Knowledge graph management and streaming in the context of edge computing

Weiqin Xu

Université Gustave Eiffel
Laboratoire d'Informatique Gaspard Monge (UMR 8049)

Knowledge Graph Management and Streaming in the Context of Edge Computing

Presented by Weiqin XU

PhD thesis in Computer science

September 2021

Fatiha Saïs	Grade	Reporter
Chan Le Duc	Grade	Reporter
Philippe Calvez	Grade	Co-supervisor
Olivier Curé	Grade	Supervisor

Contents

List of Figures

List of Tables

Acknowledgement

First and foremost, thanks to my parents, for their support and comprehension that encourage me to finish this thesis successfully.

I would like to express my sincere gratitude to my supervisors Prof. Olivier CURÉ and Dr. Philippe CALVEZ, for their invaluable guidance throughout the research. Their optimism and motivation have deeply inspired me. I've learned much from Olivier during my PhD study, not only on how to do good research, but also on the attitude to life, to work. His passion for knowledge influences me a lot. It's really a pleasure to work with him. Philippe and I have participated in ISWC 2019 in New Zealand, his humor impressed me a lot during that week, the experience of working with him is unforgettable.

My thanks also goes to Prof. Fatiha SAIS and Prof. Chan LE DUC for their acceptance to be members of my thesis committee, thanks for their sacrificing time on the lecture of my thesis and for their kind suggestions for my thesis.

I will also thank all my friends for their accompany during these three years, it would be hard without them, especially in the situation of covid-19. I will never forget the voyages, the barbecues, the parties by the Seine with them. The fun we have had will always give me the confidence to face challenges and difficulties throughout my life.

Last but not the least, I thank Antoine MEYER, Claire DAVID, Maxime THOUVENOT, Nadime FRANCIS, Samuele GIRAUDO and all the other colleagues that have helped me to adapt to course-teaching during my thesis, their patience and advice have encouraged me to overcome difficulties in teaching as a non-native French speaker. Moreover, I'm so grateful that all the colleagues around are so friendly and kind, this helps me to dive into research without distraction.

Résumé

Le Edge Computing propose de répartir le calcul et le stockage des données au plus près des sources de données d'origine. Cette technologie devient une tendance importante dans l'informatique. Ceci est principalement dû à l'émergence de l'Internet des objets (IoT) et de son ensemble d'appareils compacts, *e.g.*, capteurs, actionneurs ou passerelles, dont les capacités de calcul et de stockage ne cessent de croître. Différente du Cloud Computing, qui cible les grands centres de données, la stratégie de distribution des calculs du Edge Computing peut potentiellement réduire la pression du réseau et tirer pleinement parti de la puissance de calcul des périphériques à la bordure du réseau.

Afin de prendre en charge le traitement intelligent des données à la périphérie du réseau, une stratégie de représentation des connaissances est nécessaire. En 2021, les technologies appartenant au Web sémantique sont suffisamment matures et robustes pour apporter de l'intelligence au Edge computing. Ces technologies correspondent au modèle de données RDF (Resource Description Framework), aux langages d'ontologie RDFS (RDF Schema) et OWL (Web ontology Language), à leurs services de raisonnement associés, au langage de requête SPARQL. La pierre angulaire d'une telle approche est un système de gestion de base de données RDF compatible avec les périphériques Edge. Cependant, la plupart des systèmes de base de données du type RDF sont conçus pour des serveurs puissants ou le Cloud Computing. Ces systèmes doivent, en partie, leur efficacité à des stratégies d'indexation coûteuses, c'est-à-dire basées sur des indices multiples.

Dans le contexte du Edge computing, caractérisé par une empreinte mémoire et une puissance de calcul relativement limitées, il n'est pas raisonnable d'utiliser l'un de ces systèmes de base de données RDF. Par conséquent, un nouveau type de RDF store est nécessaire. Dans ce travail, nous considérons que certaines de ses fonctionnalités doivent être une approche en mémoire, une empreinte mémoire faible pour le système et ses données, des techniques d'optimisation des requêtes adaptées pour rendre le traitement des requêtes aussi rapide que possible. De plus, le raisonnement au moment de l'exécution des requêtes et le traitement des flux sont requis par plusieurs des cas d'utilisation que nous avons identifiés dans des situations réelles.

Dans le but de compresser les données RDF tout en maintenant la vitesse d'interrogation, nous utilisons abondamment les structures de données succinctes (SDS - Succinct Data Structure) pour bénéficier simultanément de sa compression de données et de sa vitesse élevée de récupération des données. Cela nous aide à obtenir un RDF store, nommé SuccinctEdge, compact auto-indexé qui ne nécessite pas d'opération de décompression. Notre approche de traitement des requêtes est adaptée à notre agencement de stockage et aux opérations SDS standard, à savoir `access`, `rank` et `select`. Nous prouvons la capacité de notre approche par une évaluation approfondie.

Afin d'aider à l'accélération du raisonnement RDFS, nous avons conçu notre système en utilisant une stratégie d'encodage sémantique nommée LiteMat. Ce schéma d'encodage, qui a été développé et est maintenu par notre équipe de recherche, a été étendu dans cette thèse de doctorat pour prendre en charge l'héritage multiple, les propriétés transitives et inverses. Il étend ainsi le pouvoir expressif des ontologies adressées.

Dans les cas d'utilisation réels de l'IoT, les données proviennent généralement en continu de capteurs ou d'actionneurs. Pour résoudre ce problème, une extension de Succinct-Edge a été conçue pour gérer ces données en streaming. Cette extension inclut une structure de données supplémentaire dans notre système de base de données RDF pour traiter les données numériques avec des agrégations temporelles et un processeur d'extension streaming SPARQL adapté pour permettre l'interrogation des données captées sous forme de flux. Avec l'aide de cette structure de données supplémentaire et d'un processeur de requêtes adapté, nous pouvons facilement interroger le graphe RDF dynamique par une requête SPARQL acceptant les flux. Cependant, l'exécution d'une requête sur un graphe dynamique peut imposer de nombreuses recherches répétives sur le graphe, ce qui peut fortement ralentir le système. Afin de résoudre ce problème, nous séparons une requête en une partie dynamique et une partie statique. Le résultat de la partie statique est calculé une seule fois et stocké pendant toute la durée du traitement continu de la requête. Concernant pour la partie dynamique, le résultat obtenu est combiné avec le résultat de la partie statique pour générer le résultat final de chaque exécution de requête. Nous prouvons que notre système d'extension de streaming est à faible latence et à haut débit avec de bonnes propriétés de robustesse et de correction.

Abstract

Edge Computing proposes to distribute computation and data storage closer to original data sources. This technology is becoming an important trend in IT. This is mainly due to the emergence of the Internet of Things (IoT) and its set of compact devices, *e.g.*, sensors, actuators or gateways, whose computing and storing capacities are ever-increasing. Different from Cloud Computing, which targets large data centers, Edge Computing's computation distribution strategy can potentially reduce network pressure and make full use of the computation power of edge devices.

In order to support smart data processing at the edge of the network, a knowledge representation strategy is needed. In 2021, technologies belonging to the so-called Semantic Web are mature and robust enough to bring intelligence to Edge computing. These technologies correspond to the RDF (Resource Description Framework) data model, the RDFS (RDF Schema) and OWL (Web ontology Language) ontology languages and their associated reasoning services, the SPARQL query language. A cornerstone of such an approach is an Edge device compliant RDF database management system. However, most RDF stores are designed for powerful servers or Cloud Computing. These systems partly owe their efficiency to costly indexing strategies, *i.e.*, based on multiples indexes.

