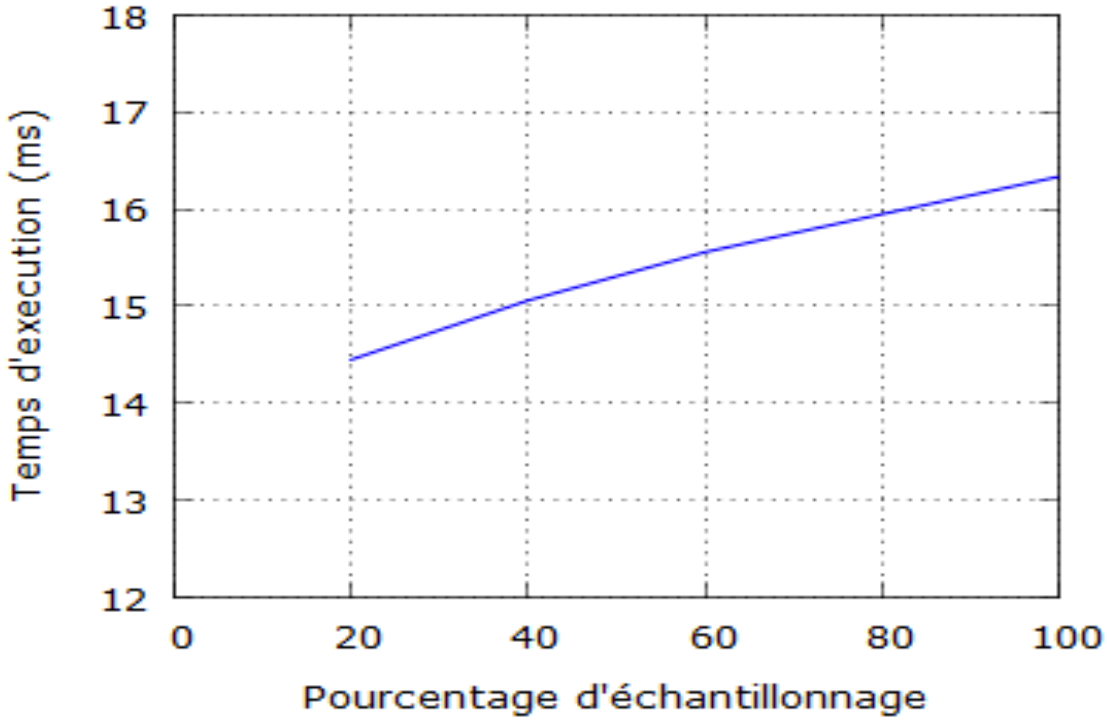


FIGURE 4.6 – Uniforme.

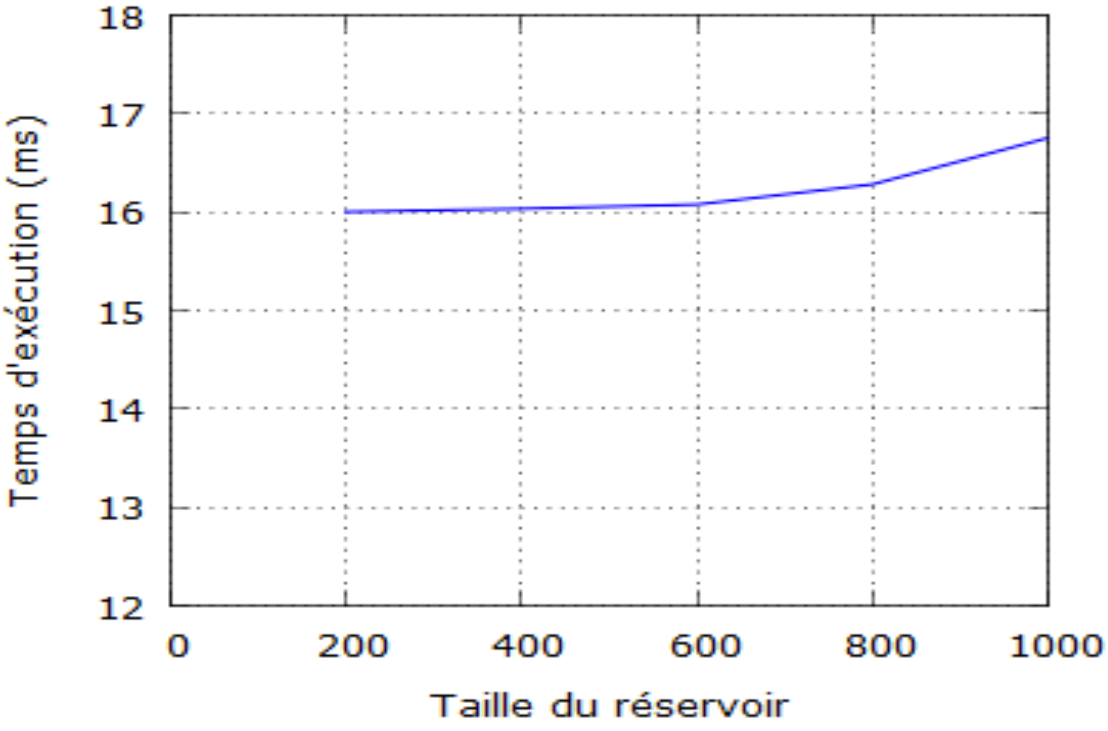


FIGURE 4.7 – Réservoir.

4.5.2 Évaluation de la préservation des liens sémantiques

Pour l'évaluation de la préservation des liens sémantiques entre les données de l'échantillon, nous considérons la requête suivante :

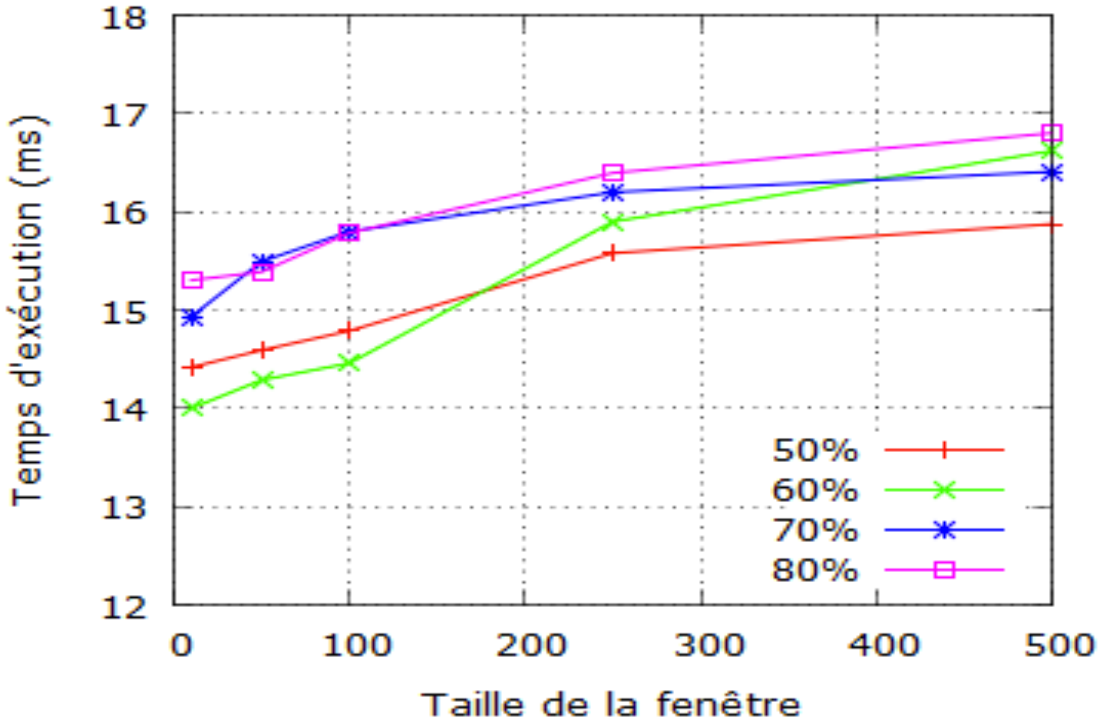


FIGURE 4.8 – Chaîne.

```
REGISTER QUERY sourceSensorID AS
PREFIX ex : <http ://waterdist.org/>
SELECT    ?sensorID    ?pressureValue
FROM STREAM <http ://waterdist.org/stream> [RANGE TRIPLES 10]
                              [SAMPlING [window] percent|size]
WHERE {?sensorID ex :hasPressure ?pressureMnemonic .
        OPTIONAL{?pressureMnemonic ex :value ?pressureValue .} }
ORDER BY    ?sensorID
```

Nous exécutons d'abord cette requête de manière continue, dans un scénario de base (sans aucune opération d'échantillonnage), puis avec un échantillonnage orienté triplet et enfin avec un échantillonnage orienté graphe. La requête sélectionne pour les 10 derniers graphes ou triplets observés dans l'échantillon, l'ID du capteur et sa valeur de pression correspondante. Dans le graphe, l'identifiant du capteur est connecté à sa valeur de pression capturée par l'intermédiaire d'un autre nœud. Nous calculons, pour chaque méthode d'échantillonnage (orienté triplet et orienté graphe), le nombre de résultats corrects et complets (par exemple, l'ID du capteur et sa valeur de pression correspondante). Nous calculons ensuite le taux de perte dans les deux cas (orienté triplet et orienté graphe) en utilisant la formule suivante :

$$\textbf{Taux de perte (\%)} = \frac{\textbf{NbrSE - NbrAE}}{\textbf{NbrSE}} * \textbf{100}, \text{ avec}$$

NbrSE = Nombre de résultats corrects et complets dans le scénario de base.
NbrAE = Nombre de résultats corrects et complets avec échantillonnage.

Le tableau 4.1 montre les taux de perte calculés en faisant varier les paramètres d'échantillonnage des méthodes uniforme et réservoir. Le nombre de résultats corrects et complets dans le scénario de base est **NbrSE=10625**.

Opérateur	Échantillonnage orienté triplet		Échantillonnage orienté graphe	
	Résultats corrects et complets	**Taux de perte (%)**	Résultats corrects et complets	**Taux de perte (%)**
UNIFORM				
P = 20%	19	**99.82**	2108	**80.16**
P = 40%	73	**99.31**	3614	**65.98**
P = 80%	181	**98.2**	7137	**32.82**
RESERVOIR				
$Size = 2$	24	**99.77**	2117	**80.07**
$Size = 4$	144	**98.64**	4210	**60.37**
$Size = 8$	522	**95.08**	7240	**31.85**

TABLEAU 4.1 – Taux de perte entre échantillonnage orienté graphe et triplet.

Avec l'échantillonnage orienté graphe, nous observons un taux de perte inférieur à celui obtenu par l'échantillonnage orienté triplet. En effet, Il est possible qu'en appliquant une approche orientée triplet, l'échantillon contient bien l'identifiant d'un capteur mais pas la valeur de pression correspondante. En revanche, l'approche orientée graphe maintient des liens sémantiques assurant que les capteurs sont toujours associés à leurs valeurs de pression correspondantes dans l'échantillon.

4.6 Conclusion

La gestion massive des flux de données est une préoccupation industrielle permanente et un défi scientifique. L'application des technologies du web sémantique pour le traitement des flux de données hétérogènes reste perturbée par les gros volumes de données et la génération rapide des flux. Dans ce chapitre, nous avons tiré parti des techniques d'échantillonnage existantes et proposé une extension du système RSP C-SPARQL afin d'y intégrer des fonctions d'échantillonnage en temps réel des flux de graphes RDF. Nous avons adopté une approche orientée graphe à la place de celle orientée triplet dans le but de préserver les liens sémantiques entre les différents triplets du graphe qui forment un événement. Cette approche nous a permis de maintenir cohérentes les données de l'échantillon et d'améliorer leur représentativité (l'exactitude des résultats de requêtes sur données échantillonnées). Nous avons également étendu la syntaxe de requête de C-SPARQL et rajouté trois opérateurs d'échantillonnage (uniform, reservoir, chain). Ces opérateurs sont implémentés dans le module de fenêtrage du système C-SPARQL (Esper) en prenant en compte toutes les variantes de fenêtres logique et physique. L'évaluation et la comparaison des résultats obtenus dans les deux scénarios (système C-SPARQL avec et sans processus d'échantillonnage) ont montré de meilleures performances en termes de temps de calcul de la requête et de la préservation de la sémantique des donnés. Ces résultats garantissent ainsi une réduction de la charge de données suivant la probabilité de sélection pour chaque flux tout en gardant la cohérence des données à traiter en garantissant l'exactitude des résultats de requêtes. Cependant, l'échantillonnage orienté graphe de données liées (sous format graphe RDF) permet de réduire la charge de données à traiter tout en préservant les liens sémantiques mais a des limites sur l'extraction d'informations pertinentes des graphes pour construire un échantillon ou un résumé du graphe. Ainsi, pour la conception d'un résumé plus représentatif, il est nécessaire d'aller au delà du simple échantillonnage en proposant une nouvelle approche qui privilégie les nœuds

et les prédicats les plus pertinents d'un graphe RDF.

Publications

[42]. Chabchoub, Yousra, Zakia Kazi-Aoul, Amadou Fall Dia, and Rayane El Sibai. "ON THE DEPENDANCIES OF QUERIES EXECUTION TIME AND MEMORY CONSUMPTION IN C-SPARQL."

[48]. Amadou Fall Dia, Zakia Kazi-Aoul, Aliou Boly and Yousra Chabchoub, "Extension de C-SPARQL pour l'échantillonnage de flux de graphes RDF."

[43]. Dia, Amadou Fall, et al."C-SPARQL Extension for Sampling RDF Graphs Streams."

Chapitre 5

2ème contribution : résumé orienté graphe de flux de données RDF

Sommaire

5.1 Introduction

A l'image des systèmes de gestion de flux de données (DSMS), les systèmes de gestion de flux de données (RSP) traitent les flux en entrée dans des fenêtres qui les suppriment immédiatement une fois la session expirée (une requête SPARQL exécutée sur le contenu d'une fenêtre de données). Il est impossible qu'une requête puisse interroger des données antérieures. Cette limite des RSPs fait que l'on remarque très souvent, des pertes énormes de l'information pertinente. De plus, si aucune des requêtes stockées n'interroge un flux de données, ces dernières sont purement et simplement supprimées. Cette perte d'information influence négativement sur l'exactitude des résultats des requêtes continues et sur d'éventuelles analyses (requêtes SPARQL et raisonnements postérieurs) sur des données antérieures d'un flux.

Dans ce chapitre, à l'instar du chapitre précédent, nous proposons une nouvelle approche de résumé de flux de graphes RDF en combinant les besoins exprimés sur les flux de graphes RDF (c'est-à-dire les requêtes SPARQL). Nous étendons un algorithme de mesure de centralité (degré de proximité) issue de l'Analyse des Réseaux Sociaux (ARS). Notre approche permet de résumer un graphe RDF complexe en tenant compte des besoins mais aussi des liens sémantiques dans les flux RDF afin de ne pas perdre les connaissances cachées dans les données. Enfin, nous implémentons un système de résumé et de gestion en continu sur les flux de graphes RDF pour d'une part, la gestion de bout en bout

des flux RDF en présence de requêtes SPARQL et d'autre part, offrir une configuration des paramètres de résumé et une visualisation des résultats.

Ce chapitre est organisé comme suit. La section 5.2 présente quelques notions sur les mesures de centralité (utilisées en Analyse des Réseaux Sociaux) dont nous en étendons une pour le résumé des graphes RDF. Nous détaillons, dans la section 5.3, notre approche de résumé à travers cinq (5) principaux algorithmes et le système développé derrière. La section 5.4 évalue les performances de notre approche en termes de ressource et de qualité du résumé. Enfin, la section 5.5 conclut ce chapitre.

5.2 Définitions et concepts en Analyse des Réseaux Sociaux (ARS)

Avant d'aller plus loin, il est important de définir quelques notions importantes qui sont essentielles à la compréhension de notre approche. En effet, dans ce chapitre, nous proposons une technique dont l'objectif principal est de réduire la charge de données à traiter et de rendre disponibles de grands volumes de données hétérogènes générées en continu à des vitesses extrêmement rapides et variables. Cet objectif peut être réalisé par le résumé de flux de graphes RDF avec l'exploitation des requêtes SPARQL fournies par les utilisateurs et les mesures de centralité issues de l'analyse des réseaux sociaux (ARS).

5.2.1 Généralités et utilités

L'analyse des réseaux sociaux en tant que théorie à part entière est souvent définie comme le fondement même de la sociologie et de la science des structures des relations sociales (axée sur la forme de relations et non sur leur contenu). Ce domaine a puisé des théories mathématiques (notamment la théorie des graphes, l'algèbre linéaire et les matrices), le pouvoir calculatoire et de représentation qui lui faisait défaut jusque là. Aujourd'hui, nous pouvons identifier l'implication des liens sociaux dans plusieurs domaines tels que la sociologie, l'éducation, la psychologie sociale, l'anthropologie, les sciences politiques, la communication, les études organisationnelles, la gestion des ressources naturelles, la biologie, l'épidémiologie, etc. L'ARS est exploitée dans l'étude de plusieurs thèmes ou concepts tels que la propagation des maladies, les réseaux de terroristes, la diffusion des idées, la gestion des connaissances, le niveau de vulnérabilité des réseaux, la découverte des communautés, etc. Les auteurs dans [121] apportent quelques définitions de concepts de l'ARS.

Définition 5.2.1. Un **Réseau Social** est défini comme étant un ensemble d'acteurs et des relations qui les relient.

Le média social Facebook illustre bien la notion de réseau social à travers son réseaux d'amis.

Définition 5.2.2. Un **acteur** est un individu ou un organisme. Un lien fait référence à une relation qui existe entre deux individus.

Les acteurs dans le réseau sont les personnes ou les groupes tandis que les liens montrent les relations ou les flux entre les nœuds.

L'ARS est la cartographie et la mesure des relations et des flux entre personnes, groupes, organismes, ordinateurs, sites web ou toute autre entité de traitement d'information ou de connaissances. Elle permet une analyse visuelle et mathématique des relations entre individus ou entités [122].

Un réseau est souvent représenté sous forme de graphe G dans lequel un nœud représente une entité sociale (personne, animal, document, machine, etc.) et un lien représente une interaction sociale (amitié, intérêt commun, relation professionnelle, etc.). Un graphe peut être orienté auquel cas les liens sont représentés par des flèches. Ces flèches peuvent être unidirectionnelles (relation asymétrique) comme le réseau de followers de Twitter ou bidirectionnelles (relation symétrique) comme le réseau d'amis de Facebook. Un graphe peut aussi être non orienté. Dans ce cas, les liens sont de simples arcs qui relient deux nœuds où l'on parlera de relation sans orientation. Deux nœuds peuvent avoir une liaison directe ou une liaison indirecte.

Définition 5.2.3. Soit un graphe $G = (V, E)$ avec V l'ensemble des nœuds dans G et E l'ensemble des liens dans G tels que $E \subseteq V \times V$.
$\forall i, j, k \in V$,

— si $\exists e = (V_i, V_j) \in E$, on dit que les nœuds V_i et V_j sont liés, connectés ou en relation directe dans G.

— Si $\exists e_{ij} = (V_i, V_j) \in E$ & $\exists e_{jk} = (V_j, V_k) \in E$ & $\nexists e_{ik} = (V_i, V_k) \in E$, on dit que les nœuds V_i et V_k sont liés, connectés ou en relation indirecte de degré 2 dans G.

L'Analyse des Réseaux Sociaux a pour objectif principal d'analyser les relations existantes ou non entre les nœuds d'un graphe afin d'en tirer des connaissances. Pour ce faire, elle s'est dotée de deux grandes familles de méthodes d'analyse de réseaux.

5.2.2 Les méthodes d'analyse dans les ARS

Il existe deux grandes familles de méthodes d'analyse : les méthodes traditionnelles et les méthodes de fouille de réseaux sociaux [123]. Les méthodes de fouille de réseaux sociaux sont diverses mais elles s'intéressent toutes à l'extraction de connaissances. Comme exemple de méthode de fouille de réseaux sociaux, nous pouvons citer :

— la classification qui a pour principe, d'affecter à chaque nœud une classe ;

— la méthode de clustering dont l'objectif est l'identification des communautés dans un réseau ;

— la recherche de motifs fréquents qui à son tour recherche dans le graphe, les sous-graphes qui sont fréquents dans le réseau.

Pour ce qui concerne les méthodes traditionnelles, il en existe deux (02) groupes. On note le groupe des mesures locales [124] qui fournit des informations sur un nœud ou un ensemble de nœuds du réseau mais de manière locale : degré de centralité, coefficient de clustering, distance moyenne, centralité d'intermédiarité, centralité de proximité, centralité de prestige, etc. On note aussi le groupe des mesures globales [125] qui fournit des informations sur l'ensemble de la structure du réseau : composantes connexes, coefficient de clustering moyen, etc.

Soit $G = (V, E)$ un graphe avec V l'ensemble des nœuds de G et E l'ensemble des liens de G tels que $E \subseteq V \times V$. $\|V\|$ représente le nombre de nœuds dans G et $\|E\|$ est le nombre de liens dans G.

5.2.2.1 Degré de centralité

Cette mesure permet de déterminer l'acteur ou le nœud central du réseau. Il s'agit de l'acteur le plus actif en ce qui concerne la communication. Il s'agit d'une simple mesure centrale qui compte le nombre de voisins d'un nœud. Si le réseau est dirigé, nous avons deux versions de la mesure.

— **In-degree** est le nombre de liens entrants, ou le nombre de nœuds prédécesseurs (Figure 5.1) ;

— **Out-degree** est le nombre de liens sortants, ou le nombre de nœuds successeurs (Figure 5.1).

En règle générale, la mesure in-degree est beaucoup plus intéressante car les liens sont donnés par d'autres nœuds du réseau, alors qu'au niveau de la mesure out-degree, les liens sont déterminés par le nœud lui-même. Avec la mesure du degré de centralité, un nœud est important s'il a beaucoup de voisins, ou, dans le cas d'un réseau orienté, s'il y a un grand nombre de nœuds qui pointent vers lui, ou s'il a des liens vers d'autres nœuds.

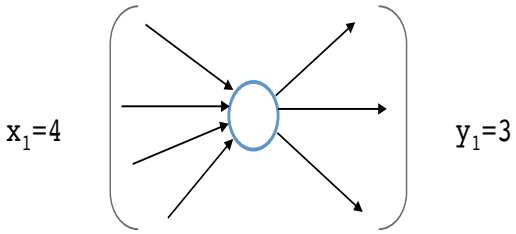


FIGURE 5.1 – un noeud avec in-degree=4, out-degree=3 et degré total 7.

— **Expressions mathématiques**. Soit A = $a_{i,j}$ la matrice de contiguïté d'un graphe orienté. La centralité **in-degree** x_i du nœud i est donnée par $x_i = \sum_k a_{k,i}$ ou bien sous forme de matrice x = IA où I est un vecteur avec tous les composants équivalent à l'unité. La centralité **out-degree** y_i du nœud i est donnée par $y_i = \sum_k a_{i,k}$ ou bien sous forme de matrice y = AI.

5.2.2.2 Degré d'intermédiarité

La centralité d'intermédiarité détermine la mesure dans laquelle un nœud se trouve sur les chemins entre d'autres nœuds. Les nœuds à forte intermédiarité peuvent avoir une influence considérable au sein d'un réseau en vertu de leur contrôle sur l'information qui passe entre les autres nœuds. Leur suppression du réseau peut perturber les communications entre les autres sommets parce qu'ils se trouvent sur le plus grand nombre de chemins empruntés par les messages.

Soient $N^i_{s,t}$ le nombre de chemins géodésiques de s à t qui passent par i et $N_{s,t}$ le nombre total de chemins géodésiques de s à t. Rappelons qu'un chemin géodésique est le plus court chemin à travers un réseau entre deux nœuds et qu'il n'est pas nécessairement unique. Ils existent uniquement entre paires de nœuds (ce qui implique que les nœuds ne peuvent pas être indépendants). Ensuite, le caractère central du degré d'intermédiarité d'un nœud i est donné par $b_i = \sum_{s,t} w^i_{s,t} = \sum_{s,t} \frac{N^i_{s,t}}{N_{s,t}}$ où, par convention, le rapport de $w^i_{s,t} = 0$ si $N_{s,t} = 0$. Notons que chaque paire de nœuds s,t contribue à la somme de i avec un poids $w^i_{s,t} \in [0,1]$ exprimant l'intermédiarité de i par rapport à la paire s,t. Cette somme peut être normalisée en la divisant par le nombre total de couples de nœuds de sorte que le degré d'intermédiarité soit strictement compris entre 0 et 1.

5.2.2.3 Degré de proximité

Le degré de proximité mesure la distance moyenne d'un sommet par rapport aux autres sommets. Supposons que $d_{i,j}$ soit la longueur d'un chemin géodésique de i à j (c'est-à-dire, le nombre d'arêtes le long du chemin), la distance géodésique moyenne pour

le sommet i est donnée par $l_i = \frac{1}{n}\sum_j d_{i,j}$ où n est le nombre total de chemins géodésiques de i à j. Cette quantité a de faibles valeurs pour les nœuds qui sont séparés des autres par une courte distance géodésique en moyenne. De tels nœuds pourraient avoir un meilleur accès à l'information à d'autres nœuds ou plus d'influences directes sur d'autres nœuds. Dans un réseau social, par exemple, une personne avec une faible distance moyenne à d'autres, peut constater que ses opinions atteignent d'autres dans la communauté plus rapidement que l'opinion de quelqu'un avec une distance moyenne plus élevée.

La distance moyenne l_i n'est pas une mesure centrale dans le sens des précédentes, car elle donne de faibles valeurs aux nœuds les plus centraux et des valeurs élevées aux nœuds les moins centraux, ce qui est à l'opposé d'autres mesures de centralité. Par conséquent, on calcule généralement son inverse $C_i = \frac{1}{l_i} = \frac{n}{\sum_j d_{i,j}}$.

Ces différentes mesures de centralité sont utilisées par des algorithmes de graphes pour caractériser les structures de graphes émergeant des usages sociaux (liens d'amis, followers, etc.) et définir les positions stratégiques d'un réseau. Cependant, pour concevoir un résumé (extraire l'information la plus pertinente) du modèle de graphe richement typé RDF, l'application des mesures de centralité aux graphes RDF nécessite d'aller au delà de l'analyse simplement structurelle des graphes RDF en considérant et en tirant partie de la sémantique des liens et de la représentation des nœuds (feuille) dans le graphe.

5.3 Approche de génération de flux de graphes RDF

Les mesures de centralité présentées dans la section précédente (sous-section 5.2.2) permettent d'analyser de grands volumes de graphes mais restent non adaptées aux données RDF encore moins aux flux RDF. En effet, dans un graphe RDF (issu par exemple d'une source de données de capteurs), l'information essentielle réside le plus souvent au niveau des nœuds feuilles.

Ainsi, dans cette section, nous proposons une extension de la mesure de centralité de proximité combinée à un algorithme qui identifie les prédicats les plus importants dans le graphe tout en prenant en compte les besoins exprimés par l'utilisateur (requêtes). Nous détaillons cette proposition à travers trois (3) principales étapes.

— Le fenêtrage pour la récupération en continu des flux de graphes RDF et des requêtes SPARQL en faisant correspondre à chaque requête le/les flux RDF qu'elle interroge.

— Le regroupement des requêtes SPARQL partageant des sous-structures communes afin de réduire le nombre de requêtes à traiter.

— Enfin, l'identification des prédicats les plus importants dans le graphe RDF.

Le système proposé permet une récupération continue, par session (une fenêtre), du flux RDF et des requêtes SPARQL. En sortie, il donne une version résumée (la partie la plus informative et la plus importante du graphe) de chaque graphe RDF d'entrée dans un flux. Le système prend en compte les besoins précédemment exprimés par les utilisateurs ainsi que l'importance définie des prédicats du graphe RDF associé. La figure 5.2 montre l'architecture des processus réalisés dans le système regroupés en différentes étapes.

1. Une fois la session terminée (fenêtre expirée), le système lance automatiquement en parallèle le traitement des flux de graphes RDF et les requêtes SPARQL enregistrés avant le début de la session.

2. Le traitement de graphe effectué à la deuxième étape consiste à identifier les prédicats les plus importants dans le graphe. Pour cela, une mesure de centralité étendue est appliquée au graphe RDF en combinaison avec notre algorithme proposé.
3. Le traitement des requêtes SPARQL de la troisième étape, consiste à regrouper les requêtes de la session partageant des prédicats (en une seule requête SPARQL) pour l'interrogation des flux de graphes RDF de cette session (fenêtre) ayant des prédicats communs, au sein d'une seule requête SPARQL. Pour chaque requête construite, nous récupérons tous les prédicats qui s'y trouvent.

Dans ce qui suit, nous détaillons le processus effectué dans chaque étape.

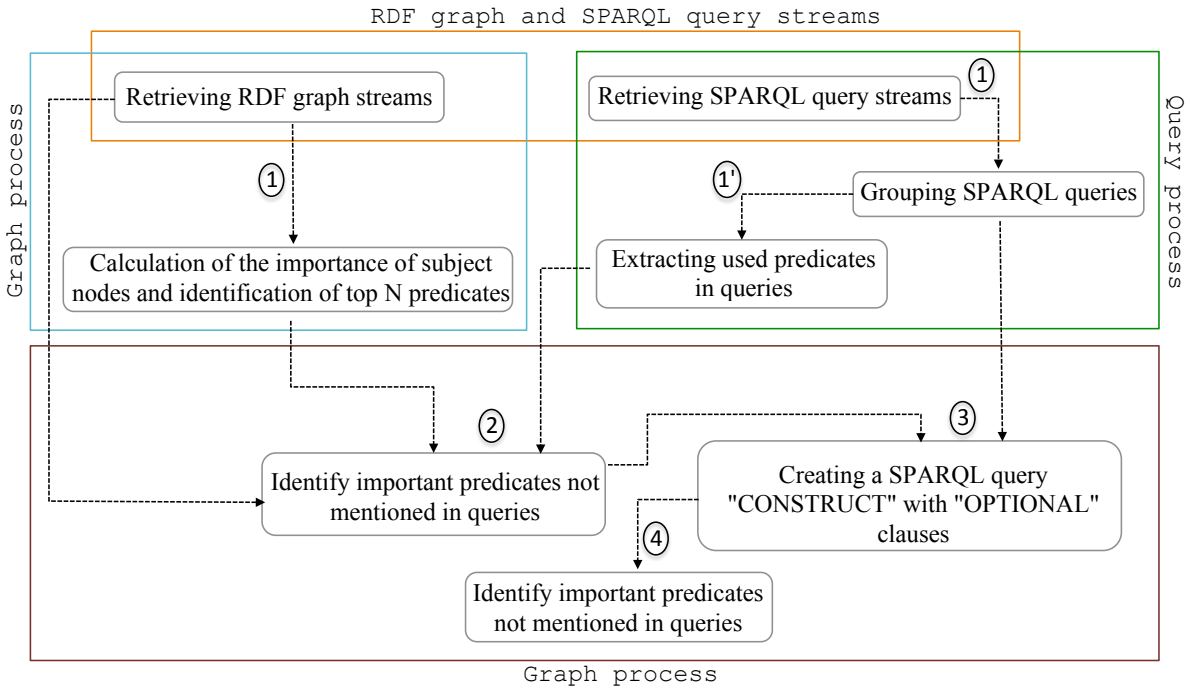


FIGURE 5.2 – Architecture du processus de résumé en utilisant l'ARS.

5.3.1 Fenêtrage sur flux RDF

Durant la phase d'initialisation, le système considère en entrée :

— Le nombre de canaux **NC** à ouvrir pour recevoir des graphes RDF à partir des flux en entrée.

— Le nombre de graphes par session **NG**. Une fois qu'une fenêtre est ouverte, elle ne sera fermée qu'après avoir enregistré les graphes RDF **NG**. Une fois la fenêtre fermée, le traitement des graphes RDF à travers l'exécution des requêtes SPARQL de la session est automatiquement démarré.

— Le nombre d'essais n. Durant le traitement des graphes, le système utilise l'algorithme 2 basé sur la distribution binomiale.

— Le nombre de succès attendu k. Durant le traitement du graphe, le système utilise l'algorithme 3 basé sur la distribution binomiale. k est le nombre attendu de succès pour n événements aléatoires et indépendants.

— Le niveau du résumé **NR**. Lors du traitement des graphes RDF, le système privilégie un certain nombre de nœuds sujets donnés dans le graphe pour les opérations

suivantes. Ce sont les **N** nœuds sujets les plus importants selon notre algorithme. **N** est obtenu en multipliant le nombre de nœuds sujets du graphe par le pourcentage du niveau de résumé **NR** choisi par l'utilisateur. Par exemple, pour un graphe RDF donné avec 1000 nœuds sujets et pour un niveau résumé choisi de 80%, nous avons,

$$\mathbf{N} = \frac{(1000*80)}{100} = 800 \text{ nœuds sujet.}$$

Après la phase d'initialisation, le système reste en attente des flux de graphes RDF et des requêtes SPARQL pour la ou les session(s) ouverte(s) en cours. Une fenêtre est utilisée pour récupérer les derniers éléments d'un flux. Une fois la fenêtre fermée, le processus est effectué uniquement sur les éléments enregistrés. Nous considérons ici les deux (2) catégories de fenêtres : la fenêtre logique basée sur un nombre défini d'éléments de flux et la fenêtre physique basée sur un intervalle de temps. Chaque flux est marqué par un identifiant unique. Nous fournissons une interface graphique pour une écriture facile des requêtes SPARQL en entrée par les utilisateurs. En utilisant cette interface graphique, chaque utilisateur peut enregistrer des requêtes SPARQL dans le système. Une fois inscrite, la requête est mise en correspondance avec la session en cours. Chaque requête SPARQL possède un identifiant unique, appartient à une session (et donc une fenêtre) donnée et interroge un ou plusieurs flux de graphe RDF donnés. Il n'y a pas de relation entre le nombre de graphes RDF dans une fenêtre et le nombre de requêtes SPARQL dans la même fenêtre. Chaque fenêtre ouverte se ferme dès qu'elle atteint une taille seuil de graphes RDF **NG**.

5.3.2 Traitement de requête SPARQL

Une fois les paramètres utilisateur **NG**, **NC**, **n**, **k** et **Nr** renseignés, le système prend respectivement en charge les flux de graphes RDF en entrée et les éventuelles requêtes SPARQL. Ces requêtes représentent les besoins actuels des utilisateurs en matière d'informations. Ils ont un rôle très important dans la construction du résumé des graphes et se positionnent dans un traitement fait en amont. Ainsi, pour optimiser les ressources utilisées durant l'exécution des requêtes, nous identifions préalablement les requêtes ayant des données communes et les regroupons en une seule requête SPARQL. Cette approche présente l'avantage de réduire considérablement le nombre de résumés de graphes à obtenir à la fin du processus. Un bon résumé devrait donner une réponse assez proche de celle donnée par le graphe d'origine pour une même requête.

Pour cela, et autant que possible, le processus de génération de résumé d'un graphe doit prendre en compte les besoins explicites des utilisateurs. Comme le montre la figure 5.2, l'étape de traitement de la requête regroupe trois opérations séquentielles. La première opération est la récupération du graphe de requête SPARQL.
Après cela, nous regroupons dans la deuxième opération les requêtes qui interrogent le(s) même(s) sources de flux et qui ont en commun un ou plusieurs prédicats dans leurs patterns de graphes de requête (BGP). La troisième opération consiste à extraire les prédicats utilisés dans les requêtes résultant de l'opération précédente. Notez que la première opération est effectuée lorsque la fenêtre est ouverte et les deux dernières opérations sont effectuées après la fermeture de la fenêtre.

Le système proposé est capable de prendre en charge une ou plusieurs requêtes SPARQL continues. Des requêtes qui interrogent les mêmes sources de flux de données partagent le plus souvent une partie de leurs patterns de requêtes. Pour éviter la redondance dans l'exécution et les doublons dans le résultat, par conséquent améliorer les performances, nous regroupons en une seule requête, toutes les requêtes interrogeant le même flux et

ayant des prédicats en commun dans leurs patterns de requête (clause WHERE). A l'issue de ce processus, pour chaque flux, nous avons des requêtes SPARQL distinctes en fonction de leurs patterns.
Comme décrit dans l'algorithme 2, le système récupère d'abord la liste l des requêtes SPARQL interrogeant un flux donné dans une session donnée. Chaque requête r_i de la liste l des requêtes en entrée est comparée à chaque requête R_j de la liste L des requêtes déjà groupées. Si la requête r_i a au moins un prédicat en commun avec la requête R_j alors elles sont fusionnées et la requête résultante remplace R_j dans L. Cependant, si r_i n'a pas de prédicat en commun avec une requête dans L, alors elle est simplement ajoutée à la liste L. Comme décrit dans la fonction de comparaison $Comp(\mathrm{R}_j, r_i)$, nous passons par la liste lr_i des prédicats de r_i et, pour chaque prédicat p de lr_i, nous vérifions sa présence dans la liste $l\mathrm{R}_j$ des prédicats déjà groupés de R_j. En fonction de la présence ou non de p dans $l\mathrm{R}_j$, la fonction retourne vrai ou faux. Une fois toutes les requêtes de l parcourues et fusionnées si possible, la liste L déjà groupée est également parcourue. Ce processus est répété tant qu'il y a au moins une paire de requêtes qui ont été fusionnées et répétées pour tous les flux.

$$Comp(\mathrm{R}_j, r_i) = \begin{cases} true & if\ \exists p \in lr_i \quad \& \quad \exists p^{'} \in l\mathrm{R}_j,\ p = p^{'} \\ false & if\ \forall p \in lr_i \quad \& \quad \forall p^{'} \in l\mathrm{R}_j,\ p \neq p^{'} \end{cases}$$

Dans la requête fusionnée, le sujet et l'objet sont créés en tant que variables. Cette approche permet d'atteindre et d'obtenir autant d'informations que possible pendant le traitement de la requête et tous les patterns du graphe de requête qui ont les prédicats spécifiés sont affectés.

Notons que nous exploitons uniquement les prédicats présents dans la clause WHERE. Ce qui implique que les opérateurs de requête tels que la projection (SELECT), le filtrage (FILTER) et le groupement (GROUP BY) ne sont pas vraiment exploités dans le système. En effet, la plupart du temps, nous ne connaissons pas le sujet ou l'objet mais nous avons encore une idée sur les prédicats que le sujet pourrait avoir. Dans l'exploitation des patterns de requêtes présents dans le WHERE, les combinaisons 2, 4 et 7 (chapitre 2, sous-section 2.2.2) sont les plus fréquemment rencontrées. Cependant, une requête avec une variable comme prédicat n'a pas d'impact négatif sur la qualité de notre résumé. Après regroupement par flux et par requête de la session en cours de traitement, le système met à disposition les prédicats qui intéressent les utilisateurs (présents dans la requête). Pour chaque flux, nous récupérons la liste des requêtes groupées de la session à partir de la liste obtenue après l'opération de regroupement dans une seule requête. Cette opération est répétée pour chaque flux identifié de la session.

5.3.3 Traitement de graphes RDF

Une fois que les requêtes ont été traitées, le système détermine les prédicats les plus informatifs du graphe : la récupération des flux de graphes RDF et le calcul de l'importance des nœuds sujet d'un graphe donné avec une identification des N premiers prédicats (étapes 1 à 2 de la figure 5.2). L'importance d'un prédicat peut être définie en fonction de ses différentes mesures de centralité telles que son caractère central, son degré de centralité, ou encore son nombre d'occurrences dans le graphe. Dans la suite, nous détaillons les principales étapes de notre approche de construction de résumé d'un graphe RDF.

Algorithme 2 Regroupement de requêtes SPARQL

```
Require: groupingQueries(sessionId, NC)
 1: groupedQueries ← null //▷ gQ
 2: groupedQueriesForStream ← null //▷ gQFS
 3: i ← 1
 4: while i < NC do
 5:    gQFS ← null
 6:    sessionQueriesForStream ← getSessionQryForStream(sessId, i) //▷ sQFS
 7:    while query ∈ sQFS do
 8:       found ← false
 9:       while gQ ∈ gQFS do
10:          comp ←compareQry(gQ, query)
11:          if comp == true then
12:             mergedResult ← mergeQuery(gQ, query) //▷ mR
13:             gQFS.remove(gQ)
14:             gQFS.add(mergedResult)
15:             found ← true
16:             Return
17:          end if
18:          if found == false then
19:             gQFS.add(mR)
20:          end if
21:       end while
22:    end while
23:    found ← true
24:    while found == true do
25:       found ← false
26:       temporalyGroupedQuery ← null //▷ tGQ
27:       listsize ← gQFS.size()
28:       i ← 0
29:       while i < listsize do
30:          gQ ← gQFS(i)
31:          query ← gQFS(i + 1)
32:          comp ←compareQry(gQ, query)
33:          if comp == true then
34:             mR ←
35:             mergeQueries(gQ, query)
36:             tGQ.add(mR)
37:             found ← true
38:          else
39:             tGQ.add(gQ)
40:             tGQ.add(query)
41:          end if
42:       end while
43:       if (listsize mod 2) ≠ 0 then
44:          gQ ← gQFS(listsize − 1)
45:          tGQ.add(gQ)
46:       end if
47:       gQFS.clear()
48:       gQFS.addAll(tGQ)
49:       i ← i + 2
50:    end while
51:    gQ.add(gQFS)
52: end while
```

5.3.3.1 Récupération des flux de graphes RDF via le fenêtrage

Un flux de données est continu et illimité dans le temps. Une gestion efficace de ces types de données produites avec une vitesse incontrôlable sans limite de taille ou de volume implique et suggère l'utilisation d'un système de fenêtrage. Techniquement, une fenêtre est un intervalle de temps ou une quantité fixe de données définies à la fin desquelles le traitement sera effectué sur les données. Dans notre contexte, ces données sont des graphes RDF. Dans ce système, la partie récupération des flux de graphe RDF est faite avec Esper qui est un système de traitement d'événements complexes open source et gratuit développé en Java. Esper travaille en temps réel et déclenche des actions lorsque les conditions sur les événements sont réalisées. Avec son langage de requête EPL (Event Processing Language), il offre la possibilité d'interroger les éléments du flux avec des fenêtres coulissantes ou sautantes. C'est ici que le paramètre **NG** (nombre de graphes par session) défini lors de l'initialisation du système est utilisé. Notons que notre système utilise Esper uniquement pour la gestion des fenêtres.

```
SELECT
FROM GraphStore.WIN:LENGTH_BATCH(NG)
GROUP BY streamID;
```

L'expression `"WIN:LENGTH_BATCH(NG)` de la requête ci-dessus définit une fenêtre sautante.Cette fenêtre contient le dernier lot de graphes RDF **NG** du flux `"GraphStore"` regroupés par l'identifiant du graphe RDF. En effet, chaque graphe RDF d'entrée a un identifiant unique qui est `streamID`, appartient à une fenêtre donnée et arrive au système via un canal spécifique. Avec la requête ci-dessus, à chaque fois qu'une fenêtre ouverte atteint le nombre **NG** de graphes RDF, elle est fermée et commence automatiquement le traitement des graphes RDF. Le but de ce traitement est de déterminer les prédicats les plus importants de chaque graphe RDF puis chaque graphe RDF dans la fenêtre sera résumé. Dans le but de simplifier, nous exposerons le traitement d'un seul graphe RDF sachant que, tous les graphes d'une fenêtre suivent exactement le même traitement à ce stade.

5.3.3.2 Importance du graphe

Dans un réseau (graphe), l'importance d'un nœud est relative au but que l'on veut atteindre. La définition de l'importance d'un nœud varie en fonction de l'information requise. Dans notre contexte, la plupart des informations qui seront nécessaires pour une extraction efficace des connaissances sont le plus souvent placées dans les nœuds feuilles du graphe. Cela devrait être pris en compte lors de la définition de l'importance d'un nœud. Un nœud a une grande importance par rapport aux autres nœuds du graphe s'il a une plus grande capacité à atteindre tous les nœuds feuilles du graphe. Cette capacité est calculée en combinant deux méthodes différentes. En effet, l'analyse des réseaux sociaux offre la possibilité de classer les nœuds du réseau en fonction du degré auquel ils sont proches de tous les autres nœuds du réseau. Néanmoins, cette mesure seule ne suffit pas pour calculer l'importance d'un nœud tel que défini ci-dessus. C'est ce qui justifie notre idée de renforcer cette mesure avec une autre spécifiquement axée sur les nœuds feuilles du graphe.

5.3.3.3 Centralité de proximité

La méthode de centralité de proximité ne prend pas en compte le fait d'avoir beaucoup d'amis, c'est-à-dire d'avoir beaucoup de relations dans le graphe. Elle ne tient pas non plus compte du fait d'être l'intermédiaire de plusieurs autres nœuds dans le graphe.

Pour cette mesure, le but est de déterminer à quel degré un nœud est au centre du graphe, c'est-à-dire à quel point il est proche de tous les autres nœuds du graphe. La proximité est ainsi basée sur la longueur du chemin moyen le plus court entre un nœud et tous les autres nœuds du graphe. Supposons que $d_{i,j}$ est la longueur d'un chemin géodésique de i à j, ce qui équivaut également au nombre d'arêtes le long du chemin. Pour rappel, la distance géodésique moyenne pour le nœud i est $l_i = \frac{1}{n}\sum_j d_{i,j}$, et la centralité de proximité du nœud i est $C_i = \frac{1}{l_i} = \frac{n}{\sum_j d_{i,j}}$.

L'idée consiste à identifier et classer les nœuds du graphe en fonction de leur capacité à transmettre l'information à tous les autres nœuds du graphe. Rappelons nous que les nœuds feuilles (ceux qui n'ont pas de lien sortant) ne satisfont pas cette condition. L'algorithme 3 permet de classer chaque nœud du graphe selon qu'il est ou non un nœud sujet ou feuille. La fonction $classifySubjectLeafNodes$ détermine les nœuds sujets et les nœuds feuilles et renvoie un tableau des deux listes.

Algorithme 3 Centralité de proximité d'un nœud sujet

Require: closenessCentrality($graph$)
$cC \leftarrow ClosenessCentrality(graph)$
$nodesValues \leftarrow null$ //▷ nV
$subjectsNodes \leftarrow$ cSLN($graph$)[0] //▷ sN
while $node \in sN$ **do**
  $cvalue \leftarrow cC.getVertexScore(node)$
  $nV.add(cvalue)$
end while
return nV
end function

Require: classifySubjectLeafNodes($graph$) //▷ cSLN
$subjectNodes \leftarrow null$
$litteralNodes \leftarrow null$
$classifiedNodes \leftarrow null$
$nodesSet \leftarrow graph.getVertices()$
while $node \in nodesSet$ **do**
  $EdgesSize \leftarrow graph.getOutEdges(node).size()$
  if $EdgesSize > 0$ **then**
    $subjectNodes.add(node)$
  else
    $litteralNodes.add(node)$
  end if
end while
$classifiedNodes.add(subjectNodes)$
$classifiedNodes.add(litteralNodes)$
return $classifiedNodes$
end function

5.3.3.4 Probabilité de l'information (distribution binomiale)

Soit G un graphe orienté (Figure 5.3) où les nœuds étiquetés 3, 10 et 7 sont des nœuds feuilles tandis que les nœuds étiquetés 1, 2, 4, 5, 6, 8 et 9 sont des nœuds sujets. La probabilité que l'information du nœud 1 atteigne spécifiquement le nœud 3 suit une distribution de Bernoulli de paramètre p associée au succès $p \Rightarrow 1 \rightarrow 3$, où 1 est le nombre de chemins quittant le nœud 1 pour le nœud 3 et 3 étant le nombre de chemins qu'on pourrait prendre à partir du nœud 1. En effet, l'information quittant le nœud 1 peut prendre le chemin $1 \rightarrow 9 \Rightarrow failure$ ou le chemin $1 \rightarrow 2 \rightarrow 4 \Rightarrow failure$ ou le chemin $1 \rightarrow 2 \rightarrow 3 \Rightarrow succes$. Il y a donc trois chemins possibles et un seul conduit au nœud 3. L'application de la distribution de Bernoulli nous donne donc la probabilité que l'information du nœud 1 atteigne spécifiquement le nœud 3. Cependant, pour assurer cette probabilité, le test est répété

n fois (n est un paramètre fourni par l'utilisateur lors de la phase d'initialisation du système).

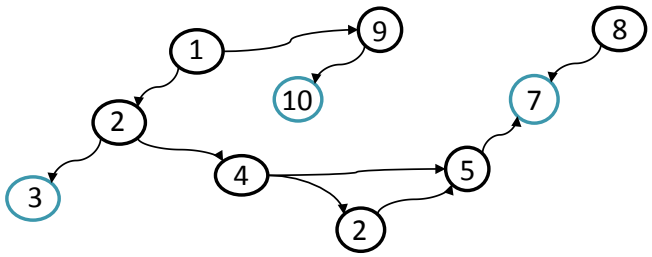


FIGURE 5.3 – Exemple de graphe orienté.

Soit un univers Ω composé de deux éventualités, S pour le succès et F pour l'échec. $\Omega = \mathrm{F}, \mathrm{S}$ sur lequel on construit une variable aléatoire discrète, "$number\ of\ successes$" comme lors d'un test, si S est réalisé alors $\mathrm{X} = 1$ et si F est réalisé alors $\mathrm{X} = 0$. On l'appelle variable de Bernoulli ou variable indicatrice, la variable aléatoire X telle que :

$$\begin{cases} \mathrm{X} & : & \Omega \to \mathbb{R} \\ \mathrm{X}(\Omega) & = & \{0,1\} \end{cases}$$

La probabilité de distribution associée à la variable de Bernoulli X telle que $\mathrm{P}(\mathrm{X}=0) = q$ et $\mathrm{P}(\mathrm{X}=1) = p$ avec $p+q=1$ est appelée distribution de Bernoulli notée $\beta(1,p)$. Supposons que l'application $\mathrm{S}_n : \Omega^n \to \mathbb{R}$ avec $\mathrm{S}_n = \sum_{i=1}^{n} \mathrm{X}_i$ où X_i est une variable de Bernoulli. La variable binomiale, S_n, représente le nombre de succès obtenus lors de la répétition de n tests identiques et indépendants, chaque test ne pouvant donner que deux résultats possibles. Ainsi, la distribution de probabilité suivie de la somme des n variables de Bernoulli, où la probabilité associée au succès est p, est la distribution binomiale des paramètres n et p dénotés $\beta(n,p)$. La probabilité que $\mathrm{S}_n = k$, c'est-à-dire, l'accomplissement de k succès dans n tests indépendants et identiques de Bernoulli est $\mathrm{P}(\mathrm{S}_n = k) = \mathrm{C}_n^k p^k q^{n-k} = \mathrm{C}_n^k p^k (1-p)^{n-k}$, car $p+q=1$.

Notons que la valeur C_n^k est appelée coefficient binomial. Pour connaître la probabilité qu'un nœud sujet donné puisse atteindre un nœud feuille donné, nous appliquons la distribution binomiale. L'algorithme 4, à travers sa fonction $subjectToLeafProbability$, prend comme paramètre le graphe orienté à résumer, le nœud sujet d'où part l'information, le nœud feuille à atteindre, la liste des nœuds sujets qui précèdent le nœud feuille dans le graphe, le coefficient binomial, le nombre attendu de succès et le nombre d'échecs. Le nombre attendu de succès k est fourni lors de la phase d'initialisation du système. Le nombre d'échecs est $\overline{k} = n - k$ avec n le nombre d'essais, c'est-à-dire le nombre de répétitions du test de Bernoulli. La liste des nœuds sujets qui précèdent le nœud feuille dans le graphe (fonction $leafNodePredecessors$) est fournie par la fonction $extractPredecessorNode$ (algorithme 5). Nous calculons la probabilité que l'information d'un nœud sujet atteigne un nœud feuille donné en utilisant la distribution binomiale. Nous calculons, d'une part, le nombre de chemins $nbSuccessPath$ en partant du nœud sujet et menant aux nœuds feuilles et d'autre part, le nombre de chemins $nbTotalPath$ que l'information pourrait prendre à partir du nœud sujet. Le calcul de ces deux nombres ($nbSuccessPath$ et $nbTotalPath$) se fait en parcourant les successeurs des nœuds qui suivent le nœud sujet dans le graphe quel que soit leur degré du moment où ils sont dans

le $leafNodesPredecessor$.

Une fois ces calculs effectués, la fonction calcule la probabilité $P(S_n = k) = C_n^k p^k (1-p)^{n-k}$ avec $p = \frac{nbSuccessPath}{nbTotalPath}$. Si $p = 1$, comme dans le cas de la transition du nœud 8 au nœud 7 du graphe G (Figure 5.3), on réduit la valeur de p à $0,99999999999$ pour éviter que $P(S_n = k)$ soit nulle.
Soient $\mathbb{SN}$ l'ensemble des nœuds sujets du graphe G et $\mathbb{LN}$ l'ensemble des nœuds feuilles du graphe G tel que $\mathbb{SN} \cup \mathbb{LN} = \mathbb{L}$, l'ensemble des nœuds du graphe G. La probabilité $P(s)$ que l'information quitte le nœud sujet s et atteigne tous les nœuds feuilles du graphe G est définie comme suit :

$$P(s) = \frac{1}{\mathbb{LN}} \sum_{f} P(S_n = k)_{sf} \forall f \in \mathbb{LN}, \text{ où :}$$

— s un nœud sujet de $\mathbb{SN}$;

— f un nœud feuille de $\mathbb{LN}$;

— $P(S_n = k)_{sf} = C_n^k p^k (1-p)^{n-k}$ la probabilité que les informations du nœud sujet s atteignent spécifiquement le nœud feuille f ;

— n le nombre d'essais de Bernoulli (événements identiques et indépendants) et k le nombre de succès envisagés.

Algorithme 4 La probabilité suivant la distribution binomiale entre nœud sujet et nœud feuille

Require: subjectToLeafProbability($graph$, $leafNodesPredecessor$, $leafNodes$, $subjectNodes$, $coef$, k, $\overline{k}$)
//▷ lNP, lN, sN
1: $subjectSuccessorNodes \leftarrow graph.getSuccessors(sN)$ //▷ sSN
2: $nbSuccessPath \leftarrow 0$ //▷ $nbSP$
3: $nbTotalPath \leftarrow$ sSN.Size() //▷ $nbTP$
4: **while** $node \in sSN$ **do**
5: **if** $lNP.contains(node)$ **then**
6: $nbTP \leftarrow nbTP - 1$
7: $nodesSet \leftarrow graph.getSuccessors(node)$ //▷ nS
8: $nbCT \leftarrow nbTP + nS.Size()$
9: **while** $nodeS \in nS$ **do**
10: **if** $sSN.notContains(nodeS)$ **then**
11: **if** $nodeS == lN$ **then**
12: $nbSP \leftarrow nbSP + 1$
13: **else**
14: $sSN.add(nodeS)$
15: **end if**
16: **else**
17: $nbSP \leftarrow nbSP + 1$
18: **end if**
19: **end while**
20: **else**
21: $sSN.remove(nodeS)$
22: **if** $nodeS == lF$ **then**
23: $nbSP \leftarrow nbSP + 1$
24: **end if**
25: **end if**
26: **end while**
27: $p \leftarrow nbSP \div nbTP$
28: **if** $p == 1$ **then**
29: $p \leftarrow 0.999999999999$
30: **end if**
31: $q \leftarrow 1 - p$
32: $proba \leftarrow coef * p^k * q^{\overline{k}}$
33: **return** $proba$
end function

5.3.3.5 Calcul de l'importance des nœuds sujets

Un nœud s du graphe a une grande importance par rapport aux autres s'il a une plus grande capacité à atteindre tous les nœuds feuilles du graphe. Cette définition offre deux

Algorithme 5 Importance du nœud sujet

```
Require: subjectNodeImportance(orientedGraph, nonOrientedGraph, n, k)
    //▷ oG, nOG
 1: subjectImportanceValue ← null //▷ sIV
 2: classifiedNodes ← //▷ cN
 3: classifySubjectLeafNode(oG)
 4: subjectNodes ← cN[0] //▷ sN
 5: leafNodes ← cN[1] //▷ lN
 6: subjectProbability ← 0.0 //▷ sP
 7: subjectCloseness ← 0.0 //▷ sC
 8: subjectImportance ← 0.0 //▷ sI
 9: binomialCoef ← //▷ bC
10: binomialCoefficient(n, k)
11: k̄ ← n − k
12: leafNodesCount ← lN.Size() //▷ lNC
13: closenessCentrality ← closenessCentrality(nOG) //▷ cC
14: while sNode ∈ sN do
15:     sP ← 0.0
16:     sC ← 0.0
17:     sI ← 0.0
18:     while lNode ∈ lN do
19:         predecessorsNodes ← ePN(oG, lNode) //▷ pN
20:         subjectLeafNodeProba ← subjectToLeafProbability(oG, pN, lN, sNode, bC, k, k̄) //▷ sLNP
21:         sP ← sP + sLNP
22:     end while
23:     sP ← sP ÷ lNC
24:     cS ← cCentrality.getVertexScore(sNode)
25:     sI ← (cS + (sP ∗ 2)) ÷ 2
26:     sIV.add(sI)
27: end while
28: return sIV
    end function
Require: extractPredecessorNode(graph, leafNode) //▷ ePN
29: leafNodePredecessors ← graph.getPredecessors(lN) //▷ lNP
30: while node ∈ lNP do
31:     nodeSet ← graph.getPredecessors(node) //▷ nS
32:     while nodeP ∈ nS do
33:         if lNP.notContains(nodeP) then
34:             lNP.add(nodeP)
35:         end if
36:     end while
37: end while
38: return lNP end function
```

angles d'analyse : la capacité C(s) du nœud s à être proche de tous les nœuds du graphe (centralité de proximité) et la capacité P(s) du nœud s de véhiculer une information précise à tous les nœuds feuilles du graphe (probabilité selon une distribution binomiale des paramètres n et p). La combinaison de ces deux valeurs reflète l'importance du nœud pour transmettre l'information à travers le graphe. Pour souligner la primauté d'atteindre tous les nœuds feuilles, P(s) sera pondérée lors du calcul de l'importance des nœuds. Ainsi, l'importance I(s) d'un nœud s dans un graphe G est la moyenne de sa centralité de proximité C(s) et sa probabilité d'information P(s) :

$$\mathrm{I}(s) = \tfrac{1}{2}(\mathrm{C}(s) + 2 * \mathrm{P}(s))$$

La fonction *subjectNodeImportance* de l'algorithme 5 illustre comment cette valeur est calculée. Pour chaque nœud sujet du graphe, le système parcourt la liste des nœuds feuilles du graphe et pour chacun d'entre eux, il récupère la liste des nœuds qui le précèdent dans le graphe orienté, c'est-à-dire tous les nœuds du graphe qui ont un chemin menant au nœud feuille en question. Il calcule donc la probabilité que l'information du nœud sujet atteigne le nœud feuille en utilisant la fonction *subjectToLeafProbability* de l'algorithme 4 expliqué dans le paragraphe précédent. Les deux dernières opérations sont répétées pour tous les nœuds feuilles du graphe. La somme de toutes les valeurs de probabilité est divisée par le nombre de nœuds feuilles du graphe, ce qui donne la probabilité d'information de s dans le graphe. Enfin, il calcule la valeur de l'importance du nœud

s dans le graphe suivant la formule indiquée ci-dessus. Pour récupérer tous les nœuds prédécesseurs d'un nœud feuille, la fonction récupère d'abord les prédécesseurs directs du nœud feuille et, pour chaque prédécesseur trouvé, il récupère ses prédécesseurs et ainsi de suite jusqu'à ce qu'il atteigne les nœuds du graphe sans prédécesseur.

Définition 5.3.1. Un nœud *o* est le **prédécesseur** d'un nœud *s* dans un graphe si *o* a un lien direct pointant vers *s* dans le graphe.

L'étape de traitement du graphe doit fournir la liste des prédicats les plus importants du graphe.

Définition 5.3.2. Un **prédicat** est **important** lorsque le nœud sujet qu'il qualifie est important.

Ainsi, après avoir calculé l'importance des nœuds sujets du graphe, ses sujets sont classés en fonction de leur importance du plus important au moins important. Avec le niveau de résumé **NR** souhaité fourni par l'utilisateur pendant la phase d'initialisation, on calcule le nombre de nœud sujets importants pour garder S_i selon la formule suivante :

$$Si = \frac{NR*|SN|}{100} \text{ avec, } |SN|.$$

Algorithme 6 La probabilité suivant la distribution binomiale entre nœud sujet et nœud feuille

Require: nonCitedImportantPredicate($importantPredicates$, $sourceQueryPredicates$) //▷ iP, sQP
1: $nonCitedImportantPredicates \leftarrow null$ //▷ nCIP
2: **while** $iPredicate \in nCIP$ **do**
3: $found \leftarrow false$
4: **while** $rPredicate \in sQP$ **do**
5: **if** $iPredicate == rPredicate$ **then**
6: $found \leftarrow true$
7: **end if**
8: **end while**
9: **if** $found == false$ **then**
10: $nCIP.add(iPredicate)$
11: **end if**
12: **end while**
13: **return** $nCIP$ **end function**

5.3.3.6 Résumé de graphes RDF

Le résumé d'un graphe est construit en tenant compte des besoins des utilisateurs mais aussi des entités les plus informatives du graphe. Le résumé du graphe est créé à partir d'une requête de type CONSTRUCT avec la clause OPTIONAL dans laquelle nous encapsulons les prédicats non mentionnés dans les requêtes. Le résumé est généré pour chaque requête d'origine sur le flux. Pour chaque requête d'origine, la liste des prédicats contenus dans les modèles de la clause WHERE sera comparée à la liste des prédicats les plus importants du graphe. Si un prédicat important est déjà cité dans la requête d'origine, il sera supprimé de la liste des prédicats importants à utiliser lors de la création de la fonction de requête CONSTRUCT $importantPredicates$ de l'algorithme 6.

Pour générer le résumé du graphe, une requête SPARQL de type CONSTRUCT est créée sur la base de la requête d'origine. Pour présenter les prédicats importants du graphe non mentionnés dans les requêtes, l'avantage de la clause OPTIONAL des requêtes SPARQL

sera exploité. En effet, la clause OPTIONAL dans une requête SPARQL permet de projeter les triplets satisfaisant aux conditions de la clause OPTIONAL même si ces triplets n'ont pas de correspondants dans la projection de requête normale.

La procédure de résumé de graphes RDF détaillée dans ce chapitre étend bien l'algorithme de mesure de centralité (degré de proximité) en prenant en compte les besoins des utilisateurs exprimés par les requêtes. Notre approche de résumé accorde également une importance aux nœuds feuilles du graphe car les nœud feuilles d'un graphe RDF renferment le plus souvent l'information la plus pertinente. Cette importance des nœuds est calculée en combinant la valeur de sa centralité de proximité avec sa probabilité d'information selon la distribution binomiale pour atteindre tous les nœuds feuilles du graphe.
Le système implémenté génère en continu des résumés de flux de graphes RDF et propose une gestion continue des flux de graphes RDF pour d'une part, la gestion de bout en bout des flux RDF en présence de requêtes SPARQL et d'autre part, une configuration des paramètres de résumé et une visualisation des résultats obtenus.

Dans la suite, nous illustrons les résultats d'évaluation du système de gestion continue des flux de graphes et de requêtes SPARQL en termes de qualité du résumé et de performance en stockage mémoire.

5.4 Évaluation

La continuité et la rapidité à laquelle sont produits et exploités les flux RDF fait que leur volume peut très vite exploser et le traitement de la totalité d'un flux devient quasi impossible. Pour réduire la charge de données liées (graphes) à traiter, on peut écrire des algorithmes de résumé de graphes qui fonctionnent correctement c'est-à-dire, des algorithmes corrects en matière de résultats de requêtes correctes et complets ou suffisamment approximatifs du résultat attendu (résultat de requêtes sur les données d'origine avant résumé). L'évaluation de ces algorithmes de résumé revient à déterminer la qualité de leur résumé. Il s'agit d'une comparaison des résultats d'une même requête sur les résumés de graphes et sur les graphes sources.
Dans cette section, nous présentons le jeu de données utilisé, et évaluons respectivement la qualité du résumé à travers des requêtes SPARQL et les performances obtenues sur le critère du stockage mémoire.

5.4.1 Description des données

Nous utilisons des données d'observation provenant de 20 000 capteurs aux États-Unis. Cet ensemble de données contient l'agrégation de données climatiques collectées à partir de différentes stations localisées au États Unis depuis 2002. Ces données sont collectées au département de météorologie de l'Université de l'Utah à l'aide du laboratoire Kno.e.sis (Ohio Center of Excellence in Knowledge Enabled Computing). La figure 5.4 donne un aperçu de l'ensemble de données utilisées et des relations entre les nœuds. Cet ensemble de données est divisé en données de capteurs et méta-données et en données provenant des observations d'ouragans et des tempêtes de neige [4].

L'ensemble des données et méta-données de capteurs contient les descriptions expressives de 20 000 stations météorologiques aux États-Unis. En moyenne, il y a environ cinq capteurs par station météorologique, d'où un total de 100 000 capteurs décrits dans l'ensemble de données. Ces capteurs mesurent des phénomènes tels que la température, la visibilité, la précipitation, la pression, la vitesse du vent, l'humidité, etc. En outre, les

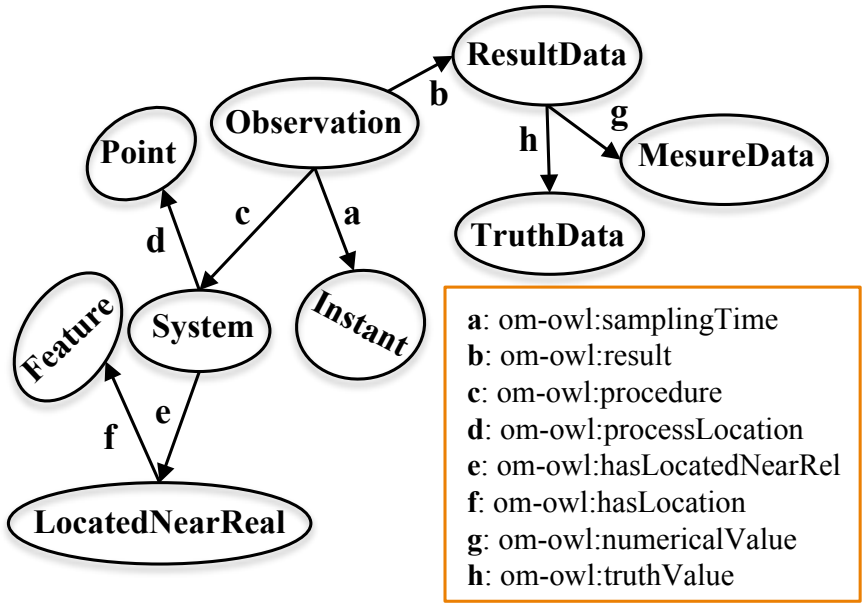


FIGURE 5.4 – Vue d'ensemble du jeu de données utilisé.

informations concernant les positions géographiques (latitude, longitude et altitude) des capteurs sont également fournies par ces capteurs.

Le deuxième sous-ensemble de données contient contient des descriptions expressives d'observations d'ouragans et de tempêtes de neige aux États-Unis. Les observations recueillies comprennent des mesures de phénomènes tels que la température, les précipitations, la vitesse du vent, l'humidité, etc. Cette base de données comprend celles observées pendant la période où plusieurs ouragans ont secoué les États-Unis. Nous remarquons les ouragans Katrina, Ike, Bill, Bertha, Wilma, Charley et Gustav ainsi que la grande tempête de neige qui a secoué le Nevada en 2003. Toutes ces observations sont produites par les stations météorologiques décrites plus haut.

5.4.2 Évaluation de la qualité du résumé

Nous évaluons d'abord les réponses obtenues après l'exécution des requêtes SPARQL sur le récapitulatif du graphe RDF. Nous évaluons ensuite la taille de la mémoire occupée par le résumé du graphe RDF par rapport à celles occupées par le graphe d'origine. La qualité du résumé g d'un graphe d'origine G, est de fournir à une requête R approximativement le même résultat qu'aurait fourni le graphe source G pour la même requête. Nous voulons donc obtenir un résumé de graphe prenant en compte les informations les plus pertinentes du graphe et les besoins des utilisateurs. Ces besoins sont exprimés au travers des requêtes SPARQL. Pour évaluer la qualité de notre approche de résumé, nous exprimons, dans le tableau 5.1, quelques requêtes dont le résultat de l'exécution sur le graphe source G est comparé au résultat de l'exécution sur le graphe résumé g. Nous avons lancé le système avec trois (03) canaux pour recevoir les flux de graphes RDF. Chaque fenêtre est composée de mille (1000) graphes RDF. Le nombre d'essais pour calculer la probabilité d'information de chaque nœud sujet du graphe est initialisé à cent (100) avec quatre-vingt (80) succès attendus. Le niveau de résumé est initialisé à 10% afin de se mettre dans le pire des cas lors de la récupération des nœuds sujets les plus informatifs du graphe. Ces 10 premiers nœuds sujets sont ceux qui seront pris en compte pour le résumé du graphe.

Requête SPARQL	Résultats	
	sur données sources	sur données résumées
Requête[A]		
$sessionId = 149$		
$StreamId = 1$	"11.0" | "2017-04-20T16 :27 :26.591Z"	"11.0" | "2017-04-20T16 :27 :26.591Z"
$graphId = \#3b6f7$		
Requête[B]		
$sessionId = 234$		
$StreamId = 1$	System_NGE_1 | 9	System_NGE_1 | 9
$graphId = \#0e9e3f$		
Requête[C]		
$sessionId = 592$		
$StreamId = 1$	#degree	null
$graphId = \#9af54z$		

[A] **SELECT** (MAX(?001) as ?MaxWind) ?003
WHERE { ?ss0 om-owl :result ?oo0. ?oo0 om-owl :floatValue ?001. ?ss0 om-owl :samplingTime ?oo2.
FILTER regex(str(?ss0), "Observation_WindSpeed", "i")
FILTER regex(str(?oo0), "MesureData_WindSpeed", "i") ?oo2 owl-time :inXSDDateTime ?oo3.}
GROUP BY ?oo3

[B] **SELECT** (?ss0 as ?Sensor) (COUNT(?oo0) as ?NbOfObservations
WHERE { ?ss0 om-owl :generatedObservation ?oo0.}
GROUP BY ?ss0

[C] **SELECT** ?oo0
WHERE { ?ss0 om-owl :uom ?oo0.
FILTER regex(str(?ss0), "MesureData_WindDirection", "i")}

TABLEAU 5.1 – Échantillon du résultat d'évaluation sur des données résumées avec les requêtes s'intéressant à [A]"la vitesse du vent la plus élevée", [B]"le nombre d'observations de capteurs générées" et [C]"l'unité de mesure de la direction du vent".

5.4.3 Évaluation des performances de stockage

L'un des principaux objectifs de ce chapitre est de réduire la charge de données à traiter. En toute logique, le stockage de l'espace mémoire utilisé par le graphe résumé devrait être plus petit que celui occupé par le graphe d'origine. En effet, nous avons fait une comparaison entre l'espace utilisé par un lot de graphes RDF et celui utilisé par tous les graphes résumés du même lot (Figure 5.5).

Nous remarquons bien évidement que les graphes résumés prennent moins de place que les graphes d'origine. Un autre constat fait est que la taille est une fonction du niveau de résumé spécifié pour l'initialisation du système. En effet, chaque fois que le niveau de résumé défini lors de l'initialisation augmente, la taille du graphe récapitulatif augmente également. Ceci s'explique par le fait que plus on a de nœuds sujets retenus pour le résumé, plus on a de prédicats à considérer et plus on se rapproche du graphe entier. Ceci explique pourquoi, à partir d'une certaine valeur du niveau de résumé (le cas spécifié est

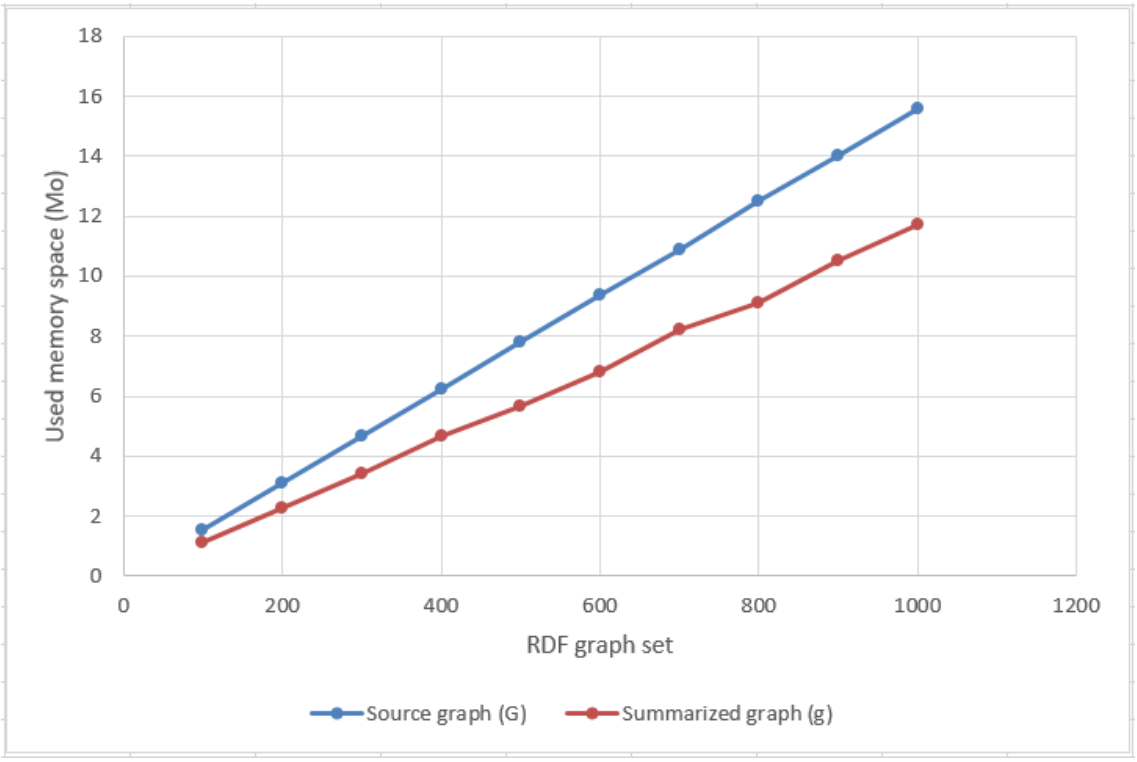


FIGURE 5.5 – Évaluation de l'espace de stockage.

de 50%), le graphe résumé atteint la même taille que le graphe source car il contient exactement les mêmes informations que celui-ci.

5.5 Conclusion

Dans ce chapitre, nous avons exposé une nouvelle approche de résumé de flux de graphes RDF. En effet, le traitement à la volée des données issues des SGFD (Systèmes de Gestion de Flux de Données) ne permet pas toujours une bonne exploitation en temps réel des flux de données. Toutes les données, à l'exception des plus informatives, ne doivent pas être stockées. Puisque ces données peuvent être représentées comme des graphes RDF, la meilleure solution est de résumer les flux de graphes RDF. Nous avons vu dans le chapitre 3 qu'il existe deux catégories d'approches récapitulatives des graphes. Les premières sont basées sur l'agrégation et le groupement et les secondes sont basées sur l'extraction de la structure. Parmi d'autres manquements, nous avons noté que les techniques utilisées dans ces différentes approches n'accordent pas une importance particulière aux nœuds feuilles du graphe. Pour résoudre ce problème, nous avons proposé un nouveau résumé de graphe RDF qui prend en compte les besoins exprimés par l'utilisateur et les nœuds du graphe qui sont les plus importants en matière de proximité avec tous les autres nœuds du graphe. En effet, nous réutilisons et étendons une mesure de centralité d'un nœud ou d'un prédicat (centralité de proximité) issue de l'Analyse des Réseaux Sociaux. Nous avons déterminé avec cette mesure les nœuds et les prédicats les plus pertinents, c'est-à-dire, les plus importants tenant compte de leur centralité mais également du besoin de l'utilisateur (prédicats présents dans la requête SPARQL). Nous avons étendu cette mesure afin de définir l'importance d'un nœud feuille, auquel les mesures de centralité existantes n'accordent pas suffisamment d'importance dans l'analyse du graphe et qui pourtant, renferment le plus souvent l'information la plus utile dans un graphe RDF.

Nous avons évalué les résultats produits par le système suivant la qualité du résumé généré et des performances obtenues relatives au stockage mémoire (en temps réel). Ces deux axes de mesures ont consisté à déterminer la capacité d'un graphe résumé à fournir sensiblement les mêmes réponses que le graphe source pour les requêtes SPARQL et sa capacité à être plus léger en stockage mémoire (durant le traitement temps réel) que le graphe source.

L'extension d'une mesure de centralité pour la génération de résumés de graphes RDF nous a permis d'extraire les nœuds sujet, objet et les prédicats les plus informatifs d'un graphe avec comme critères la pertinence du nœud ou du prédicat dans le graphe ainsi que le besoin utilisateur (requête SPARQL). Cependant, suivant un pourcentage élevé de sélection du niveau de résumé et des cas de jointures entre flux RDF et graphes RDF stockés dans des dépôts locaux ou distants, les informations pertinentes (les résumés générés) peuvent avoir une faible performance (comparée aux données d'origine) en matière d'espace de stockage. L'adoption d'un format compressé de graphes RDF résumés et d'une approche d'interrogation de graphes RDF compressés sans phase de décompression pourrait d'avantage optimiser les performances dans le traitement des requêtes SPARQl continues. Nous présentons cela dans le chapitre suivant.

Publication

[46]. Dia, A. , Togbe, M. , Boly, A. , Aoul, Z. , Metais, E.. "Graph-Oriented Summary for Optimized Resource Description Framework Graphs Streams Processing."

Chapitre 6

3ème contribution : interrogation de flux de données RDF compressées au format RDSZ

Sommaire

6.1 Introduction

Les systèmes RSP ont proposé différentes approches de traitement centralisé des flux RDF. Dans ce contexte, nous avons proposé des solutions pour échantillonner (chapitre 4) ou résumer (chapitre 5) les flux de graphes RDF. Cependant, ces techniques restent insuffisantes dans certains domaines utilisant les flux de données RDF, les applications ayant besoin de garder la totalité du flux. Ainsi, ces domaines d'application requièrent des techniques de compression de données RDF afin de réduire la taille des données échangées et d'optimiser le temps de traitement des requêtes. La compression de données RDF a donné lieu à des travaux récents sur des données RDF statiques (stockées) et dynamiques (flux). Les approches dites "statiques" ([126], [127], [128], [129] et [130]) nécessitent une ou plusieurs lectures globales de larges volumes de données stockées. Ces techniques ne conviennent pas aux flux RDF où deux solutions de compression de flux RDF, [40] RDSZ et ERI [41] sont proposées.

Cependant, les données issues des algorithmes de compression RDSZ ou ERI ne peuvent pas être traitées (exécution de requêtes SPARQL) sans phase de décompression préalable.

De plus, la phase de décompression augmente naturellement le coût de traitement en matière d'échanges de données entre composants du systèmes, de temps de calcul des requêtes et de consommation mémoire.

Dans ce chapitre, nous proposons une extension de l'algorithme RDSZ permettant d'interroger en continu les données compressées. Nous présentons trois algorithmes pour le traitement de requêtes SPARQL simples, avec opérateur(s) de filtrage (FILTER) et avec opérateur(s) d'agrégation (AVG, COUNT, SUM, MIN, MAX) sur des flux RDF compressés au format RDSZ. Nous avons évalué notre approche en utilisant un jeu de données RDF réelles qui contient une description des capteurs expressifs de 20 000 stations météorologiques.

Ce chapitre est organisé comme suit. La section 6.2 présente les deux techniques de compression de flux de données RDF les plus connues ERI et RDSZ. Nous présentons ensuite notre approche d'interrogation de flux RDF compressés au format RDSZ dans la section 6.3. Cette approche est évaluée dans la section 6.4. Enfin, la section 6.5 conclut ce chapitre.

6.2 Compression de flux de données RDF

La pertinence des flux RDF dans de nombreux domaines d'application a motivé l'émergence de deux nouveaux travaux sur la compression de flux RDF que sont RDSZ et ERI.

6.2.1 RDSZ

L'algorithme RDSZ (RDF Differential Stream compressor based on Zlib) [131] est un algorithme de compression de flux RDF sans perte de données. RDSZ tire profit sur le fait que les items d'un flux RDF présentent des similitudes structurelles qui peuvent être exploitées par un mécanisme d'encodage différentiel, de sorte que les nouveaux éléments dans le flux peuvent être représentés sur la base des éléments précédemment traités.

6.2.1.1 Étapes de compression dans RDSZ

L'algorithme RDSZ réalise la compression en utilisant trois principaux composants :

1. **Un codeur différentiel.**
   Le codeur différentiel prend en entrée une séquence d'éléments d'un flux RDF. Les éléments RDF à l'entrée sont traités séquentiellement et séparément par l'encodeur. Le premier traitement effectué sur un élément est sa décomposition en un motif de triplets (appelé **pattern**) et un tableau de variables et valeurs correspondantes (appelé **bindings**).