In the context of Edge Computing, characterised by relatively limited memory footprint and computing power, it is not reasonable to use any of these RDF stores. Hence, a novel kind of RDF store is needed. In this work, we consider that some of its features must be an in-memory approach, low-memory footprint for both the system and its managed data, adapted query optimization techniques to make query processing as fast as possible. Moreover, reasoning at query run-time and stream processing are required by several of the use cases that we have identified in real-world situations.

For the aim of compressing RDF data while maintaining querying speed, we make an extensive use of Succinct Data Structure (SDS) data structures to benefit from its data compression and high data retrieving speed simultaneously. This help us to design a self-indexed compact RDF store, named SuccinctEdge, which does not require decompression operation. Our query processing approach is adapted to our storage layout and to standard SDS operations, namely `access`, `rank` and `select`. We prove the efficiency of our approach with a thorough evaluation.

In order to help the acceleration of RDFS reasoning, we have designed our system based on a semantic-aware encoding strategy named LiteMat. This encoding scheme, which has been developed and is maintained by our research team, has been extended in this PhD thesis to support multiple inheritance, transitive and inverse properties. It thus extends the expressive power of addressed ontologies.

In real IoT use cases, data are usually continuously coming from sensors or actuators. To address this issue, an extension of SuccinctEdge has been designed to handle those streaming data. This extension includes an extra data structure in our RDF store to process numeric data with time-based aggregations and an adapted SPARQL streaming extension processor to permit the querying of streaming data. With the help of this extra data structure

and the adapted query processor, one can easily query the dynamic RDF graph with streaming compliant SPARQL queries. However, query execution on a dynamic graph may have many repeating graph searches, which may heavily impact the performance of the system. In order to solve this problem, we separate a query into dynamic part and static part. The result of the static part is computed once and stored all along the duration of the continuous query processing. Concerning the dynamic part, the corresponding result is combined with the static part result to generate the final result of each query execution. We prove that our streaming extension system is of low latency and of high throughput with good robustness and correctness properties.

Chapter 1

Introduction

La volonté trouve, la liberté choisit.
Trouver et choisir, c'est penser.

– Victor Hugo

1.1 Introduction

With the explosion of information technology in the late 20th century, the Internet has become an important part of our daily life. In order to facilitate communications through the Internet, the World Wide Web (WWW), which contains a set of widely established standards, is designed to guarantee interoperability with the Internet at various levels[15]. The first generation of the WWW was aiming at human direct accessibility. With the appearance of the Semantic Web[6], the goal is now to make the Web data machine process-able. Further work is needed to complete this vision even if many standards and good practices are known and available.

This need for web data processing is especially true with the emergence of the Internet of Things (IoT) where small connected devices, *e.g.*, sensors and actuators, are blooming. In this context, one can question about the interactions of the IoT and the Semantic Web as it is clear that the latter can help in managing "smartly" the data produced by the former.

Considering the IoT, the wide installation and running of small devices causes an explosion of the amount of produced data. Until now, this data has generally been transferred to a central powerful machine or to a set of machines to get it processed. We can consider that such an approach is no longer tenable for ecological, performance and financial reasons. Intuitively, round-trips of information on the network are consuming a lot of energy and prevent fast decision making. Moreover, either buying and maintaining or renting on the cloud powerful machines has its economical cost that could be prevented when some computations can be performed on small devices which are already installed and accessible.

The small devices reside at the Edge of the network. And data processing performed at the Edge is usually denoted Edge Computing. This concept gradually enters into the view of researchers in this domain. This proposition aims at processing data where it is generated in order to reduce data transmission in the network.

It then seems obvious that combining Semantic Web and Edge Computing may become an interesting trend where smart, autonomous decisions could be taken at the edge of the network. Very few research work have investigated the interface of these two computer science domains. Nevertheless, we consider that, motivated by its potential in real world use cases, it will gain much more attention in the near future.

In this PhD thesis, we address some important questions toward an integration of Semantic Web approaches to Edge Computing. In the next section, we present three of these research questions.

1.2 Problem statement

When one considers the combination of the Semantic Web and Edge Computing, one of the first problem that comes into our mind is how to adapt RDF data management to run on Edge devices. Existing popular and production-ready RDF stores are mostly designed for centralized, powerful machines which generally have a large memory space and high computational power, *i.e.*, CPUs with multiple cores.

These machines can easily run an RDF store that depends on multiple indexes, *i.e.*, which requires a potentially large memory footprint. However, considering the limited resources on an Edge device, a compact RDF store is needed. Obviously, this should not come at the cost of query execution performance nor querying capacities. Moreover, to benefit from the "intelligence" of the Semantic Web, this RDF store should also support reasoning services. With the same consideration, an efficient and compact reasoning mechanism is required. This reasoning mechanism will support fast reasoning and, in the meantime, can handle sufficient logical expressiveness to support standard use cases. Another fact that should be taken into consideration is that the data generated by sensors are coming continuously. Nevertheless, traditional RDF systems can only handle static RDF graph which are not adapted to sensors' data.

Hence, our goal in this thesis is to design and implement an novel kind of RDF store with the following characteristics: 1) requiring a small memory footprint without losing of query processing performance, 2) supporting fast reasoning services with enough logical expressiveness, 3) processing data streams coming from sensors, actuators or gateways.

1.3 Research questions

In this section, with the problem stated previously, we will present the main research questions studied in this thesis. There are three main questions waiting to be answered:

RQ1: How to accelerate reasoning services with RDF Schema (RDFS) and some of its extensions?

Traditionally, RDFS reasoning is performed following the 14 RDF entailment rules presented in the RDF 1.1 Semantics W3C Recommendation[1]. In [24], the author is emphasizing that the entailment of RDFS is decidable, NP-complete in the general case and in P in the absence of blank nodes. These computational complexity results motivated the identification of RDFS fragments where reasoning is efficient, *i.e.*, preferably tractable, in the general case. One solution that tackles this issue and also addresses practical use cases is the work presented in [31]. In this work, ρdf is presented as a simple and minimal RDFS fragment that focuses specifically on the entailments that real-world situations are interested in. In fact, it mainly focuses on reasoning services based on the concept and property hierarchies.

In the context of an RDF store, a naïve approach to retrieve all subconcepts of a given concept C would be: i) search all direct subconcepts of C, *i.e.*, with $\{D1 \sqsubseteq C\}$, $D1$ should be in the answer set. ii) if the TBox also contains $\{D2 \sqsubseteq D1\}$ then $D2$ is an indirect subconcept of C and should also be in the answer set. 3) We repeat the second step until no more indirect subconcepts of C are discovered.

This approach is obviously quite costly. For example, consider that in the context of an RDF store, we have an ontology with concepts: $\{A, B, C, D\}$ together with their subsumption relationships (represented using a Description Logic[3] formalism): $\{B \sqsubseteq A, C \sqsubseteq B, D \sqsubseteq C\}$. In order to reason over this hierarchy, for instance asking whether $C \sqsubseteq A$ holds, we need to execute a query like

$$(?x, subClassOf, ?y) \bowtie (?y, subClassOf, ?z)$$

Moreover, in order to answer whether $D \sqsubseteq A$ holds, a query like

$$(?x, subClassOf, ?y) \bowtie (?y, subClassOf, ?z) \bowtie (?z, subClassOf, ?t)$$

should be computed. As we can see, the amount of work to infer a hierarchical relationship is exponential considering the concept hierarchy depth. With a deep hierarchical relationship, this will become a very costly approach.

In general, RDF stores are reasoning using either one of the two following approaches: materialization and query rewriting. The former proposes to perform the reasoning services before querying, which may a large memory footprint while the latter proposes to reason at query-runtime which may further slow down the query processing. In order to solve these problems, a encoding scheme LiteMat has been proposed. It can be considered a mix of the two approaches since it precomputes the inferred graphs of the concept and property hierarchies and attributes unique identifiers to each mapping entries. Then it uses this encoding to rewrite queries in need for some reasoning. Thus, it accelerates the reasoning services of ρdf and saves storage space at the same time. However, LiteMat is unable to handle the multi-inheritance cases and it can only support reasoning services with RDFS. How to improve LiteMat to handle multi-inheritance and how to extend LiteMat towards more expressive ontology languages are the two investigations that we are highlighting in this first set of research questions.