   Ce composant retourne comme résultat une chaîne de caractères qui représente le motif obtenu en sortie du processus de remplacement et un tableau de variables de liaison qui lie chaque variable à sa valeur particulière de l'entrée. Après avoir obtenu le pattern et le binding, l'encodeur a besoin de déterminer si cet élément peut être représenté en se basant sur un élément déjà traité dans le flux ou non.
   Pour cela, le codeur utilise les informations sur les éléments précédemment traités qui sont stockés dans un cache appelé LRU (Least Recently Used). Pour chaque motif, les variables associées et un identifiant unique sont stockés. Après la lecture de la mémoire cache, le codeur a deux options : si le motif de l'élément RDF en cours de traitement est déjà dans le cache, cela signifie qu'un autre élément avec le même

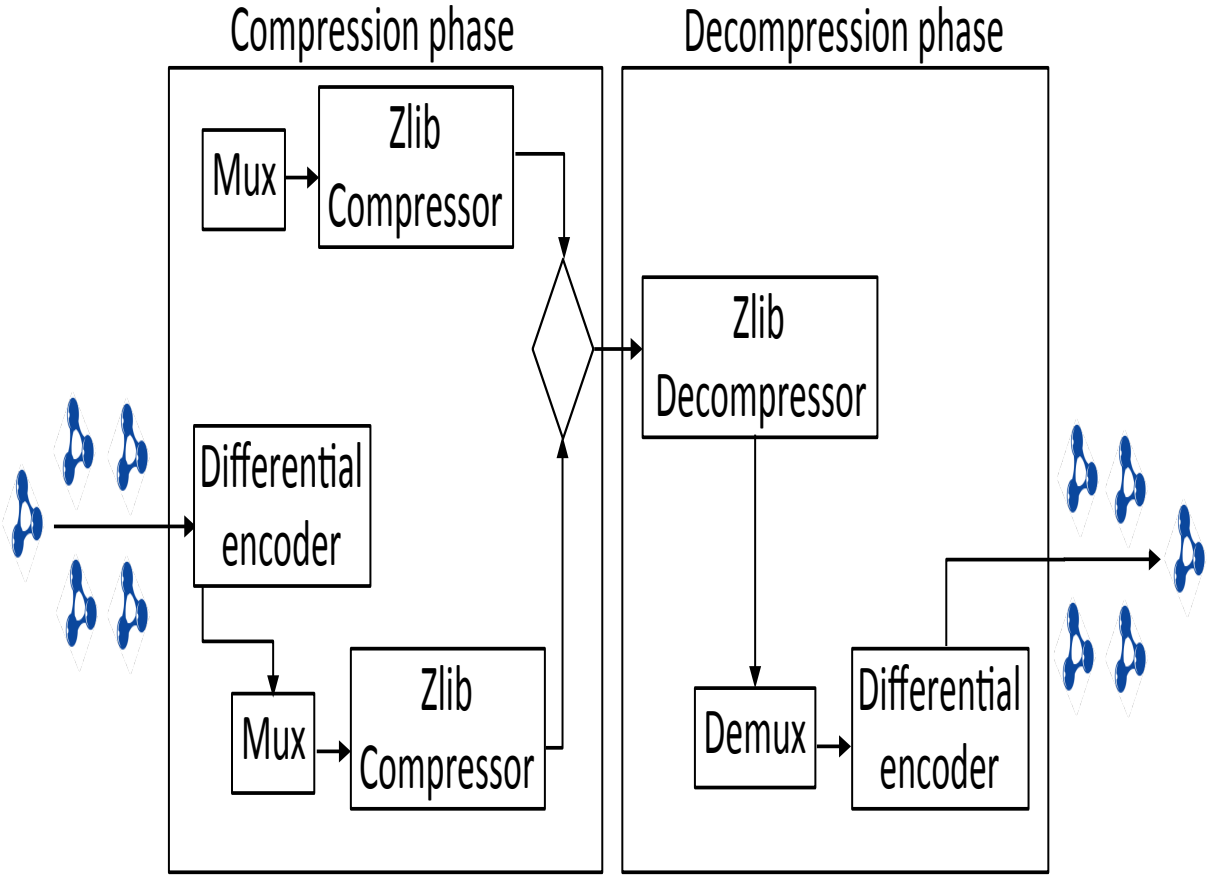


FIGURE 6.1 – Phase de compression et de décompression dans RDSZ.

modèle a été récemment traité. Ainsi, l'élément actif est codé sur la base du précédent. Comme les deux éléments ont le même motif, il n'y a pas besoin d'envoyer à nouveau toutes les données du motif au décompresseur. Seulement l'identificateur de motif sera inclus dans l'élément codé. Pour ce qui concerne les variables, le nouvel élément et le précédent peuvent avoir les mêmes valeurs ou non. Si pour une variable, la valeur est la même, il n'y a pas besoin de l'envoyer à nouveau. Dans le cas contraire, la valeur est incluse dans l'élément codé. Le résultat du processus de codage dans ce cas est une chaîne qui contient une ligne pour l'identificateur du modèle et une ligne pour chaque variable.
Enfin, le codeur différentiel met à jour la mémoire cache en stockant les informations à propos de l'élément RDF qui vient d'être traité. En particulier, le cache mappe le motif de l'élément aux variables associées et à l'identificateur de l'élément. La valeur de cet identificateur dépend du fait que le motif était déjà dans la mémoire cache ou non. Dans le cas où le motif était déjà dans le cache, son identifiant précédent est réutilisé. Dans le cas où le modèle n'a pas été précédemment dans le cache, un nouvel identificateur est généré. Une fois que tous les éléments d'entrée RDF sont traités, le résultat du processus d'encodage est retourné en tant que sortie à traiter par l'élément suivant dans le pipeline du compresseur qui est le multiplexeur.

2. **Un multiplexeur.** Le multiplexeur prend en entrée une séquence d'éléments (codés ou non) et la convertit en une seule chaine par concaténation du texte de sérialisation des éléments. Un délimiteur spécial est utilisé pour marquer les limites de chaque élément, de sorte que le décompresseur puisse les séparer à nouveau.
3. **Un Compresseur ZLIB.** Le compresseur prend en entrée la chaîne générée par le multiplexeur et compresse cette chaîne. Le compresseur Zlib implémente l'algorithme Deflate qui est un algorithme de compression de données sans perte qui couple l'algorithme LZ77 et le codage de Huffman. LZ77 utilise le principe d'un dictionnaire qui trouve les séquences de mots répétés dans le texte à compresser et les remplace par un index dans un dictionnaire. Le codage de Huffman fonctionne avec un arbre pour associer des codes courts à des séquences de bits très fréquentes.
Ainsi après l'obtention des deux résultats, celui avec la plus petite taille est choisi. La décompression est l'inverse de la compression. Les données passent d'abord par un décompresseur Zlib puis par un démultiplexeur et à la fin par un décodeur différentiel.

6.2.2 ERI

Le format ERI (Efficient RDF Interchange) [132] est une représentation compacte du format RDF pour tirer parti de la redondance des structures et des données inhérentes des flux RDF. ERI s'inspire de l'approche RDSZ et exploite le fait que dans la plupart des flux RDF la structure de l'information est bien connue et le nombre de variations dans la structure est limité. ERI considère un flux RDF comme un flux continu de blocs (avec une taille maximale prédéfinie) de triplets. Chaque bloc est modularisé en deux grands ensembles de canaux pour atteindre de grands gains spatiaux. les **canaux structurels** codent les sujets dans chaque bloc et, pour chacun, les propriétés structurales des triplets associés, en utilisant un dictionnaire de structures. Le dictionnaire structurel détient un catalogue dynamique de tous les différents modèles structurels trouvés sur un *dataset* de triplets appelés molécules (ensemble de triplets). Les molécules sont des éléments unitaires de codage. Chaque molécule sera codifiée par un identifiant correspondant (ID) dans le dictionnaire de structures et des valeurs de données concrètes détenues par chaque prédicat. La méthode de regroupement par défaut utilisée est celle des molécules sujettes qui

groupe les triplets ayant le même sujet. Les c**anaux de valeur** codent les valeurs de données concrètes détenues par chaque prédicat dans le bloc compact. L'approche ERI réalise également l'étape de compression en trois principales étapes :

1. **Division en bloc et identification des molécules.**
   ERI divise d'abord les données RDF entrantes en blocs contigus de taille maximale `blockSize` prédéfinie. Chaque bloc possède un dictionnaire de structures qui contient l'ID, les prédicats, le nombre d'objets dans chaque prédicat et les valeurs pour les prédicats discrets. Chaque bloc a également un `Preset` qui comprend toute la configuration et les métadonnées qui sont fournies par le fournisseur de données ou déduites au moment de l'exécution. Ensuite, les molécules (groupes) dans chaque bloc sont identifiées en fonction de la politique de regroupement mis en place.
2. **Multiplexage des molécules.**
   Les canaux structuraux contiennent les informations sur la structure des molécules dans le bloc avec lesquels il va mettre à jour le dictionnaire. Leur fonctionnement est le suivant : le canal `ID-Structures` répertorie les entiers (ID) représentant l'entrée dans le dictionnaire. Les nouvelles entrées sont identifiées au moyen d'un canal supplémentaire appelé `New Structure Marker`. Ce canal a un bit pour chaque ID dans le canal `ID-Structures` : un `0-bit` indique que l'ID correspondant est déjà dans le dictionnaire, alors qu'un `1-bit` montre que l'ID pointe sur une nouvelle entrée qui est récupérée dans le nouveau canal de structures. Chaque entrée du dictionnaire est codifiée comme des prédicats dans la structure, le nombre d'objets pour chaque prédicat et les valeurs concrètes des propriétés pour les prédicats discrets. Pour améliorer la compacité dans la représentation, un dictionnaire de prédicats est utilisé, un ID pointant sur une entrée de prédicat dans le dictionnaire. S'il y a un nouveau prédicat jamais vu auparavant dans un bloc, il est répertorié dans un canal `New` comme prédicat supplémentaire. Les canaux de valeur organisent les valeurs concrètes dans les molécules du bloc pour chaque prédicat non discret. ERI considère essentiellement un canal par prédicat, énumérant tous les objets. Avoir des valeurs de propriété d'un prédicat regroupées peut aider les parseurs à récupérer directement des informations correspondant à des prédicats spécifiques.
3. **Compression des canaux.**
   ERI utilise Zlib à chaque fois que des informations textuelles sont présentes dans les canaux, à savoir dans les principaux termes de molécules, `new structures`, `new predicates` et `new terms channels`. Le flux ERI complet se compose d'un en-tête ERI suivi d'un corps ERI. L'en-tête d'ERI comprend l'identification du flux et les Presets initiales. Le corps d'ERI porte le contenu du flux représentant chaque bloc comme un ensemble de méta-données identifiant le bloc et mettant à jour les changements potentiels dans les Presets, et les canaux compressés. Le processus de décodage est exactement le contraire, c'est-à-dire que le corps du flux est décompressé par des canaux et dé-multiplexé dans des blocs contenant des molécules.

6.2.3 Comparaison des approches RDSZ et ERI

Afin de faire un choix entre les deux algorithmes, plusieurs paramètres de comparaison comme le temps de compression, la taille des données après l'étape de compression et l'exactitude des résultats des requêtes sur les données après compression/décompression ont été fixés. La désignation de l'approche la plus performante d'un de ces deux algorithmes est un compromis entre temps, espace et exactitude des résultats. Le tableau 6.1 résume les résultats d'évaluation comparative des deux algorithmes. Cette évaluation est

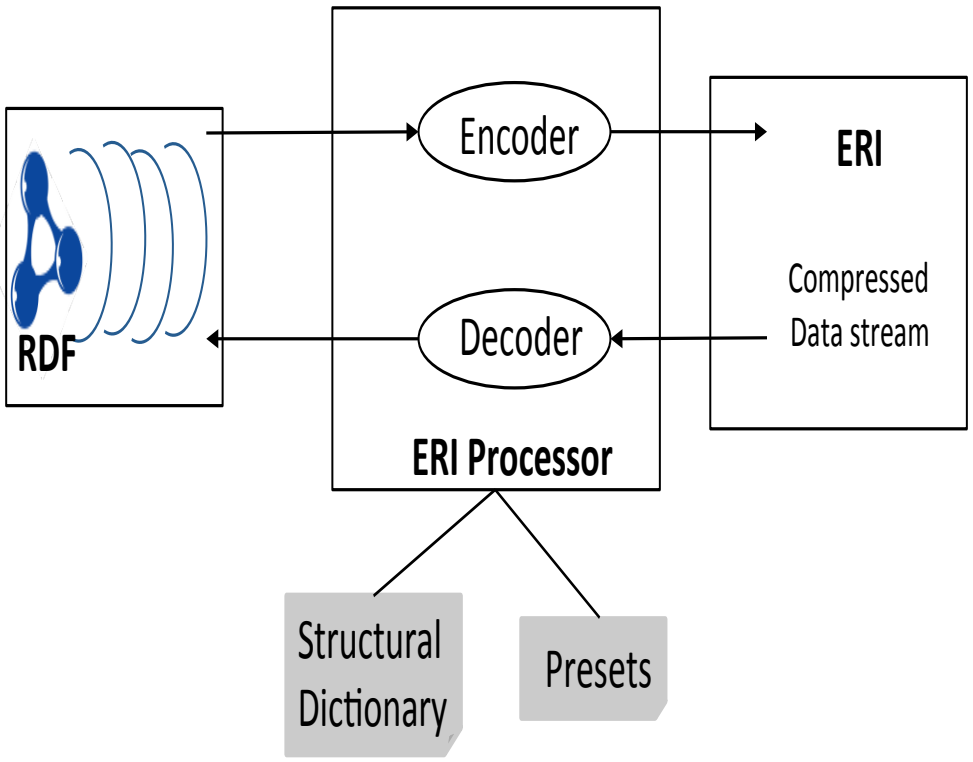


FIGURE 6.2 – Phase de compression et de décompression dans ERI.

Algorithme	Temps de compression	Temps de décompression	Taille des données après compression	Précision des résultats
ERI	-	+	+/-	-
RDSZ	+	-	+/-	+

TABLEAU 6.1 – Tableau comparatif des performances d'ERI et de RDSZ.

effectuée sur la base des critères suivants.

— La **taille des données obtenues après compression**. Dans certains cas, ERI surpasse RDSZ car, lors de la division des ensembles de données sous format graphe (RDSZ prend en entrée un flux de graphes), les éléments ayant des similitudes sont séparés. En revanche, la compression RDSZ surpasse légèrement ERI dans des cas particuliers où le nombre de prédicats est très limité. Dans de tels cas, ERI paie le coût d'avoir plusieurs canaux de compression.

— **Temps de compression et de décompression**. Pour le temps de compression, ERI est nettement plus rapide que RDSZ. En effet, RDSZ traite et fournit des séquences de graphes alors qu'ERI traite des successions de triplets. Ainsi, la compression de RDSZ peut être affectée par le fait qu'il doit traiter potentiellement de larges graphes avec des milliers de triplets car le processus de codage différentiel prend un temps plus long. En revanche pour le temps de décompression, RDSZ est plus rapide qu'ERI car ERI le fait en plusieurs canaux et sort tous les triplets en un seul bloc.

L'approche ERI s'inspire de l'algorithme RDSZ. Cependant, RDSZ présente plus d'avantages qu'ERI dans la mesure où les schémas des données sont bien connus des sources de

données et que le nombre de variations est limité. De ce fait, contrairement à ERI, RDSZ élimine les redondances dans le flux et reste sans perte d'information après la phase de compression. Enfin, l'implémentation des phases de compression et de décompression dans RDSZ est assez modulaire et peut être facilement étendue.
Ainsi, dans la suite, nous avons porté le choix sur RDSZ pour l'extension de SPARQL pour l'interrogation continue de flux de données RDF compressées (c'est-à-dire, sans phase de décompression).

6.3 Interrogation de flux de données RDF compressées au format RDSZ

Une solution pour réduire le coût de l'opération de requête sur les flux RDF consiste à exécuter les requêtes SPARQL immédiatement après l'étape de compression. Dans cette section, nous présentons notre solution que nous avons implémentée pour trois types de requêtes : requête SPARQL simple (jointure simple de patterns), requête SPARQL avec opérateur(s) de filtrage et requête SPARQL avec fonction(s) d'agrégation(s).

6.3.1 Architecture du système

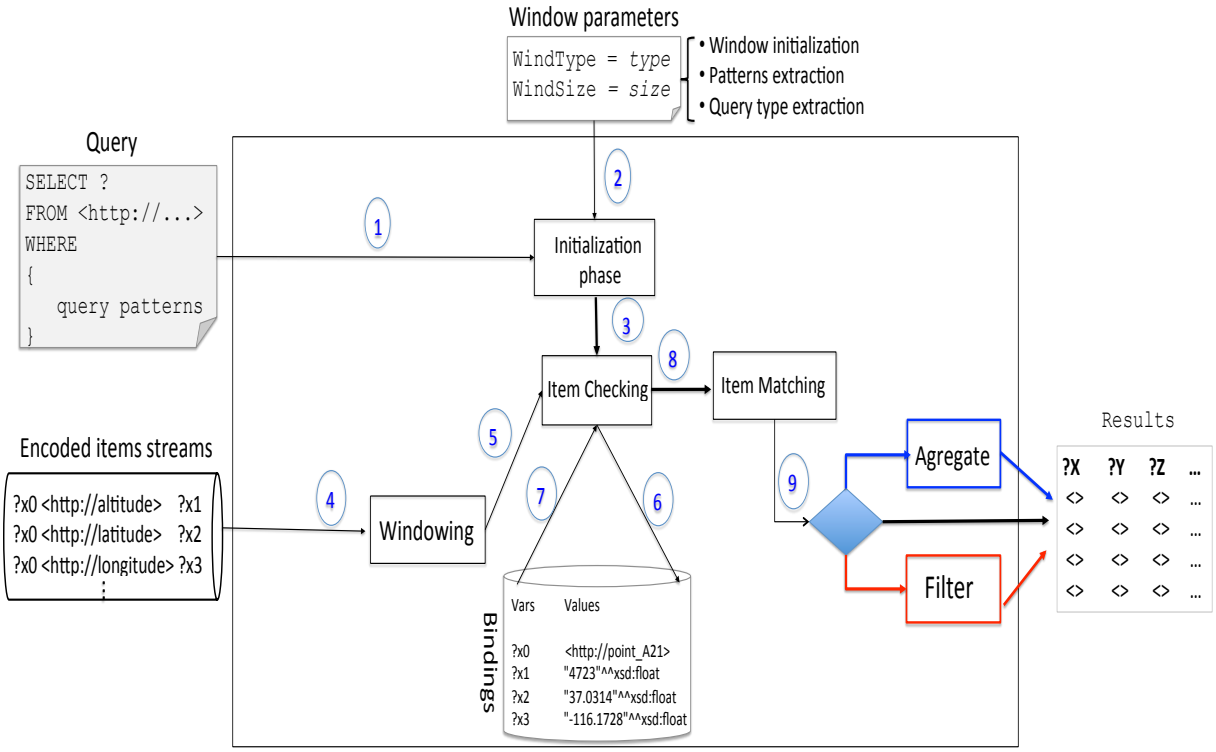


FIGURE 6.3 – Architecture de traitement optimisé de flux de données RDF compressés au format RDSZ.

Comme décrit à la Figure 6.3, l'architecture de notre système est composée de plusieurs modules.

6.3.2 Module 1 : initialisation

Durant la phase d'initialisation, le système prend en entrée trois paramètres : la requête SPARQL, le type de fenêtre et la taille de la fenêtre. Pendant la phase d'initialisation, trois tâches sont effectuées en fonction de ces paramètres :

1. La première tâche consiste à récupérer des modèles à partir de la requête SPARQL. En effet, une requête SPARQL peut se composer de plusieurs patterns que nous analysons au cours de chaque nouvelle phase (c'est-à-dire, à l'arrivée d'une nouvelle requête). Les patterns sont récupérés et mis dans une liste qui est utilisée à l'étape suivante.
2. La deuxième phase est l'initialisation de la fenêtre avec la taille donnée en entrée. La fenêtre nous permet de définir le nombre d'éléments à considérer avant de commencer le traitement.
3. La troisième et dernière tâche de cette phase est la récupération du type de la requête. En effet, comme mentionné plus haut, le système peut gérer trois types de requêtes. Le premier type concerne les requêtes simples. Ces requêtes sont de la forme SELECT ... WHERE {...} sans filtre ni fonction d'agrégation. Le second type de requête concerne les requêtes avec opérateur(s) de filtre. Nous gérons deux types de filtres : le filtrage arithmétique SELECT ... WHERE { ... FILTER (...) } et les expressions régulières SELECT ... WHERE { ... FILTER regex(...) }.

Le filtrage des expressions arithmétiques est appliqué aux entiers, aux nombres réels, etc. Plusieurs types d'opérateurs peuvent être appliqués à ces types tels que la supériorité, l'infériorité ou l'égalité. Pour les requêtes avec filtrage d'expression régulière, nous gérons également deux cas : l'égalité entre deux chaînes et la capacité d'une chaîne à être sous-chaîne d'une chaîne donnée. Le dernier type de requête concerne les opérateurs d'agrégation tels que COUNT, SUM et AVG. Dans ce qui suit, nous détaillons le traitement de ces trois types de requêtes.

Après la phase d'initialisation, le système attend que les triplets du flux en entrée soient compressés par l'algorithme RDSZ. En effet, l'utilisateur définit une fenêtre contenant les éléments à coder par RDSZ. Le traitement des items contenus dans la fenêtre débute une fois celle-ci remplie. Dans la suite, nous détaillons ce traitement.

6.3.3 Module 2 : vérification du pattern

La vérification des patterns renvoyés par l'algorithme RDSZ est une étape essentielle du traitement de la requête. En effet, lors de l'étape de compression RDSZ, les éléments peuvent être codés en fonction de ceux déjà traités. Comme montré à la section 6.2, l'algorithme RDSZ utilise la redondance sur les patterns des jeux de données pour compresser les données RDF. Ainsi, lorsqu'un élément est reçu, il peut être encodé de deux manières : soit le même pattern a été précédemment traité, soit il ne l'a pas été. Dans le premier cas, l'identifiant du pattern précédent est utilisé, tandis que dans le second, un nouvel identifiant est généré. Ensuite, l'objectif du **Module 2** est de vérifier le type d'encodage utilisé pour chaque item. Le **Module 2** passe par les patterns reçus et, pour chacun d'entre eux, vérifie s'il a été codé en fonction des patterns précédemment traités par le codeur ou non. Si l'élément a été codé avec un pattern précédent alors le pattern est extrait du cache. En effet, comme chaque élément est codé en fonction de l'élément précédent, le pattern est récupéré avec l'identifiant de cet élément. La sortie du **Module 2** est envoyée au module suivant.

6.3.4 Module 3 : comparaison entre les patterns de la requête et ceux des items entrants

La comparaison effectuée par ce module permet de récupérer les items dont les patterns correspondent à ceux définis dans la requête. En effet, les patterns de la requête nous permettent de faire un filtre sur les items à retourner après l'exécution de la requête. Comme décrit dans l'algorithme 7, le travail passe par les patterns et le binding de chaque item envoyé par le **Module 2** en récupérant le sujet, le prédicat et l'objet du pattern. Ensuite, il parcourt l'ensemble des patterns de la requête qui ont été récupérés dans la phase d'initialisation avec le sujet, le prédicat et l'objet du pattern. Les patterns de la requête peuvent être présentés selon les huit (8) différentes combinaisons montrées dans le chapitre 2 (sous-section 2.2.2). Dans l'algorithme 7, seule la combinaison 8 n'est pas prise en charge. Une comparaison est faite entre les patterns de la requête et ceux de l'item

Algorithme 7 Appariement des items

```
Require: matchedItem (encodedItem, cache)
result ← null
while (item in encodedItem) do
    diffmodel ← checkItem(item)
    itemPatterns ← diffmodel.getPattern()
    itemBindings ← diffmodel.getBindings()
    queryPatterns ← query.getPattern
    while (pattern in queryPatterns) do
        queryS ← pattern.getS()
        queryP ← pattern.getP()
        queryO ← pattern.getO()

        if queryS.IsVar & queryP.IsVar & queryO.IsVar then
            while (itemPattern in itemPatterns) do
                itemS ← Bindings.getS(patternQuery)

                itemP ← patternQuery.getP
                itemO ← Bindings.getO(patternQuery)

                // add item in the result
            end while
        else if (queryS.IsVar & queryP.IsVar & queryO.IsValue) then
            while (itemPattern in itemPatterns) do
                itemS ← Bindings.getS(patternQuery)

                itemP ← patternQuery.getP

                itemO ← Bindings.getO(patternQuery)

                if (itemObject == queryObject) then
                    // add item in the result
                end if
            end while
        end if
    end while
end while
```

courant si au moins un des trois éléments (sujet, prédicat ou objet) du pattern de la requête est une constante. Dans ce cas, nous comparons le(s) élément(s) des deux patterns : s'ils sont égaux, nous ajoutons le pattern de l'item courant dans le résultat de la requête en remplaçant le sujet et l'objet par leurs valeurs récupérées des bindings, sinon, nous n'ajoutons pas le pattern de l'objet. Ce traitement est effectué sur tous les patterns de la requête avant de passer au pattern suivant de l'item en cours. Après avoir traité tous les patterns de l'élément actuel, nous passons aux patterns de l'élément suivant.

6.3.5 Module 4 : requête avec opérateur de filtre

Les opérateurs de filtrage permettent d'appliquer des restrictions sur les nombres, les chaînes, les dates, etc. Dans notre approche, nous avons implémenté les deux premiers types de filtres, à savoir les filtres sur les nombres et ceux sur les chaînes de caractères. Lorsqu'une requête contient un opérateur de filtre, nous vérifions s'il s'agit d'un filtre d'expression arithmétique ou d'un filtre d'expression régulière. S'il s'agit d'une expression arithmétique, nous récupérons les deux opérandes et les opérateurs (algorithme 8). Par contre, pour une expression régulière, on extrait uniquement les deux opérandes (algorithme 9). Ainsi, l'exécution de la requête se fait en deux étapes : dans la première, nous éliminons les filtres et l'exécutons comme une simple requête et appliquons ensuite le filtrage décrit dans les algorithmes 8 et 9. Selon le type de filtre (expression arithmétique ou régulière), différents traitements de données sont effectués.

6.3.5.1 Filtre d'expression arithmétique

Les opérations arithmétiques implémentées sont `supérieur`, `inférieur` et `égal`. Après l'exécution de la requête sur les éléments n, le résultat est filtré à l'aide des deux opérandes et opérateurs qui ont été récupérés dans la phase d'initialisation. Nous passons par le résultat retourné par le **Module 3** (comparaison entre les patterns de la requête et ceux des items entrants) et pour chaque triplet, nous vérifions s'il répond aux conditions observées dans la clause FILTER, et l'ajoutons dans le résultat final. Sinon, nous l'éliminons.

Algorithme 8 Filtre arithmétique

```
Require: numericFilter (Items)
 1: filterValue ← query.getFilterValue()
 2: operator ← query.getFilterOperator()
 3: if (operator is " = ") then
 4:    while (item in Items) do
 5:       if (item.object is filterValue) then
 6:          // add item in result filter
 7:       end if
 8:    end while
 9: else if (operator is " > ") then
10:    while (item in Items) do
11:       if (item.object > filterValue) then
12:          // add item in result filter
13:       end if
14:    end while
15: else if operator is " < " then
16:    while (item in Items) do
17:       if (item.object < valueFilter) then
18:          // add item in result filter
19:       end if
20:    end while
21: end if
```

6.3.5.2 Filtre d'expression régulière

Nous avons implémenté deux types de filtre d'expressions régulières que sont l'équivalence entre deux chaînes de caractères données et l'extraction de sous-chaîne d'une chaîne donnée. C'est le même principe qu'avec les filtres arithmétiques où nous passons par le résultat retourné par le **Module 3** (comparaison entre les patterns de la requête et ceux des items entrants) et pour chaque triplet, nous vérifions s'il respecte les conditions

qui figurent dans le filtre. Dans l'affirmative, nous l'ajoutons au résultat final, sinon nous l'éliminons.

Algorithme 9 Filtre avec expression régulière

```
Require: filter (Items, query)
filterType ← getFilter(query)
if (filterType is regexContains) then
    regexContainsFilter(Items)
else if (filterType is numeric) then
    numericFilter(ResultItem)
end if
function regexContainsFilter (Items)
filterVar ← query.getFiterVar()
filterValue ← query.getFilterValue()
if (varFilter is subject) then
    while (item in Items) do
        if (item.subject.
        contains(filterValue)) then
            // add item in the result filter
        end if
    end while
else if (filterVar is predicate) then
    while (item in Items) do
        if (item.predicate.
        contains(filterValue)) then
            // add item in the result filter
        end if
    end while
else if (filterVar is object) then
    while (item in Items) do
        if (item.object.
        contains(filterValue)) then
            // add item in the result filter
        end if
    end while
end if
end function
```

6.3.5.3 Requête avec opérateur d'agrégation

Les opérateurs d'agrégation nous permettent d'agréger le résultat d'une requête. Lorsque la requête à exécuter contient des opérateurs d'agrégation, nous récupérons les variables qui sont dans la clause `GROUP BY` et celles sur lesquelles s'appliquent les fonctions d'agrégation. L'exécution d'une requête avec des opérateurs d'agrégation se fait en deux étapes. Dans la première, nous éliminons d'abord l'agrégation et l'exécutons comme une requête simple. La deuxième étape concerne l'agrégation des items et est décrite par l'algorithme 10. Selon le type (COUNT, SUM, AVG, MAX, MIN), différents modules (vérification du pattern et comparaison entre les patterns de la requête et ceux des items entrants) sont exécutés avec le même principe. Nous passons par le résultat retourné par le **Module 3** (comparaison entre les patterns de la requête et ceux des items entrants) de chaque fenêtre et nous regroupons les variables de la clause `GROUP BY` que nous avons récupérées. Ainsi, pour chaque groupe, nous appliquons la fonction d'agrégation. À la fin, nous retournons le résultat pour chaque groupe.

Algorithme 10 Fonctions d'agrégation

```
Require: Agregator (resultItem, query)
aggrFunction ← query.getAgregateFunction()
var ← query.getGroupByVar()
if (aggrFunction is count) then
  if (var is subject) then
    while (result1 in resultItem) do
      while (result2 in resultItem) do
        if (result1.subject.equals(result2.subbject)) then
          count ← count + 1
        end if
      end while
      // add result_1 subject and count number in the result
      count ← 0
    end while
  else if (var is predicate) then
    while (result1 in resultItem) do
      while (result2 in resultItem) do
        if (result1.predicate.equals(result2.predicate)) then
          count ← count + 1
        end if
      end while
      // add result_1 predicate and count number in the result
      count ← 0
    end while
  else if (var is object) then
    while (result1 in resultItem) do
      while (result2 in resultItem) do
        if (result1.object.equals(result2.object)) then
          count ← count + 1
        end if
      end while
      // add result_1 object and count number in the result
      count ← 0
    end while
  end if
end if
```

6.4 Evaluation

Nous avons implémenté un système d'interrogation de données compressées au format RDSZ dont l'architecture a été présentée dans la section 6.3.1. Nous utilisons ce système pour valider notre approche en orientant notre évaluation suivant deux aspects : le temps de traitement et l'espace de stockage mémoire. L'évaluation des performances relatives au temps de traitement revient à évaluer le temps de compression des données et le temps d'exécution de la requête. Toutes les expérimentations sont réalisées sur une machine équipée d'un processeur AMD E2-1800 APU à 1,70 GHz et de 4Go de RAM tournant avec un système Windows 10.

Nous utilisons 9757 graphes RDF issus de données d'observations de 20 000 stations météorologiques décrites dans le chapitre 5 (section 5.4). Ces données contiennent l'agrégation de données climatiques collectées dans différentes stations aux États-Unis depuis 2002. Ces données sont collectées au département de météorologie de l'Université de l'Utah à l'aide du laboratoire Kno.e.sis (Centre d'excellence de l'Ohio en matière de connaissances).

Dans ce qui suit, nous montrons les performances relatives au temps de traitement (section 6.4.1) et à l'espace de stockage mémoire (section 6.4.2) de notre solution comparée à l'algorithme de base RDSZ.

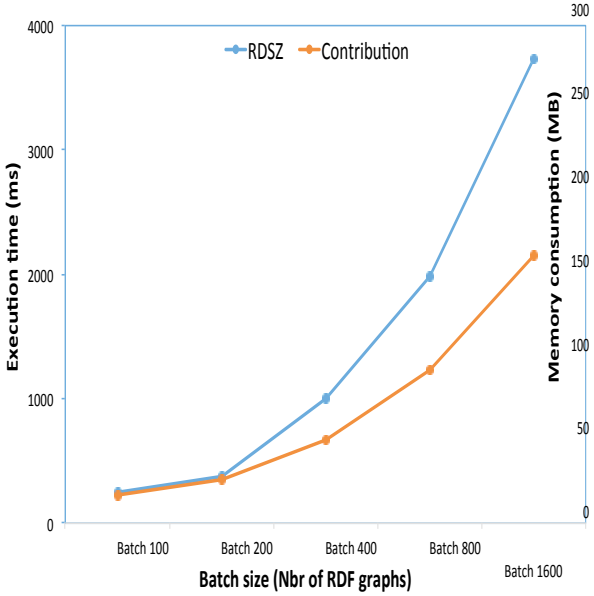


FIGURE 6.4 – Évaluation du temps d'exécution d'une requête SPARQL.

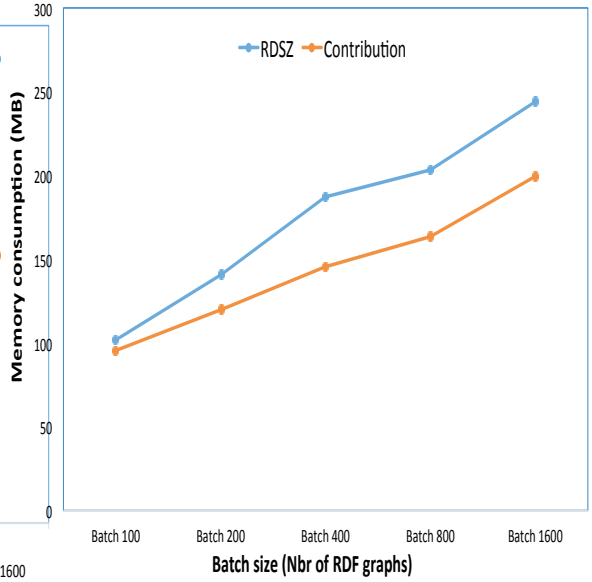


FIGURE 6.5 – Évaluation de la consommation de mémoire d'une requête SPARQL.

6.4.1 Évaluation du temps de traitement

Le temps d'exécution d'une requête dépend de plusieurs paramètres, tels que la taille du batch ou la taille des données. La taille du batch est le nombre d'items que le système doit recevoir avant de commencer le traitement des données. Nous évaluons le temps d'exécution en faisant varier la taille du batch. Nous considérons comme temps d'exécution de base celui observé après l'exécution de la même requête d'origine exécutée avec RDSZ. Notons que les requêtes SPARQL ne peuvent pas interroger directement les données compressées avec RDSZ, et nécessitent donc une phase de décompression.

Ainsi le temps d'exécution d'une requête avec RDSZ est la somme des temps nécessaires pour la phase de compression, la phase de décompression et celui d'exécution de la requête, tandis que le temps d'exécution d'une requête avec notre approche est la somme du temps nécessaire pour la phase de compression plus celui requis pour notre modèle d'exécution. Les requêtes sont exécutées sur différentes tailles de batch (100, 200, 400, etc.) afin de voir l'évolution du temps lorsque la taille du batch augmente. Le graphe de la figure 6.4 illustre que plus la taille du batch augmente, plus le temps de calcul augmente. Cet effet est dû au fait que lorsque la taille du batch augmente, le système doit attendre qu'un plus grand nombre d'items commence son traitement et par conséquent le temps de traitement augmente. Les temps d'exécution observés avec notre approche sont plus petits. En effet, nous exécutons les requêtes juste après l'étape de compression de RDSZ, ce qui nous permet d'éliminer le temps de décompression. Par conséquent, nous avons des temps d'exécution plus petits.

6.4.2 Évaluation de l'espace de stockage mémoire

L'espace mémoire utilisé par RDSZ est calculé en prenant le maximum de l'espace mémoire moyen utilisé entre la phase de compression et de décompression et l'exécution de la requête pour chaque élément d'un batch. La Figure 6.5 montre les résultats obtenus

où nous notons que plus la taille du batch augmente, plus l'espace mémoire utilisé est important. L'espace mémoire utilisé par notre algorithme demeure moins important que celui utilisé avec l'algorithme RDSZ. En effet, lors de l'exécution de la requête avec l'algorithme RDSZ de base, les données sont décompressées, occupant ainsi une partie de l'espace mémoire alloué à la machine virtuelle Java (JVM). En éliminant cette phase, l'espace qui devait être utilisé par cette étape est non utilisé. Il convient de noter que l'augmentation de la taille des données de test conduit à plus de gains de performance (en espace mémoire plus précisément).

6.5 Conclusion

Les formats de représentation de données RDF peuvent être très verbeux et la rapidité de génération des flux RDF peut très vite exploser le volume de données à gérer. Pour cela, il est de plus en plus nécessaire de réduire le format et la charge de données à traiter. Ainsi, l'espace de stockage mémoire utilisé peut être considérablement réduit en adoptant un mécanisme de compression des flux de données RDF (approches RDSZ et ERI). Cependant, quelle que soit l'approche de compression, les systèmes RSP nécessitent une phase de décompression avant le traitement des requêtes continues. Ce mécanisme rajoute un coût supplémentaire au temps de calcul annulant ainsi les gains obtenus en espace de stockage.
Dans ce chapitre, nous avons proposé une approche permettant d'éliminer la phase de décompression de données RDF compressées au format RDSZ afin d'exécuter directement les requêtes SPARQL continues sur un format compressé. Nous avons inclus la phase de traitement de requêtes SPARQL simples (requêtes contenant uniquement des opérateurs de jointure dans sa clause WHERE) et complexes (requêtes avec opérateurs de filtre et/ou d'agrégation) après la phase de compression de l'approche RDSZ.

L'évaluation de notre approche et sa comparaison au scénario de base (traitement avec phase de décompression incluse) a montré que l'interrogation continue des données RDF est réalisée avec plus d'espace de stockage mémoire requis et un temps de calcul largement plus important comparé au temps de calcul nécessaire pour le traitement immédiat après la phase de compression. Le système implémenté est modulable et peut être adapté ou inclus dans un système RSP afin d'optimiser en temps réel l'échange de données entre composants du système, le stockage mémoire et accélérer le temps de calcul des requêtes SPARQL continues.

Bilan d'étape. Nos propositions sur l'échantillonnage de flux de graphes RDF, sur le résumé orienté graphe RDF et sur l'interrogation de flux données RDF compressées au format RDSZ constituent notre premier groupe de contributions sur la réduction de la charge de traitement. Ces approches sont implémentées sur des systèmes de traitement centralisé comme C-SPARQL. Cependant, les performances de ces systèmes baissent considérablement en cas de large volumes de flux de données RDF, de plusieurs requêtes concurrentes, d'enchaînement de requêtes, de croisement de données statiques et dynamiques, etc. Dans la suite de cette thèse, nous présentons les études et nos travaux qui concernent notre second groupe de contribution sur la réduction de la charge de traitement. Ces travaux concernent les systèmes et approches pour la gestion distribuée des données et du traitement. Ainsi, le chapitre suivant présente l'état de l'art sur les systèmes de traitement distribués de flux de données RDF existants.

Publication

[45]. Déme, Ndéye Bousso, Amadou Fall Dia, Aliou Boly, Zakia Kazi-Aoul, and Raja Chiky. "An Efficient Approach for Real-Time Processing of RDSZ-Based Compressed RDF Streams."

Chapitre 7

État de l'art sur les systèmes et approches distribués pour la gestion de flux de données RDF

Sommaire

7.1 Introduction

La croissance de la génération de données en temps réel et de données stockées nous amène à penser constamment aux défis des 3V du big data : volume, vélocité et variété. Les systèmes de traitement centralisé de flux RDF existants ont levé le verrou de la variété en définissant un modèle commun pour la production, la transmission et l'interrogation continue des données suivant le modèle RDF.

Étant donné que l'objectif final est la réduction de la charge de traitement, nous avons dans les chapitres précédents proposé des solutions pour réduire cette charge en gardant un système centralisé et en proposant soit de faire des résumés soit de compresser les données. Nous nous intéressons dans ce chapitre à une autre manière de réduire cette charge en parallélisant et en distribuant le traitement des données et des requêtes. Plusieurs solutions existent. Quatre principaux systèmes de traitement distribué de flux de données RDF ont été proposés autour des plateformes de traitement continu et distribué telles que Apache Storm, Apache S4 et Apache Spark. Il s'agit des moteurs CQELS Cloud [20], C-SPARQL on S4 [21], DIONYSUS [23] et Strider [22]. Ces systèmes apportent a priori une capacité de traitement en temps réel de grandes masses de données RDF en mode

parallèle et distribué. Dans ce chapitre, nous étudions ces systèmes de traitement distribués en dressant leurs avantages et limites.

La suite de ce chapitre est structurée comme suit. La section 7.2 introduit les principaux outils de traitement de larges volumes de données utilisés dans cette thèse ou par les principaux systèmes de traitement distribué de flux de données RDF. Nous étudions ensuite ces systèmes RSP distribués dans la section 7.3. Enfin, la section 7.4 conclut ce chapitre.

7.2 Outils de traitement de gros volumes de données

Dans cette section, nous étudions les principaux systèmes de file d'attente et de messagerie distribuée (kafka), de traitement en parallèle et en temps réel (Storm, S4 et Spark), de bases de données clé/Valeur orientées mémoire (Redis) utilisés par les systèmes RSP distribués et/ou dans nos contributions (chapitres 8 et 9).

7.2.1 Apache Kafka

Apache Kafka [133] est un système distribué de transmission de messages proposé à l'origine par LinkedIn. Il est capable de prendre en charge un nombre important de messages par seconde (106/seconde). Il est plus rapide que les systèmes JMS classiques (Apache ActiveMQ, JBossMQ, etc.) car la distribution des messages ne prend pas en compte l'ordre et l'accusé de réception. Apache kafka présente quatre principaux composants.