[1] https://www.w3.org/TR/rdf11-mt/

RQ2: How to build an RDF store adapted to small devices?

Most RDF stores have been designed to store large amount of RDF data. To manage them in an efficient manner, powerful devices equipped with large memory spaces and high computational power are needed. These compute resources help improve the performance of an RDF store by, for instance, adding meta-data such as indexes and by providing a query optimizer that will find an efficient execution plan for given SPARQL query, *e.g.*, , to find satisfying join and triple pattern execution orders. This does not correspond to the characteristics encountered in Edge devices.

In devices with limited resources, saving memory space is a primary consideration. But one can go a step further by using a compression method that reduces the required memory footprint. At the moment, many data compression methods impose a decompression step, *i.e.*, to recreate the original form of the data set, to perform any processing. It thus slows down the information retrieval process and implies additional computations during query processing. With the emergence of decompression-free data structures, it should be possible to propose a storage layout for RDF triples that is both compact and capable of supporting efficient query processing.

Additionally, finding the most optimal join order takes much calculation and requires some extra meta-data. With limited resources and limited accessible meta-data, how can we make a compromise between the most optimal join order and query optimization speed?

Finally, in order to benefit from the logical deduction ability of RDF with ontology, an efficient reasoning process should be implemented with an adaptive query processing, this reasoning service is quite important as inference can make a system intelligent in real use cases. Thus the third question is how to implement this reasoning service in the RDF store for small devices.

RQ3: How to extend the RDF store for edge devices to handle the streaming numeric data generated by sensors in real cases?

In real-world IoT use cases, small devices that perform some computations, like anomaly detection, are often connected to sensors. These sensors continuously generate various forms of measures, *e.g.*, numerical data like pressure in gas distribution network. Thus an RDF store designed for these small devices should have the ability to handle streaming data with an adapted query logic. In such cases, standard SPARQL queries are generally not sufficient and a continuous extension is required. The integration of such a component in the context of our Edge Computing RDF store is a research question by itself.

Based on this, the question of efficient query processing is emerging. In fact, in practical continuous IoT queries, a query is generally handling some static part, *e.g.*, sensor characteristic, and a dynamic part, *e.g.*, the latest measure. Our last question deals with how to make a distinction between the static and dynamic portion of a continuous query in order to reduce repeated information retrieving comes to be a question.

1.4 Contributions

The contributions we present in this section address the three research questions. In this section, we are going to list the contributions according to each of the three questions.

- For question RQ1, we have extended LiteMat toward RDFS++, which is an extension of RDFS together with `owl:sameAs` and `owl:transitiveProperty` , and to support multiple inheritance cases. More concretely, we have proposed an encoding solution of individuals involved in chain-like and tree-like transitive structures with an adapted query processing strategy. We have also proposed a simple ABox transformation together with an ID transformation while looking up dictionaries to support `owl:inverseOf` properties.
- For question RQ2, we have designed SuccinctEdge, a compact RDF store for Edge Computing with the support of rapid query processing and reasoning services. The basic data structure implemented in SuccinctEdge is based on Succinct Data Structure (SDS) which is a family of data structures that support efficient data compression and quick data retrieving. We have also implemented LiteMat into SuccinctEdge to support rapid reasoning services. As for query optimization, we have applied a left-deep join based on query graph analyses and heuristic to make a compromise between data optimization speed and query performances.
- For question RQ3, we have extended SuccinctEdge to query unbounded graph. This makes it possible to realize anomaly and risk detection with a continuous analysis of events received from sensors. During query processing, we have distinguished static and dynamic portion of a continuous query to avoid unnecessary computations of parts of a query execution. We have also conducted evaluations of our system in the aspects of correctness, robustness, latency and throughput.

1.5 Publications

In this section, we are going to list the publications corresponding to the contributions mentioned above of this thesis.

1. **Extending LiteMat toward RDFS++**. Olivier Curé, Weiqin Xu, Hubert Naacke, Philippe Calvez. LASCAR@ESWC 2019: 54-64
2. **LiteMat, an Encoding Scheme with RDFS++ and Multiple Inheritance Support**. Olivier Curé, Weiqin Xu, Hubert Naacke, Philippe Calvez. ESWC (Satellite Events) 2019: 269-284
3. **Multiple Inheritance of Ontology Concepts in a Semantic-Aware Encoding Scheme**. Weiqin Xu, Olivier Curé, Philippe Calvez. ISWC Satellites 2019: 105-108
4. **SuccinctEdge: A Succinct RDF Store for Edge Computing**. Weiqin Xu, Olivier Curé, Philippe Calvez. Proc. VLDB Endow. 13(12): 2857-2860

5. **Knowledge Graph Management on the Edge**. Weiqin Xu, Olivier Curé, Philippe Calvez. EDBT 2021: 229-240

1.6 Thesis organization

This thesis is organized in six chapters. The main research contributions are presented in Chapters 3, 4 and 5. More precisely, the chapters can be described as follows:

- In the second chapter, we will introduce some background knowledge corresponding to our research. This includes the Semantic Web, Resource Description Framework(RDF), the SPARQL query language which is designed for querying RDF data and some extensions of SPARQL. Then we will also introduce RDF Schema with its extensions and LiteMat, a encoding scheme to accelerate RDFS reasoning services. Some other important concepts are Edge Computing and RDF stores together with Succinct Data Structures, a compact data structure which plays an important role in our RDF system.
- In the third chapter, we will introduce our extensions of LiteMat, which is a semantic-aware encoding scheme. These extensions include i) how to solve the multi-inheritance problem of LiteMat, ii)how to support `owl:inverseOf` and iii)how to support `owl:transitiveProperty`. We will also illustrate query processing concerning the extended logic.
- Chapter four starts with a general presentation of SuccinctEdge (our prototype RDF store for Edge Computing) and a motivating example of our use case. Then we step into details of the system architecture and data structure especially designed to compress data while maintaining query speed. To fully improve the system performance, we propose a heuristic-based join order optimization and design some algorithms for fast retrieval of RDF data.
- Chapter five extends SuccinctEdge towards streaming processing. In order to adapt the system to streaming cases, we add some streaming data structures to support data aggregation and extend SPARQL to streaming query with the inspiration of C-SPARQL. To fully exploit the potential of this extension, we separate the query graph model into static and dynamic elements and have the static part calculated once to avoid repeating searches.
- In the last chapter, we conclude this research work and propose a list of some future work and perspectives.

Chapter 2

Background knowledge

The man who asks a question is a fool for a minute, the man who does not ask is a fool for life.

– Confucius

This chapter introduces several notions which are needed to understand the research and implementation work that is presented in this PhD thesis.

2.1 Introduction

In this chapter, we first present some important Semantic Web related W3C recommendations, *i.e.*, RDF, RDFS, OWL and SPARQL, as well as some stream processing SPARQL extensions. Then, we introduce an encoding strategy for RDFS knowledge bases, *i.e.*, LiteMat. In another section, we consider concepts pertaining to the IoT and more precisely to Edge Computing. Important research in the domain RDF data management is presented. This considers both RDF stores running on standard machines as well as systems especially designed to run on edge devices. Finally, we conclude this chapter with a presentation of Succinct Data Structures (SDS), a family of data structures that we are widely using in our SuccinctEdge RDF store system.

2.2 Semantic Web

The "Semantic Web" concept, which was first proposed by Tim Berners-Lee in 1999[5], aims at supporting intelligent services on the Web. The principal approach toward reaching this goal is to enable machines to automatically interpret Web data via defining vocabularies' semantics used to describe this data. This extension of the Web, which is also called "Web 3.0", is considered as the next generation of the Web. Unsurprisingly, it gained much attention from a certain research community. The Semantic Web is built around a set of W3C recommendations. For instance, it uses the Resource Description Framework (RDF) as the basic data modeling framework. A suitable query language (SPARQL Protocol and RDF Query Language - SPARQL) has been carefully designed to query RDF data. Moreover, in order to ensure reasoning mechanisms over RDF data, researchers proposed to add logical restrictions which are based on different logical languages, *e.g.*, based on Description Logic (DL)[3]. These restrictions form an ontology language which help to infer implicit consequences from explicit data represented in RDF.

2.2.1 Resource Description Framework

The birth of RDF can be traced back to 1997[39]. The design of was influenced by various predecessor languages and its first official appearance was in a W3C publication in 1999[21]. From that time, RDF has become a W3C recommendation for representing web resources and their metadata.

The RDF data model is based on the notion of triples of the form (subject, predicate, object). A set of triples forms a directed graph where subjects and objects are nodes and predicates are directed edges. Given three distinct sets U, B and L respectively corresponding to sets of URIs, blank nodes and literals, the signature of a triple (subject, predicate, object) is :

$$(U \cup B) \times (U) \times (U \cup B \cup L)$$

Intuitively, URIs are similar to Uniform Resource Locator(URL) but URIs don't necessarily correspond to actual web pages. Blank Nodes are just nodes in an RDF graph which

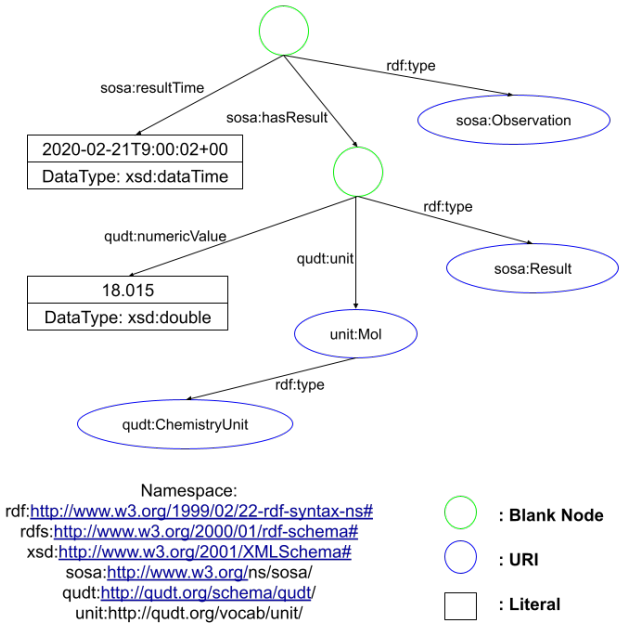

Figure 2.1: RDF graph example

don't have intrinsic names. Literals consist of a lexical form with a datatype URI (with a language tag in addition if the datatype URI is *http://www.w3.org/1999/02/22-rdf-syntax-ns#langString*). Literals are used for values such as strings, numbers, and dates. This signature indicates that a subject could be a URI or a Blank Node, a predicate could only be a URI and a object could be a URI, a Blank Node or a Literal.

Figure 2.1 shows an example of an RDF graph where blank nodes correspond to green circles, URIs to labelled black circles and literals are rectangles. This figure describes an observation of a sensor in an IoT use case. We have an observation which is generated at a certain time, this observation has a measure result which is 18.015 Mol. From this RDF graph, one can easily retrieve information using a query language such as SPARQL. We will present SPARQL in the next sub-section.

2.2.2 RDF query languages

Once the Semantic knowledge model standard is settled, an adapted query language is needed to retrieve information from an RDF graph. SPARQL[1] is the official W3C recommendation for querying RDF data.

SPARQL

SPARQL is a query language especially designed for RDF data. Similar to SQL, SPARQL also uses **SELECT...WHERE...** clauses to represent a searching query. Unlike SQL, the

[1]https://www.w3.org/TR/sparql11-query/

?t	?v	?u
"2020-02-21T9:00:02+00"^^xsd:dataTime	"18.015"^^xsd:double	unit:Mol

Table 2.1: Result table of query graph 2.2

content of the **WHERE** clause is a Basic Graph Pattern(BGP) which consists of multiples Triple Patterns(TPs). Hence, the graph represented in a WHERE clause is matched to an RDF graph to produce the query result. Each TP in a BGP is represented in the form of **(Subject Predicate Object)** where some elements can be replaced with variables. Given the following distinct sets of URIs U, Blank Nodes B, Literals L and variables V, the signature of a SPARQL TP:

$$Tp = (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$$

A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be bound to the variables and the result is an RDF graph equivalent to the subgraph[43]. Once all the matched subgraphs are found, we can output the answer set according to the required variables that are indicated in the **SELECT** clause.

From the SPARQL query presented in Query 2.1, we can construct the BGP of Figure 2.2. This graph can be matched to Figure 2.1. The query execution will yield some variable bindings for **?t**, **?v** and **?u** such as displayed in table 2.1.

```
SELECT ?t ?v ?u WHERE{
    ?a sosa:resultTime ?t .
    ?a sosa:hasResult ?b .
    ?b qudt:numericValue ?v .
    ?b qudt:unit ?u .
    ?u rdf:type qudt:ChemistryUnit .
}
```

Query 2.1: SPARQL query example

SPARQL extensions

Although SPARQL is relatively rich for querying RDF data, it lacks some features for querying streaming data, *i.e.*, unbounded data that are coming with at a certain velocity. Some research have been conducted in order to fill this gap and there exists some SPARQL extensions aiming at querying RDF streaming data, *e.g.*, Streaming SPARQL[7], CQELS[27] and C-SPARQL[4].

Streaming SPARQL extends the SPARQL language to allow the definition of time and count based windows over data stream[7]. It usually uses a **FROM STREAM** clause with

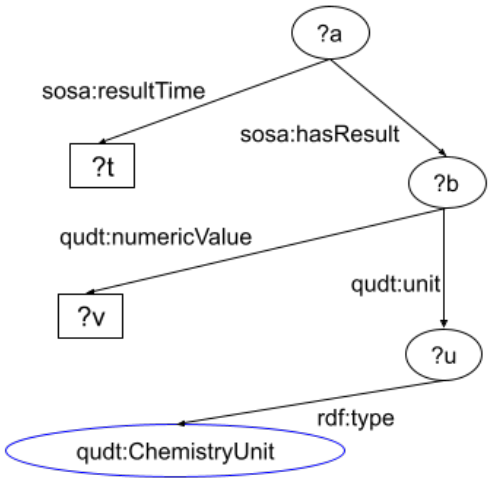

Figure 2.2: SPARQL graph pattern example

an URI to set an input as a data stream. The **RANGE** key word serves to give out the data window size and the **SLIDE** key word indicates the delay of each data window movement.

C-SPARQL also uses a **FROM STREAM** clause to indicate the input data stream and **RANGE** key word is also applied to indicate the data window size. However, C-SPARQL assumes a data window could be either sliding or tumbling. With the sliding mode, the data window is moved by the frequency indicated by **STEP** key word. The tumbling mode applies a consume-and-drop strategy which slides the data window by its range.

Unlike the two previous proposals, CQELS applies the **STREAM** key word to a portion of graph pattern. It is designed to handle queries that also depend on some static data. Each **STREAM** key word with a portion of graph pattern is called a Stream Graph Pattern. Like Streaming SPARQL, a Stream Graph Pattern can also use **RANGE** and **SLIDE** clauses to define a time-based stream window. Moreover, the **RANGE** key word can be replaced by **Triple**, **NOW** and **ALL**. **Triple** indicates a triple-based window whose size is determined by the number of triples, **NOW** refers to the triples at the current timestamp and **ALL** keeps all the triples.