1. **Topic**. Il permet une classification des messages dans des catégories (par exemple animal, voiture, etc.) prédéfinies par l'utilisateur. Afin de faciliter la parallélisation des traitements, un topic est découpé en partitions qui sont répliquées sur plusieurs machines.
2. **Producteur**. Il envoie un message à un topic. Ce message est assigné à l'une des partitions par défaut de manière uniforme (algorithme d'ordonnancement round-robin).
3. **Consommateur**. Chaque consommateur est associé à un groupe souscrit à un topic. Une partition ne peut être consommée que par une unique instance d'un consommateur d'un groupe donné. Le nombre maximum de consommateurs travaillant en parallèle est déterminé par le nombre de partitions dans chaque topic. Kafka permet un "re-balancing" des partitions pour les répartir parmi les consommateurs présents. Chaque consommateur utilise un pointeur de lecture (offset) pour tracer et contrôler sa position de lecture.
4. **Brokers**. Ils représentent les serveurs du cluster où les partitions sont répliquées. Chaque message persiste dans le broker durant une période de rétention prédéfinie par l'utilisateur. Chaque broker est capable de traiter des téraoctets de messages en même temps.

La coordination entre les brokers est assurée à travers Apache Zookeeper [134]. Elle permet de désigner un leader pour une partition d'un topic donné ou d'introduire (à chaud) un nouveau broker dans le cluster. Zookeeper se charge également de piloter les consommateurs et de stocker les derniers points de lecture de chaque consommateur dans chaque partition.

7.2.2 Apache Storm

Apache Storm [135] est un système de traitement parallèle et temps réel de flux de données qui vise à éliminer le vide laissé par les systèmes Hadoop et MapReduce [1]. Storm a été créé par Backtype, une société acquise par Twitter en 2011. Il s'agit d'un projet libre et open source sous licence Eclipse Public License et qui est écrit en Clojure et Java. Hadoop et MapReduce fournissent un framework pour le traitement par batch, tandis que Storm fournit un framework pour le traitement en temps réel. Storm est utilisé pour analyser, filtrer et normaliser les informations et de nombreux autres filtres en expression régulière sur les journaux en temps réel. Apache Storm garanti la tolérance aux pannes et la non perte de données entre composants (spout et bolts). Dans plusieurs domaines d'application, il peut être utilisé pour traiter un flux de données entrantes et mettre à jour des bases de données en temps réel, mais aussi pour interroger en continu et diffuser les résultats aux clients en temps réel et exécuter en parallèle et à la volée plusieurs requêtes SPARQL continues. Dans nos quatrième et cinquième contributions, nos systèmes sont implémentés au dessus de Apache Storm. Dans ce que suit, nous définissons ainsi les principaux concepts de Storm que nous utilisons dans nos implémentations.

7.2.2.1 Concepts

7.2.2.1.1 Topologie

Une topologie storm est un ensemble de tâches définies par des **bolt** et des **spouts** par des flux d'informations (figure 7.1). Elle peut être comparé aux tâches *Map* et *Reduce* de *Hadoop*. La topologie peut être vue comme des éléments d'un graphe orienté, où les tâches (regroupés au sein des **bolts** et **spouts**) représentent des sommets et les flux représentent des arêtes dirigées. Dans la mesure où une topologie est supposée traiter un flux en temps réel, elle traite les messages aussitôt qu'ils arrivent. La seule façon d'arrêter une topologie storm en cours d'exécution est d'envoyer la commande "kill" ou de définir un délai d'expiration ultérieur.

7.2.2.1.2 Cluster

Un cluster Storm peut être considéré comme un cluster Hadoop dans la mesure où il existe deux types de nœuds : "maître" (master) et "esclaves" (workers).

7.2.2.1.3 Tâches

Une **tâche** Storm implémente un mécanisme de traitement autour de composants appelés **bolt** et **spout**. Le **worker** est conçu comme une boucle implicite, de ce fait, les fonctions de **bolts** et des **spouts** sont appelées dans une boucle par le **worker**. La **tâche** Storm déclare les types de champs dans des flux sortants appelés **flux**. Un **flux** est défini comme une suite illimitée de **tuples** où un **tuple** désigne une liste nommée de valeurs. Ces valeurs peuvent avoir des types différents. Un **flux** peut être divisé et son **tuple** peut être dirigé vers plusieurs **bolts** pour un traitement parallèle. Un identifiant est attribué à chaque **flux** et à chaque **tuple** pour sélectionner un **flux** et reconnaître un **tuple** pour la fiabilité du traitement.

7.2.2.1.3.1 Spout

Un **Spout** est défini comme source d'un ou plusieurs **flux** émettant des **tuples**. Un **flux** peut être utilisé pour générer des **tuples** de n'importe où, (par exemple, fichier, base de données, topic kafka, etc.).

1. http ://hadoop.apache.org/

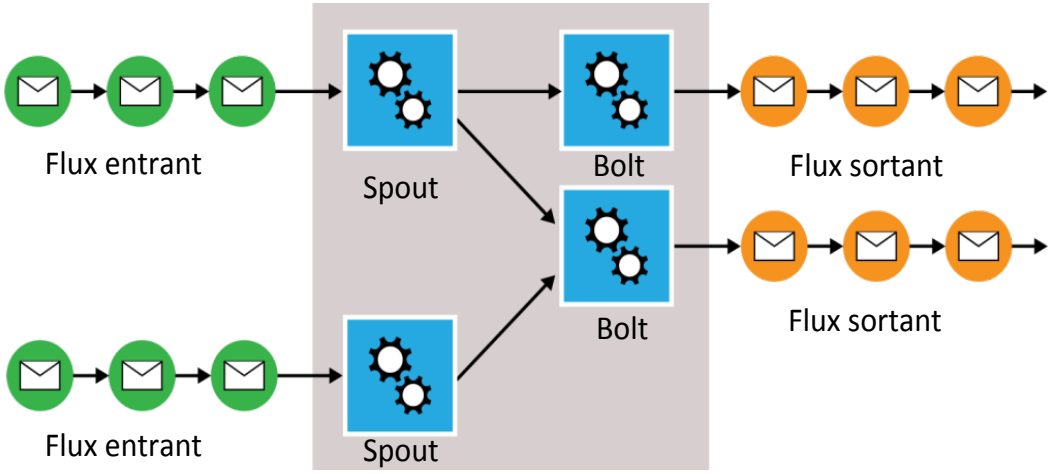


FIGURE 7.1 – Exemple de topologie Storm.

7.2.2.1.3.2 Bolt

Un **Bolt** est défini comme étant un élément de traitement acceptant en entrée un ou plusieurs **flux** et n'en émettant aucun, un ou plusieurs. Il peut effectuer tout type de traitement sur des **tuples** entrants, par ex. agrégation, filtrage, transformation, etc. Un identifiant de composant est attribué à toutes les **tâches** (**spout** et **bolt**), qui servent à identifier sa position dans la topologie, et un ID de tâche, qui permet de localiser une instance de tâche dans le cluster et de lui acheminer ses **flux**.

7.2.2.1.3.3 Groupement de flux

Le groupement de flux est la méthode de répartition des tuples d'un flux parmi les instances parallèles d'un bolt. Un bolt enrégistre plusieurs flux et peut définir un groupe de flux différents pour chacun. Les groupes de flux fournis par Storm sont les suivants :

— **Shuffle grouping**. Distribution uniforme des tuples au niveau des bolt.

— **Fields Grouping**. Il défini comme un ensemble de clés de tuples échangés entres bolts. Les tuples avec des clés égales sont mappés sur la même instance de bolt.

— **All grouping**. Le flux est répliqué sur toutes les instances de boulons (broadcast).

— **None grouping**. Dans ce cas précis, la distribution des tuples est indéfinie. L'implémentation actuelle de storm renvoie cette configuration au **Shuffle grouping**.

— **Direct grouping**. La tâche émettrice spécifie le bolt recepteur pour chaque tuple émis.

— **Local or shuffle grouping**. Les boulons situés sur le même **worker** que la tâche émettrice sont préférés, ce qui reste équivalent à un **Shuffle grouping** pour les bolts assez distants.

7.2.2.1.4 Fiabilité

Storm fournit un framework fiable en garantissant le traitement de tous les tuples. En effet, chaque tuple émis par un spout est associé à un identifiant et est vu comme la racine d'un arbre. Lorsqu'un tuple est traité par un bolt, les nouveaux tuples résultants sont ajoutés à l'arbre en tant que tuple enfant. Ajouter des tuples enfant à l'arbre doit être fait explicitement en ancrant le tuple émis dans un autre tuple. A chaque étape traversant l'arborescence, un accusé de réception ou un accusé de réception négatif (fail) est envoyé à une tâche de suivi appelée **Acker**. Si tous les tuples enfants sont ajoutés jusqu'à ce que les feuilles soient reconnues avec succès, l'arbre entier est considéré comme entièrement traité. Sinon, si un échec est envoyé ou un tuple arrive à expiration, l'arborescence entière est considérée comme ayant échoué et l'ID du tuple racine est transmis à un gestionnaire de tâches interrompues dans le spout.

7.2.3 S4

S4 [136] est une plateforme de traitement en continu qui permet aux développeurs de créer des applications de gestion parallèle, distribuée et sans limite de flux de données. Ceci est accompli en écrivant de petits sous-programmes, appelés éléments de traitement (**PEs**), capables de consommer des événements d'un certain flux, et de publier directement leurs résultats ou d'émettre de nouveaux événements sur un autre flux, qui peut ensuite être consommé par d'autres PE. Les flux sont distribués entre les PE selon une clé donnée (c'est-à-dire que tous les événements d'un flux donné et d'une clé donnée seront traités par le même PE). Cela signifie que les PE peuvent être répartis sur différents nœuds, ce qui permet une approche parallèle. Lorsque les clés sont affectées de manière intelligente, le partitionnement des événements de flux sera uniforme, ce qui aboutit à un système performant et passant à large échelle.

7.2.4 Apache Spark

Spark est un framework de traitement en grappe de type MapReduce qui propose une collection d'éléments parallèles tolérants aux pannes, appelée RDD (Resilient Distributed Dataset) [26]. Un RDD est divisé en plusieurs partitions sur différents nœuds d'un cluster, de sorte que les opérations peuvent être effectuées en parallèle. Spark permet des calculs parallèles sur des machines et gère automatiquement les tâches de planification tenant compte de la tolérance aux pannes et de l'équilibrage de charge. Spark Streaming étend le RDD au flux discrétisé (DStream) [27] et permet de prendre en charge le traitement de données en temps quasi réel en créant des microbatches de durée τ. DStream représente une séquence de RDDs auxquels chaque RDD se voit attribuer une estampille. Semblable à Spark, Spark Streaming décrit les logiques informatiques comme un modèle RDD de graphe acyclique orientée DAG (Directed Acyclic Graph). Chaque batch génère une instance en fonction de ce modèle pour une exécution ultérieure du travail. Le modèle d'exécution en microbatch offre une latence en dessous de la seconde et une capacité de traitement élevée.

7.2.5 Redis

Redis (REmote DIctionary Server) [137] est présentement considéré comme l'une des principales bases de données NoSQL dans la catégorie des bases de données clé/valeur en mémoire [2]. Dans le cadre général, la motivation pour l'adoption de ce système repose sur la :

2. https ://db-engines.com/en/ranking

— **Robustesse**. La version initiale du système date de 2009. Ce système est utilisé par des groupes tels que Alcatel Lucent, Twitter, Uber, SnapChat, Github, Docker, AirBnB, Pinterest et Square.

— **Facilité de persistance**. Redis propose une approche de persistance périodique qui évite de perdre des volumes importants de données causée par une panne du système.

— **Maintenance**. Le système, qui est maintenant en version 4, a été amélioré, optimisé et étendu avec de nouvelles fonctionnalités par une grande communauté de programmeurs.

— **Distribution**. Le système est capable de partager les partitions sur un ensemble de machines en adoptant une approche maître-esclave avec réplication.

Le système est performant grâce au stockage en mémoire, facile d'installation, libre et fournit plusieurs interfaces de langage de programmation incluant plusieurs langages fonctionnant sur la JVM (Java, Scala, etc.).

Au-delà de ses performances extrêmes, Redis est reconnu pour sa simplicité et sa polyvalence. Il peut être utilisé pour stocker des données à haute vitesse de génération afin d'éviter de les perdre avant l'analyse. L'utilisation de Redis comme base de données pour gérer les volumes de données élevés, consolider les écritures de données ou fournir un débit rapide réduit le nombre de bases de données sur disque à exécuter. Il va également réduire les coûts d'exploitation et de maintenance. Redis est l'une des structures de données particulièrement importantes dans plusieurs domaines d'applications de larges volumes de flux de données RDF. Elle propose certaines opérations utiles sur les ensembles triés : récupérer des éléments dans l'ordre croissant ou décroissant, obtenir des éléments en fonction d'une plage, c'est-à-dire un intervalle de valeurs ou supprimer des éléments via une clé ou une plage.

Les systèmes RSP ont gagné quelques années d'avance en réutilisant les plateformes de traitement distribué présentées ci-haut (Storm, Spark et S4) pour la gestion parallèle et distribuée de flux de données RDF. Dans la suite, nous présentons les principaux systèmes RSP implémentés au dessus de ces plateformes distribuées.

7.3 Systèmes RSP distribués

7.3.1 CQELS Cloud

Les auteurs de CQELS ont introduit un nouveau système appelé CQELS Cloud [20]. Le système proposé est parmi les premiers accomplissant le traitement élastique et scalable pour les données LOD [3]. **Streaming SPARQL** [15], **C-SPARQL** [16], **SPARQL**$_{stream}$ [1] et **CQELS** [18] utilisent un modèle de données RDF similaire et une sémantique de requête, mais sont conçus pour fonctionner sur une seule machine. CQELS Cloud distribue le traitement et propose un algorithme et une architecture adaptés au cloud computing. Le système CQELS Cloud repose sur un modèle d'exécution élastique et des algorithmes de parallélisation pour le calcul incrémental d'opérateurs de requête continue.

— **Modèle d'exécution pour l'exécution de requêtes continues sur flux RDF** : le modèle d'exécution de CQELS Cloud gère un ensemble de requêtes continues dans la syntaxe de langage de requête CQELS (CQELS-QL [18]) pour produire un résultat d'ensemble au format de résultat SPARQL (CONSTRUCT, DESCRIBE, SELECT ou

3. Linked Open Data

ASK). Le **coordinateur d'exécution** met en correspondance les opérateurs entre les nœuds de traitement. Ce modèle d'exécution minimise les coûts de communication en utilisant d'abord l'approche de codage du dictionnaire de CQELS pour la compression et le second déploiement sur la même machine. Cette stratégie garantit une utilisation optimale des ressources, la tolérance aux pannes, l'élasticité et la scalabilité.

— **Algorithmes de parallélisation pour le traitement incrémental des opérateurs de requête continue** : chaque opérateur continu du langage CQELS-QL (par exemple agrégation, jointure et filtrage) est considéré comme complexe en raison de l'ensemble des appariements et des résultats intermédiaires nécessaires pour exécuter une requête. Les auteurs ont effectué une évaluation incrémentale pour les opérateurs de fenêtres glissantes qui distribuent les tâches incrémentales sur plusieurs nœuds.

Pour le traitement parallèle sur l'infrastructure du cloud computing, les auteurs ont implémenté leur modèle d'exécution élastique et leurs algorithmes parallèles en utilisant Zookeeper, Storm comme couche en ligne et HBase comme couche de stockage. CQELS Cloud gère la disparition ou l'apparition d'un ou plusieurs nœuds et est capable, dans le premier cas, de récupérer le traitement en réaffectant les tâches aux autres nœuds du réseau en conséquence et prend évidemment en charge l'élasticité. La figure 7.2 montre l'architec-

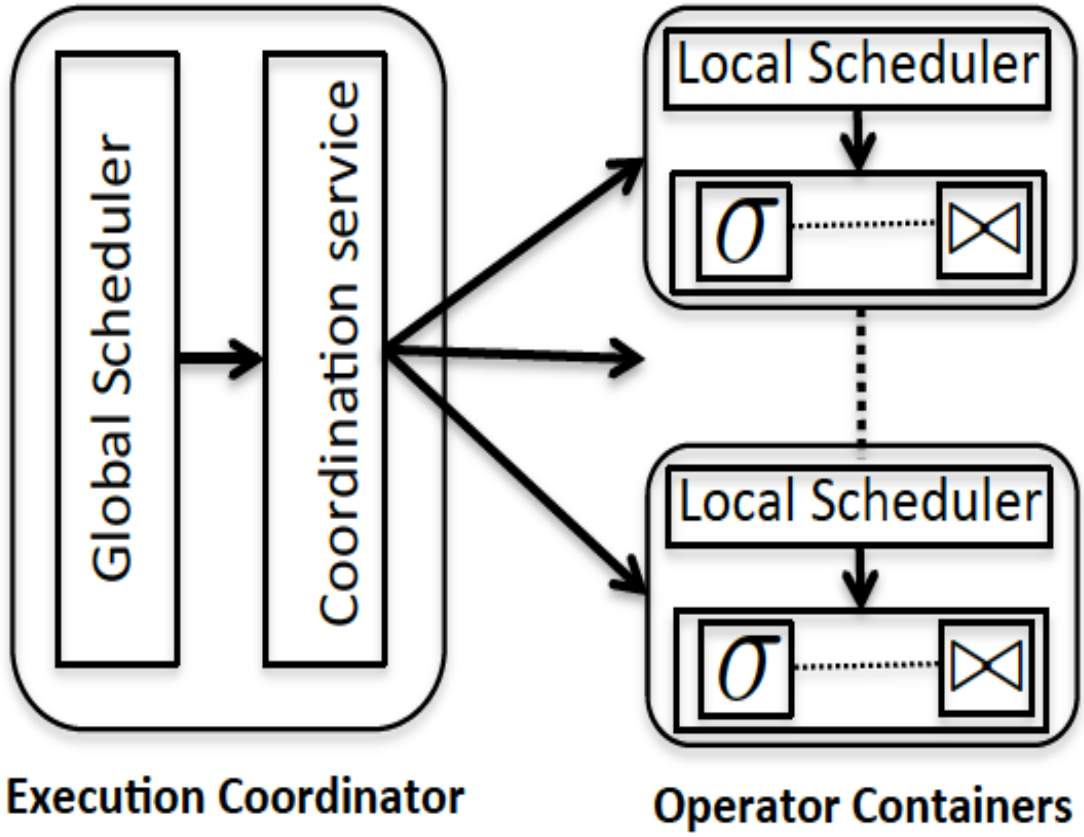


FIGURE 7.2 – Architecture de CQELS Cloud.

ture de CQELS Cloud. Le **coordinateur d'exécution** coordonne le groupe de nœuds **OCs** ((Operator Container) en utilisant les services de coordination fournis par Storm et HBase

qui partagent le même cluster Zookeeper. Le **Global Scheduler** utilise **Nimbus 5** pour déployer le code des opérateurs sur les OCs et surveille les défaillances. Chaque nœud OC exécute un superviseur **Storm** qui écoute les tâches de traitement continu assignées à sa machine via **Nimbus**. Les tâches de traitement qui doivent traiter les données persistantes utilisent le composant client **HBase** pour accéder aux données stockées dans HBase. Les nœuds exécutant un OC hébergent également les **DataNodes HDFS** du cluster d'**HBase**. Les DataNodes sont accessibles via le composant l'OC **HRegionServer** de HBase.

L'évaluation du système montre comment le débit évolue linéairement avec le nombre de nœuds de traitement. Les auteurs évaluent principalement la scalabilité de l'opérateur (en augmentant le nombre de machines exécutant des OCs) et la parallélisation des requêtes (en traitant plusieurs requêtes simultanées en utilisant le scénario de LSBench [70]). Concrètement, CQELS Cloud peut évoluer jusqu'à un million d'entrées par seconde avec 100 000 requêtes simultanées sur un cluster de 32 nœuds de traitement EC2.

CQELS Cloud reste le système de référence dans la pile des moteurs de traitement de flux RDF distribués. Toutefois, il considère uniquement les opérateurs de correspondance, de jointure et d'agrégation et n'inclut aucune tâche de raisonnement.

7.3.2 C-SPARQL on S4

C-SPARQL on S4 (High-performance Distributed Stream Reasoning using S4) [21] présente une nouvelle implémentation de C-SPARQL basée sur la plateforme de traitement distribué de flux S4 [4]. Le système fonctionne sur un schéma RDF fixe et implémente un raisonnement RDFS partiel pour fournir un motif efficace correspondant aux flux RDF, ajoutant de la complexité au système. Les auteurs distribuent une tâche continue sur un ensemble de nœuds en considérant deux sous-tâches :

— **Exécution parallèle** : pour la distribution de triplets, les auteurs présentent un nouveau schéma amélioré qui réduit le nombre de fois que chaque triplet doit être traité, améliore les propriétés d'équilibrage de charge du système et limite le nombre d'éléments de processus intermédiaires lors du traitement d'un triplet donné.

— **Formatage du résultat d'une requête** : les auteurs développent un ensemble d'opérateurs de flux qui peuvent être utilisés pour prendre en charge des fonctionnalités clés dans C-SPARQL.

Le paradigme de Yahoo S4 [136] pour réaliser des applications de traitement continu en parallèle distribuées est l'écriture de petits sous-programmes, appelés éléments de traitement **PEs** (**Processing Elements**). Les PEs peuvent consommer des événements d'un flux donné et publier leurs résultats directement, ou émettre de nouveaux événements sur un autre flux, qui peuvent ensuite être consommés par d'autres PEs. Les flux sont répartis entre les PEs selon une clé donnée (<sujet, prédicat, objet> du flux RDF). Cela signifie que les PEs peuvent être répartis sur différents nœuds, ce qui permet une approche parallèle. Lorsque les clés sont attribuées intelligemment, cela entraîne un partitionnement uniforme des événements de flux, ce qui se traduit par un système hautement scalable.

Les auteurs implémentent les opérateurs de projection, de jointure, de sélection, de filtrage et d'agrégation. Le système prend en charge le fenêtrage basé sur le temps mais n'implémente pas la fenêtre basée sur le nombre de tuples. Leur implémentation ne prend pas en charge la jointure temporelle complexe et ne peut non plus répondre aux requêtes continues avec chevauchement d'événements temporels. L'approche est principalement

4. http ://incubator.apache.org/s4/

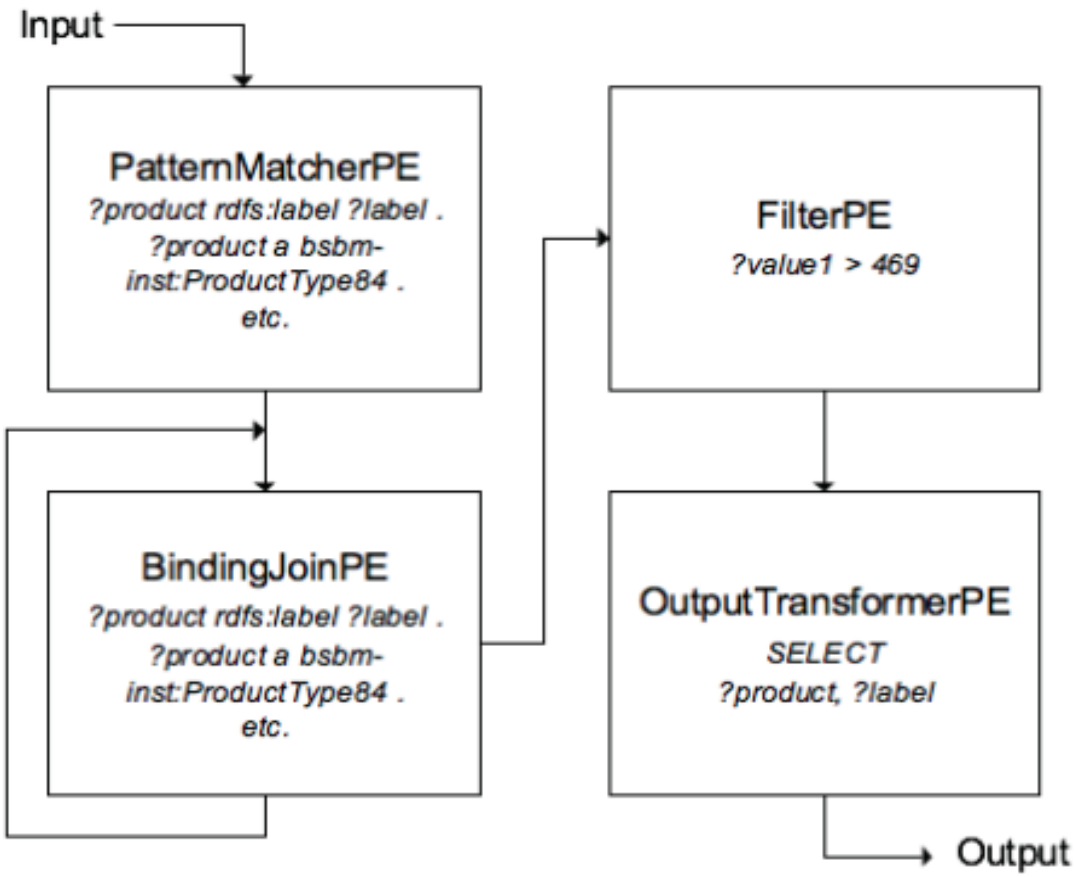


FIGURE 7.3 – Architecture de C-SPARQL on S4.

axée sur la création d'un certain nombre de composants pouvant être combinés pour traduire un sous-ensemble de requêtes C-SPARQL dans un plan d'exécution parallèle. La figure 7.3 illustre un résumé de l'ensemble du processus d'un PE. Initialement, les données en entrée sont traitées par un ensemble de **PatternMatcherPEs**, qui lient les variables du flux d'entrée. La sortie est envoyée à un réseau de **BindingJoinPEs**, qui effectuent les jointures dans la requête. Il est également possible d'appliquer certaines fonctions de filtrage en utilisant un FilterPE placé après ou avant le **BindingJoinPE**. Les résultats finaux sont produits par un **OutputTransformerPE**. Les auteurs décrivent chaque PE que nous incluons brièvement dans ce qui suit.

— **PatternMatcherPE.** Il filtre les triplets qui ne correspondent à aucun pattern de triplet dans le graphe et crée des liaisons de variables. Étant un PE sans clé, il existe une instance de **PatternMatcherPE** sur chaque nœud de traitement, qui reçoit tous les triplets qui entrent dans la partie de traitement SPARQL sur ce nœud spécifique. Cela rend la recherche de triplets hautement parallèle. **PatternMatcherPE** génère les liaisons de variables, triées par nom de variable.

— **BindingJoinPE.** Pour chaque variable de la requête, un ensemble de PE est créé. Les variables reçoivent un certain ordre. Chaque ensemble de PE traite les jointures pour une variable donnée. Ceci est implémenté en écoutant tous les **PatternMatcherPE** et en maintenant les liaisons dans une liste par motif. Lorsqu'une liaison de variables apparaît dans toutes les listes, une nouvelle liaison est créée et émise vers les PE pour la variable suivante. Ceci est répété jusqu'à ce qu'il n'y ait plus de variables.

— **FilterPE.** Pour chaque liaison entrante, **FilterPE** vérifie si elle correspond à la condition de filtre fournie dans la configuration. Si cela correspond, la liaison est envoyée à son flux de sortie, sinon elle est ignorée.

— **Unions.** Afin de prendre en charge d'autres modèles de graphe (union), aucun composant supplémentaire n'a besoin d'être introduit. Les sorties de deux paires **PatternMatcher**/**BindingJoinPE** peuvent utiliser le même flux de sortie, ce qui entraînera l'insertion de l'union des deux dans ce flux de sortie.

— **OutputTransformerPE.** Il créera finalement une projection des variables distinguées et, dans le cas d'une requête **CONSTRUCT**, une extension. Il est également responsable du formatage des résultats.

Les auteurs ont évalué leur implémentation sur 32 nœuds en utilisant les données du benchmark LUBM [138] pour tester le raisonnement et le benchmark partiel de BSBM [5] pour tester le résultat des requêtes. Les auteurs font d'abord une simple évaluation avec une simple requête `passthrough` (c'est-à-dire, une requête sans opération complexe) sur seulement 3 nœuds de traitement, puis observent un débit élevé (un peu plus de 150 000 triplets par seconde). Dans d'autres expériences, ils utilisent des requêtes allant du simple `passthrough` au plus complexe (avec plusieurs jointures et filtrage) incorporant le raisonnement RDFS et mesurant un haut débit (plus de 60 000 triplets par secondes) et une extensibilité assez linéaire jusqu'à 8 nœuds. Néanmoins, ces performances se dégradent au delà de 8 nœuds.

A notre connaissance, ce système que nous appelons CSPARQL sur S4 est un travail préliminaire sur le raisonnement sur flux de données RDF distribuées. Il est assez scalable avec un débit global assez élevé pour des requêtes de raisonnement simples ou complexes mais n'est pas un système complet (au niveau opérateur) comme l'est CQELS Cloud. S4 et Storm sont des plateformes de traitement continu et distribué de flux de données. Cependant, l'utilisation de S4 crée une nouvelle instance du bloc de construction pour chaque valeur de regroupement tandis que Storm gère plusieurs valeurs de regroupement par instance et est plus susceptible de fournir de meilleures performances.
Le système permet l'analyse de différents types de requêtes, de flux et de séquences à travers une seule interface de requête. Il fournit également la complétude sémantique et la transparence des localisations. Cela implique l'ajout de cohérence (par exemple, une nouvelle source de données ajoutée) et de stratégies d'optimisation des requêtes et de réduction du trafic réseau en utilisant des stratégies locales d'optimisation et de traitement.

7.3.3 DIONYSUS

DIONYSUS [23] est un prototype qui fournit une interface permettant à l'utilisateur de décrire des requêtes d'analyse (par exemple, une requête décrivant le nombre d'appareils actifs et leur consommation d'énergie dans une maison), de séquence (par exemple, déterminer la séquence de consommation d'énergie par des appareils électroniques dans une maison) et dynamiques sur des graphes RDF distribués. Afin de gérer les problèmes de variété et d'hétérogénéité des sources de données, les auteurs ont d'abord modélisé les données sous format graphe RDF et ont basé leur conception sur quatre objectifs principaux. Le premier est la définition d'un stockage CBGP-store (Common Basic Graph Pattern) qui est assigné à un BGP (Basic Graph Pattern) générique et généré manuellement ou automatiquement à partir d'une ontologie de domaine. Ce modèle de stockage permet un traitement incrémental et indexé sur des fenêtres glissantes pour une analyse efficace des données à la fin. Leur deuxième objectif est d'optimiser localement et de traiter un graphe

5. www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/

de requêtes EQGs (Exact Query Graphs) sur son CBGP-store correspondant (prévu). Un EQG est la version orientée requête d'un CBGP-store (c'est-à-dire plus sélectif) avec des modèles de triplets distribués et des opérateurs SPARQL1.1 (SELECT, OPTIONAL, UNION, FILTER, GROUP BY, etc.).

Cependant, les auteurs de DIONYSUS décrivent un prototype et ne fournissent pas d'évaluation du système ni de code du système. Le système proposé demeure aujourd'hui une projection sur les opportunités qu'offriraient leur approche pour le traitement de flux de graphes RDF.

7.3.4 Strider

Un récent travail propose Strider [22] où les auteurs présentent un moteur de traitement distribué de flux RDF qui permet une évaluation continue des requêtes SPARKQL, basée sur Spark Streaming [6] (utilisé pour traiter les opérations de requête) et Apache Kafka [7] (utilisé pour gérer les flux de données). Ils ajoutent des composants aux requêtes existantes au lieu d'en créer une nouvelle et l'exécuter séparément. Les requêtes sont réexécutées à chaque intervalle microbatch spécifié par la clause BATCH.

Les auteurs proposent deux algorithmes adaptatifs pour générer le plan d'exécution de la requête. Ils démontrent dans leurs expériences un avantage en terme de débit de traitement comparée aux autres systèmes existants. Strider peut être ainsi considéré comme le premier système capable de gérer à grande échelle un débit élevé de flux de données RDF avec une latence relativement faible.

Cependant, la technique d'optimisation de Strider reste exclusivement orientée requête et les auteurs concentrent et limitent essentiellement leurs tâches à l'expressivité du langage SPARQL et non à la distribution du traitement en se basant sur la distribution et la disposition des nœuds dans les graphes RDF. De plus, Strider ne prend pas en charge l'interrogation de données RDF statiques (stockées).

7.4 Conclusion

Les systèmes de traitement distribué de flux de données RDF existants sont toujours à leur stade de développement. Ils apportent a priori une capacité de traitement, en temps réel, de grandes masses de données RDF en mode parallèle et distribué. CQELS Cloud est aujourd'hui le système de référence en matière de traitement distribué de flux RDF. Cependant, Strider est un moteur RSP distribué très récent et son expérimentation revendique son avantage sur les autres systèmes distribués tels que CQELS Cloud pour faire face à un débit élevé des flux. Cependant, Strider ne prend pas en charge l'interrogation de données statiques.

Ces systèmes RSP distribués restent limités au niveau des opérateurs de fenêtrage associés au langage de requête et proposent une réévaluation des requêtes par microbatch (c'est le cas de Strider). De plus, la jointure distribuée entre données statiques et dynamiques ainsi que les résultats intermédiaires générés durant les opérations de jointure (notamment les systèmes RSP CQELS Cloud et C-SPARQL on S4) dégradent considérablement les performances des systèmes RSP distribués. Enfin, les performances de ces systèmes laissent à désirer en présence de requêtes concurrentes et/ou dépendantes. Ils ne

6. https ://spark.apache.org/streaming
7. https ://kafka.apache.org

prennent pas en charge le rafraîchissement à des fréquences lentes (jour, semaine, mois, etc.) de données statiques.

Il est indispensable dès lors d'adopter des approches orientées partitionnement de requêtes pour un traitement parallèle et distribué des données RDF tout en réduisant au mieux les résultats intermédiaires générés et/ou échangés entre les nœuds de traitement du système. C'est ce que nous proposons dans les deux prochains chapitres.

Chapitre 8

4ème contribution : Système de traitement distribué de flux de graphes RDF (DRSS)

Sommaire

8.1 Introduction

Le groupe de travail du W3C RSP (RDF Stream Processing) a étendu le language de requête SPARQL pour l'interrogation continue de flux de données RDF via un certain nombre de systèmes de traitement de flux RDF C-SPARQL [64], $\text{SPARQL}_{\text{stream}}$ [1], CQELS [18], EP-SPARQL [17] et Sparkwave [19]. Ces systèmes apportent des solutions permettant de lever le verrou de l'hétérogénéité et mentionnent certains aspects de montée en charge [70] tels que les requêtes concurrentes, une fréquence rapide de génération des flux de données, le volume de données statiques et l'enchaînement de requêtes (cas d'une ou plusieurs requêtes nécessitant le résultat d'une autre).

Cependant, les performances des systèmes RSP laissent à désirer quand ces métriques de montée en charge dépassent certains seuils. En effet, les capacités en débits des systèmes C-SPARQL et EP-SPARQL atteignent difficilement un million de triplets (dans quel cas les système risquent de tomber en panne). De plus, la plupart des systèmes RSP ne peuvent pas dépasser un débit de 100 triplets par seconde pendant que le nombre de requêtes concurrentes dépasse les 100 [70, 20]. Pour pallier ces limites, nous avons proposé des solutions de réduction de la charge de flux de données à traiter par l'échantillonnage (chapitre 4), le résumé et l'interrogation de flux de graphes compressés (chapitre 6). Cependant, ces approches de réduction de la charge restent non adaptées à différents domaines d'application nécessitant de garder la totalité des données ou d'effectuer des analyses (le raisonnement par exemple) non conformes au format compressé des flux RDF. De

plus, en raison de la nature libre et verbeuse des flux de données, l'exécution d'une requête SPARQL fait souvent appel à d'intenses tâches de jointures générant d'innombrables résultats intermédiaires qui réduisent considérablement les performances des systèmes de traitement centralisé. Ainsi, ces systèmes de traitement centralisé ne peuvent pas faire face aux larges volumes de données statiques et dynamiques car ne bénéficiant pas des tâches de parallélisme et de distribution du traitement sur plusieurs nœuds de traitement. Les approches de traitement distribué de flux RDF proposées, dont les principales sont CQELS Cloud [20], SPARQL on S4 [21], le prototype DIONYSUS [23] et Strider [22] présentent des techniques avancées de montée en charge et de parallélisme. Cependant, ces récentes propositions restent limitées en matière **(1)** d'opérateurs de fenêtrage sur flux RDF, **(2)** de jointures entre données statiques et dynamiques, **(3)** d'optimisation de la distribution des données pour la réduction des quantités d'informations intermédiaires échangées entre nœuds de traitement, **(4)** de prise en charge de requêtes concurrentes et d'enchaînement de requêtes et **(5)** de rafraîchissement de données "statiques".

Dans ce chapitre, nous présentons **DRSS** (**D**istributed **R**DF **S**PARQL **S**treaming), un système de traitement distribué de flux de graphes RDF. **DRSS** apporte des solutions aux principales limites (**(1)**, **(3)**, **(4)** et **(5)**) des systèmes de gestion distribuée de flux RDF abordés ci-dessus en proposant :

— un nouveau langage de requête continue qui inclut trois nouvelles variantes de fenêtrage des flux RDF **(1** et **4)** et d'un opérateur de rafraîchissement de données "statiques" **(5)** ;

— un algorithme de `réécriture` des requêtes SPARQL continues non concurrentes partageant des sous-structures communes (patterns) afin d'éviter la surcharge et la redondance des données issues d'une même source mais interrogées par des requêtes différentes **(3)** ;

— un algorithme de `partitionnement de requêtes` (SPARQL orienté nœud de jointure pour une parallélisation des opérations de jointure **(3)**) ;

— et un algorithme de **partitionnement de graphes RDF** basé requête pour un traitement local des graphes RDF afin de minimiser les données échangées entre nœuds de traitement **(3)**.

Le partitionnement et l'optimisation des partitions en vue de la distribution efficace des flux de graphes RDF sont effectués en offline alors que les graphes en entrée sont dynamiquement distribués sur les nœuds de traitement. Les graphes RDF suivent un mécanisme de stockage clé-valeur en mémoire pour alléger le coût des opérations de jointure.

La suite de ce chapitre est organisée comme suit. La section 8.2 présente notre extension du langage de requête SPARQL. La section 8.3 présente l'architecture de DRSS proposée et les algorithmes de gestion distribuée des données et du traitement. La section 8.4 fournit les détails et les résultats d'évaluation du système DRSS. Enfin, la section 8.5 conclut ce chapitre.

8.2 Langage de requête DRSS

Dans cette section, nous présentons notre proposition d'extension du langage de requête SPARQL (langage DRSS) à exécuter dans le système DRSS. DRSS regroupe tous les opérateurs de requêtes présents dans SPARQL1.1. Ainsi, dans un premier temps, nous avons défini une nouvelle syntaxe de requête native, proche de celle de C-SPARQL. L'anatomie d'une requête DRSS peut être vue comme suit :

```
QUERY queryID AS
    {
          *PREFIX          name:<namespaceIRI>
          SELECT           vars aggr expressions  |
          CONSTRUCT        graph template         |
          DESCRIBE         resource               |
          ASK
          *FROM [NAMED] STREAM <streamSourceIRI> WINDOW
          *FROM [NAMED]        <staticSourceIRI>
          WHERE
          {
                                FILTER
               Triple           OPTION
               Pattern          AND
                                UNION
          }
          GROUP BY       expression
          HAVING         expression
          ORDER BY       expression
    }
          [REFRESH   time timeUnit]
```

Ce langage de requête inclut toutes les variantes de fenêtre pouvant être appliquées sur un flux de données. En plus des types de fenêtres existantes et prises en compte par les systèmes C-SPARQL et CQELS, notre syntaxe prend en compte cinq (5) variantes de fenêtres (WINDOW) que nous détaillons à travers des exemples de requête appliquées au cas d'utilisation concernant des données de gestion d'un réseau de transport et de distribution d'eau potable (pression, débit, taux de chlore, etc.).

Q1. [**RANGE EVENTS BATCH** 1000] retourne une **collection de 1000 valeurs** de pression observées dans un secteur donné (exemple 8.2.1).

Q2. [**RANGE EVENTS** 1000] retourne les **1000 dernières valeurs** de pression observées dans un secteur donné (exemple 8.2.2).

Q3. [**RANGE** $10m$ **STEP** $1m$] retourne toutes les minutes, toutes les valeurs de pression observées dans un secteur donné sur les 10 dernières minutes de l'estampille temporelle **implicite** (exemple 8.2.3).

Q4. [**EXT RANGE** $10m$ **STEP** $1m$] retourne toutes les minutes, toutes les valeurs de pression observées dans un secteur donné sur les 10 dernières minutes de l'estampille temporelle **explicite** (exemple 8.2.4).

Q5. [**RANGE** $10m$ **EVENT BATCH** 1000] retourne les 1000 premières valeurs de pression collectées dans un secteur donné **ou** celles observées au bout de 10 minutes et qui n'auraient pas atteint la valeur 1000. Cette dernière fenêtre est dite hybride et nous permet de prévenir les cas de pic car la première des conditions satisfaites (1000 events ou 10 minutes) est aussitôt traitée (exemple 8.2.5).

L'opérateur **REFRESH** est optionnel pour les fenêtres 3 et 4 et obligatoire pour les fenêtres 1, 2 et 5 où il représente la fréquence de rafraichissement de l'exécution de la requête.

Exemple 8.2.1. RANGE EVENTS BATCH

```
QUERY pressuresInSectorLouve AS
        {
PREFIX    waves: <http://waves-rsp.org/resource#>
PREFIX    ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX    rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT   ?pressureSens ?value
FROM STREAM    <http://waves-rsp.org/obs> [RANGE EVENTS BATCH 1000]
FROM   <http://waves-rsp.org/wavesrepo>
WHERE
          {
                    ?sector rdfs:label "Louveciennes" .
                    ?pressureSens ssn:onPlatform ?sector .
                    ?event ssn:isProducedBy ?pressureSens ;
                         ssn:hasValue ?observation .
                    ?observation a "pressure" ;
                         waves:numericValue ?value .
          }
        }
```

Exemple 8.2.2. RANGE EVENTS

```
QUERY pressuresInSectorLouve AS
        {
PREFIX    waves: <http://waves-rsp.org/resource#>
PREFIX    ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX    rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT   ?pressureSens ?value
FROM STREAM    <http://waves-rsp.org/obs> [RANGE EVENTS 1000]
FROM   <http://waves-rsp.org/wavesrepo>
WHERE
          {
                    ?sector rdfs:label "Louveciennes" .
                    ?pressureSens ssn:onPlatform ?sector .
                    ?event ssn:isProducedBy ?pressureSens ;
                         ssn:hasValue ?observation .
                    ?observation a "pressure" ;
                         waves:numericValue ?value .
          }
        }
```

Exemple 8.2.3. RANGE STEP

```
QUERY pressuresInSectorLouve AS
        {
PREFIX    waves: <http://waves-rsp.org/resource#>
```

```
PREFIX     ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX     rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT    ?pressureSens ?value
FROM STREAM     <http://waves-rsp.org/obs> [RANGE 10m STEP 1m]
FROM    <http://waves-rsp.org/wavesrepo>
WHERE
          {
                    ?sector rdfs:label "Louveciennes" .
                    ?pressureSens ssn:onPlatform ?sector .
                    ?event ssn:isProducedBy ?pressureSens ;
                         ssn:hasValue ?observation .
                    ?observation a "pressure" ;
                         waves:numericValue ?value .
          }
        }
```

Exemple 8.2.4. Ext RANGE STEP

```
QUERY pressuresInSectorLouve AS
        {
PREFIX     waves: <http://waves-rsp.org/resource#>
PREFIX     ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX     rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT    ?pressureSens ?value
FROM STREAM     <http://waves-rsp.org/obs> [Ext RANGE 10m STEP 1m]
FROM    <http://waves-rsp.org/wavesrepo>
WHERE
          {
                    ?sector rdfs:label "Louveciennes" .
                    ?pressureSens ssn:onPlatform ?sector .
                    ?event ssn:isProducedBy ?pressureSens ;
                         ssn:hasValue ?observation .
                    ?observation a "pressure" ;
                         waves:numericValue ?value .
          }
        }
```

Exemple 8.2.5. RANGE `value` **EVENTS BATCH**

```
QUERY pressuresInSectorLouve AS
        {
PREFIX     waves: <http://waves-rsp.org/resource#>
PREFIX     ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX     rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT    ?pressureSens ?value
FROM STREAM     <http://waves-rsp.org/obs> [RANGE 10m EVENTS BATCH 1000]
FROM    <http://waves-rsp.org/wavesrepo>
WHERE
          {
                    ?sector rdfs:label "Louveciennes" .
                    ?pressureSens ssn:onPlatform ?sector .
                    ?event ssn:isProducedBy ?pressureSens ;
```

```
                    ssn:hasValue ?observation .
                ?observation a "pressure";
                    waves:numericValue ?value .
        }
    }
```

8.3 Architecture de DRSS

L'architecture proposée pour DRSS est illustrée par la figure 8.1 et comporte une liste de requêtes DRSS continues (composant a), une ou plusieurs sources de flux de graphes RDF et une ou plusieurs sources de données statiques stockées dans des repository locaux ou distants (composants (d) et (c)). Ces sources de requêtes continues et de données sont associées à deux principaux composants. Le composant (b) constitue la partie **offline** tandis que le composant (e) représente la partie **temps réel** du système.

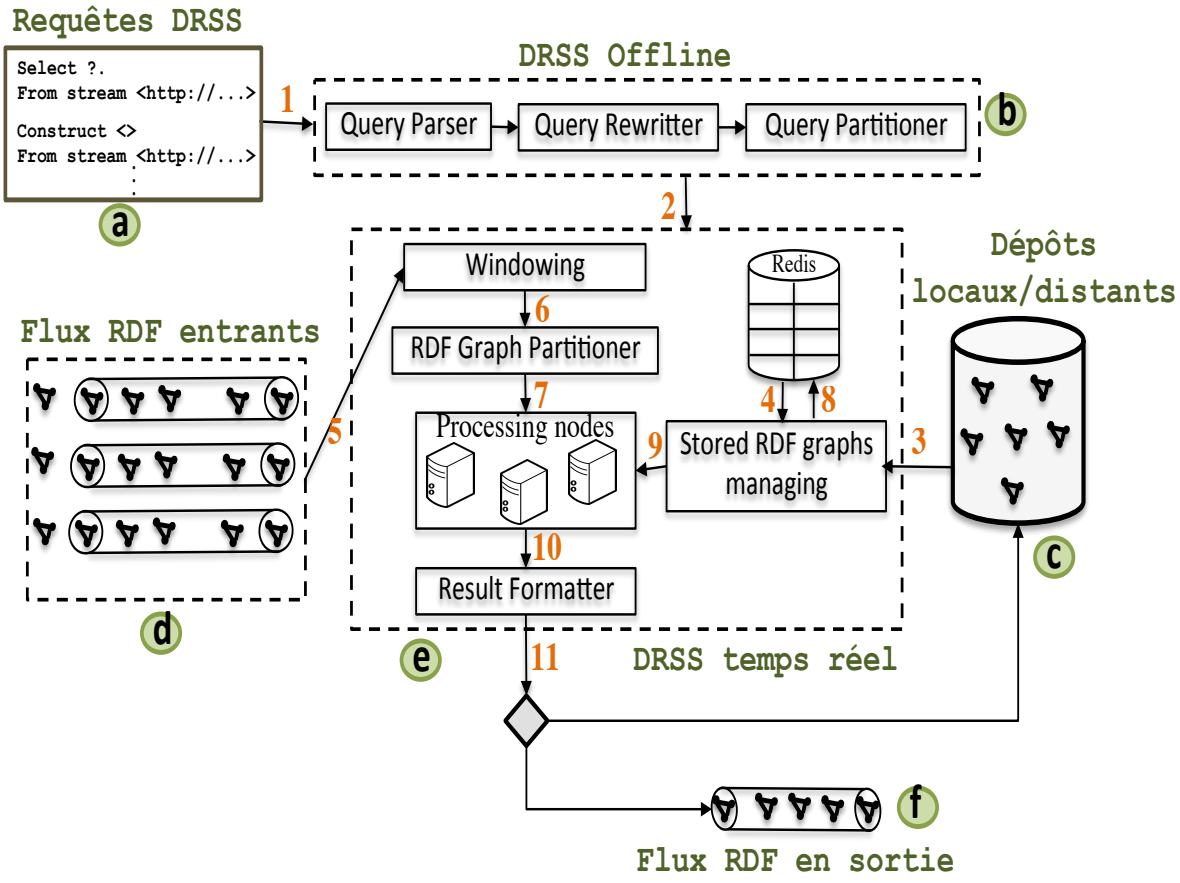


FIGURE 8.1 – Architecture de DRSS.

8.3.1 Composant "Offline"

La partie **offline** est essentiellement orientée requête et est composée de trois modules.

8.3.1.1 Module "Query Parser"

La partie dynamique contient tous les identifiants des flux et des paramètres de fenêtrage (type, valeur, etc.). Ce module prend en entrée une liste de requêtes continues concurrentes, dépendantes ou non et n'a pas de limite sur le nombre de requêtes. Le **Query Parser** analyse chaque requête et vérifie si elle est conforme à la syntaxe de requête de DRSS présentée dans la section 8.2. Il extrait ensuite les parties statiques et dynamiques de la requête.

— La partie statique comporte l'identifiant de la requête DRSS et la requête SPARQL contenue dans la requête DRSS. La requête SPARQL est parsée suivant le type de retour, les opérateurs d'agrégation et de groupement, les opérateurs de filtre et le BGP (les patterns de requête contenus dans la clause WHERE). Si la requête est imbriquée, alors la ou les requêtes les plus internes sont considérées comme prioritaires par rapport aux requêtes les plus externes (enchaînement de requêtes). Les patterns de requête extraits sont transmis au module `Query Rewritter` puis au module `Query Partitioner`.

— La partie dynamique est composée des identifiants de la requête DRSS et de ceux de chaque source de flux interrogée dans la requête, des paramètres de fenêtrage ((type de fenêtre (section 8.2), valeur(s), etc.)) de chaque flux et de la fréquence de rafraîchissement de l'exécution de la requête continue.

8.3.1.2 Module "Query Rewriter"

Le module `Query Rewriter` utilise l'algorithme 11 pour réécrire toutes les requêtes SPARQL partageant une ou des sous-structures (pattern(s)). Cet algorithme regroupe tous les patterns communs de différentes requêtes SPARQL avant de les réécrire en une seule requête. Ses avantages sont l'évaluation unique des parties communes à plusieurs requêtes et la non-duplication d'un sous-graphe de requêtes observé dans plusieurs requêtes DRSS. Toutes les requêtes réécrites sont ensuite transmises au module `Query Partitioner`.

8.3.1.3 Module "Query Partitioner"

Le module `Query Partitioner` constitue le cœur de notre contribution sur le système DRSS. En utilisant l'algorithme 12, le module partitionne chaque requête SPARQL en se basant sur l'opération de jointure. Nous énumérons tous les nœuds de jointure présents dans le pattern de graphe de chaque requête avant de définir les notions de `light pattern` (pattern de requête comportant un seul nœud de jointure) et de `full pattern` (pattern de requête comportant exactement deux nœuds de jointure). Les `light pattern` sont immédiatement affectés aux nœuds de jointure correspondants (et donc aux partitions correspondantes) et les `full pattern` sont dupliqués vers leurs différentes partitions. Cette méthode permet de traiter parallèlement des parties du graphe sans pour autant nécessiter une communication entre les nœuds de traitement. Nous générons le plan d'exécution le plus optimal de la partition de requête en utilisant l'algèbre syntaxique et sémantique de SPARQL [57].

8.3.2 Composant temps réel

La partie `temps réel` démarre avec le module `RDF Graph Partitioner`.

Algorithme 11 Module **Query Rewritter**

Require: Requêtes SPARQL $Q_{init} = \{Q_{i1}, Q_{i2}, ..., Q_{in}\}$

$S_p \Leftarrow \emptyset$ ensemble de chemins de patterns de graphes de requêtes
while $Q \in Q_{init}$ **do**
  if Q.head is "select" **then**
    $V \Leftarrow$ ensemble de variables
    $G \Leftarrow$ ensemble d'expressions "group by"
    $O \Leftarrow$ ensemble d'expressions "order by"
  end if
  $Tp \Leftarrow$ patterns de triplets
  $Sp \Leftarrow$ chemin de sous-patterns de triplets
  Construire le chemin du pattern de graphe de Q
  $P_Q = \{V, G, O, Tp, Sp\}$
  S_p.add(P_Q)
end while
Initialiser l'ensemble des indexeurs de chemin des patterns de requêtes
$\mathscr{I}_Q = \emptyset$ for each Q
$i \Leftarrow 0$
$j \Leftarrow 0$
while $i < S_p$.size() **do**
  $j \Leftarrow i + 1$
  while $j < S_p$.size() **do**
    if P_{Q_i}.sharePatWith(P_{Q_j}) **then**
      $\mathscr{I}_{Q_i}$.add(Id_{Q_j})
    end if
  end while
end while
while $p \in P_Q$ **do**
  Trouver tous les chemins de patterns de graphe p partageant un index de chemin pour les combiner
  $Q_{path} \Leftarrow$ liste de tous les patterns partagés
  $Q_{opt} \Leftarrow$ liste de tous les patterns non partagés (à mettre dans des clauses **Optional**)
  Q_{Rewrit}.add(Q_{path}, Q_{opt})
end while
return Ensemble de requêtes SPARQL réécrites $Q_{Rewrit} = \{Q_1, Q_2, ..., Q_k\}$

Algorithme 12 Module **Query Partitioner**

Require: une requête SPARQL Q, degré de partitionnement N

$GraphPatt \Leftarrow$ tous les patterns de graphes (BGP) de Q
$joinNode \Leftarrow$ {nœud $\in$ BGP, $Degr$(nœud) $> 1 \wedge$ nœud $\neq$ prédicat}
$lightPattern \Leftarrow$ lien sémantique entre 2 nœuds où l'un d'entre eux n'est pas un $joinNode$
$fullPattern \Leftarrow$ lien sémantique entre 2 $joinNode$
$Nbr \Leftarrow joinCount$(BGP)
$DuplicatePattern \Leftarrow \emptyset$
$i \Leftarrow 0$
while $i < Nbr$ **do**
  créer une partition de requête vide $Q_i \Leftarrow \emptyset$
end while
while $pattern \in$ BGP **do**
  if $pattern$ is $lightPattern$ **then**
    assigner $pattern$ à la partition correspondante Q_i du $joinNode$ courant
  else
    assigner $pattern$ aux 2 partitions correspondantes Q_i et Q_j, $(i \neq j)$ des 2 $joinNode$ liés
  end if
  while Partition Q_i **do**
    if currentDegree(Q_i) $\geq$ N **then**
      Q_p.add(Q_i)
    end if
  end while
end while
return Ensemble de partitions de requêtes SPARQL $Q_p = \{Q_1, Q_2, ..., Q_n\}$

8.3.2.1 Module "RDF Graph Partitioner"

Le module RDF Graph Partitioner utilise l'algorithme 13 pour partitionner tout graphe RDF entrant. Le procédé de partitionnement reste basé sur les informations recueillies à partir de l'étape de partitionnement de la requête. Avant de déployer chaque partition de graphe RDF dans son nœud de calcul prévu (d → e), nous la transmettons au module de fenêtrage (en utilisant les paramètres de fenêtrage contenus dans sa requête d'origine). Chaque partition de graphe RDF est redirigée vers le nœud destiné à la traiter dès qu'elle quitte le module de fenêtrage. Les graphes RDF statiques sont préalablement importés (durant le processus `offline`) et stockés dans Redis [1] (c → e). Les requêtes figurant dans les différentes partitions sont formées avec l'entête CONSTRUCT de la syntaxe SPARQL. Les résultats intermédiaires sont pris en charge par le `Result Formatter` et, en fonction de l'en-tête de la requête DRSS, seront destinés à un triplestore local ou distant ou bien écrits sur un flux sortant (topic kafka), c'est-à-dire e → c ou e → f.

Algorithme 13 Module **RDF Graph Partitioner**

Require: graphe RDF G, partition(s) de requête(s) SPARQL Q_p

1: M_{QP} = Map<Q_p, List<$Patterns$>>
2: $G_{Q_p} \Leftarrow \emptyset$ le graphe associé à une partition de requête donnée
3: **while** triple ∈ G **do**
4: **while** Q ∈ Q_p **do**
5: **if** $qryGraph(M_{QP}, Q)$ contient 1 triplet **then**
6: $G_{Q_p} \Leftarrow addPat(G_Q, Pat)$
7: updateGraphSet(G_{Q_p}, queryId(Q))
8: **end if**
9: **end while**
10: **end while**
11: **return** Ensemble de partitions de graphes mappées à l'Id de la partition de requête G_p = $\{(G_1, Q_p id), (G_2, Q_p id), ..., (G_n, Q_p id)\}$

8.3.2.2 Module "Windowing"

Le module de fenêtrage extrait du flux les derniers graphes du flux de graphe "ouvert". L'extraction est **physique** (un nombre donné de graphes (fenêtres **1** et **2**)) ou **logique** (un nombre variable de triplets qui se produisent pendant un intervalle de temps donné (fenêtres **3** et **4**)) ou **hybride** (combinaison des fenêtres physique et logique(fenêtre **5**)). Notre module **Windowing** attribue à chaque identifiant de flux (correspondant à une seule et unique source de flux) une instance de fenêtre logique, physique ou hybride (fenêtre présente dans la clause **FROM STREAM**). Les instances de fenêtre sont traitées parallèlement et leurs résultats sont générés séparément et transmis au module de partitionnement de graphes.

8.3.3 Illustration de DRSS

Le module de partitionnement de requête représente la partie centrale du composant offline de l'architecture de DRSS. Dans cette section, nous illustrons par un exemple le modèle de partitionnement d'une requête dans DRSS.

La requête de l'exemple 8.3.1 interroge un flux de données issues d'un réseau de transport et de distribution d'eau potable. Comme résultat, la requête retourne toutes les minutes, pour chaque identifiant de capteur, la moyenne des valeurs de pression mesurées dans le secteur "@@@Louve###" sur les 10 dernières minutes ou sur les 1000 derniers

1. https://redis.io/

graphes RDF entrants.

Exemple 8.3.1. Requête DRSS.

```
QUERY avgPressure AS
  {
PREFIX    waves: <http://waves-rsp.org/resource#>
PREFIX    ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX    rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT  ?sensorID  (AVG( ?value) as ?avgPressure)
FROM STREAM <http ://waves-rsp.org/obs> [RANGE 10m EVENTS BATCH 1000]
WHERE
      {
            ?eventID inZone ?zoneID .
            ?eventID isProducedBy ?sensorID .
            ?eventID hasValue ?result.
            ?eventID hasObservation ?obsID .
            ?result type ?obsID.
            ?result numericValue ?value .
            FILTER ( ?value > 0)
            ?obsID type observationValue .
            ?obsID hasName "pressure" .
            ?obsID unit "mH20" .
            ?sensorID inSector ?sectID.
            ?sectID hasName "@@@Louve###" .
      }
GROUPBY ?sensorID
HAVING ( ?avgPressure >= 30)
ORDER BY avgPressure
  }
REFRESH 1m
```

Une fois la requête validée par le module **Query Parser** (les parties statiques et dynamiques extraites) et les opérations de réécriture de requêtes ayant des sous-structures communes (algorithme 11), le module **Query Partitioner** va procéder comme suit :

1. Nous construisons tout d'abord le graphe de requête correspondant aux BGP de la requête DRSS dans l'exemple 8.3.1. La figure 8.2 présente ce graphe dans lequel nous mettons en relief les nœuds de jointure (nœuds **2**, **3**, **4**, **5** et **12**) présents dans le BGP de la requête. Ces nœuds de jointure vont constituer le centre des partitions du graphe de requête qui seront construites.
2. La deuxième étape consiste à appliquer l'algorithme 12 de partitionnement de requête où nous détectons d'abord, pour chaque nœud de jointure, ses **full** et **light pattern(s)**. Chaque **light pattern** est directement affecté à son nœud de jointure. Ainsi, les **light patterns a**, **f**, **h** et [**i**, **j**, **k**, **l**] sont respectivement affectés au nœuds de jointure **2**, **12**, **4** et **5**.

   Les **light patterns** ne sont jamais dupliqués entre les nœuds de traitements.
3. Dans la troisième étape, nous maintenons les liens sortant des nœuds de jointure source c'est-à-dire, un **full pattern** appartient à son nœud de jointure source est dupliqué pour le nœud de jointure cible. Notons que les notions de nœud source

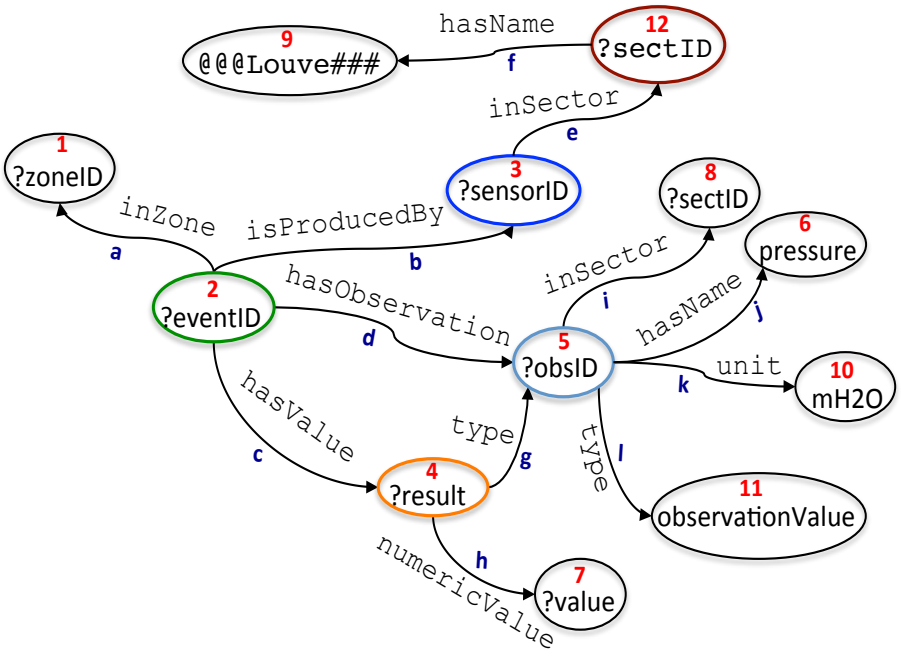


FIGURE 8.2 – Exemple de graphe de la requête de l'exemple 8.3.1.

et cible sont relatives au sens d'orientation de la flèche. Ainsi, pour compléter la construction des différentes partitions, nous procédons comme suit :

- les **full patterns b**, **c** et **d** sont affectés au nœud de jointure **2** et respectivement dupliqués pour les partitions des nœuds de jointure **3**, **4**, et **5** ;
- le full pattern **g** est affilié au nœud de jointure **4** et dupliqué pour la partition du nœud de jointure **5** ;
- enfin le full pattern **e** est affecté au nœud **3** et dupliqué pour la partition qui porte le nœud de jointure **12**.

Cette procédure de partitionnement et de duplication des patterns de la requête est illustrée par la figure 8.3 qui génère cinq (5) partitions de la requête montrées dans les figures 8.4a, 8.4b, 8.5a, 8.5b et 8.6.

4. La dernière étape construit la requête finale de chaque partition avec une clause CONSTRUCT. Pour éviter des blocages (dus aux retards dans la synchronisation) dans le traitement des différentes partitions de requête, nous encapsulons chaque pattern de requête de chaque partition par une clause OPTIONAL qui est l'équivalent de la jointure de gauche (ou de droite) en SQL.

Les résultats (graphes RDF) intermédiaires sont pris en charge par le **formatter** et, en fonction de l'entête de la requête DRSS de départ, seront destinés à un dépôt local ou distant ou écrits sur un flux sortant.
Ces partitions de requête sont exécutées parallèlement sans nécessiter un nœud central ou synchroniseur. De plus, les données échangées entre les nœuds de traitement sont quasi inexistantes.
Notons que le système DRSS initialise une configuration contenant initialement toutes les initialisations de la topologie, des identifiants des flux sortants ou des dépôt locaux ou distants. Cette configuration est également mise à jour avec les informations extraites après les opérations de partitionnement des requêtes DRSS.

Au cours de la transmission entre les composants (b → e), la partie `offline` partage toutes les informations telles que les identifiants des partitions de requête, des requêtes

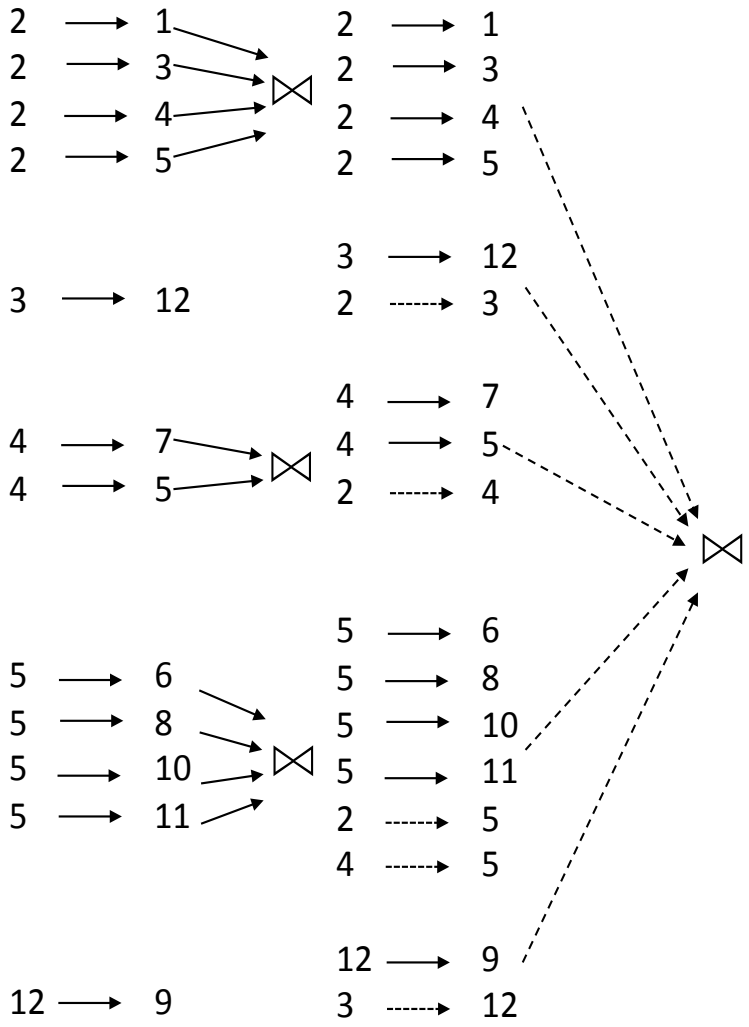
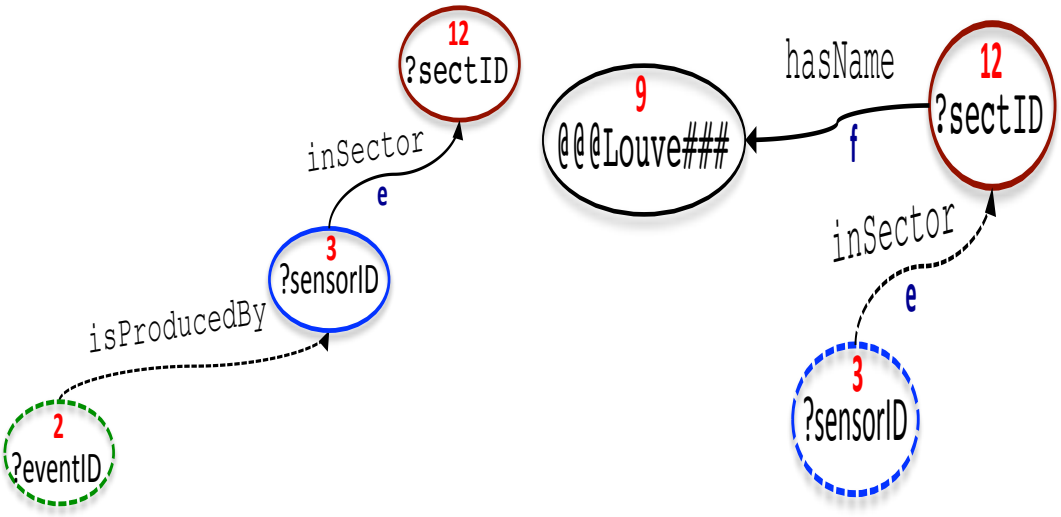


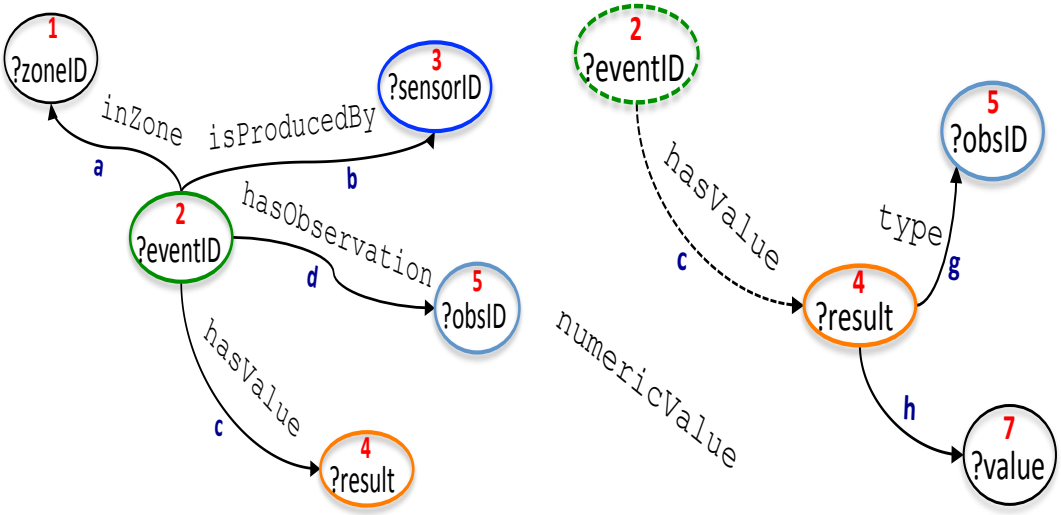
FIGURE 8.3 – Procédure de partitionnement et de duplication de patterns.

d'origine et des paramètres de fenêtrage et de leur type. Nous avons besoin de tous ces éléments pour gérer correctement le partitionnement en temps réel des graphes RDF en entrée.



(A) Partition de requête 1. (B) Partition de requête 2.

FIGURE 8.4 – Partitions de requête 1 et 2.



(A) Partition de requête 3. (B) Partition de requête 4.

FIGURE 8.5 – Partitions de requête 3 et 4.

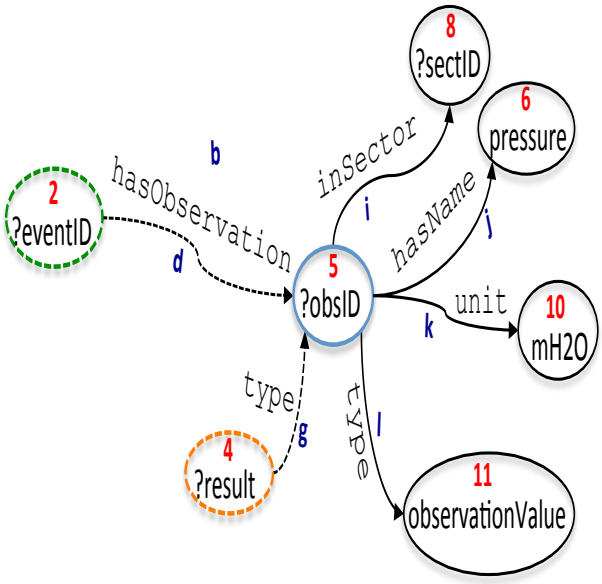


FIGURE 8.6 – Partition de requête 5.

8.4 Implémentation et évaluation

Dans cette section, nous allons évaluer notre stratégie de parallélisation et de distribution à travers un ensemble de données synthétiques RDF générées pour répondre au besoin de complexité (graphes avec des liaisons multiples). Cette complexité n'est pas disponible dans la plupart des jeux de données de la communauté RSP (LSBench, SRBench, CSRBench et CityBench) [2]. Notre jeu de données concerne la supervision en temps réel du réseau de transport et de distribution d'eau potable.

8.4.1 Choix technologique et implémentation

Nous avons implémenté notre système en utilisant Apache Zookeeper, Apache kafka, Apache Storm et Redis (voir section 7.2, chapitre 7). Un graphe RDF en entrée est partitionné et chaque partition est routée vers le même `bolt`. Pour définir comment les graphes RDF doivent être partitionnés parmi les bolts, nous avons utilisé deux des huit paramètres de regroupement de flux intégrés dans Storm que sont le `fields grouping` et le `all grouping`. Le `fields grouping` est utilisé pour regrouper toutes les partitions de graphes RDF partageant le même Id de partition de requête dans la même tâche de traitement (nœud de traitement). Le `all grouping` est utilisé pour la réplication d'un `full pattern` pour une jointure locale et parallèle sans communication entre les nœuds.

Nous avons entièrement implémenté DRSS en utilisant Java et toutes les expériences sont effectuées en utilisant un cluster de 9 machines tournant sur Linux. Chaque machine a un processeur à 4 cœurs de 2,4 GHz et 4 Go de RAM. Parmi ces 9 machines, il y en a une que nous nommons `faux-nœud`. Le `faux-nœud` n'est pas un nœud central ou master, mais est utilisé pour l'exécution du planificateur de requêtes `offline` (la réécriture de requête, le partitionnement de requête et l'optimisation de requête avant de prendre le rôle de formatage des résultats (`Result formatter`)). Ce dernier reçoit en permanence des résultats de graphes RDF à partir des nœuds de traitement où seules les sous-requêtes `Construct` sont exécutées quelle que soit l'entête de la requête initiale (Select, Describe

2. http ://streamreasoning.org/resources

ou aussi Construct). Si une jointure entre graphes RDF stockés avec des flux RDF est requise, l'importation des données est effectuée pendant le processus `offline` et les données sont stockées dans Redis. Cette procédure évite le risque de blocage dans le réseau lors de l'interrogation de triplestores locaux ou distants. Le stockage en mémoire local nous permet de joindre localement des données RDF statiques à des flux de graphes RDF en quasi temps réel. Il prend également en compte un mécanisme de rafraîchissement des données stockées. En effet, nous considérons les données RDF statiques comme des flux de données avec une faible fréquence de rafraîchissement (par exemple toutes les 24 heures ou toutes les semaines, etc.).

8.4.2 Résultats d'évaluation

Nous avons considéré tout d'abord quatre requêtes DRSS **Q1**, **Q2**, **Q3** et **Q4**. Pour chaque requête, nous avons mesuré le temps d'exécution moyen compte-tenu du nombre de nœuds de calcul et de la taille de la fenêtre. Le temps nécessaire au formatage des résultats (c'est-à-dire, l'intégration des sous-résultats et l'écriture des résultats finaux sur le(s) topic(s) de sortie de kafka ou vers un stockage temporaire pour une future procédure de rafraîchissement des dépôts locaux ou distants) n'est pas considéré. Pour tester la scalabilité, nous avons fait varier la taille du cluster et avons exécuté les quatre requêtes avec respectivement 1, 2, 4, 6 et 8 nœuds de calcul. Nous avons normalisé le temps d'exécution des requêtes par rapport à ceux considérés dans notre exécution de base où nous avions traité toutes les requêtes sur un seul nœud.

Dans la figure 8.7, nous observons un temps d'exécution plus bas à mesure que le nombre de nœuds de calcul augmente. Cela est dû au traitement local d'une partition de requête. Il n'y a aucun résultat intermédiaire partagé entre les nœuds durant le processus et le nombre de nœuds de jointure dans la requête n'a pas une grande influence à partir de 2 machines. Cela est dû au fait que la jointure se fait en parallèle en dupliquant les patterns de jointure (`full pattern`).

Les expériences ci-dessus ont été effectuées avec une taille de fenêtre fixe. Pour tester la scalabilité en terme de taille de données à traiter dans une fenêtre, nous reprenons une évaluation similaire en modifiant la taille des données de la fenêtre. Les résultats de cette évaluation sont montrés par la figure 8.8 où nous notons une évolution du temps d'exécution lorsque nous augmentons la longueur de la fenêtre en matière du nombre de graphes RDF. Nous avons fixé le débit d'entrée à 300 graphes/seconde. Naturellement, à mesure que le nombre de graphes RDF à traiter dans une fenêtre augmente, le temps d'exécution de chaque requête augmente également. Le temps d'exécution observe une évolution presque linéaire confirmant la scalabilité de notre système (figure 8.7).

Nos dernières évaluations concernent des requêtes concurrentes (**Q1** à **Q5**) et la combinaison de plusieurs sources de flux dans une requête (**Q5**). Les requêtes données ne sont pas interdépendantes mais, nous configurons leurs sorties de telle manière qu'elles dépendent les unes des autres. Nous avons considéré tout d'abord un ensemble de requêtes concurrentes (c'est-à-dire des requêtes qui nécessitent des résultats fournis par une ou plusieurs requêtes dans la même topologie en cours d'exécution) et avons mesuré les débits de graphes RDF (graphes RDF/seconde) tout en fixant la taille de fenêtre à 100 000 graphes pour toutes les requêtes et en modifiant le nombre de nœuds de calcul.

Dans la figure 8.9, nous observons une croissance linéaire du débit lorsque nous augmentons le nombre de nœuds de calcul. Toutefois, le débit est négativement proportionnel au nombre de requêtes dont dépend une requête. Mais cette évolution peut toujours être compensée par des nœuds de calcul supplémentaires. Étant donné que **Q5** n'utilise pas de jointure externe à gauche et que la requête interne (requête dont dépend **Q5**) ne renvoie pas de résultat, la requête **Q5** ne renvoie aucun résultat pendant le temps de traitement. De même, nous pouvons faire la même observation avec la figure 8.10 pour la variation du nombre de flux sources joints dans la requête **Q5**.

Le calcul distribué de larges flux de graphes RDF conduit à la mise en place d'une technique de partitionnement efficace des graphes et des requêtes sur plusieurs nœuds de traitement tout en minimisant les communications entre eux. Contrairement aux systèmes distribués existants, notre système combine le traitement des données RDF dynamiques et statiques et plus des nœuds de calcul sont ajoutés, plus les performances attendues croissent. Ces expériences confirment la scalabilité de notre stratégie de parallélisation du traitement et de distribution des données.

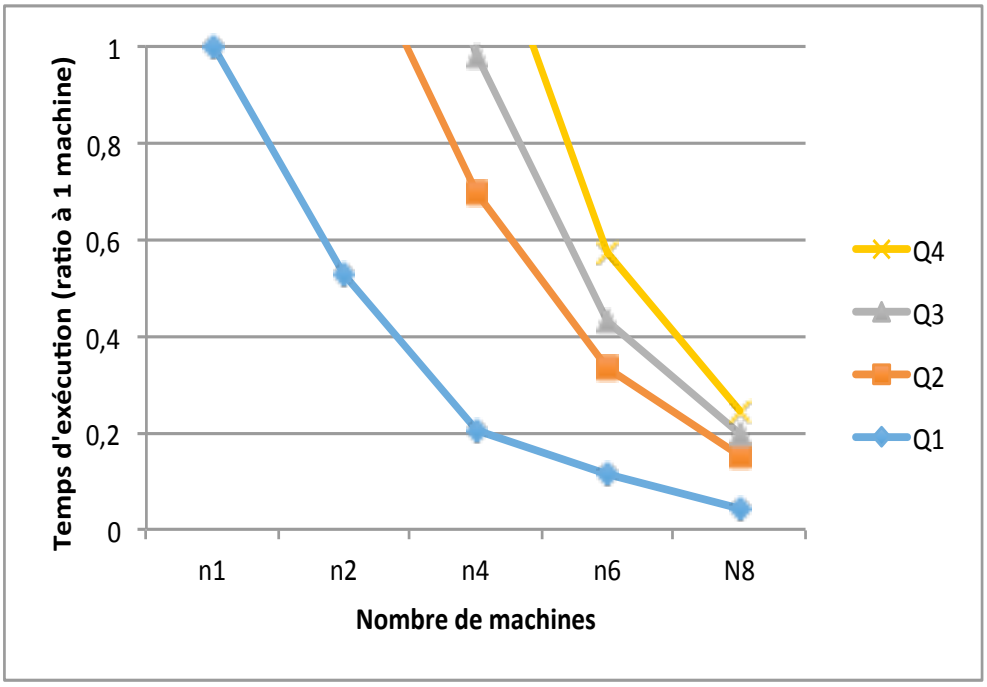


FIGURE 8.7 – Évaluation de la scalabilité par variation du nombre de machines.

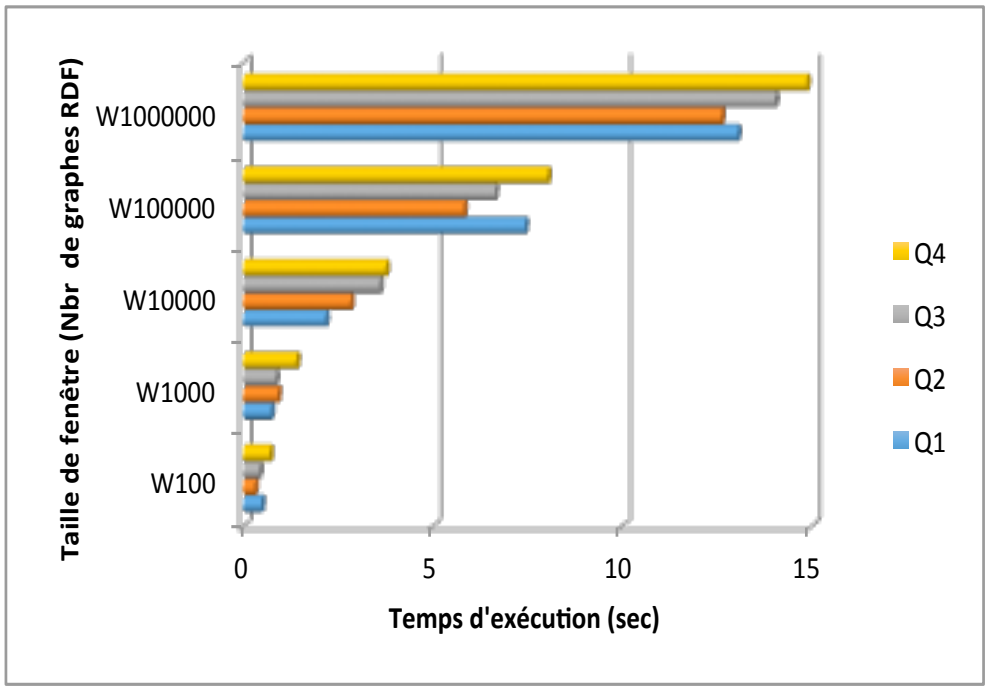


FIGURE 8.8 – Évaluation de la scalabilité par variation de la taille de la fenêtre.