2.2.3 Ontology languages

Reasoning is the functionality that provides some "intelligence" to RDF data,*i.e.*, based on a logic-based ontology, we can obtain implicit consequences from explicit data. The less expressive W3C ontology language is RDFS. But different fragments of Description Logic (DL) have been used to design some additional ontology languages for the Semantic Web. These different fragments became the standards for the OWL profiles.

In RDF stores, the reasoning process is usually addressed by two distinct approaches: materialization (a.k.a. graph saturation) and query rewriting (a.k.a. query reformulation). A combination of these solutions is possible. Materialization and query-rewriting are the

two representatives. Materialization suggests to pre-infer all the knowledge with the given ontology and store all the results in the database. This makes the query processing efficient as the only thing that needs to be done is searching into the database. However, a large amount of memory footprint is demanded by materialization because the newly generated data often takes much more space than the original data. With query rewriting, instead of pre-inferring all the knowledge, the system rewrites the original query using the knowledge's ontology. Query rewriting processes inferences online at query run-time. Although it reduces the overall memory footprint of the RDF store, this heavily increases query execution duration.

RDF Schema (RDFS)

In an RDF graph, an element denoted an instance could be the subject of a triple containing **rdf:type** as the predicate and the object of this triple indicates the class this instance belongs to. However, knowing the basic class of an instance is not enough in many cases, people also want to infer some extra terminological knowledge. In order to enable RDF graph with terminological knowledge, RDFS was equipped with reasoning rules to infer concept as well as property hierarchies. Moreover, RDFS is also capable to infer concept typing via the `rdfs:domain` and `rdfs:range` properties. Focusing on these inference rules correspond to addressing the ρdf RDFS fragments. RDFS also provides some other inference rules but they are less relevant in real-world use cases.

RDFS presents to be the most basic support of logical inference with RDF data, it is also the most widely used logical syntax in real use cases.

Ontology Web Language (OWL)

The Ontology Web Language, which is abbreviated as OWL, is a W3C recommendation for the modeling of ontologies. It provides a greater expressiveness compared to that of RDFS. The first version of OWL contains 3 sub-languages which are denoted as OWL Full, OWL DL and OWL Lite witht he former the more expressive and the latter the less expressive. OWL Full has a very high expressive power but is undecidable. It is thus almost never used in real-world use cases. The set of constructors in OWL DL is the same as in OWL Full but it adds constraints on the descriptions of concept and property. As a consequence, OWL DL is decidable and there exists some relatively efficient algorithms for reasoning with it. OWL Lite presents to be the least expressive language among the three. It was intended to be an easier language to use than OWL DL since its set of constructs is a sub set of OWL DL. Nevertheless, practically it was considered to be easier to use for a standard Semantic Web end-user/developper.

There also exists OWL 2, referring to the second version of OWL, which adds several new features to OWL[22]. OWL 2 contains 5 sub languages which are OWL 2 FULL, OWL 2 DL, OWL 2 EL, OWL 2 QL and OWL 2 RL. OWL 2 DL is compatible with OWL 1 DL with the support of some extra features. OWL 2 EL permits to process all standard inference types with polynomial time algorithms. It is designed for creating a very large ontology

with a limited amount of OWL features. OWL 2 QL is designed for data-driven applications, thus all of its standard inference types can also be processed with polynomial time algorithms. Different from OWL 2 El, OWL 2 QL allows conjunctive query answering by using conventional relational database[21]. OWL 2 RL is designed for using rule-based reasoning engine to make inferences in polynomial time. OWL 2 FULL contains both OWL 2 DL features and RDFS, it is the most expressive sub language of OWL 2. However, similar to OWL FULL, it is also undecidable.

OWL enhances RDF with a great power of logical deduction and the cost of reasoning is a direct consequence of the expressive power of the underlying ontology. Finding a trade-off between the computational complexity of reasoning and the expressive power of the ontology language has been an important aspect of the research on the Semantic Web.

RDFS++

Although RDFS offers an interesting entry into reasoning with RDF data, it is not enough for some real world use cases. Yet, OWL is often too powerful to handle these cases. In order to make a compromise between RDFS and OWL, ontology designers frequently mix some RDFS with some OWL constructs. This is often referred as RDFS++.

A common, but not strict (*i.e.*, the set of OWL constructs is open), description of RDFS++ is RDFS plus **owl:sameAs** and **owl:transitiveProperty**. **owl:sameAs** states that two URIs actually refer to the same thing, this property is usually used to map the identical instances in two different RDF graphs. **owl:transitiveProperty** indicates a property is transitive,*e.g.*, given three instances {a,b,c} and a transitive property {p} with the facts {(a,p,b), (b,p,c)}, a new triple {(a,p,c)} can be inferred based on the transitive principal.

RDFS++ can satisfy many use cases without reaching the complex reasoning services of OWL2 DL. This makes a light weight RDFS system with the support of logical artificial intelligence possible.

2.2.4 LiteMat, an encoding scheme for RDFS

Due to the string lengths of URIs and literals, most RDF stores adopt an encoding approach for the representation of RDF triples. Even with the extra payload of associated dictionaries, this approach generally yields a smaller memory footprint for the stored data. A typical approach is to attribute an arbitrary identifier to each RDF element, *e.g.*, URIs (including instances, concepts and properties), literals and blank nodes. LiteMat suggests to encode the identifiers of the concept (respectively property) hierarchy with a semantic aware method. A direct impact of this approach is to improve query reformulation at query run-time.

LiteMat[11] is an encoding scheme for RDF and RDFS data that offers a trade-off between materialization (of inferred triples) and query rewriting, in order to obtain complete result sets from queries requiring some inferences. It uses an integer interval based encoding for the Knowledge Graph (KG) elements that efficiently and effectively captures in a compressed manner cliques and hierarchical structures. In [11], researchers are apply-

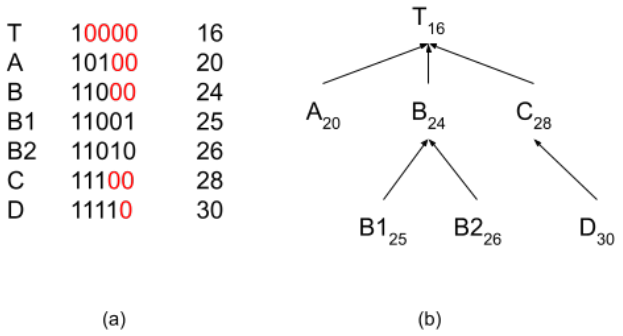

Figure 2.3: LiteMat Encoding example

ing this encoding to the ρdf[30] fragment of RDFS, *i.e.*, supporting inferences associated to `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range` properties. More recently, $Strider^R$[47] extended LiteMat to support the `owl:sameAs` property which is quite popular in the Linked Data community. Intuitively, a special encoding (a tuple containing clique ID and local ID) was applied to all individuals present in `owl:sameAs` cliques and a representative of this clique was automatically selected among them. Like LiteMat, the work presented in [48] models the concept and property subsumption hierarchies with an intelligent integer identification that is used to rewrite SQL queries in the ontology-based data access Quest system [49]. With the latest extension of LiteMat, researchers go further with a smart identification solution for individuals involved in `owl:sameAs` cliques as well as support for inverse properties and transitive structures taking the form of chains and trees.