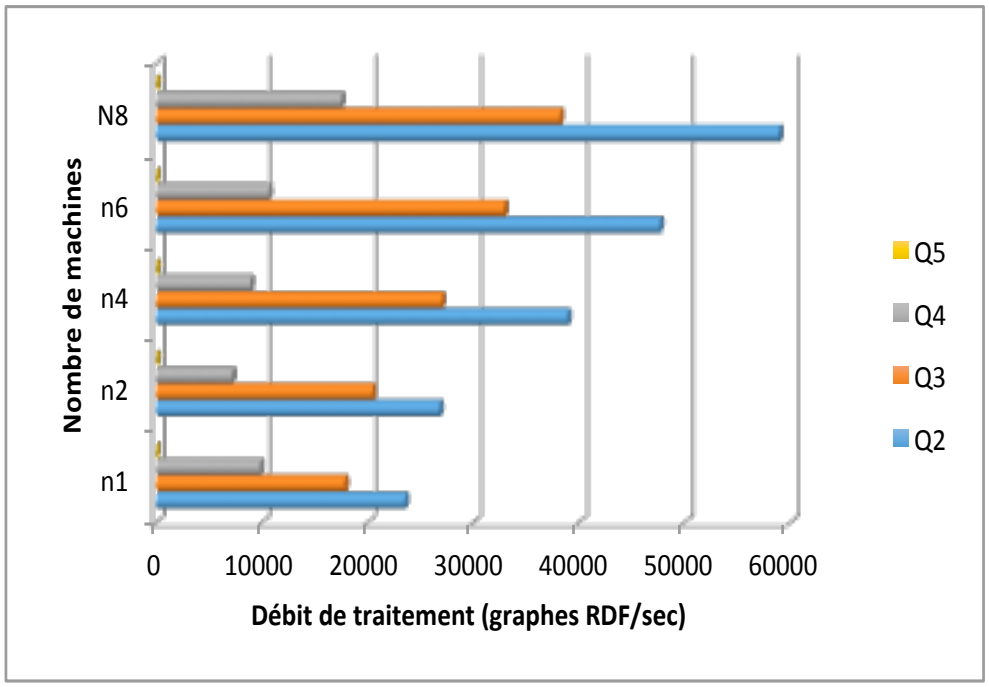


FIGURE 8.9 – Évaluation de requêtes concurrentes.

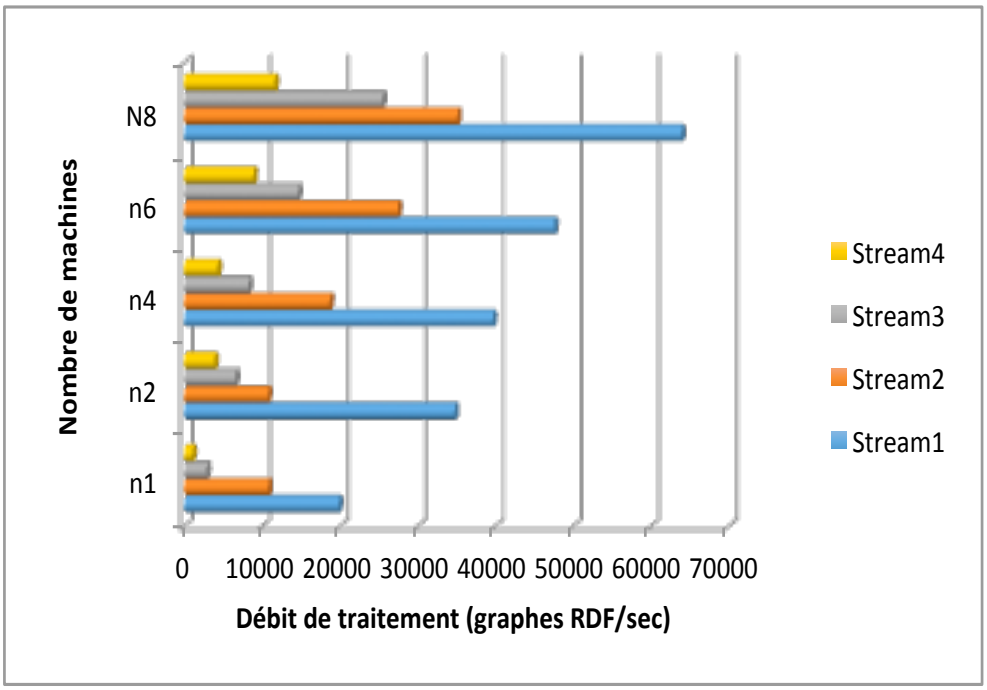


FIGURE 8.10 – Évaluation de flux multiples.

8.5 Conclusion

Dans ce chapitre, nous avons présenté DRSS, un système de traitement distribué de flux de graphes RDF. Nous avons proposé trois algorithmes pour la réécriture de requêtes (algorithme 11), le partitionnement de requêtes (algorithme 12) et le partitionnement de graphes RDF (algorithme 13). Les expérimentations montrent une excellente scalabilité en terme de temps d'exécution de requêtes concurrentes et de requêtes interrogeant plusieurs sources de flux de données RDF. Cependant, DRSS n'atteint pas les objectifs fixés dans le croisement des données RDF statiques et dynamiques. En effet, dans DRSS, notre mécanisme d'importation, de stockage et de jointure des données statiques reste identique à l'approche adoptée pour les flux de graphes RDF entrants. Cette option constitue une limite dans l'optimisation du traitement dans un contexte requérant des croisements répétitifs de données RDF statiques et dynamiques. Ainsi, dans le prochain chapitre, nous proposons une solution de jointure optimisée entre graphes RDF statiques et dynamiques.

Publication
[**44**]. Dia, Amadou Fall, Zakia Kazi-Aoul, Aliou Boly, and Elisabeth Métais. "DRSS : Distributed RDF SPARQL Streaming."

Chapitre 9

5ème contribution : approche de jointures optimisées entre graphes RDF statiques et dynamiques (JSS-RDF)

Sommaire

9.1 Introduction

Les données RDF stockées sont très importantes dans le contexte des flux de données RDF. Ces données concernent les ontologies de domaine associées à un domaine d'application des flux de données, des données RDF résumées et/ou historiées des données RDF à fréquence de rafraîchissement très lent (hebdomadaire, mensuel, semestriel, trimestriel, etc.). Dans ce dernier cas nous pouvons noter les données de localisation de capteurs déployés dans un réseau de transport et de distribution d'eau potable (latitude, longitude), ses caractéristiques (date d'installation, voltage, propriétés, type de mesures, etc.). Ces données stockées demeurent indispensables dans plusieurs domaines d'application tels que l'e-santé, la biologie, l'étude de la biodiversité, etc., où le croisement de données statiques et dynamique est requis.

Pour la gestion combinée de données RDF statiques et dynamiques, plusieurs systèmes RSP tels que C-SPARQL, CQELS et CQELS Cloud prennent en charge le croisement (jointure) entre données RDF statiques et dynamiques. Ces systèmes réalisent la jointure

au cours du traitement en important les données statiques depuis des fichiers ou des dépôts locaux ou distants. Nous avons abordé cette problématique dans le système **DRSS** (chapitre 8) où nous traitons les données statiques en les chargeant et les partitionnant depuis des fichiers durant la phase de traitement `offline` . Dans DRSS, l'un de nos défis principaux était de proposer une extension du langage et un mécanisme pouvant prendre en compte la combinaison de données statiques et dynamiques. Ainsi, sur ce point, DRSS souffre des mêmes limites que les systèmes existants. En effet, les systèmes RSP ont besoin de lire l'ensemble des données RDF stockées localement ou à distance tandis que les flux de données RDF arrivent en continu et requièrent un traitement quasi temps réel. Ainsi, les approches de jointure utilisées dans les systèmes RSP centralisés et distribués restent évidement non optimisées et non évolutives pour effectuer des opérations de jointure entre les deux types de données. Ceci est dû à la nature le plus souvent verbeuse des données RDF statiques qui, nous (systèmes RSP) parviennent à des tailles allant de l'ordre des gigaoctets (comme les endpoints DBpedia [1], GeoNames [2], Wikidata [3], etc.). Les temps de latence causés par des opérations de jointure non adaptées aux deux natures de données à combiner entraînent des latences qui affectent négativement les performances en matière de traitement continu. Ils provoquent également de multiples goulots d'étranglement dans le réseau de nœuds de traitement dans un environnement distribué. Cela rend également difficile le rafraîchissement ou la mise à jour (à fréquences lentes) des données statiques (résumé, historisation, etc.).

Dans ce chapitre, nous proposons un processus de jointure distribué rapide et adaptatif entre les graphes RDF statiques et dynamiques. Ce système est regroupé autour d'une nouvelle architecture appelée **JSS-RDF** (Join between Streams and Stored RDF graphs). Dans la partie partitionnement de requête, nous étendons les approches utilisées dans le système DRSS avec une nouvelle méthode de jointure qui accorde cette fois-ci les mêmes privilèges aux deux natures de données RDF. En effet, nous utilisons les filtres de Bloom pour réaliser la jointure en mémoire, de chaque triplet RDF entrant dans le(s) flux avec les données RDF statiques sans nécessiter l'interrogation à temps réel de dépôts locaux ou distants. Les graphes RDF stockés sont préalablement importés et stockés à l'aide des filtres de Bloom. Nous les combinons ensuite aux flux de graphes de manière continue en réduisant considérablement les résultats intermédiaires générés en mémoire et les sous-requêtes pré-calculées selon les variables de requêtes SPARQL sélectionnées entre ces deux types de graphes RDF. Notre approche mise sur le stockage en mémoire optimisé à l'aide des filtres de Bloom ainsi qu'à une opération de jointure clé-valeur rapide évaluée de l'ordre d'une seule interrogation ($\mathcal{O}(2)$).

Le reste de ce chapitre est structuré comme suit. Dans la section 9.2, nous fournissons les notions essentielles sur les filtres de Bloom et les travaux sur la jointure entre données RDF statiques et dynamiques. La section 9.3 présente l'architecture de notre système, y compris notre approche de stockage, d'indexation et de traitement des requêtes. La section 9.4 fournit les détails de l'évaluation du système et les performances obtenues. Enfin, la section 9.5 conclut ce chapitre.

1. https ://wiki.dbpedia.org/
2. https ://www.geonames.org/
3. https ://www.wikidata.org/wiki/Wikidata :Main_Page

9.2 Concepts et approches existantes

Dans cette section, nous présentons la structure de données des filtres de Bloom avant de passer en revue les travaux connexes sur la jointure entre flux RDF et données RDF statiques.

9.2.1 Filtres de Bloom (FB)

Un filtre de Bloom (FB) [139] est une structure de données probabiliste qui permet de tester si un élément est présent dans un ensemble. Le FB est adapté à la plupart des applications impliquant des listes et des ensembles nécessitant une grosse mémoire. Ils utilisent le vecteur de bits comme structure de données interne qui permet de représenter de grandes quantités d'éléments dans une mémoire plus petite. Sa représentation probabiliste compacte peut entraîner des faux positifs, c'est-à-dire déclarer qu'un élément y fait partie d'un ensemble S alors qu'il ne l'était pas (c'est-à-dire, $y \notin$ S alors qu'il rapporte $y \in$ S). Il n'y a pas de faux négatifs dans le test d'appartenance, c'est-à-dire déclarer qu'un élément inséré est absent de l'ensemble ($y \in$ S alors qu'il rapporte $y \notin$ S). Cependant, le test d'appartenance du FB pour un élément réellement inséré dans le filtre passera toujours.

Mathématiquement, un FB représente un ensemble S = $\{y_1, y_2, ..., y_n,\}$ de n éléments dans une liste F de m bits (m est la taille du filtre), tous initialement mis à zéro. Un FB utilise une constante k de fonctions de hachage indépendantes $h_1, h_2, ..., h_k$. Chaque fonction de hachage doit faire correspondre les éléments d'un ensemble S = $\{y_1, y_2, ..., y_n,\}$ à un nombre aléatoire uniforme dans l'intervalle $1, ..., m$ comme indices dans le filtre F. Cela signifie que pour chaque élément $y \in$ S, les bits $h_i(y)$ sont mis à 1 pour $1 \leq i \leq k$. La phase d'insertion d'un élément y dans F suit ce mécanisme en plaçant les bits sous les indices $h_i(y)$ en sortie à 1. La figure 9.1 montre un exemple d'insertion de 3 éléments x, y et z de S dans F avec la taille $m = 30$ bits et $k = 3$ fonctions de hachage. Tous les bits ont été initialement définis à 0.

$h_1(x), h_2(x), h_3(x) \rightarrow 001000000100000000000001000000$
$h_1(y), h_2(y), h_3(y) \rightarrow 000001000000000100000000000100$
$h_1(z), h_2(z), h_3(z) \rightarrow 000000010000100000010000000000$

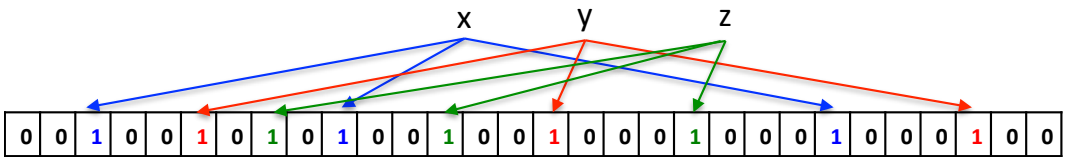


FIGURE 9.1 – Exemple d'insertion dans un filtre de Bloom.

La tâche inverse consistant en un test d'appartenance d'un élément y à un ensemble S est effectuée en vérifiant si tous les bits $h_i(y)$ sont définis. Soulignons que l'élément y est haché k fois et que les sorties des fonctions de hachage sont utilisées pour générer des indices sur le filtre F. Si un ou plusieurs bits $h_i(y)$ ne sont pas définis, alors l'élément est définitivement considéré comme n'étant pas dans l'ensemble S. Par exemple, interroger le FB dans la figure 9.1 pour les éléments y et t, les trois bits pour $h_i(y)$ seront positionnés alors qu'au moins un des $h_i(t)$ ne le sera pas. Ainsi, (y) est supposé être présent et (t) est garanti de ne pas faire partie de l'ensemble S. Le FB construit derrière un ensemble S de taille n permettant de faire des requêtes d'adhésion rapides en $\mathcal{O}(1)$ au lieu de $\mathcal{O}(n)$, ce qui implique un accès rapide, un stockage optimisé et un gain d'espace.

9.2.2 Approches existantes

Les données RDF statiques sont stockées dans des dépôts RDF locaux ou distants et interrogées via des requêtes SPARQL. Les langages de requête des systèmes RSP non distribués et distribués existants prennent en charge les requêtes continues combinant des flux RDF et des graphes RDF stockés. Ils étendent tous SPARQL, puis incluent la clause SERVICE de SPARQL qui permet un accès à distance des graphes RDF stockés. Pendant l'évaluation continue, leur mécanisme de jointure utilisé entre les graphes RDF stockés et les flux RDF, adopte l'approche de fédération SPARQL, c'est-à-dire que les données distantes sont fréquemment appelées (sur chaque évaluation de requête) avant de les joindre pour déplacer les données RDF dans une ou plusieurs fenêtres. Cette approche expose les systèmes à quatre limites majeures.

1. **Disponibilité**. Étant donné que les données distantes sont invoquées à chaque évaluation de la requête, une simple défaillance réseau peut considérablement augmenter le temps de traitement, allant jusqu'à perdre les données nécessaires pour satisfaire une requête.
2. **Charge de données à traiter élevée**. La taille élevée des données RDF extraites à partir d'un grand nombre de dépôts distants et leur verbosité peuvent surcharger le système et par conséquent ralentir le traitement et augmenter les temps de réponse.
3. **Performance (limites ARQ).** Des systèmes comme C-SPARQL délèguent la partie statique au moteur ARQ de SPARQL. ARQ fédère, de manière non optimisée, les requêtes SPARQL et plusieurs de ses extensions sont proposées pour optimiser l'évaluation de la requête avec la clause SERVICE [4]. Cette clause a été introduite avec la version 1.1 de SPARQL. Elle fédère les sous-requêtes sur différents dépôts. En utilisant ce mot-clé, les utilisateurs peuvent fédérer leurs propres requêtes sur l'interface des dépôts et est pratique au premier abord. Cependant, elle peut devenir problématique comme expliqué dans [140].
4. **Rafraîchissement des données**. De nombreux domaines d'application requièrent des données stockées mises à jour. Les systèmes RSP n'assurent pas le rafraîchissement des données stockées avant et après l'évaluation de la requête en fonction d'une fréquence donnée. Par conséquent, il n'y a pas de possibilité pour d'autres traitements et analyses tels que l'agrégation ou le résumé de données stockées.

Dans [141], une approche récente propose une optimisation en matérialisant des données distantes dans des vues locales afin de limiter le nombre de clauses SERVICE. Cette méthode initialise le système en extrayant les résultats du premier appel de SERVICE. Cependant, la méthode peut ne pas garantir un temps de réponse rapide en raison du rafraîchissement des données consultées. Une approche plus récente est présentée dans [142]. Les auteurs proposent un système proactif qui permet un traitement rapide des requêtes en invoquant une fois les données distantes et en les actualisant avant l'évaluation de la requête. Le système proposé délègue le traitement de flux à C-SPARQL et un intégrateur de résultat combine les résultats obtenus à partir de l'évaluation C-SPARQL et de la sous-requête SERVICE sur un processus de réplication. Les auteurs présentent enfin un temps d'exécution moyen de traitement d'une requête donnée avec variation du nombre de clauses SERVICE. Leur évaluation démontre un gain de 5 secondes par rapport à la méthode proposée dans [141]. Plus précisément, l'évaluation de la requête a pris 5 secondes, 1 seconde, 6 secondes et 11 secondes pour respectivement 2 `SERVICE`, 4`SERVICE`, 8 `SERVICE` et 16 `SERVICE`.

4. https ://www.w3.org/TR/sparql11-service-description/

De plus, tous les systèmes proposés utilisent la correspondance de sous-graphe pour les requêtes SPARQL stream-stream, stream-static et static-static. Cependant, comme mentionné dans [38], la correspondance de sous-graphes devient très coûteuse (nombre élevé de résultats de requête intermédiaires) avec le nombre d'opérations de jointure.

9.3 Jointure entre graphes RDF statiques et dynamiques

Dans cette section, nous présentons une architecture distribuée qui repose sur les filtres de Bloom et qui répond au besoin de réaliser des jointures entre graphes RDF statiques et des flux RDF.

9.3.1 Architecture du système

Le système DRSS que nous avons présenté dans le chapitre précédent (chapitre 8) a apporté une première solution dans laquelle nous avons levé certains des verrous des systèmes RSP centralisés et distribués existants. Plus précisément, DRSS a apporté des solutions aux principaux limites des systèmes RSP distribués que sont :

— les opérateurs et mécanismes de fenêtrage plus adaptés et optimisés ;

— l'optimisation de la distribution des données afin de limiter les résultats intermédiaires échangés entre les nœuds de traitement ;

— la prise en charge de requêtes concurrentes et dépendantes en limitant au mieux les baisses de performance qui pourraient se produire (ce qui est le cas avec les systèmes RSP centralisés et distribués) ;

— enfin, un mécanisme de mise à jour de données statiques.

Dans DRSS nous avons proposé un algorithme efficace de partitionnement des requêtes dont dépend celui du partitionnement des graphes RDF. Cependant, l'approche de jointure dans les partitions de requête engendre dans DRSS, à l'image des systèmes RSPs existants, des baisses de performance non négligeables. Afin de traiter cette limite, nous avons étendu l'architecture de DRSS et en proposant un mécanisme de jointure optimisée entre flux de données statiques et dynamiques.

JSS-RDF désigne l'architecture distribuée du système que nous proposons et que nous illustrons dans la figure 9.2. Le système traite des requêtes SPARQL continues suivant la syntaxe de la requête de DRSS présentée dans [44]. Cette syntaxe est conforme aux fonctionnalités de requêtes RSP [143] telles que l'identifiant de flux, les opérateurs S2R (c'est-à-dire fenêtrage) et la grammaire du langage de requête SPARQL [57] comme SELECT, CONSTRUCT, FILTER, OPTION, UNION, etc.

En guise d'exemple pour la suite, nous donnons ci-dessous un exemple de données RDF extraites d'un flux de données issues d'un réseau de transport et de distribution d'eau potable.

```
sensorJ06 measure     resultJ06 .
sensorJ06 designation AJ06 .
sensorJ06 observation pressure .
sensorJ06 inZone      zoneId43H .
sensorJ06 contains    factoryAF4D .
resultJ06 type        pressure .
```

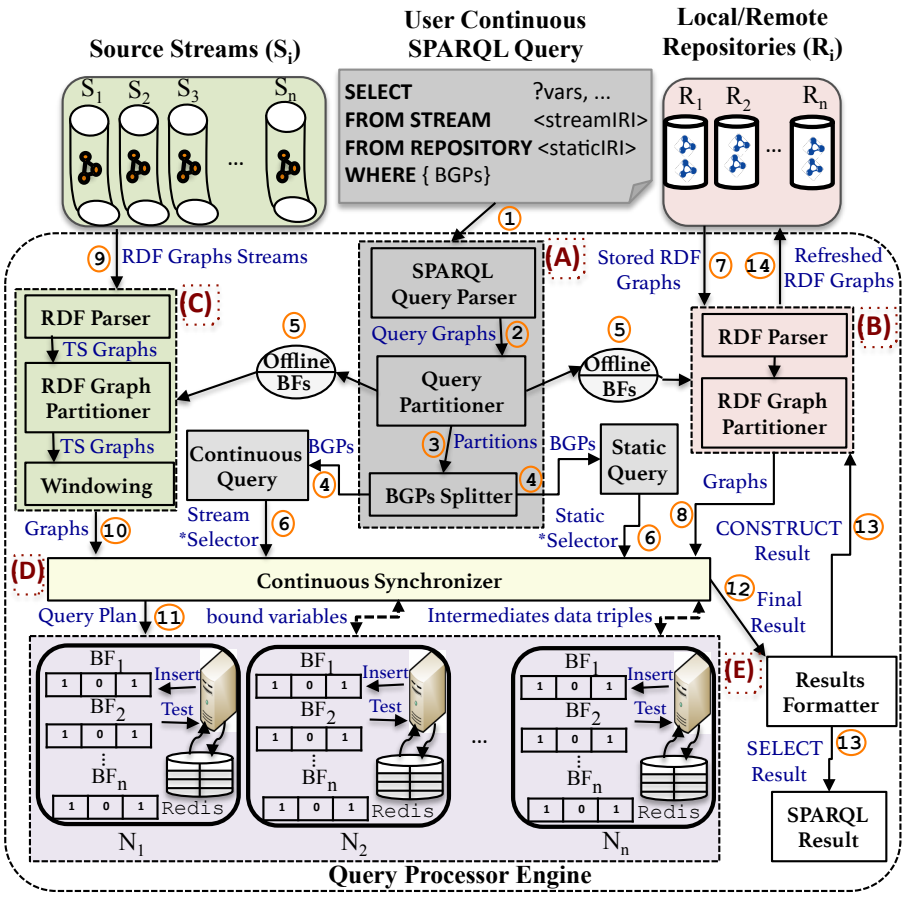


FIGURE 9.2 – Architecture du système.

```
resultJ06 uom         mH2O .
resultJ06 value       37^^xsd:integer .
sensorJ07 measure     resultJ07 .
sensorJ07 designation AJ07 .
resultJ07 type        pressure .
resultJ07 uom         mH2O .
resultJ07 value       29^^xsd:integer .
sensorJ02 measure     resultJ02 .
sensorJ02 designation AJ02 .
resultJ02 type        pressure .
resultJ02 uom         mH2O .
resultJ02 value       31^^xsd:integer .
```

Pour interroger ces données, nous considérerons dans la suite la requête SPARQL donnée par l'exemple 2.2.2 et son graphe correspondant donné par la figure 2.2.2 (chapitre 2 que nous rappelons dans la figure 9.3).

Exemple 9.3.1. Requête SPARQL sur les données RDF de l'exemple 2.2.1.

```
SELECT ?sensor ?value WHERE {
?zoneId contains factoryAF4D.   → TP1
?sensor inZone ?zoneId.         → TP2
?sensor observation pressure.   → TP3
```

```
?sensor measure ?result.          → TP4
?result type pressure.            → TP5
?result value ?value.}            → TP6
```

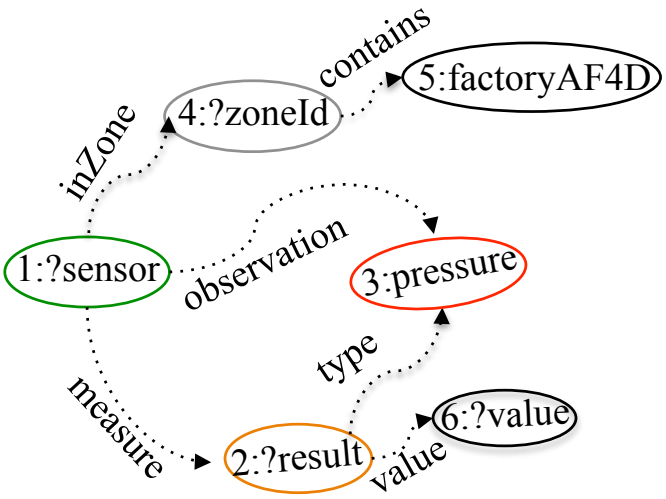


FIGURE 9.3 – Graphe de requête de l'exemple 2.2.2.

Les cinq (5) principaux composants du système sont **(A)** le module de gestion de requêtes SPARQL (**offline SPARQL query manager** avec `parser`, `partitioner` et `splitter`), **(B)** le module de gestion de données stockées (**offline stored data manager** avec `parsing`, `partitioning` et `refreshing`), **(C)** le module de gestion des flux de données en temps réel (**real-time data streams manager** avec `parsing`, `partitioning` et `windowing`), **(D)** le module de synchronisation continue (**continuous synchronizer**) assurant la communication inter-nœuds de traitement, le partage des résultats intermédiaires et le formatage final des résultats) et **(E)** le moteur de requête (**the query processor engine**) assurant le stockage des graphes RDF statiques dans les FBs des graphes RDF statiques et le traitement parallèle des partitions de requête. Nous détaillons dans ce qui suit tous ces modules.

— **(A)**. Le **module de gestion de requêtes SPARQL** analyse d'abord les requêtes SPARQL continues en entrée et génère les graphes de requête correspondants. Il prend également en entrée le paramètre d'exactitude de FB qui sera détaillé dans ce qui suit (9.3.3). Les graphes de requête sont partitionnés en fonction de l'opérateur de jointure. Le partitionneur (Query Partioner) répertorie d'abord tous les nœuds de jointure présents dans le graphe de requête et détecte les patterns de triplets formés exactement par deux nœuds de jointure (pattern complet). Les patterns complets sont dupliqués sur les partitions formées et les patterns incluant un seul nœud de jointure sont directement affectés au motif complet auquel il est lié. La figure 9.4 donne un exemple de partitions de graphes de requête de la figure 9.3. Les BGPs dans les partitions de requête sont divisés en parties statique (la partie de la requête liée aux graphes RDF stockés) et continue (la partie de la requête uniquement liée aux flux de graphes RDF). Les BGPs statiques et dynamiques sont respectivement transmis à un module de gestion de données stockées et un module de gestion de flux de données en temps réel. Les modules intermédiaires relatifs à la requête continue et statique génèrent les éléments `*Selector`. Ce dernier contient le BGP de la partie dynamique (Stream) ou statique (Static) ainsi que toutes les variables non appariées (non encore matchées ou résolues) dans le SELECT.

— **(B)**. Le **module de gestion de données stockées** importe les graphes RDF stockés en fonction des BGPs statiques. Les graphes importés sont analysés et partitionnés en fonction des informations collectées à partir de l'étape de partitionnement des requêtes. Chaque partition de requête reçoit son propre FB pour un stockage distribué des sous-graphes de données en mémoire dans les nœuds de traitement. Ce module gère également le mécanisme de rafraîchissement des données (si nécessaire) qui consiste en un graphe de sortie résultant destiné à mettre à jour, si nécessaire, un dépôt local ou distant de données statiques. Notons que ce module utilise les paramètres également les paramètres d'exactitude du FB que nous détaillons dans la sous-section 9.3.3.

— **(C)**. Le **module de gestion des flux de données en temps réel** démarre l'activité "temps réel'" du système. Les mêmes méthodes d'analyse et de partitionnement des graphes RDF stockés sont utilisées sur des flux de graphes RDF. Chaque partition de graphe est ensuite dirigée vers le module de fenêtrage auquel elle est connectée avant d'être stockée en mémoire selon le mécanisme de partitionnement de requête. Tout comme le module **(B)**, ce module utilise aussi les paramètres d'exactitude du FB.

— **(D)**. Le **module de synchronisation continue** assure le parallélisme et la cohérence dans le processus de traitement. Il transforme également les requêtes `*Selector` en plans de requête contenant chacun un ensemble de patterns de triplets. Enfin, il échange avec le moteur de traitement de la requête des variables (remplissant déjà une condition dans le plan de requête) et des résultats intermédiaires (triplets de données RDF déjà appariés) et formate en continu les résultats finaux (12). Le résultat final dépend de la requête et peut être un résultat formaté SELECT ou CONSTRUCT destiné à rafraîchir des différents dépôts locaux ou distants connus.

— **(E)**. Le **moteur de traitement de requête** renferme notre stratégie d'exécution de jointure parallèle et distribuée sur un cluster de FBs stockés en mémoire. Nous détaillons dans ce qui suit les approches de stockage et de jointure de FB utilisées dans ce module. A noter que les partitions de requêtes sont traitées localement (sur un nœud de traitement) sur des partitions de données RDF tout en minimisant autant que possible les informations échangées avec le synchroniseur.

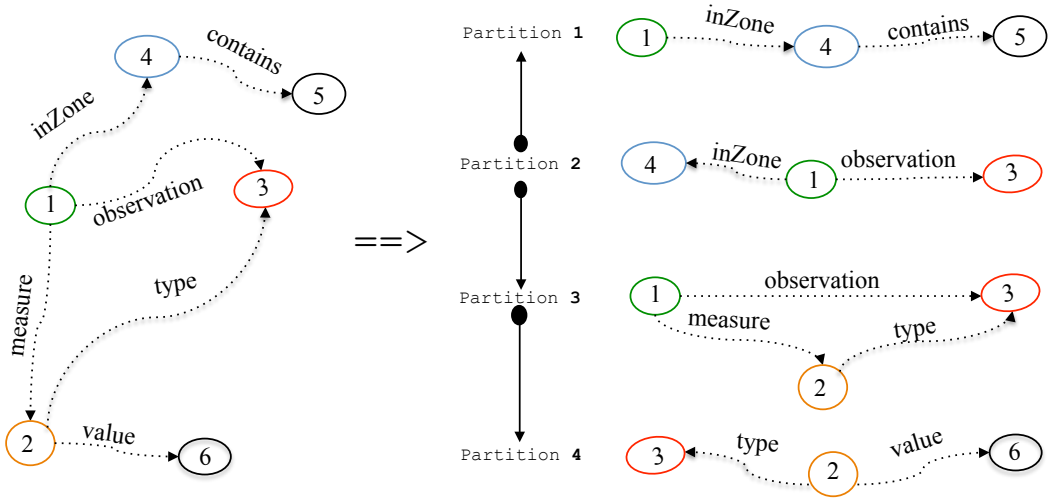


FIGURE 9.4 – Partitions de graphe de la requête de l'exemple 9.3.1 de la figure 9.3.

9.3.2 Procédure de jointure

Nous détaillons dans cette section l'approche adoptée pour notre opération de jointure distribuée en mémoire entre les graphes RDF stockés et continus. Selon les variables non appariées dans la clause WHERE, l'objectif de la requête est de trouver les bindings qui correspondent aux conditions (jointure) dans la clause. Notre approche de traitement des requêtes exploite quatre (4) des huit (8) combinaisons de patterns de triplets distincts dans une clause WHERE c'est-à-dire (i) les trois éléments sont des constantes `<s, p, o>`, (ii) seul le sujet est une variable `(<?s, p, o>)`, (iii) seul l'objet est une variable `(<s, p, ?o>)` et (iv) seul le prédicat est une constante `(<?s, p, ?o>)` . Les combinaisons non supportées sont (v) seul le sujet est une constante `(<s, ?p, ?o>)`, (vi) seul l'objet est une constante (<?s,?p, o>), (vii) seul le prédicat est une variable (<s?p, o>) et ($viii$) les trois composantes sont des variables (<?s, ?p, ?o>), étant donné que les combinaisons (ii), (iii) et (iv) sont les plus fréquemment rencontrées.

Chaque pattern de graphe RDF est haché à l'aide de fonctions de hachage SHA-512. Chaque triplet haché et sa valeur de hachage sont stockés en mémoire dans la base de données clé-valeur, qui est apparié en même temps à l'indice du triplet stocké dans le FB. Soulignons que le FB, au niveau taille des données, stocke en permanence les graphes RDF fenêtrés (c'est-à-dire valides pendant un temps donné et supprimés après l'expiration de la session).
Notre approche génère d'abord un plan de requête (plan de requête statique et continu) pour chaque requête continue et `*selector`. Le plan de requête est composé d'un ensemble de patterns de triplets (par exemple $\texttt{TP}_i$ dans le BGP de la requête SPARQL de l'exemple 9.3.1). Chaque variable non appariée de l'ensemble des variables dans le plan de requête est affectée à son propre FB initialement vide. L'algorithme 14 montre les étapes impliquées dans la procédure de jointure. Chaque plan de requête est à son tour mappé à ses variables non appariées (une variable peut être localisée dans plusieurs plans de requêtes). Pour chaque plan de requête statique mappé à une variable non appariée donnée, nous itérons et testons l'appartenance des variables non appariées communes aux plans de requêtes continues. Si au moins une variable commune est trouvée, les FBs continus et statiques sont respectivement interrogés et les bindings des variables trouvés. Une fois que les variables communes sont appariées, nous itérons sur chaque appariement de variable de requête en remplaçant la variable commune par leur valeur concrète. Enfin, nous ajustons la taille du FB et le nombre de fonctions de hachage pour minimiser la probabilité de faux positifs afin d'obtenir le paramètre d'exactitude du FB le plus optimisé.

Ce processus réalise uniquement la procédure de jointure entre les premières variables communes sélectionnées. Ensuite, pour une procédure de jointure complète et optimale, nous réduisons la taille des résultats intermédiaires pour le reste des opérations de jointure des variables non partagées. En effet, nous générons un seul plan de requête (plan de partition de requête final) dans lequel les variables partagées sont remplacées par leurs bindings. A partir du plan final, nous sélectionnons la variable suivante (non partagée), la parcourons et récupérons les bindings depuis le FB statique ou continu. Cependant, lorsque les triplets de la variable ont deux variables non appariées dont l'une est déjà liée à son binding durant les itérations précédentes alors le binding final de la variable traitée est immédiatement utilisé pour gérer la seconde (c'est-à-dire, les patterns de triplets sont récursivement réorganisés pour éviter les itérations coûteuses et réduire les tâches répétées).

Par exemple, considérons le graphe de requête de la figure 9.3 dont les partitions sont

Algorithme 14 Jointure FB multi-variables de graphes RDF

Require: Query plan qP,
Previous var binding pVB,
Proba false positives P_{fp}
1: **while** Variable T ∈ qP **do**
2: tPs ← getAllTriplePatternsOf(var);
3: **while** Triple pattern tp ∈ tPs **do**
4: Non shared vars nC ← tp.getListOfVars() without var;
5: ListOfAlreadyAddedBindings Bind;
6: **while** Triple tr ∈ Bind **do**
7: **if** pVB.contains(tr.getBindingFor(nC)) && pVB.contains(tr.getBindingFor(var)) **then**
8: Bind.add(tr.getBindingFor(var))
9: **end if**
10: **end while**
11: **end while**
12: Bloom Filter BF ← createFlterJoin(Bind, P_{fp});
13: pVB.add(var, BF);
14: **end while**
15: **return** pVB;

représentées par les BGP de la figure 9.4. Nous illustrons notre approche de traitement de jointure par FB des partitions de requête dans la figure 9.5. Les patterns de chaque partition sont stockés dans son propre FB. En utilisant l'algorithme 14, les variables non appariées sont linéairement appariées s'il existe une ou des dépendance(s) entre elles. Sinon, les variables qui ne dépendent d'aucune autre variable sont appariées en parallèle. Une fois `?zoneId` (nœud 4) a été traité avec le pattern de triplet TP_1 et que `?sensor` (nœud 1) est la variable suivante qui apparaît dans le pattern de triplet TP_2 où les deux variables apparaissent, alors le binding de `?zoneId` est immédiatement ajouté à celui de `?sensor`. L'opération de jointure du FB est en cascade et, si nécessaire, les résultats intermédiaires sont partagés avec le FB en mémoire et le stockage clé-valeur (Redis) dans d'autres nœuds de traitement. La même opération est ainsi répétée pour les variables non encore appariées `?result` (nœud 2) et `?value` (nœud 6). Le synchroniseur traite la jointure finale et stocke les triplets continus finaux dans des FBs temporaires qui renvoient l'identifiant de capteur (`sensorJ06`) et sa valeur de pression mesurée (`37^^xsd:integer`).

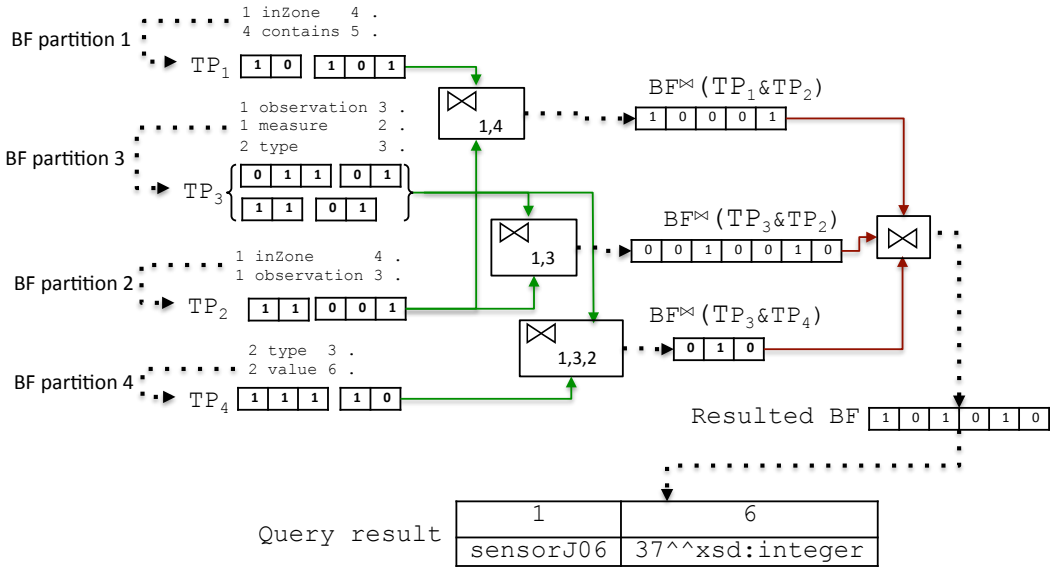


FIGURE 9.5 – Procédure de jointure statique-continue de l'exemple de graphe de requête de la figure 9.3.

9.3.3 Exactitude du FB

Comme mentionné précédemment, la requête pourrait indiquer par erreur que l'élément t fait partie de S. Ce défaut appelé faux positif, constitue le point faible connu des FBs mais est compensé par l'économie d'espace mémoire. Cependant, les faux positifs peuvent être acceptables même tolérés dans de nombreuses applications à condition que le taux de probabilité soit suffisamment faible. Rien ne peut prévenir les faux positifs ($h_1(y) = h_2(y) = ... = h_k(y)$) et la probabilité de l'avoir peut être estimée et réduit en fonction de la taille de l'ensemble n, la taille configurée du filtre m et le nombre de fonctions de hachage k (c'est-à-dire $\mathrm{P}_{false\_positives} \rightarrow (n, m, k)$). Si m indique la taille du FB (avec tous les bits initialisés à 0) et k le nombre de fonctions de hachage. Après avoir inséré tous les éléments n de S dans le filtre F, la probabilité qu'un bit donné reste égal à 0 est $(1 - \frac{1}{m})^{kn}$. Cela peut être démontré comme suit.