The idea of LiteMat is to implement the hierarchical information into the encoding of an ontology element by using a bit-wise strategy. In Figure 2.3(a), we present LiteMat's encoding for the TBox in 2.3(b). In a first step, the assignment of an identifier, using a binary representation, for each concept is performed in a top-down recursive manner, *i.e.*, it starts by setting the top concept ($\top$) at 1, and proceeds level-wise on the element hierarchy until all leaves have been processed. Intuitively, for each concept denoted α, we count the number N of direct sub concepts (including α itself), *e.g.*, in our running example $\top$, we have N=4 for $\top$. At this level, $\lceil$ log(N) $\rceil$ provides the number of bits necessary to represent each sub concept. Then, these sub concepts (excluding α) are prefixed by the binary identifier of α and uniquely get a binary representation of a value $\in [1, N-1]$. For instance, the concept A is prefixed with '1' ($\top$'s identifier and is assigned the value 1 on 2 bits, i.e., '01', yielding the binary string '101'). Finally, a normalization step makes sure that all identifiers are encoded on the same binary string length. This is performed as follows: once all concepts have been encoded in the first step, we get the size L of the longest encoding string (*i.e.*, 5 in our running example). Then all concept identifiers with an encoding length lower than L are appended with '0' until their length reaches L. This normalization step is represented with red '0' in Figure 2.3(a). The last column of this figure provides the integer identifier corresponding to each concept.

LiteMat proposes an efficient query processing approach that takes advantage of the

semantic-aware encoding. In fact, whenever a query requires to reason over the concept or property hierarchies, the system simply introduces new variables that are filtered on the identifiers of the encoding scheme.

Consider the concept hierarchy of Figure 2.3(b) and the following BGP: {?x rdf:type B}. A frequent rewriting that ensures to retrieve an exhausitve answer set would be {?x rdf:type B} UNION {?x rdf:type B1} UNION {?x rdf:type B2}. Although costly on a query processing point of view, this rewriting also requires to access the ontology to discover sub concepts of B. In LiteMat, the system would identify that B has several sub concepts (only requiring an access to the concept dictionary), replace B with a new variable (*e.g.*, ?y) and add a FILTER clause that restricts the accepted values of this new variable. This restriction corresponds to an interval of integer values where the lower bound is the identifier of B and the upper bound is easily computing (*i.e.*, using 2 bit shift operations and an addition) from the identifier of B. This computation requires an identifier metadata stating the index on the bit string where the normalization has started. Considering that the identifiers of B, $B1$ and $B2$ are respectively 24, 25 and 26, the LiteMat rewriting would be: {?x rdf:type ?y. FILTER (?y$\geq$24 && ?y$<$28)}. This rewriting is also applied when sub property relationships are used. In general, the more complex the query, the more efficient the rewriting and its performance execution.

2.3 Edge Computing

Edge Computing is a concept related to the IoT. A general architecture of IoT is shown in Figure 2.4, different devices compose the three displayed layers. The first layer is the cloud, which usually consists of powerful machines with very large bandwidth internet connection. The second layer, Fog computing, has more devices than the cloud. These devices are generally less powerful than those of the cloud but still maintain enough computation power. The last layer is the Edge where the devices are even less powerful but the number of these devices is much larger than that of the other two layers. Many real use cases have implemented services for this layer, *e.g.*, smart wearable, smart home and smart city[41].

Other use cases such as smart farming and smart grids[16] are still the tip of the iceberg. It is estimated in [23] that at the end of 2021, there will be 10.07 billion IoT devices installed world wide. These devices are usually micro controllers, Arduinos[2] or Raspberry Pis[3]. Micro-controllers and Arduinos are very small computers which are energy-efficient but function-limited. They can not even run an operating system which is often required in many real use cases. Raspberry Pis are more powerful than the two previous and are relatively still resource-limited. With such a huge number of these small devices (billions over the whole IoT), transferring all the data to the cloud to get it processed will not only consume enormous calculation power in the cloud but also occupy large network bandwidth. With this background, the need to design systems and services for Edge Computing becomes an important consideration for the future of IT.

Edge Computing[2] corresponds to a processing paradigm that brings storage, manage-

[2]https://www.arduino.cc/

[3]https://www.raspberrypi.org/

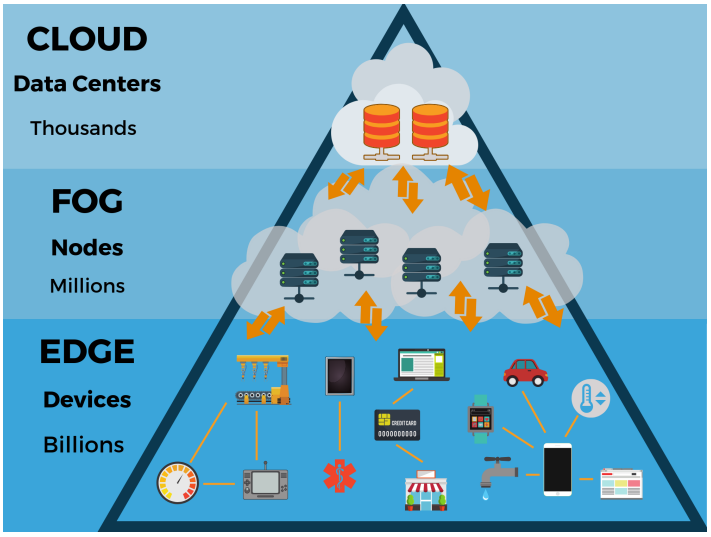

Figure 2.4: IoT general architecture

ment, and processing of huge amounts of data closer to the location where it needs to be performed. This location indicates the edge devices where the data is generated. Although Edge computing is not a new concept[18], it has gradually become a hop topic since the ubiquity of Cloud Computing. Unlike Cloud Computing which does all the computation in a data center equipped with powerful machines, Edge Computing distributes computation to a large number of small devices which are closer to data sources and only the minimum information is transferred through the network. This can efficiently reduce the network pressure which is usually the bottle neck of a linked system. More over, distributed computation can also reduce the server's pressure which, while doing a huge amount of calculation, is limited by the thermal problem and hardware technology. Especially, Edge Computing can also, in some aspects, contribute to data security where the Edge device may not have all the information of the whole system.

As such, this emerging trend complements a cloud computing approach by supporting the design of highly local context aware and responsive services, hence eliminating round trips to the Cloud, as well as mask cloud computing outages.

2.4 RDF stores

RDF has shown its ability to model Web resources, including Semantic Web ones. But it does not provide any hints on how to manage RDF data. That is issues such as how to store RDF data in efficient compressed manner and process SPARQL queries in a cost-effective way are still open. Many attempts have been made to answer these questions. Existing RDF stores are mostly designed for classical usage, which means they are designed for powerful machines with sufficient resources such as memory, computational power, *etc.* , or even for a cluster of machines. However, in recent years, more and more attention has been paid for RDF stores running on small devices which are resource-limited. In the following part of this section, we are going to present RDF stores within these two categories: classical RDF

stores and RDF stores for the edge.

2.4.1 Classical RDF stores

Classical RDF stores are often designed for powerful, centralized machines which aim to handle a large amount of data and to do very fast query processing. These machines always have enough memory space and other computing resources. That is the reason why they often maintain multiples indexes and uncompressed data structure which are persisted on disks. Some representatives of classical RDF stores are Jena TDB, RDF4J[8], RDF-3X[35][37], RDFox[34] and Hexastore[53] which are implemented for research use. Others such as Stardog, GraphDB, AllegroGraph and Oracle are mainly for commercial purpose. In order to adapt to the requirement of Cloud computing in recent years, some previously mentioned systems also offer the distributed version, *e.g.*, Stardog, AllegroGraph.

Classical RDF stores are able to handle very large data sets while in real case the data may come from widely spread small devices, thus collecting data from these devices may become the bottle neck of the system performance. Moreover, with the growth of the devices' number, the scalability and robustness of a knowledge graph management system based on classical RDF stores can potentially become a problem as the pressure of the network becomes very high. That is why in recent years, RDF stores for edge devices and edge computing in knowledge graph management gradually gained attention.

2.4.2 RDF stores for edge devices

RDF stores for edge devices often require a compressed data structure, few indexes with a light-weight query optimizer while still be able to handle sufficient-size data set. Some other requirements such as device-adapted storage strategy, streaming processing and energy efficiency are also taken into consideration in some existing systems. Typical RDF stores for edge devices are RDF4Led[51], Fed4Edge[38], μRDF Store[9] and Wiselib TupleStore[20]. RDF4Led is a disk-based RDF store that has been evaluated on Raspberry Pis. It relies on indexes and is equipped with rich query optimizer nevertheless it does not support reasoning services. Fed4Edge is a decentralized streaming RDF engine, the system's optimizations mainly focus on query federations. μRDF Store and Wiselib TupleStore are dedicated to micro-controllers where the optimizations are designed for extremely resource-limited environments. Each of these systems has its own consideration for their adapted environment and adapted optimizations for the environment, *e.g.*, data store structure, adapted data buffer for disk-based storage, *etc.* . We will more details on these systems in detail in Section 4.5.

2.5 Succinct Data structures

Succinct Data structures(SDS) represents a family of data structures that stores data in a compact way, but still allows some efficient data access operations without decompression.

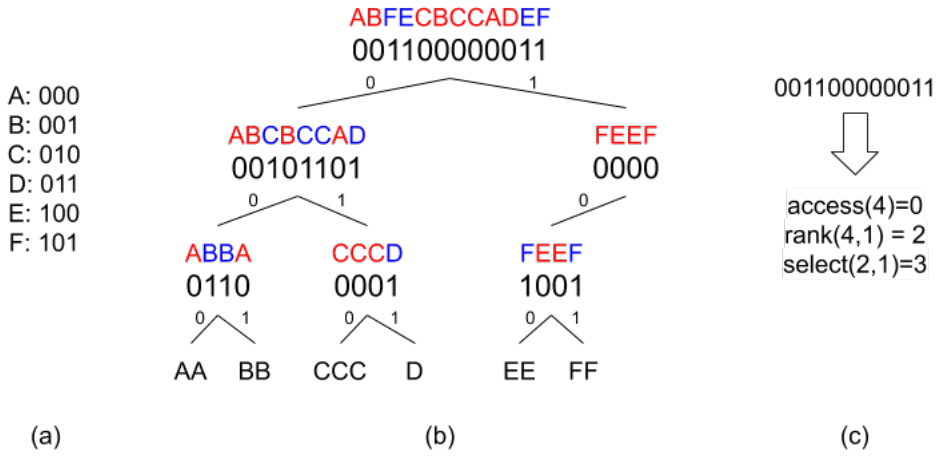

Figure 2.5: Wavelet Tree example with its dictionary

There are different types of SDS, among which we consider Wavelet Tree (WT) and BitMap (BM). SuccinctEdge represents an RDF graph into a combination of these two structures to reach a very compact storage layout without loss of query efficiency.

BM is the most basic SDS. It is a sequence of bits with some extra information to support the efficient execution of SDS operations. BM is the basic building block of WT's nodes (as each node in the tree is a BM), but it also relates different WTs in SuccinctEdge's triple representation (further details in Section 4.3).

WT [33], whose name reveals some affinity with the idea of the wavelet packet decomposition in signal processing, refers to a data structure which decomposes a data sequence into a set of nodes of a balanced binary tree. An example of a WT is given in Figure 2.5b. Suppose that we have a sequence $ABFECBCCADEF$, where each letter is mapped with an identifier in an incremental order, *e.g.*, A is denoted with 0, B is denoted with 1 (see dictionary in Figure 2.5a). A tree structure is constructed from this sequence as follows: each level of this tree divides the sequence of previous nodes into two sub-sequences by the corresponding bit. For example, from root to the first level, $ABFECBCCADEF$ is divided into $ABCBCCAD$ and $FEEF$ by the first bit of each identifier entry. This strategy is applied recursively until each leaf is computed.

SDS support three operations to access data: $Rank$, $Select$ and $Access$. Given a sequence S, the operation $S.\ Access(i)$ (also denoted as $S[i]$) refers to the $(i+1)^{th}$ element in S. $S.Rank(i, c)$ returns the number of occurrences of c from S's beginning to index i. Finally, $S.Select(i, c)$ returns the index of i^{th} occurrence of element c in S. These operations can be computed in O(1) for BM and O(log n) for WT where n is the size of the vocabulary. Figure 2.5(c) provides an example over a simple BM. The $Access(4)$ operation returns the 5th bit in a BM, which is 0. The operation $Rank(4, 1)$ asks for the number of 1 occurrences from index 0 to index 4, of which the result will be 2. The operation $Select(2, 1)$ searches for the position where 1 appears the 2nd time, this returns 3 as the result. Some reserach have demonstrated that the help of RRR vectors [44], the $Rank$, $Select$ and $Access$ can be processed very efficiently over a BM.

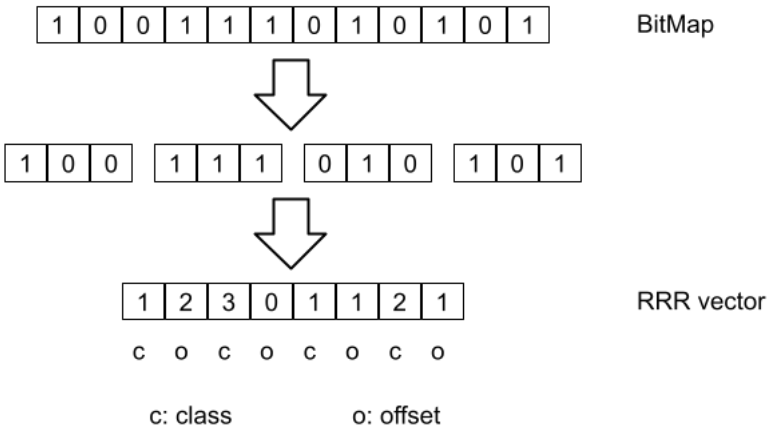

Figure 2.6: RRR vector example

A RRR vector proposes to divide a bitmap into blocks. An example shown in figure 2.6 highlights how to store a bitmap in the form of an RRR vector. A BM is split into small blocks which are of given size, here of size 3. For each block, we store the popcount and the offset corresponding to the block's permutation order. Moreover, we can also use a superblock to store a certain number of blocks to further reduce calculation. With the help of RRR vector, we can calculate $Rank$ and $Select$ operation over a BM in $O(1)$ time[32][45].

As $Rank$, $Select$ and $Access$ over a BM can be executed in $O(1)$ time, we can now analyse these operations over a WT.

The algorithm to compute $Rank(i, c)$ over a WT proposes to compute $r_i = Rank(r_{i-1}, b_i)$ recursively, where r_i corresponds to the result of current iteration and b_i corresponds to the i-th bit of c. This algorithm can help to execute $Rank(i, c)$ over a WT with a complexity of $O(log(n))$. For example, to compute $Rank(6, B)$ over figure 2.5(b), we can 1) compute $Rank(6, 0)$ (0 corresponds to the first bit of B) over the root level of the WT, thus we get 4 as the result. Then, the second step is to compute $Rank(4 + 1, 0)$ (0 corresponds to the second bit of B) over the left branch of the root, thus we get the new result 3. Finally we compute $Rank(3 + 1, 1)$ (1 corresponds to the third bit of B) to get $Rank(6, B) = 2$.

$Select$ operation can also be done in $O(log(n))$ time. The idea is to inverse the steps of a $Rank$ operation. For example, to compute $Select(2, B)$, we 1) compute $Select(2, 1)$ (1 corresponds to the last bit of B) over the supernode SN of the leaf B, this give us 2 as the result. The next step is 2) compute $Select(2 + 1, 0)$ (0 corresponds to the 2nd last bit of B) over the supernode of SN to get the result 3. The last step is 3) compute $Select(3 + 1, 0)$ (0 corresponds to the first bit of B) over the root, and we have the final result $Select(2, B) = 5$.