1. Pour la première insertion d'un élément dans F, la probabilité qu'une position de bit ne soit pas définie sur 1 est $1 - \frac{1}{m}$.
2. Ensuite, la probabilité que l'une des positions ne soit pas définie sur 1 après k hachage est $(1 - \frac{1}{m})^k$.
3. Enfin, la probabilité après avoir inséré les éléments n est $(1 - \frac{1}{m})^{kn}$.

De manière triviale, la probabilité que le bit soit défini sur 1 est
$1 - (1 - \frac{1}{m})^{kn}$.
Lors de l'interrogation du FB, la probabilité de faux positifs $\mathrm{P}_{falsePositives}$ pour tout test d'appartenance à un élément peut être évaluée comme suit :

$$\mathrm{P}_{falsePositives} = (1 - (1 - \frac{1}{m})^{kn})^k$$

Dans [144], les auteurs démontrent l'approximation suivante : $(1 - \frac{1}{m})^{kn} \approx e^{\frac{-kn}{m}}$. Cela implique que $\mathrm{P}_{falsePositives} \approx (1 - e^{\frac{-kn}{m}})^k$. Nous pouvons obtenir une faible probabilité de faux positifs en augmentant la taille du filtre m et le nombre de fonctions de hachage. Le taux de probabilité augmente lorsque n augmente. Cependant, le nombre d'éléments à stocker dans le F peut être maximisé tout en gardant le taux de probabilité de faux positifs en dessous d'une certaine valeur ρ. Dans [145], les auteurs fournissent des valeurs approximatives de n et k en fonction de la probabilité de faux positifs donnés $\mathrm{P}_{falsePositives}$ et de la taille du filtre m :

- $n \approx m \frac{\ln p \ln(1-p)}{-\ln \mathrm{P}_{falsePositives}}$, with $p \approx 1 - e^{\frac{-n}{m}}$

Par conséquent, avec $p = \frac{1}{2}$, nous obtenons :

- $n \approx m \frac{(\ln 2)^2}{|\ln \mathrm{P}_{falsePositives}|}$

et

- $k = log_2(\frac{1}{p})$

La figure 9.6 détaille en profondeur les opérations de jointure BF des patterns de triplets de la figure 9.5. Nous fixons en particulier la probabilité de faux positifs ($\mathrm{P}_{falsePositives} = 10^{-3}$, voir section 9.4.4) pour optimiser l'opération de jointure FB. Chaque opération génère toutes les combinaisons possibles de paires de variables non appariées. Le synchroniseur ne choisit que les variables conservant une signature dans le `*Selector`.

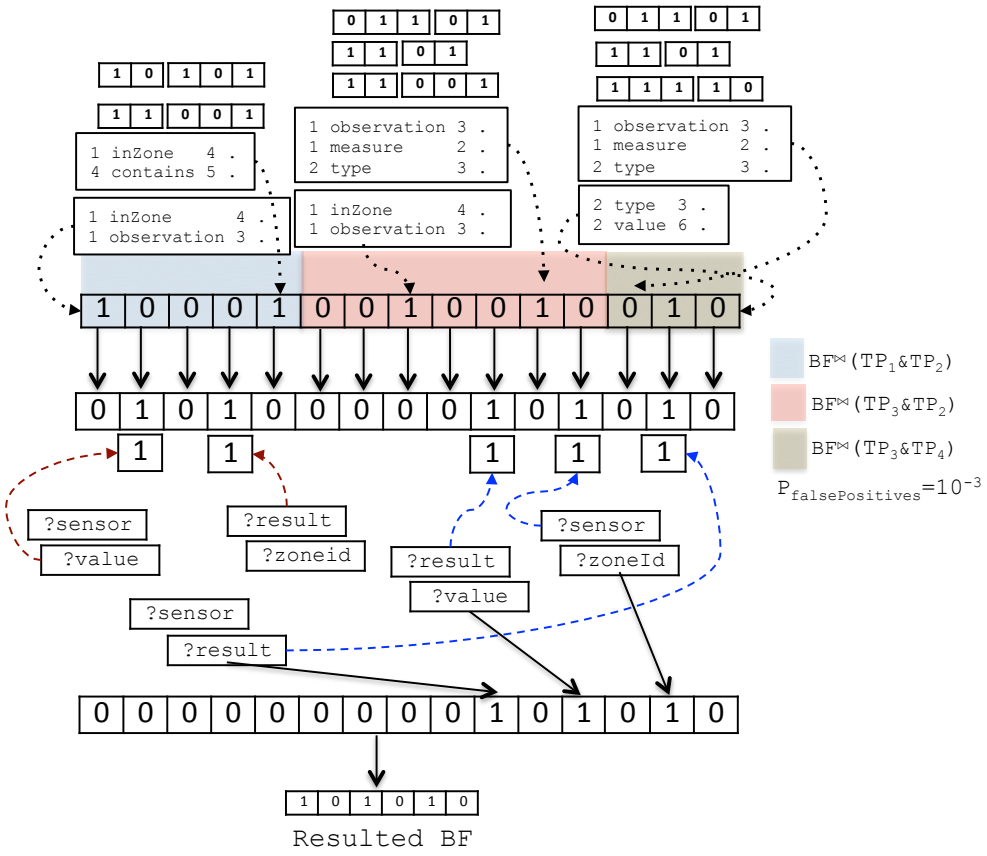


FIGURE 9.6 – Jointure BF des patterns de triplets de la figure 9.5 en utilisant une probabilité de faux positifs 10^{-3}.

9.4 Évaluations

Cette section évalue le système proposé à partir de la section 9.3 dans trois (3) expériences. Nous comparons, dans un premier temps, JSS-RDF aux systèmes RSP principaux qui prennent en compte l'opération de jointure entre les données RDF continues et statiques en matières de débit et de latence dans le traitement de la requête. La deuxième expérience évalue la scalabilité de JSS-RDF en le comparant au système de base (un seul nœud de traitement). Enfin, la troisième expérience se concentre sur l'exactitude des résultats par le débit et l'erreur relative par rapport à la probabilité de faux positifs ($P_{falsePositives}$).

9.4.1 Jeux de données, requêtes et configuration

9.4.1.1 Jeux de données

Nous utilisons trois ensembles de données du monde réel utilisés dans les benchmarks SRBench [4]. Ces jeux de données contiennent (*i*) des données RDF en continu collectées depuis les stations météorologiques des US (LinkedSensorData [5]), (*ii*) des données RDF stockées décrivant l'emplacement des stations (GeoNames [6]) et (*iii*) les localisations de GeoNames décrites dans un autre jeu de données RDF stocké (DBpedia [7]). La Figure 9.7 présente un aperçu des flux de données utilisés et des jeux de données stockées. En tant que flux de fichiers RDF, nous utilisons un ensemble de données Linked-

5. http ://wiki.knoesis.org/index.php/LinkedSensorData
6. http ://www.geonames.org/ontology/documentation.html
7. http ://wiki.dbpedia.org/develop/datasets

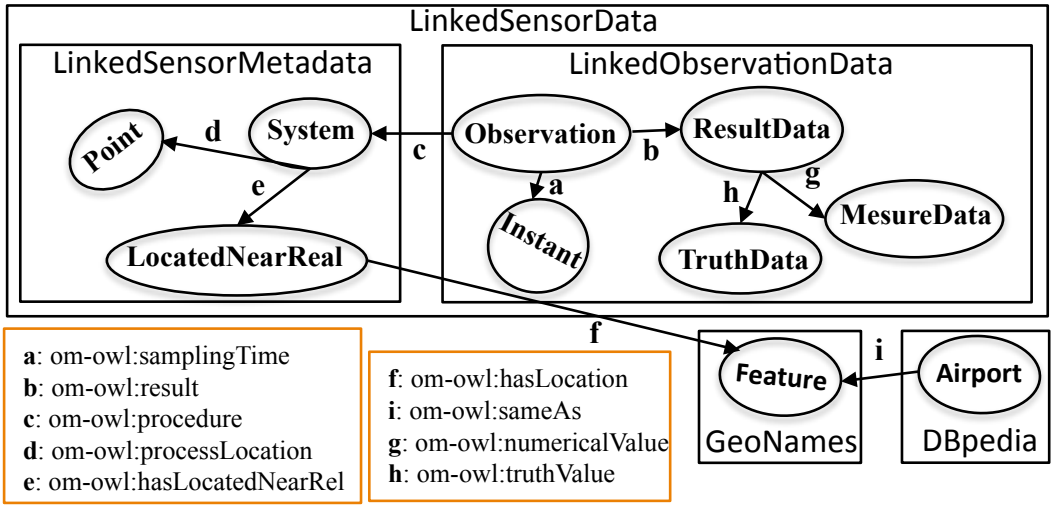


FIGURE 9.7 – Vue d'ensemble des jeux de données utilisés et de leurs relations [4].

SensorData, fourni par le laboratoire kno.e.sis [8]. Il s'agit d'un jeu de données RDF réelles contenant l'agrégation de données météorologiques (température, visibilité, précipitations, pression, vitesse du vent, humidité et de localisation telles que la latitude, la longitude et l'élévation) recueillies durant l'année 2002. C'est jusqu'à présent le plus grand ensemble de données de flux RDF puisqu'il contient environ 1,7 milliard de triplets RDF. Le LinkedSensorMetadata (LSM) regroupe la description de 100 000 capteurs chacun lié à des emplacements dans GeoNames qui sont à proximité de chaque station météorologique. Le LinkedObservationData (LObD) contient tous les phénomènes de mesure (température, pression, précipitation, vitesse du vent, etc.) particulièrement observés pendant les périodes où les États-Unis ont fait face à des ouragans majeurs : Katrina, Ike, Bill, Bertha, Wilma, Charley et Gustav. Il contient 159 millions d'observations décrites à travers 2 milliards de triplets RDF.

Comme données RDF stockées, nous utilisons un ensemble de données de l'ontologie GeoNames (version 3.1 (novembre 2012)) qui contient des informations sémantiques géospatiales d'environ **8 millions** de lieux répartis dans **11 millions** de toponymes GeoNames. Il contient plus de **100 millions** de triplets [9]. Nous utilisons également un ensemble de données de DBpedia qui contient une base de connaissances d'informations structurées de Wikipedia. La version anglaise de la version 2015-04 de DBpedia contient une description d'environ **5,9 millions** d'entités classées dans un total de **4 millions** de ressources (lieux, entreprises, etc.). Dans cette expérience, nous extrayons seulement un échantillon de **203 millions** de triplets RDF parmi les **737 millions** de triplets RDF [10] de la version anglaise de DBpedia de la version 2015-04. Le tableau 9.1 résume les statistiques (taille globale sur disque et nombre de triplets) de chaque jeu de données RDF stockées et utilisé dans cette expérience. Nous appliquons un traitement temps réel en mémoire sur les graphes RDF continus et statiques.

8. (the Ohio Center of Excellence in Knowledge Enabled Computing
9. https ://old.datahub.io/dataset/geonames-semantic-web
10. http ://wiki.dbpedia.org/dbpedia-data-set-2015-04

	LObD	**LSM**	**GeoNames**	**DBpedia**
Type	Hurricane Ike	–	–	–
Période	1 – 13 Sep. 2008	–	–	–
Observations	34 Millions	–	–	–
Triplets	374 Millions	185,383	–	203 Millions
taille sur disque	35 Go	40 Mo	19,2 Go	31,2 Go

TABLEAU 9.1 – Statistiques des données RDF statiques et continues utilisées dans notre évaluation.

9.4.1.2 Requêtes

Comme requêtes d'évaluation, nous utilisons et adaptons les requêtes du benchmark SRBench. Ce benchmark contient un ensemble de 17 requêtes numérotées de **Q1** à **Q17** [11] [4]. De toutes ces requêtes, nous nous focalisons tout d'abord sur celles qui couvrent à la fois les flux de données (LObD) et les métadonnées de capteurs stockées (LSM) : de **Q8** à **Q11**. Nous utilisons ensuite des requêtes du benchmark couvrant l'ensemble des données présentées (LObD, LSM, GeoNames et DBpedia), c'est-à-dire de **Q12** à **Q17**. Nous adaptons les requêtes choisies en fonction de notre système implémenté sans modifier les expressions SPARQL internes (voir en annexe B, la liste des requêtes réadaptées pour notre système).

Les requêtes SRBench s'intéressent aux principaux aspects de la requête SPARQL, c'est-à-dire de l'opérateur de jointure naturelle simple ("." ou " ;") au plus complexe (UNION et OPTIONAL). Cependant, la première partie de notre évaluation se concentre sur l'opération de jointure entre des graphes RDF continues et statiques plutôt qu'avec la complexité des différents opérateurs dans les requêtes. Dans la suite, nous listons les besoins complets exprimés par ces deux classes de requêtes de benchmark avec leur numérotation d'origine.

- **Classe 1** : en utilisant les jeux de données **LObD** et **LSM**.
  - **Q8.** Fournir la température quotidienne minimale et maximale de l'air observée par le capteur à un endroit donné.
  - **Q10.** Fournir les endroits où une forte chute de neige a été observée ces derniers jours.
  - **Q11.** Détecter si une station a produit des mesures significativement différentes de celles des stations voisines.
- **Classe 2** : en utilisant les jeux de données **LObD**, **LSM**, **GeoNames** et **DBpedia**.
  - **Q12.** Fournir la température moyenne de l'air et l'humidité des grandes villes.
  - **Q14.** Fournir le ou les aéroport(s) situé(s) dans la même ville que le capteur qui a observé une visibilité extrêmement faible dans la dernière heure.
  - **Q15.** Fournir les emplacements où la vitesse du vent dans la dernière heure est plus élevée qu'un ouragan connu.
  - **Q17.** Estimer les dommages lorsqu'un ouragan a été observé.

11. https ://www.w3.org/wiki/SRBench

9.4.1.3 Configuration

Nous avons importé tous les jeux de données précédemment définis dans le cloud d'AWS et utilisons les requêtes précédentes dans deux catégories : nous comparons le débit global de traitement des requêtes de la **classe 2** et le taux de latence dans l'exécution de requête entre JSS-RDF (c'est-à-dire en utilisant notre approche de jointure avec les FBs) et deux des principaux systèmes RSP existants (C-SPARQL et SPARQL*stream*) en mode centralisé (un seul nœud de traitement). Les requêtes de la **classe 2** sont également utilisées pour l'évaluation de la scalabilité de JSS-RDF. En utilisant les requêtes de la **classe 1**, nous réalisons la même comparaison entre JSS-RDF et CQELS. Nous définissons enfin la requête **Q10** pour l'évaluation de l'exactitude des résultats. Dans chaque expérience, le débit d'exécution, la latence du traitement de la requête et l'erreur relative moyenne sont calculés et sauvegardés.

Les requêtes de la **classe 2** sont complètement exécutées dans le cloud EC2 d'Amazon sur un cluster de 15 nœuds (1 synchronizer + 14 executor). Nous utilisons Java 8, Storm (version 0.9.1), Zookeeper (version 3.4.8), kafka (version 0.8.x), Redis (ElastiCache pour Redis 3.2.6) pour un stockage clé-valeur en mémoire des triplets intermédiaires et bibliothèques nécessaires. Toutes les expériences sont orientées sur un équilibre entre ressources de traitement, mémoire et réseau. En effet, les requêtes et les résultats intermédiaires sont traités dans des instances EC2 "m3.xlarge" de grande taille et non extensibles, avec une plate-forme 64 bits, 15 Gio RAM, 2 x 40 Go de stockage d'instance, 4 processeurs Intel Xeon E5-2670 (Sandy Bridge) tournant à 2,6 GHz et une bande passante d'E/S élevée (500 Mo/s). Le cluster est configuré sur un (1) nœud synchronizer pour l'échange de données et de résultats intermédiaires, deux (2) nœuds pour la gestion des flux de graphes RDF, deux (2) nœuds pour la gestion des graphes RDF locaux/distants (1 nœud pour l'import et 1 autre pour l'export (rafraîchissement des dépôts)). Chaque graphe RDF est importé ou exporté via le serveur de file d'attente kafka. Enfin, 9 nœuds sont utilisés comme nœud de traitement de la topologie Storm (spout et bolts) et le dernier est consacré au formatage des résultats de requêtes SARQL. Tous les composants internes et externes de notre expérience sont implémentés en Java 8.

Chaque procédure d'exécution enregistre une seule requête qui contient les URI des flux (l'identifiant des flux sources correspondant chacun à un topic kafka). Une fois la phase à froid (offline) traitée, le système démarre la partie temps réel (real-time) et donc l'envoi des flux définis via kafka avec des débits d'entrée aléatoires (le système ne doit pas contrôler l'arrivée des flux de données) (graphes/s).

Les évaluations en mode centralisé (avec C-SPARQL, SPARQL*stream* et CQELS) sont effectuées en utilisant une seule instance de m3.xlarge (également considérée comme notre référence) tandis que les évaluations dans un environnement distribué placent les partitions de requêtes et de graphes RDF sur les 9 nœuds de traitement. Pour chaque requête, nous répétons 4 traitements centralisés et distribués avant de calculer les résultats moyens. Pour tester l'amélioration de la scalabilité dans l'évaluation en mode distribué, nous supprimons et ajoutons au hasard un nombre aléatoire (entre 1 et 4) de nœuds sans arrêter le traitement. Pour le traitement parallèle, le synchronizer alloue des partitions de requêtes selon la configuration de topologie, c'est-à-dire le nombre de **workers** à créer, le nombre d'**executors** à générer par composant et le nombre de **tasks** à créer par composant. Enfin, toutes les fenêtres des flux sont fixées à 20 secondes, ce qui peut également être suffisant mais également nécessaire pour le chargement, au premier stade, des données par les systèmes RSP.

Pour les métriques de performance globale, nous mesurons le débit de traitement moyen (par exemple combien de triplets RDF peuvent être traités par seconde) et la latence dans le traitement (le délai entre l'entrée du graphe RDF et l'obtention du résultat final souhaité) pendant qu'une métrique particulière se focalise sur l'exactitude des résultats de la requête. Pour cela, nous suivons trois (3) critères de comparaison majeurs :

— **Vitesse du traitement.** Ce premier critère évalue le débit et la latence par rapport aux systèmes RSPs existants dans des conditions normales (sans modification de paramètres ni retrait d'un nœud de traitement). Nous mesurons la situation où le traitement de requête suivant notre scénario (c'est-à-dire en utilisant un FB) est plus rapide ou plus lent que son traitement en utilisant DRSS. Ce critère est suffisant pour démontrer l'optimisation du traitement des requêtes fournie par notre approche.

— **Scalabilité.** Nous démontrons les propriétés d'extensibilité en poussant à fond les paramètres de distribution et de parallélisme par rapport à la ligne de base (mode centralisé). Nous fixons d'abord la taille de la fenêtre à 60 secondes, faisons varier le nombre de nœuds de manière contrôlée et exécutons une requête à la fois. Nous avons exceptionnellement fait varier le débit des flux d'entrée pour vérifier uniquement l'ordre de grandeur de notre système atteignant le goulot d'étranglement comparé au mode centralisé.

— **Exactitude des résultats.** Ce critère d'évaluation est complémentaire à la mesure du débit de traitement. Cependant, dans cette évaluation, en plus de l'évaluation de l'exactitude des résultats, nous ajoutons l'erreur relative à la relation de probabilité $P_{falsePositives}$. En effet, comme démontré précédemment dans la sous-section 9.3.3, le choix de la probabilité $P_{falsePositives}$ (minimal, maximal) peut négativement affecter le résultat correct durant le processus de jointure avec le FB. Nous déterminons approximativement un intervalle de seuils garantissant qu'il n'y a aucun `faux positif` possible. Nous modifions de plus en plus la valeur de la probabilité $P_{fasePositive}$ selon les paramètres suivants :

$$n \approx m \frac{(\ln 2)^2}{|\ln P_{falsePositives}|} \text{ et } k = log_2(\frac{1}{p}).$$

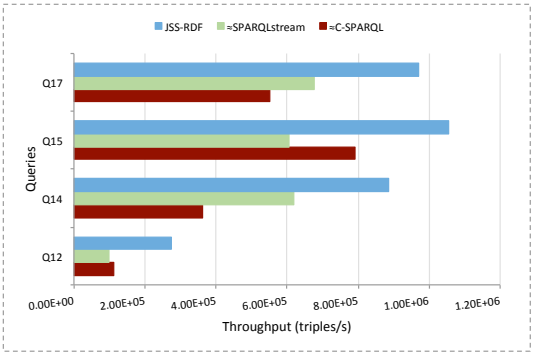


FIGURE 9.8 – Comparaison de débit entre systèmes JSS-RDF et ≈RSP.

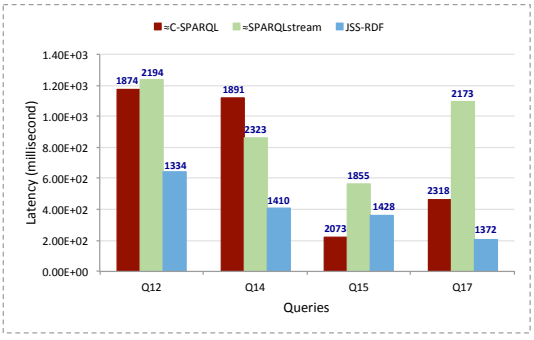


FIGURE 9.9 – Comparaison de latence entre systèmes JSS-RDF et ≈RSP.

Enfin, nous n'avons pas été en mesure de maintenir le même débit de flux d'entrée (trop élevé) pour les systèmes RSP. En effet, ces systèmes ne sont pas destinés à ce niveau de traitement de données à grande échelle [70]. Notre solution consiste à partager précisément les mêmes débits d'entrée aux premières entrées, puis à déterminer de façon "exponentielle", le débit des flux en entrée approximatif correspondant lorsque les

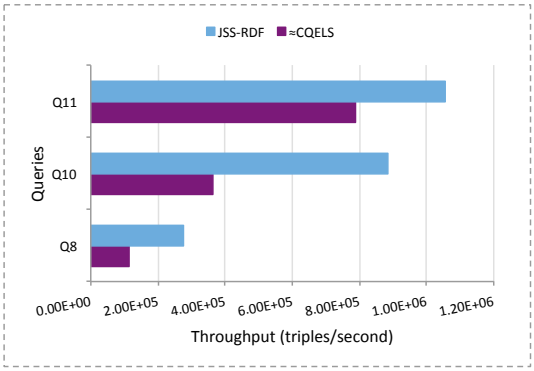


FIGURE 9.10 – Comparaison de débit : JSS-RDF et ≈CQELS.

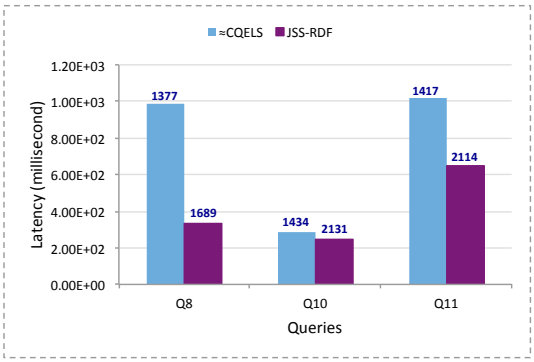


FIGURE 9.11 – Comparaison de latence entre JSS-RDF et ≈CQELS.

systèmes RSP ont atteint leurs limites. Nous adoptons donc une appellation ≈C-SPARQL et ≈SPARQL*stream* pour ces systèmes. Cela signifie qu'il s'agit plutôt d'une simulation de leur comportement en ce qui concerne le débit mais garde leur propre procédure de traitement.

9.4.2 Performances en mode centralisé

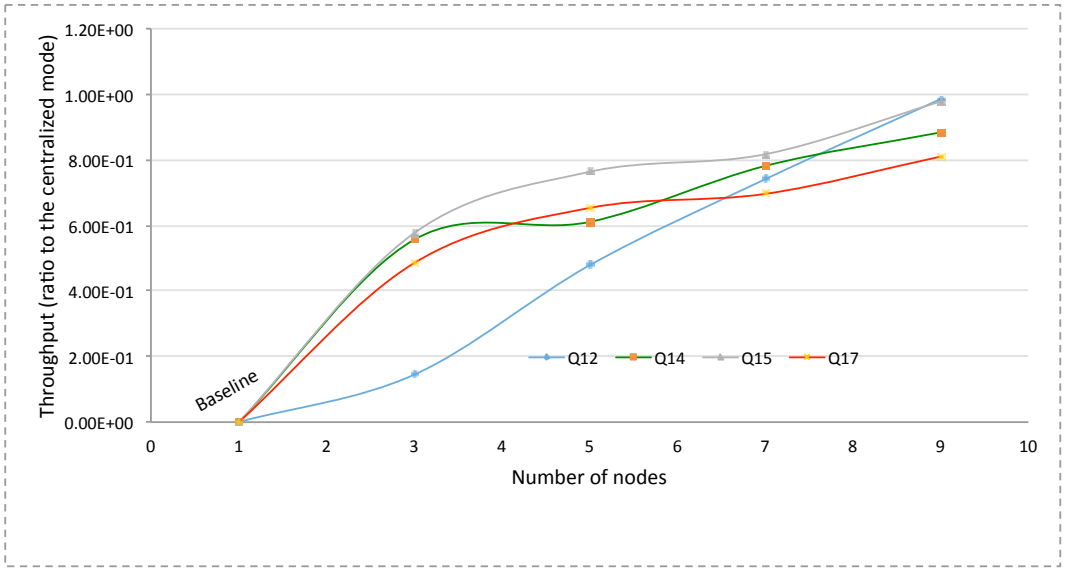


FIGURE 9.12 – Débit avec un nombre variable de nœuds.

Les figures 9.8 et 9.9 présentent respectivement les résultats moyens de la comparaison du débit et de la latence entre JSS-RDF et ≈C-SPARQL et ≈SPARQL*stream* en mode centralisé (1 nœud de traitement). Nous utilisons les requêtes SRBench **Q12**, **Q14**, **Q15** et **Q17** avec différents niveaux de complexité. Nous nous intéressons en particulier à la complexité liée à la combinaison des données RDF statiques et dynamiques et moins à celle associée aux opérateurs de requête. JSS-RDF présente un débit pouvant atteindre **1 million de triplets** en quelques secondes. Les systèmes RSP, ainsi que les systèmes existants, restent beaucoup moins efficaces en ce sens. Le JSS-RDF présente également une faible

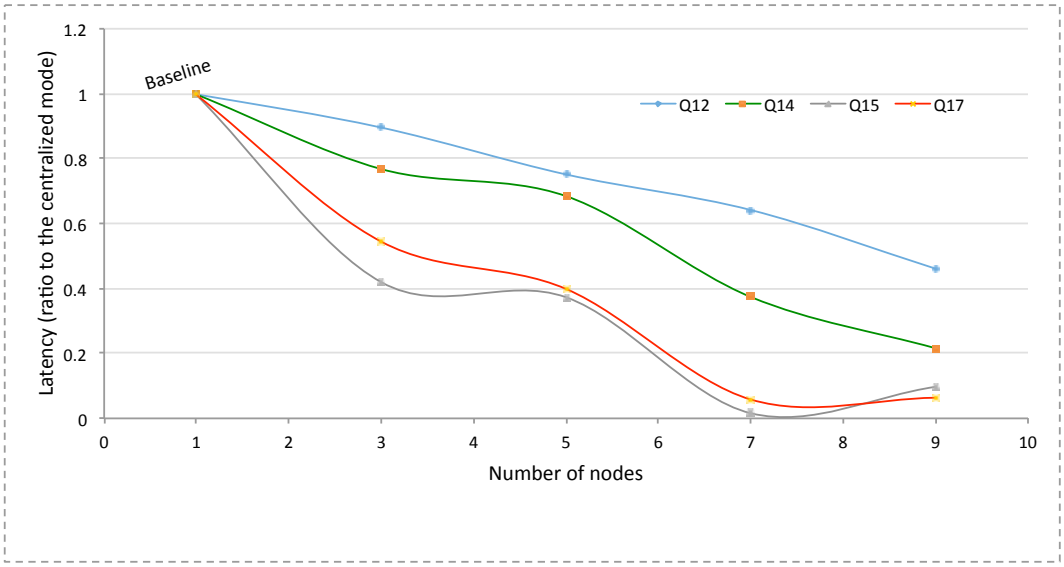
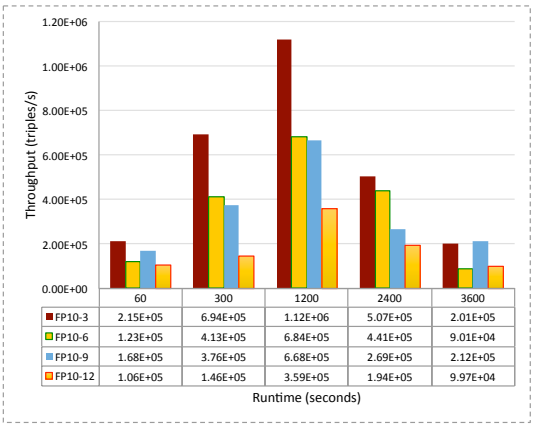
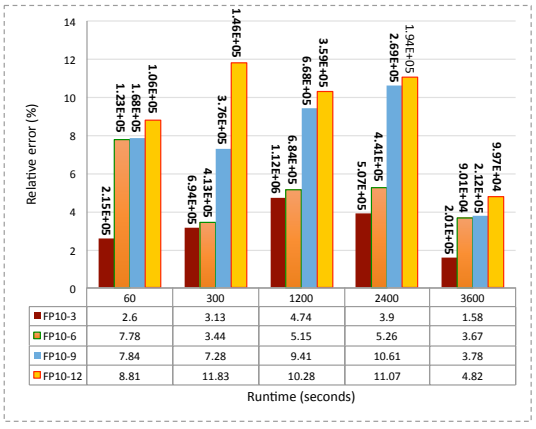


FIGURE 9.13 – Latence avec un nombre variable de nœuds.



	60	300	1200	2400	3600
FP10-3	2.15E+05	6.94E+05	1.12E+06	5.07E+05	2.01E+05
FP10-6	1.23E+05	4.13E+05	6.84E+05	4.41E+05	9.01E+04
FP10-9	1.68E+05	3.76E+05	6.68E+05	2.69E+05	2.12E+05
FP10-12	1.06E+05	1.46E+05	3.59E+05	1.94E+05	9.97E+04

FIGURE 9.14 – Débit avec une probabilité de FP variable.



	60	300	1200	2400	3600
FP10-3	2.6	3.13	4.74	3.9	1.58
FP10-6	7.78	3.44	5.15	5.26	3.67
FP10-9	7.84	7.28	9.41	10.61	3.78
FP10-12	8.81	11.83	10.28	11.07	4.82

FIGURE 9.15 – Erreur relative du résultat avec une probabilité de FP variable.

latence durant un échantillon de temps d'exécution par rapport aux requêtes qui combinent en particulier les flux et les données RDF stockées. Bien que les systèmes RSP évalués soient matérialisés dans leur condition optimale (≈C-SPARQL et ≈SPARQL*stream*), et JSS-RDF dans sa condition de performance minimale (1 seul nœud de traitement, 3 threads (par exemple 1 source de flux + 2 dépôts)), les systèmes RSP conduisent à une latence dans le traitement plus élevée ainsi qu'à un débit non optimisé. Ceci est dû au stockage en mémoire du FB et à l'opération de jointure rapide entre les graphes RDF stockés et continus. Les figures 9.10 et 9.11 présentent le résultat des mêmes évaluations réalisées cette fois sur CQELS en utilisant les requêtes SRBench **Q8**, **Q10** et **Q11**, étant donné que les requêtes précédentes utilisées n'ont pas d'opérateurs correspondants dans la syntaxe de requête de CQELS. Nous avons réalisé les mêmes expériences et avons obtenu les mêmes résultats et conclusions sur le débit d'exécution et la latence.

9.4.3 Scalabilité

Les figures 9.12 et 9.13 montrent respectivement que sur tous les jeux de données (millions de flux de graphes RDF et grands volumes de graphes RDF déjà stockés), le débit moyen augmente évidemment avec le nombre de nœuds de traitement alors que la latence moyenne des requêtes diminue. Notre approche d'exécution parallèle et distribuée combinée à l'accès rapide aux triplets et à la jointure avec les FBs des deux sources de données est clairement capable d'atteindre un débit élevé d'environ **1 million de triplets** sur un cluster de nœuds de traitement. Les résultats intermédiaires sont relativement petits et JSS-RDF échelonne sur 10 nœuds (1 synchronizer et 9 nœuds de traitement) avec le même nombre de workers storm (6 pour chaque machine) et fournit une faible latence.

Cependant, nous n'avons pas été en mesure d'effectuer des évaluations comparatives de certains systèmes distribués car ils ne sont pas disponibles ou nécessitent des configurations trop complexes pour être déployées sur EC2 dans les mêmes conditions que celles requises par JSS-RDF. En outre, CQELS Cloud [20] évalue les performances de ses opérateurs de requête tout en mettant l'accent sur la scalabilité et l'impact de l'exactitude des résultats de la combinaison des données RDF stockées et continues.

9.4.4 Exactitude des résultats

Les figures 9.14 et 9.15 exécutent des requêtes avec différentes probabilités $\mathrm{P}_{falsePositives}$ (FP) pour déterminer si ce paramètre du FB important affecte les résultats de la requête. La figure 9.14 montre en particulier qu'un paramètre FP bas (c'est-à-dire augmentant la taille du filtre m et le nombre de fonctions de hachage k) présente des impacts positifs sur l'exactitude des résultats (moins de perte de données). Cela est dû au fait qu'une valeur FP basse provoque moins d'éléments incorrects à travers le FB déjà existant avec un effet cascade croissant ou décroissant sur tous les filtres suivants. En fonction de la complexité de la requête, le temps d'exécution peut augmenter et affecter l'exactitude des résultats (c'est-à-dire que les données dans les flux sont perdues après l'expiration d'une session de fenêtre). Cependant, l'exécution de la requête s'accélère avec un FP bas. Dans la figure 9.14, 10^{-3} est la valeur FP la plus optimale pour l'initialisation du paramètre FP après des tests antérieurs non rapportés dans cette section. La figure 9.15 montre l'erreur relative des résultats obtenus à partir de la figure 9.14. L'erreur relative peut être définie comme le rapport en pourcentage du "nombre de triplets obtenus avec le FB" sur le "résultat obtenu sans FB" :

$$Relative\,Error = \frac{Execution\,With\,\mathrm{BF}}{Execution\,Without\,\mathrm{BF}} * 100.$$

La figure 9.15 indique que l'erreur relative diminue avec un FP bas. La baisse apparente du débit dans la figure 9.14 n'est pas proportionnelle à l'erreur relative, mais seulement à des données manquantes dans le jeux de données. Cependant, notre test pré-observé sur l'erreur relative, montre qu'un FP très faible peut augmenter l'erreur relative (jusqu'à **34,016%**) et affecte sérieusement la complétude et l'exactitude des résultats.

Les résultats expérimentaux montrent que JSS-RDF fournit un débit d'exécution élevé, une faible latence dans le traitement, une exactitude et une complétude des résultats qui sont d'excellentes métriques de "scalabilité". À notre connaissance, ces performances n'ont été obtenues par aucun système existant dans un contexte de flux RDF combinés aux graphes RDF stockés dans des repository.

9.5 Conclusion

Dans ce chapitre, nous avons présenté une approche pour la jointure en mémoire entre des graphes RDF stockés et continus basés sur des filtres Bloom. Nous avons suivi pour cela un mécanisme original pour gérer les opérations de jointure implémentées dans un système complet au-dessus de la plateforme Apache Storm. Par rapport aux systèmes RSP existants, nous pouvons affirmer que notre algorithme de jointure et notre système proposé ont fourni des performances bien meilleures dans un environnement combinant des données RDF statiques et dynamiques. Nous l'avons démontré à travers l'évaluation qui a montré une excellente scalabilité pour raisonner avec un haut débit et une faible latence dans le traitement de la requête. Nous avons également considéré l'exactitude et la complétude des résultats en évaluant la plage de valeurs FP la plus optimale pour réduire l'erreur relative du résultat de la requête. Contrairement aux principaux systèmes et approches de traitement distribué de flux de données RDF existants, JSS-RDF apporte une nette contribution dans l'optimisation des opérations de jointures entre données RDF statiques et dynamiques. Cependant, la limite JSS-RDF est négligeable car elle réside principalement dans le choix de la valeur FP la plus adaptative en fonction de tous les paramètres (débit, fréquence de rafraîchissement, taille de fenêtre et paramètres de distribution).

Publication
[47]. Dia, Amadou Fall, Zakia Kazi Aoul, Aliou Boly, and Elisabeth Métais. "Fast SPARQL join processing between distributed streams and stored RDF graphs using bloom filters."

Chapitre 10

Conclusion et Perspectives

Sommaire

10.1 Synthèse

Dans cette thèse, nous avons présenté nos travaux concernant la réduction de la charge de traitement de flux de données RDF.
Après avoir exposé et détaillé les principaux systèmes et approches de traitement centralisé (C-SPARQL, CQELS, $\text{SPARQL}_{\text{stream}}$, EP-SPARQL et Sparkwave) et distribué (CQELS Cloud, C-SPARQL on S4, DINYSUS et Strider) de flux de données RDF, nous avons présenté les principales approches d'échantillonnage, de résumé et de compression de flux de données. Ces techniques centralisées sont étudiées afin de proposer des approches de réduction de la charge de traitement des flux de données RDF par l'échantillonnage et le résumé de flux de graphes RDF et par l'interrogation de flux RDF compressés sans phase de décompression. De ces techniques, nous avons proposé une approche de réduction de la charge de traitement de flux de données RDF en mode centralisé suivant trois (3) contributions.

10.1.1 1ère contribution : échantillonnage de flux de graphes RDF

Dans le chapitre 4, nous avons étendu le langage et le système RSP centralisé C-SPARQL pour l'échantillonnage de flux de graphes RDF. Cette extension adopte une approche orientée graphe où chaque graphe RDF est formé d'un ou de plusieurs triplets et représente un événement. Le format graphe assure la préservation des liens sémantiques (c'est-à-dire des liens entre les sujets et les objets) après une opération d'échantillonnage. Cette approche nous a permis d'améliorer l'extraction d'échantillons cohérents à la place des techniques existantes qui utilisent des méthodes d'extraction naïves (c'est-à-dire sur des successions de triplets RDF). La préservation des liens sémantiques permet de garantir la complétude et l'exactitude des résultats des requêtes sur les graphes présents dans

l'échantillon. Ainsi, au delà de la préservation des liens sémantiques, notre extension de C-SPARQL ajoute un opérateur d'échantillonnage dans la syntaxe de requête et un module d'échantillonnage en continu dans l'architecture de C-SPARQL. Les trois (3) opérateurs d'échantillonnage choisis (**uniform**, **reservoir** et **chain**) sont ajoutés dans la clause **FROM STREAM** de la syntaxe de requête où l'utilisateur précise les paramètres d'échantillonnage (taux et fenêtre d'échantillonnage). Ces opérateurs correspondent à des méthodes d'échantillonnage implémentées au sein du module de traitement continu de C-SPARQL (Esper). Nous avons pris en compte toutes les variantes de fenêtre logique et physique de C-SPARQL. Le mécanisme d'échantillonnage réduit la charge de traitement dans C-SPARQL en réduisant la volumétrie et en s'adaptant au taux d'arrivée souvent élevé des flux RDF.

L'évaluation et la comparaison des résultats obtenus dans les deux scénarios (avec et sans processus d'échantillonnage) ont montré de meilleures performances en matière de temps de calcul de la requête et de la préservation de la sémantique des donnés. Ces résultats garantissent ainsi une réduction de la charge de traitement suivant la probabilité de sélection pour chaque flux tout en gardant la cohérence des données à traiter et garantit l'exactitude des résultats.

10.1.2 2ème contribution : résumé orienté graphe de flux de données RDF

Dans le chapitre 5, nous avons approfondi l'approche précédente en proposant une technique orientée graphe. L'échantillonnage permet de réduire la charge de traitement en terme de temps de traitement et de ressource mémoire tout en préservant les liens sémantiques mais est limité. Nous proposons une nouvelle approche de résumé de flux de données RDF qui exploite les propriétés du graphe en privilégiant les nœuds et les prédicats les plus pertinents. De plus, le résumé généré est orienté besoin (requêtes des utilisateurs). Notre approche tire profit des mesures de centralité (degré de proximité) issues de l'analyse des réseaux sociaux ARS. Ces mesures de centralité nous ont permis de caractériser les structures de graphe en définissant les positions stratégiques des nœuds afin d'extraire les informations les plus pertinentes. Cette extraction priorise la pertinence d'un nœud ou d'un lien dans le graphe en considérant le degré de proximité d'un nœud par rapport aux autres (c'est-à-dire, sa capacité à être proche des autres nœuds et à transmettre l'information au plus vite) ainsi que sa probabilité d'information (c'est-à-dire, sa probabilité d'atteindre tous les nœuds feuille qui, renferment le plus souvent l'information). Ainsi, l'extension de la mesure "centralité de proximité" est combinée à un mécanisme qui identifie les prédicats les plus importants d'un graphe RDF donné tout en les combinant aux prédicats présents dans la requête (ces prédicats expriment le besoin de l'utilisateur). Enfin, nous avons défini et mis en œuvre "l'importance d'un nœud" d'un graphe RDF. Cette mesure détermine à quel degré un nœud est proche de tous les autres.

Le système implémenté génère en continu des résumés de flux de graphes RDF et démontre la capacité de notre approche à fournir approximativement les mêmes réponses que le graphe source pour les requêtes SPARQL avec un minimum d'espace de stockage occupé en mémoire.

10.1.3 3ème contribution : interrogation de flux de données RDF compressées au format RDSZ

Dans le chapitre 6, nous avons adopté un format compressé des graphes RDF. Après une étude détaillée et une comparaison des deux principaux approches de compression de flux de données RDF que sont RDSZ et ERI, nous avons effectué un choix justifié de l'algorithme RDSZ pour la compression des flux de données RDF. Ce format de données

nous a conduit à la proposition d'une approche d'interrogation de graphes RDF compressés sans phase de décompression. En effet, nous avons éliminé la phase de décompression qui augmente naturellement le coût du traitement en matière de temps de calcul des requêtes, de consommation mémoire et de transfert de données entre composants du système. Il s'agit d'optimiser le traitement des données RDF compressées au format RDSZ en exécutant les requêtes SPARQL continues immédiatement après la phase de compression. Nous avons implémenté une solution typique aux trois (3) catégories de requêtes SPARQL (requête SPARQL simple, requête SPARQL avec opérateur(s) de filtrage et requête SPARQL avec fonction(s) d'agrégation(s)) que nous avons définie. Le système a été implémenté sur plusieurs modules prenant en compte les opérateurs de filtre, les expressions arithmétiques, les expressions régulières et les opérateurs d'agrégation.

Nous avons validé notre approche avec une évaluation des performances relativement au temps de traitement et à l'espace de stockage mémoire. Nous avons comparé le temps de calcul et l'espace mémoire utilisé d'une requête continue avec notre approche (c'est-à-dire, la somme des temps nécessaires pour les phases de compression à la volée des données d'un flux et d'exécution de la requête sur les données compressées) au temps de calcul de la requête continue avec le scénario de base (c'est-à-dire, la somme des temps nécessaires pour les phases de compression, de décompression et de calcul de la requête). Notre algorithme présente un temps de calcul et un espace de stockage mémoire plus faibles comparé au scénario de base. Enfin, notre système est assez modulaire et peut être intégré dans un système RSP centralisé.

10.1.4 4ème contribution : traitement distribué de flux de graphes RDF

Dans le chapitre 8, nous avons proposé DRSS (Distributed RDF SPARQL Streaming) comme système socle mais aussi comme nouveau langage de requête continue et approche de distribution des données et du traitement. Il s'agit d'abord de définir un langage de requête avec de nouveaux opérateurs. Ce nouveau langage regroupe tous les opérateurs de requête présents dans SPARQL 1.1, trois (3) nouvelles variantes de fenêtre (en plus de celles proposées par C-SPARQL) parmi lesquelles une fenêtre hybride (fenêtre qui combine des intervalles de temps et de valeurs) et une clause permettant de définir une fréquence de rafraîchissement de données statiques. Notre extension de requêtes s'exécute sur un système implémenté sur une architecture distribuée comportant deux (2) principaux composants : le composant "offline" pour le partitionnement des requêtes des utilisateurs et le composant "temps réel" qui distribue et traite dynamiquement les graphes. La distribution des graphes RDF est basée sur les informations obtenues du partitionnement de la requête DRSS. Cette stratégie facilite la parallélisation du traitement en réduisant considérablement les informations (résultats intermédiaires) échangés entre les nœuds de traitement. En plus des modules de partitionnement de requête, de partitionnement dynamique de graphes RDF et de fenêtrage, le système DRSS inclut un module de réécriture des requêtes partageant des sous-structures (patterns de requête) communes et l'interrogation et le rafraichissent données statiques stockées dans des dépôt locaux ou distants. DRSS est multi-requêtes et le regroupement des patterns de requête communs dans une seule requête a permis une évaluation unique des parties communes à plusieurs requêtes ainsi que la non-duplication de sous-graphes de données.

Nous avons évalué notre approche en matière de scalabilité du système en fonction de la variation du nombre de nœuds de traitement, de la taille de fenêtre, du nombre de requêtes concurrentes et du nombre de sources de flux RDF interrogées. Les résultats obtenus confirment la scalabilité de notre stratégie de parallélisation et de distribution des données et du traitement.

10.1.5 5ème contribution : jointure entre graphes RDF statiques et dynamiques

Dans le chapitre 9, nous nous sommes attaqués à la problématique de croisement des données statiques et dynamiques. Les systèmes existants n'abordant pas suffisamment cette problématique avec les mêmes efforts que ceux concernant la gestion continue des flux RDF, nous avons approfondi le système DRSS en proposant JSS-RDF (Join between Streams and Stored RDF graphs). JSS-RDF est une approche et un système de gestion continue et distribuée des flux RDF combinés à des données statiques stockées dans des dépôts locaux ou distants. Pour cela, nous avons réalisé une séquence optimisée de jointure de données statiques et dynamiques en mémoire en utilisant les filtres de Bloom ainsi qu'une opération de jointure clé-valeur de l'ordre d'une seule interrogation en $\mathcal{O}(2)$. Nous avons utilisé les filtres de Bloom car ils permettent de représenter de grandes quantités d'éléments dans une mémoire plus petite ainsi que des tests d'appartenance (interrogation du filtre de Bloom) en $\mathcal{O}(1)$. Ainsi, notre approche accélère les opérations de jointure entre flux de sources différentes, entre données statiques de sources différentes et entre données statiques et dynamiques en réduisant considérablement les résultats intermédiaires générés en mémoire et les sous requêtes pré-calculés. Dans JSS-RDF, nous avons réutilisé l'approche de distribution des données et des requêtes de DRSS. Nous assurons la gestion de bout en bout des requêtes utilisateur à travers cinq (5) principaux composants pour la gestion des requêtes SPARQL continues, des données RDF statiques (importation et rafraichissement des dépôts), des flux de données RDF, de la synchronisation continue (pour la communication entre nœuds de traitement et l'échange de résultats intermédiaires) et le moteur de requête avec notre approche de jointure basée sur les filtres de Bloom.

Le mécanisme de jointure dans JSS-RDF présente de meilleures performances comparé à ceux utilisés dans les systèmes RSP centralisés qui prennent en compte la jointure entre données statiques et dynamiques. Cette évaluation est faite en matière de débit et de latence dans le traitement de la requête. Notre deuxième évaluation a révélé une scalabilté de JSS-RDF comparé au système de base. Enfin, nous avons évalué la proportion de faux positifs qui peuvent impacter l'exactitude des résultats. Nous avons ainsi déterminé l'erreur relative à considérer avec une variation de faux positifs afin de mieux garantir une exactitude des résultats.

10.2 Perspectives associées

Nos principales perspectives visent à étendre nos solutions de résumé de graphes RDF pour les rendre plus génériques de telle sorte qu'elles soient indépendantes de tout paramètre de contexte.

10.2.1 Exploration et élagage de graphes RDF

La première perspective associée à ce travail concerne l'exploration et l'élagage continus de flux de graphes RDF pour résoudre les requêtes SPARQL continues. Les framework Jena , RDF4J, RDFSuite, etc. utilisent des tables relationnelles pour stocker les triplets et effectuent également les jointures relationnelles sur ces triplets. Ceci provoque, entre autres inconvénients, la génération d'innombrables résultats intermédiaires dont on n'aura jamais besoin dans la résolution des requêtes. Ainsi, plus on a d'opérateurs de jointures plus on générera ces données "inutiles", plus on occupera de l'espace mémoire, plus on ralentira le traitement. Notre idée est de poursuivre l'approche de traitement continu et distribué de requêtes SPARQL en se passant catégoriquement des solutions telles que Jena, RDF4J, etc. et en adoptant une approche native d'exploration et d'élagage de graphes RDF

suivant les patterns présents dans les requêtes SPARQL.

Nous visons à proposer le système **G**$^{\text{ExP}}$ (**R**DF **G**raphs streams **E**xploration and **P**runing) qui présentera deux principales contributions pour un traitement continu et scalable de l'opération de jointure sur les données RDF statiques et dynamiques. Tout d'abord, nous introduirons un élagage basé requête de graphe RDF qui réalise un traitement de matching parallèle de sous graphes sans l'opération coûteuse de jointure de SPARQL. Cette approche élimine les résultats intermédiaires superflus et garantit que la collecte des nœuds (sujet ou objet) correspond toujours à ceux nécessaires à la procédure de jointure. Nous proposerons ensuite un nouveau planificateur heuristique qui fournit le plan d'exécution le plus optimisé basé sur le modèle élagué généré tout en minimisant les communications entre différents nœuds de traitement. L'approche sera mise en œuvre dans un prototype 3C (Continu, Combiné et Cohérent) et adaptée au contexte de la combinaison des flux RDF (contrainte de passage unique) et des graphes RDF stockés (millions de triplets stockés dans des dépôts locaux ou distants).

10.2.2 Application des fonctions d'oubli aux flux de données sémantiques

La volumétrie des données statiques telles que les données médicales, environnementales (GBIF [146, 147]), ou encore géographiques ainsi que le taux d'arrivée des flux de données tels que les réseaux sociaux, les données de capteurs ou de consommations électriques imposent de nouveaux paradigmes de résumé tenant compte des contraintes liés au temps de traitement et à l'espace de stockage limité. Dans une vue plus générale, le résumé vise à garder des informations compactes sur un flux de données dans son intégralité temporelle, sans envisager de problématique a priori. Des principaux travaux réalisés récemment ont permis de mettre en place des algorithmes de compression des flux RDF dans [117, 148], et nous avons proposé des extensions au système de gestion de flux de données dans C-SPARQL [48]. Nous visons à poursuivre ces deux travaux afin de proposer des fonctions d'oubli appliquées à des flux de données sémantiques. L'objectif est de spécifier et d'implémenter des fonctions d'oubli dans le cadre des données sémantiques statiques (base de connaissances) et dynamiques (flux), potentiellement dans un environnement distribué, en étendant le langage SPARQL afin de déterminer les données à garder à chaque instant et celles à "oublier" sur un ou plusieurs flux de données sémantiques de différentes natures (statiques ou dynamiques) en entrée, et arrivant à des vitesses différentes. Cette extension devra permettre d'historiser les flux de données avec une granularité variable en fonction du temps et en assurant un espace de stockage limité. Cette contribution consistera à la mise en place de fonctions d'oubli qui aboutiront à la construction de résumés généralistes sur flux de données RDF.

Bibliographie

[1] J.-P. Calbimonte, O. Corcho, and A. J. Gray, "Enabling ontology-based access to streaming data sources," in *The Semantic Web–ISWC 2010*. Springer, 2010, pp. 96–111.

[2] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," *Next generation data mining*, p. 212 :191–212, 2003.

[3] S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello, "Introducing rdf graph summary with application to assisted sparql formulation," *23rd International Workshop on Database and Expert Sytems Applications*, 2012.

[4] Y. Zhang, P. M. Duc, O. Corcho, and J.-P. Calbimonte, "Srbench : a streaming rdf/sparql benchmark," in *International Semantic Web Conference*. Springer, 2012, pp. 641–657.

[5] "Chiffres internet - 2017," https://www.blogdumoderateur.com/chiffres-internet/.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[7] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[8] L. Ma, W. Nutt, and H. Taylor, "Condensative stream query language for data streams," in *Proceedings of the eighteenth conference on Australasian database-Volume 63*. Australian Computer Society, Inc., 2007, pp. 113–122.

[9] K. Towne, Q. Zhu, C. Zuzarte, and W.-C. Hou, "Window query processing for joining data streams with relations," in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. IBM Corp., 2007, pp. 188–202.

[10] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream : the stanford stream data manager (demonstration description)," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 665–665.

[11] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah, "Telegraphcq : An architectural status report," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 11–18, 2003.

[12] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora : a new model and architecture for data stream management," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[13] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams : a new class of data management applications," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 215–226.

[14] A. Arasu, S. Babu, and J. Widom, "Cql : A language for continuous queries over streams and relations," in *Database Programming Languages*. Springer, 2004, pp. 1–19.

[15] A. Bolles, M. Grawunder, and J. Jacobi, "Streaming sparql-extending sparql to process data streams," *The Semantic Web : Research and Applications*, pp. 448–462, 2008.

[16] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "C-sparql : Sparql for continuous querying," 2009.

[17] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "Ep-sparql : a unified language for event processing and stream reasoning," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 635–644.

[18] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *The Semantic Web–ISWC 2011*. Springer, 2011, pp. 370–388.

[19] S. Komazec, D. Cerri, and D. Fensel, "Sparkwave : continuous schema-enhanced pattern matching over rdf data streams," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, 2012, pp. 58–68.

[20] D. Le-Phuoc, H. N. M. Quoc, C. Le Van, and M. Hauswirth, "Elastic and scalable processing of linked stream data in the cloud," in *International Semantic Web Conference*. Springer, 2013, pp. 280–297.

[21] J. Hoeksema and S. Kotoulas, "High-performance distributed stream reasoning using s4," in *Ordring Workshop at ISWC*, 2011.

[22] X. Ren and O. Curé, "Strider : A hybrid adaptive distributed rdf stream processing engine," in *International Semantic Web Conference*. Springer, 2017, pp. 559–576.

[23] S. Gillani, G. Picard, and F. Laforest, "Dionysus : Towards query-aware distributed processing of rdf graph streams." in *EDBT/ICDT Workshops*. Citeseer, 2016.

[24] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 411–422.

[25] A. Harth, J. Umbrich, A. Hogan, and S. Decker, "Yars2 : A federated repository for querying graph structured data from the web," *The Semantic Web*, pp. 211–224, 2007.

[26] O. Erling and I. Mikhailov, "Virtuoso : Rdf support in a native rdbms," in *Semantic Web Information Management*. Springer, 2010, pp. 501–519.

[27] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the mapreduce software framework : the shard triple-store," in *Programming support innovations for emerging distributed applications*. ACM, 2010, p. 4.

[28] M. Cai and M. Frank, "Rdfpeers : a scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 650–657.

[29] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi, "Atlas : Storing, updating and querying rdf (s) data on top of dhts," *Web Semantics : Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 271–277, 2010.

[30] M. F. Husain, P. Doshi, L. Khan, and B. M. Thuraisingham, "Storage and retrieval of large rdf graph using hadoop and mapreduce." *CloudCom*, vol. 9, pp. 680–686, 2009.

[31] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H 2 rdf+ : High-performance distributed joins over large-scale rdf graphs," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 255–263.

[32] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Adaptive partitioning for very large rdf data," *arXiv preprint arXiv :1505.02728*, 2015.

[33] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "Triad : a distributed shared-nothing rdf engine based on asynchronous message passing," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 289–300.

[34] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[35] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1894–1905, 2013.

[36] B. Wu, Y. Zhou, P. Yuan, L. Liu, and H. Jin, "Scalable sparql querying using path partitioning," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 795–806.

[37] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 517–528.

[38] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale rdf data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 4. VLDB Endowment, 2013, pp. 265–276.

[39] R. Wang and K. Chiu, "A graph partitioning approach to distributed rdf stores," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 2012, pp. 411–418.

[40] N. Fernández, J. Arias, L. Sánchez, D. Fuentes-Lorenzo, and Ó. Corcho, "Rdsz : an approach for lossless rdf stream compression," in *European Semantic Web Conference*. Springer, 2014, pp. 52–67.

[41] J. D. Fernández, A. Llaves, and O. Corcho, "Efficient rdf interchange (eri) format for rdf data streams," in *International Semantic Web Conference*. Springer, 2014, pp. 244–259.

[42] Y. Chabchoub, Z. Kazi-Aoul, A. F. Dia, and R. El-Sibai, "On the dependancies of queries execution time and memory consumption in c-sparql," in *Proceedings of the IADIS International Conference Applied Computing*. AC 2015 Proceedings, 2015.

[43] A. F. Dia, Z. Kazi-Aoul, A. Boly, and Y. Chabchoub, "C-sparql extension for sampling rdf graphs streams," in *Advances in Knowledge Discovery and Management*. Springer, 2018, pp. 23–40.

[44] A. F. Dia, Z. Kazi-Aoul, A. Boly, and E. Métais, "Drss : Distributed rdf sparql streaming," in *International Conference on Software Engineering Research, Management and Applications*. Springer, 2017, pp. 125–145.

[45] N. B. Déme, A. F. Dia, A. Boly, Z. Kazi-Aoul, and R. Chiky, "An efficient approach for real-time processing of rdsz-based compressed rdf streams," in *International Conference on Software Engineering Research, Management and Applications*. Springer, 2017, pp. 147–166.

[46] A. F. Dia, M. U. Togbe, A. Boly, Z. K. Aoul, and E. Metais, "Graph-oriented summary for optimized resource description framework graphs streams processing," vol. 12, no. 7, p. 2703, 2018. [Online]. Available : http://waset.org/abstracts/Computer-and-Information-Engineering

[47] A. F. Dia, Z. K. Aoul, A. Boly, and E. Métais, "Fast sparql join processing between distributed streams and stored rdf graphs using bloom filters," in *2018 12th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 2018, pp. 1–12.

[48] A. F. Dia, Z. Kazi-Aoul, A. Boly, and Y. Chabchoub, "Extension de c-sparql pour l'échantillonnage de flux de graphes rdf," *Revue des Nouvelles Technologies de l'Information*, 2015.

[49] T. Berners-Lee and M. Fischetti, *Weaving the Web*, 1st ed., Ed. HarperCollins Publishers, 1999.

[50] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 28–37, May 2001.

[51] *UNIFORM RESOURCE IDENTIFIER*, Wikipedia, September 29 2009.

[52] M. D. (W3C) and M. S. M. Corporation), *Identifiants de ressource internationalisée (IRI)*, Groupe de travail Réseau, Request for Comments : 3987, Catégorie : Standards Track, janvier 2005.

[53] K. Rohloff, M. Dean, I. Emmons, D. Ryder, and J. Sumner, "An evaluation of triple-store technologies for large data stores," ., .

[54] A. Seaborne and G. Manjunath, *SPARQL/Update : A language for updating RDF graphs*, Digital Media Systems Laboratory, HP Laboratories Bristol, HPL-2007-102, July 3, 2007.

[55] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Working Draft, Tech. Rep. [Online]. Available : http://www.w3.org/TR/rdf-sparql-query/

[56] C. O. chimezie@gmail.com, *SPARQL 1.1 Graph Store HTTP Protocol*, W3C® (MIT, ERCIM, Keio, Beihang), March 2013.

[57] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, p. 16, 2009.

[58] N. Koudas and D. Srivastava, "Data stream query processing," *In ICDE*, vol. 5, p. 1145, 2005.

[59] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope : a stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 647–651.

[60] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq : A scalable continuous query system for internet databases," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 379–390.

[61] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *International Conference on mobile Data management*. Springer, 2001, pp. 3–14.

[62] D. J. Abadi, W. Lindner, S. Madden, and J. Schuler, "An integration framework for sensor networks and data stream management systems," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 1361–1364.

[63] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying rdf streams with c-sparql," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 20–26, 2010.

[64] D. F. Barbieri, D. Braga, S. Ceri, and M. Grossniklaus, "An execution environment for c-sparql queries," in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 441–452.

[65] D. F. Barbieri, D. Braga, S. Ceri, E. D. VALLE, and M. Grossniklaus, "C-sparql : a continuous query language for rdf data streams," *International Journal of Semantic Computing*, vol. 4, no. 01, pp. 3–25, 2010.

[66] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus, "Continuous queries and real-time analysis of social semantic data with c-sparql," in *In Proceedings of Social Data on the Web Workshop at the 8th International Semantic Web Conference*. Citeseer, 2010.

[67] C. Y. Brenninkmeijer, I. Galpin, A. A. Fernandes, and N. W. Paton, "A semantics for a query language over sensors, streams and relations," in *Sharing Data, Information and Knowledge*. Springer, 2008, pp. 87–99.

[68] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in etalis," no. 4, 2012.

[69] ——, "Real-time complex event recognition and reasoning–a logic programming approach," *Applied Artificial Intelligence*, vol. 26, no. 1-2, pp. 6–57, 2012.

[70] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, "Linked stream data processing engines : Facts and figures," in *The Semantic Web–ISWC 2012*, 2012, pp. 300–312.

[71] C. Rete, "A fast algorithm for the many pattern/many object pattern matching problem," *Artificial Intelligence*, vol. 19, pp. 17–37, 1982.

[72] M. Perry, P. Jain, and A. P. Sheth, "Sparql-st : Extending sparql to support spatiotemporal queries," in *Geospatial semantics and the semantic web*. Springer, 2011, pp. 61–86.

[73] M. Koubarakis and K. Kyzirakos, "Modeling and querying metadata in the semantic sensor web : The model strdf and the query language stsparql," in *Extended Semantic Web Conference*. Springer, 2010, pp. 425–439.

[74] A. Rodriguez, R. McGrath, Y. Liu, and J. Myers, "Semantic management of streaming data," in *Proceedings of the 2nd International Conference on Semantic Sensor Networks-Volume 522*. CEUR-WS. org, 2009, pp. 80–95.

[75] J. Tappolet and A. Bernstein, "Applied temporal rdf : Efficient temporal querying of rdf data with sparql," in *European Semantic Web Conference*. Springer, 2009, pp. 308–322.

[76] F. Grandi, "T-sparql : A tsql2-like temporal query language for rdf." in *ADBIS (Local Proceedings)*. Citeseer, 2010, pp. 21–30.

[77] B. Motik, "Representing and querying validity time in rdf and owl : A logic-based approach," *Web Semantics : Science, Services and Agents on the World Wide Web*, vol. 12, pp. 3–21, 2012.

[78] D. Dell'Aglio, J.-P. Calbimonte, M. Balduini, O. Corcho, and E. Della Valle, "On correctness in rdf stream processor benchmarking," in *The Semantic Web–ISWC 2013*, 2013, pp. 326–342.

[79] B. Csernel, "Résumé généraliste de flux de données," Ph.D. dissertation, Télécom Paris-Tech, 2007.

[80] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," *Data Engineering, 2004. Proceedings. 20th International Conference on*, p. 350–361, 2004.

[81] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," *Proceedings of the 29th international conference on Very large data bases*, vol. 29, p. 309–320, 2003.

[82] N. Gabsi, F. Clérot, and G. Hébrail, "Interrogation des résumés de flux de données," *Revue des Nouvelles Technologies de l'Information, Extraction et Gestion des Connaissances*, pp. RNTI–E–19 : 247–254, 2010.

[83] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, p. 11(1) :37–57, 1985.

[84] ——, "Faster methods for random sampling," *Communications of the ACM*, p. 27(7) : 703–718, 1984.

[85] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," *Proceedings of the 32nd international conference on Very large data bases*, p. 607–618, 2006.

[86] B. Babcock, M. Datar, , and R. Motwani, "Sampling from a moving window over streaming data," *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, p. 633–634, 2002.

[87] R. Gemulla and W. Lehner, "Sampling time-based sliding windows in bounded space," *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, p. 379–392, 2008.

[88] R. Gemulla, W. Lehner, and P. J. Haas, "A dip in the reservoir : Maintaining sample synopses of evolving datasets," *Proceedings of the 32nd international conference on Very large data bases*, p. 595–606, 2006.

[89] S. Chaudhuri, R. Motwani, and V. Narasayya, "On random sampling over joins," *ACM SIGMOD Record*, vol. 28, p. 263–274, 1999.

[90] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, p. 40–51, 2003.

[91] R. Féraud, F. Clérot, and P. Gouzien, "Sampling the join of streams," *Classification as a Tool for Research*, p. 307–314, 2010.

[92] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," *ACM SIGMOD Record*, vol. 28, p. 287–298, 1999.

[93] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "How to summarize the universe : Dynamic maintenance of quantiles," *Proceedings of the 28th international conference on Very Large Data Bases*, p. 454–465, 2002.

[94] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Record*, vol. 30, p. 58–66, 2001.

[95] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch : an efficient data clustering method for very large databases," *ACM SIGMOD Record*, vol. 25, p. 103–114, 1996.

[96] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," *Proceedings of the 29th international conference on Very large data bases*, vol. 29, p. 81–92, 2003.

[97] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *ACM Sigmod Record*, vol. 25, no. 2. ACM, 1996, pp. 294–305.

[98] V. Poosala, V. Ganti, and Y. E. Ioannidis, "Approximate query answering using histograms," *IEEE Data Eng. Bull.*, vol. 22, no. 4, pp. 5–14, 1999.

[99] S. Guha, N. Koudas, and K. Shim, "Data-streams and histograms," *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, p. 471–475, 2001.

[100] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," *Internaational Conference on Very Large Databases (VLDB'98)*, 1999.

[101] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," *Proceedings of the 28th international conference on Very Large Data Bases*, p. 346–357, 2002.

[102] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, p. 31(2) :182–209, 1985.

[103] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, p. 13(7) :422–426, 1970.

[104] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, p. 101(6) :255–261, 2007.

[105] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache : a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, p. 8(3) :281–293, 2000.

[106] J. Han, Y. Cai, Y. Chen, G. Dong, J. Pei, B. Wah, and J. Wang, "Multi-dimensional analysis of data streams using stream cubes," *Data Streams*, p. 103–125, 2007.

[107] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, "Multidimensional regression analysis of time-series data streams," *Proceedings of the 28th international conference on Very Large Data Bases*, p. 323–334, 2002.

[108] B. Csernel, F. Clerot, and G. Hébrail, "Datastream clustering over tilted windows through sampling," *Knowledge Discovery from Data Streams*, p. 127, 2006.

[109] J. Émile Symphor, A. Mancheron, L. Vinceslas, and P. Poncelet, "Le fia : un nouvel automate permettant l'extraction efficace d'itemsets fréquents dans les flots de données," *EGC'08*, p. 157–168, 2008.

[110] L. Vinceslas, J. Émile Symphor, A. Mancheron, and P. Poncelet, "Spams, une nouvelle approche incrémentale pour l'extraction de motifs séquentiels fréquents dans les data streams," *EGC'09 : Extraction et Gestion des Connaissances*, p. 205–216, 2009.

[111] M. Zneika, C. Lucchese, D. Vodislav, and D. Kotzinos, "Summarizing linked data rdf graphs using approximate graph pattern mining." in *19th International Conference on Extending Database Technology, Mar 2016, Bordeaux, France. Proc. 19th International Conference on Extending Database Technology (EDBT)*, 2016.

[112] S. Gurajaday, S. Seuferty, I. Miliarakiy, and M. Theobald, "Using graph summarization for join-ahead pruning in a distributed rdf engine," *SWIM'14*, 2014.

[113] M. Aydar, S. Ayvaz, and A. Melton, "Automatic weight generation and class predicate stability in rdf summary graphs," *Kent State University, Department of Computer Science, 241 Math and Computer Science Building. Kent, OH 44240, USA.*

[114] Šejla ČebiriĆ, F. Goasdoué, and I. Manolescu., "Query-oriented summarization of rdf graphs," in *[Research Report] RR-8920, INRIA Saclay ; Université Rennes 1.<hal-01325900v3>*, 2017.

[115] S. Vijayakumar, Q. Zhu, and G. Agrawal, "Dynamic resource provisioning for data streaming applications in a cloud environment," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on.* IEEE, 2010, pp. 441–448.

[116] J. Cao, W. Zhang, and W. Tan, "Dynamic control of data streaming and processing in a virtualized environment," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 2, pp. 365–376, 2012.

[117] F. Belghaouti, A. Bouzeghoub, Z. Kazi-Aoul, and R. Chiky, "Échantillonnage de flux de données sémantiques : Une approche orientée graphe." in *EGC*, 2015, pp. 485–486.

[118] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[119] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[120] B. Babcock, M. Datar, and R. Motwani, "Sampling from a moving window over streaming data," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2002, pp. 633–634.

[121] S. Morris, A. Verville, and L. Vasseur, "Comprendre les connexions sociales dans les communautés : comment utiliser l'analyse des réseaux sociaux? guide pratique (traduit par e. lucia). alliance de recherche universités-communautés-défis des communautés côtières," 2014.

[122] B. Divjak and P. Peharda, "Social network analysis of study environment," *JIOS, VOL.34, NO.1 UDC 378.147 :378.18*, 2010.

[123] E. Stattner, "Contributions à l'étude des réseaux sociaux : propagation, fouille, collecte de données," Ph.D. dissertation, Université des Antilles-Guyane, 2012.

[124] C. Ducruet, "Les mesures locales d'un réseau," *halshs-00546814v2*, 2010.

[125] ——, "Les mesures globales d'un réseau," *halshs-00541902*, 2010.

[126] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, "Rdf compression : basic approaches," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 1091–1092.

[127] A. K. Joshi, P. Hitzler, and G. Dong, "Logical linked data compression," in *Extended Semantic Web Conference*. Springer, 2013, pp. 170–184.

[128] J. Urbani, J. Maassen, N. Drost, F. Seinstra, and H. Bal, "Scalable rdf data compression with mapreduce," *Concurrency and Computation : Practice and Experience*, vol. 25, no. 1, pp. 24–39, 2013.

[129] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto, "Compressed k2-triples for full-in-memory rdf engines," *arXiv preprint arXiv :1105.4004*, 2011.

[130] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary rdf representation for publication and exchange (hdt)," *Web Semantics : Science, Services and Agents on the World Wide Web*, vol. 19, pp. 22–41, 2013.

[131] N. Fernández, J. Arias, L. Sánchez, D. Fuentes-Lorenzo, and Ó. Corcho, "Rdsz : an approach for lossless rdf stream compression," in *European Semantic Web Conference*. Springer, 2014, pp. 52–67.

[132] J. D. Fernández, A. Llaves, and O. Corcho, "Efficient rdf interchange (eri) format for rdf data streams," in *International Semantic Web Conference*. Springer, 2014, pp. 244–259.

[133] N. Garg, *Apache Kafka*. Packt Publishing Ltd, 2013.

[134] A. ZooKeeper, "What is zookeeper," 2014.

[135] "Apache storm," http://storm.apache.org/.

[136] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4 : Distributed stream computing platform," in *2010 IEEE International Conference on Data Mining Workshops*. IEEE, 2010, pp. 170–177.

[137] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[138] Y. Guo, Z. Pan, and J. Heflin, "Lubm : A benchmark for owl knowledge base systems," *Web Semantics : Science, Services and Agents on the World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[139] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[140] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo, "A fine-grained evaluation of sparql endpoint federation systems," *Semantic Web*, vol. 7, no. 5, pp. 493–518, 2016.

[141] S. Dehghanzadeh, D. Dell'Aglio, S. Gao, E. Della Valle, A. Mileo, and A. Bernstein, "Approximate continuous query answering over streams and dynamic linked data sets," in *International Conference on Web Engineering*. Springer, 2015, pp. 307–325.

[142] S. Chun, J. Jung, X. Jin, S. Yoon, and K.-H. Lee, "Proactive replication of dynamic linked data for scalable rdf stream processing."

[143] M. I. Ali, F. Gao, and A. Mileo, "Citybench : A configurable benchmark to evaluate rsp engines using smart city datasets," in *International Semantic Web Conference*. Springer, 2015, pp. 374–389.

[144] A. Broder and M. Mitzenmacher, "Network applications of bloom filters : A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[145] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[146] T. Robertson, M. Döring, R. Guralnick, D. Bloom, J. Wieczorek, K. Braak, J. Otegui, L. Russell, and P. Desmet, "The gbif integrated publishing toolkit : facilitating the efficient publishing of biodiversity data on the internet," *PloS one*, vol. 9, no. 8, p. e102623, 2014.

[147] H. Saarenmaa, "Sharing and accessing biodiversity data globally through gbif," in *ESRI User Conf*. Citeseer, 2005.

[148] F. Belghaouti, A. Bouzeghoub, Z. Kazi-Aoul, and R. Chiky, "Patorc : Pattern oriented compression for semantic data streams," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2016, pp. 193–209.

Annexe A: requêtes DRSS

A.1 Requête Q1

```
Query q1 AS
SELECT ?eID ?obsID ?result ?sID
FROM STREAM <http://ex.org/obs> [RANGE EVENTS 5000]
WHERE {?eID zone ?zID . ?eID startTime "2014-01-01T00 :00 :00.000+01 :00" .
    ?eID isProducedBy ?sID .
    ?eID hasValue ?result .
    ?eID hasObservation ?obsID .
    ?result type ?ObsID .
    ?result numValue 1.3e-1 .
    ?obsID inSector ?sectID ?obsID type observationValue .
    ?obsID unit "CubicMeterPerHour".}
```

A.2 Requête Q2

```
Query q2 AS
SELECT ?pressureSens ?value
FROM STREAM <ex.org/obs> [RANGE 60s STEP 20s]
FROM <http://exorg/staticrepo>
WHERE {?sector rdfs :label "Lou" .
    ?pressureSens ssn :onPlatform ?sector .
     ?event ssn :isProducedBy ?pressureSens ;
    ssn :hasValue ?observation.
     ?observation a "pressure" ;
    w :numValue ?value.}
```

A.3 Requête Q3

```
Query q3 AS
SELECT ?pressureSens ?value
FROM STREAM <ex.org/obs> [RANGE 60s STEP 20s]
FROM <http://exorg/staticrepo>
WHERE {?sector rdfs :label "Lou" .
    ?pressureSens ssn :onPlatform ?sector .
     ?event ssn :isProducedBy ?pressureSens ;
    ssn :hasValue ?observation.
     ?observation a "pressure" ;
    w :numValue ?value.}
```

A.4 Requête Q4

```
Query q4 AS
SELECT ?sensor ?value
FROM STREAM <http://ex.sh> [RANGE EVENTS BATCH 1000]
WHERE {?eventID ssn :hasValue observation ;
       ssn :isProducedBy ?sensor ;
      a ssn :SensorOutput.
       ?observation qudt :numValue ?value ;
       qudt :unit "http://qudt.org/unit#CubicMeterPerHour" ;
      a ssn :ObservationValue .}
```

A.5 Requête Q5

```
QUERY q5 AS
SELECT ?aSub ?anObj (count( distinct ?subject) as ?count)
FROM STREAM <http://example.org/observations1> [RANGE EVENTS 10000]
FROM NAMED <http://repository.org/biblio.rdf>
FROM STREAM <http://example.org/observations2> [RANGE 1m STEP 30s]
FROM NAMED <http://repository.org/book.rdf>
WHERE {?subject weather :type ?object .
       ?aSub sens-obs :aProp ?anObj ;
       sens-obs :anotherProp ?anotherObj ;
         { select ?name (count( distinct ?object) as ?count2)
         FROM STREAM <http://example.org/observations2> [RANGE 2m STEP 30s]
         FROM STREAM <http://example.org/observations3> [RANGE 10m TUM-
BLING]
         FROM NAMED <http://repository.org/biblio.rdf>
         WHERE {?object weather :name ?name ;
           sens-obs :aProp ?aName .}
         GROUP BY ?name }}
GROUP by ?aSub ?anObj }
```

Annexe B: requêtes SPARQL dans JSS-RDF

[Requêtes SRBench réadaptées (QX).]

B.1 Requête Q8

```
SELECT (MIN( ?temperature) AS ?minTemperature)
       (MAX( ?temperature) AS ?maxTemperature)
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
WHERE {?sens om-owl :processLocation ?sensLocation ;
              om-owl :generatedObs ?obs .
?sensLocation wgs84_pos :alt "%Altitude%"^^xsd :float ;
              wgs84_pos :lat "%Latitude%"^^xsd :float ;
wgs84_pos :long "%Longitude%"^^xsd :float.
?obs om-owl :observedPrty weather :_AirTemp ;
          om-owl :result [om-owl :floatValue ?temp].}
GROUP BY ?sens
```

B.2 Requête Q10

```
SELECT DISTINCT ?lat ?long ?alt
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
WHERE {?sensor om-owl :generatedObservation [a weather :SnowfallObservation] .
?sensor om-owl :processLocation ?sensorLocation .
?sensorLocation wgs84_pos :alt ?alt ;
              wgs84_pos :lat ?lat ;
              wgs84_pos :long ?long .}
```

B.3 Requête Q11

```
SELECT DISTINCT ?sensor
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
WHERE {?sensor om-owl :generatedObservation ?observation ;
    om-owl :hasLocatedNearRel [om-owl :hasLocation ?nearbyLocation].
?observation a ?observationType ;
    om-owl :observedProperty ?observationProperty ;
```

```
    om-owl :result [ om-owl :floatValue ?value ].
    SELECT AVG( ?value2) AS ?avgValue
    WHERE { ?sensor2 om-owl :generatedObservation ?observation2 ;
          om-owl :hasLocatedNearRel [om-owl :hasLocation ?nearbyLcation2].
        FILTER ( sameTerm( ?nearbyLocation, ?nearbyLocation2))
        ?observation2 a ?observationType ;
          om-owl :observedProperty ?observationProperty ;
          om-owl :result [om-owl :floatValue ?value2].
FILTER ( ABS( ?value - ?avgValue) / ?avgValue> "0.10"^^xsd :float)
}}
```

B.4 Requête Q12

```
SELECT ?name (AVG( ?temperature) AS ?avgTemperature) (AVG( ?humidity) AS ?avgHumidity)
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
FROM REPOSITORY <GeoNames>
WHERE { ?sensor om-owl :generatedObservation ?temperatureObservation ;
    om-owl :generatedObservation ?humidityObservation ;
    om-owl :hasLocatedNearRel [om-owl :hasLocation ?nearbyLocation].
?temperatureObservation om-owl :observedProperty weather :_AirTemperature ;
     om-owl :result [ om-owl :floatValue ?temperature ] .
?humidityObservation om-owl :observedProperty weather :_RelativeHumidity ;
    om-owl :result [om-owl :floatValue ?humidity].
  { SELECT ?name
  WHERE { ?nearbyLocation gn :featureClass ?featureClass ;
      gn :name | gn :officialName ?name ;
      gn :population ?population.
    FILTER ( ?population > 15000 && REGEX( ?featureClass, "P" , "i"))} } UNION { SELECT ?name
  WHERE ?nearbyLocation gn :parentFeature+ ?parentFeature.
    ?parentFeature gn :featureClass ?parentClass ;
      gn :name | gn :officialName ?name ;
      gn :population ?parentPopulation.
  FILTER ( ?parentPopulation > 15000 && REGEX( ?parentClass, "P" , "i"))}}}
GROUP BY ?name
```

B.5 Requête Q14

```
SELECT DISTINCT ?airportName ?lat ?long
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
FROM REPOSITORY <GeoNames>
WHERE { ?airport gn :featureClass ?airportClass ;
    wgs84_pos :lat ?lat ;
    wgs84_pos :long ?long ;
    gn :name|gn :officialName ?airportName ;
    gn :parentFeature+ ?city.
 ?city gn :featureClass ?cityClass.
```

```
 FILTER ( REGEX( ?airportClass, "S.AIRP" , "i") &&
    REGEX( ?cityClass, "P" , "i"))
 ?sensor om-owl :generatedObservation ?observation ;
    om-owl :hasLocatedNearRel [om-owl :hasLocation ?city].
 ?observation om-owl :procedure ?sensor ;
    a weather :VisibilityObservation ;
    om-owl :result [om-owl :floatValue ?value].
 FILTER ( ?value < "10"8sd :float)
 UNION { ?observation om-owl :procedure ?sensor ;
    a weather :RainfallObservation ;
    om-owl :result [om-owl :floatValue ?value ].
 FILTER ( ?value > "30"^^xsd :float)}
 UNION { ?observation om-owl :procedure ?sensor ;
    a weather :SnowfallObservation.}}
```

B.6 Requête Q15

```
SELECT ?lat ?long ?alt (AVG( ?windSpeed) AS ?avgWindSpeed)
FROM STREAM <LObD>
  [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
FROM REPOSITORY <DBpedia>
WHERE {
 ?observation a weather :WindspeedObservation ;
    om-owl :procedure ?sensor ;
    om-owl :result [om-owl :floatValue ?windSpeed].
 ?sensor om-owl :processLocation ?sensorLocation.
 ?sensorLocation wgs84_pos :alt ?alt ;
    wgs84_pos :lat ?lat ;
    wgs84_pos :long ?long.
 ?hurricane rdf :type/rdfs :subClassOf* yago :Hurricane111467018 ;
    dbpprop :1MinWinds ?hurricaneWindSpeed.
FILTER( ?windSpeed > ?hurricaneWindSpeed)}
GROUP BY ?lat ?long ?alt
```

B.7 Requête Q17

```
SELECT ?damage
FROM STREAM <LObD> [RANGE 60s TUMBLING]
FROM REPOSITORY <LSM>
FROM REPOSITORY <GeoNames>
FROM REPOSITORY <DBpedia>
WHERE { ?observation a weather :WindspeedObservation ;
    om-owl :procedure ?sensor ;
    om-owl :result [ om-owl :floatValue ?windSpeed].
 FILTER ( ?windSpeed >= "74"^^xsd :float)
 ?sensor om-owl :hasLocatedNearRel [om-owl :hasLocation ?nearbyLocation].
 ?hurricane dbpprop :areas [foaf :name ?areaName] ;
```

```
    rdf :type/rdfs :subClassOf* yago :Hurricane111467018 ;
    dbpprop :damages ?damage.
 ?nearbyLocation gn :parentFeature* ?area.
 ?area gn :name|gn :officialName ?areaName.}
```