$Access$ over WT is also of complexity $O(log(n))$. Given an example as $Access(5)$ over figure 2.5b, the first step is to compute $Access(5)$ over the root. Once having 0 as the result (corresponding to the first bit of the final result), we then compute $Rank(5+1, 0) = 4$ thus we get the corresponding index as $4 - 1 = 3$ in the second layer. Because we have 0 as the result's first bit, we go down to the left branch LB and execute $Access(3) = 0$. Up to this step we have 00 as the first two bits of the result. With the same strategy, we compute $Rank(3 + 1, 0) = 3$ over LB and $Access(3) = 1$ over the left branch of LB. Finally, the

result of $Access(5)$ over this WT comes to be 001 which corresponds to B.

Chapter 3

LiteMat, an encoding scheme for RDFS++

La seule bonne monnaie est la pensée.

– Platon

LiteMat is an important component of SuccinctEdge. It provides a compact representation for knowledge bases and an efficient approach to RDFS reasoning via query rewriting. Considering compactness, the binary encoding of knowledge base entries, *i.e.*, ABox individuals, ontology concepts and properties, is compatible with emerging data structures which are adapted to in-memory storage, self-indexing and a decompression-free approach. The chapter's main objective is to present our latest LiteMat contributions: support for multiple inheritance and an ontology expressiveness extension with the integration of transitive and inverse properties.

3.1 Introduction

As introduced in Section 2.2.4, LiteMat corresponds to an encoding schema for RDF data and RDFS ontologies. Its main advantage compared to other encoding approaches, *e.g.*, those found in most RDF stores [1][54][35][19], is that it can be qualified as semantic-aware. This means that the integer values associated to ontology concepts and properties are conveying the semantics of their respective hierarchies. As a result, systems that are using LiteMat can efficiently support the most common RDFS reasoning services via a query reformulation approach.

Nevertheless, LiteMat has its set of limitations. One of them is the multiple inheritance problem which appears in some real-world ontologies. Essentially encountered in concept hierarchies, it corresponds to the fact that a concept has more than one super concept. Given LiteMat's top-down, single-ancestor directed encoding scheme, providing an efficient solution to the multiple inheritance problem is not that simple. Precisely speaking, the idea of LiteMat is to encode an element by applying the encoding of its direct ancestor as the prefix. This idea leads to a problem where if an element has more than one direct ancestor, it may have multiple possible encodings which breaks the rule that an element can only have a unique identifier in the dictionary. In this work, we propose a solution that fits nicely with LiteMat's encoding approach, *i.e.*, it keeps its binary encoding strategy and limits data structures overload. Moreover, it provides good performance measures on SPARQL query processing.

Another LiteMat extension that we are presenting in this chapter concerns ontology expressiveness. In fact, we propose a solution for transitive and inverse properties. Considering `owl:inverseOf` properties, we apply a simple transformation of the encoded ABox and a property dictionary look-up at query processing-time. Our approach considering the `owl:transitiveProperty` property is more involved and is based on i) an encoding solution of the individuals involved in chains and trees of these properties and ii) an associated query processing strategy. Due to a lack of an efficient solution, directed acyclic graphs (DAG) of transitive properties, which are relatively rare in practice, are currently being materialized.

The contributions presented in this chapter permit to extend LiteMat toward RDFS++ expressiveness and to support multiple inheritance in the ontology hierarchies. These extensions concern both an encoding scheme that results in a more compact KG representation and an adapted query processing.

3.2 Multiple inheritance in LiteMat

In this section, we present LiteMat's encoding scheme using the ontology concept hierarchy displayed in Figure 3.1(b) where both plain and dashed arrows represent a subsumption relationship, *e.g.*, $A1 \rightarrow A$ corresponds to $A1 \sqsubseteq A$ in the DL formalism. Note that this method can also be applied to property hierarchies.

We can observe that two multiple inheritance situations occur in this concept hierarchy.

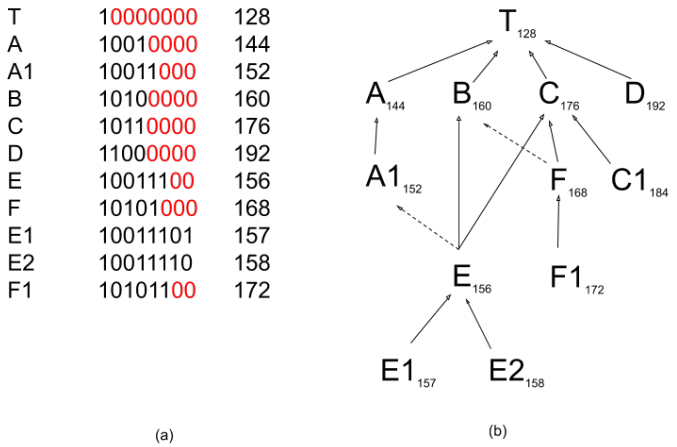

Figure 3.1: LiteMat Encoding (a dashed arrow points to a representative)

In fact, concepts E and F have respectively 3 and 2 super classes. With LiteMat's encoding scheme, it is clear that multiple inheritance poses a problem since each ontology concept must have a single identifier. For instance, in our running example, we can provide three different identifiers to E: one computed from $A1$, another one from B and a last one from C.

The prefix, *i.e.*, before the normalization step, identifier of E computed from $A1$ will be 100111, as a consequence, the identifiers of $E1$ and $E2$ will have respectively 10011101 and 10011110 as a prefix. However, if we compute the identifier of E from B, the prefix comes to be 101001, then the prefix identifiers attributed to $E1$ and $E2$ will be 10100101 and 10100110 respectively. Another possibility is to compute the identifier of E based on C, thus we obtain 101101 as E's prefix identifier and, therefore, distribute 10110101 and 10110110 to $E1$ and $E2$. It is obvious that the identifier of a concept will vary if it has multiple super concepts. Further more, this will also have impact on its sub concept hierarchies recursively.

Multi-inheritance poses issues to LiteMat's encoding. Thus in the following section, we propose a solution to this issue.

3.2.1 Encoding scheme

Our support of multiple inheritance is based on the notion of a representative. A representative, denoted C_r, is selected among the super concepts C_1,..,C_n of a concept SC. That is the integer identifier of SC will be computed following Litemat's approach based on the C_r identifier. It is obvious that with this approach SC loses every connection to its non-representative direct and indirect super concepts. Hence it is necessary to keep track of these super concepts. In the following, the remaining super classes of SC, i.e., $\{C_1,..,C_n\}\backslash C_r$ are considered as non-representative of SC.

Let consider that the representative of the concept E is $A1$. Hence, the non-representatives of E are the concepts B and C. In Figure 3.1(b)(where each concept has its identifier in subscript), a representative is pointed by a dashed arrow and we can observe that the identifiers

of concepts E, $E1$ and $E2$ (resp. 156, 157 and 158) are computed from $A1$'s identifier (i.e., 152).

To support an efficient query processing, we require a key/value data structure, denoted nonRep. Intuitively, each non representative of the ontology is an entry in that data structure and the value associated to a key corresponding to a set containing all the sub concepts involved in a multiple inheritance with the key concept as one of its super concept. In our running example, nonRep(B) = {E} and nonRep(C) = {E,F}.

3.2.2 Query processing and optimization

The query processing presented in Section 2.2.4 is extended to produce complete and correct result sets for ontologies involving multiple inheritance. The extension coincides to the addition of disjunction in the generated FILTER clause of the rewritten SPARQL queries. Like in the original rewriting approach of Section 2.2.4, the disjuncts correspond to interval descriptions for a given query variable. An interval is computed using the nonRep data structure. Note that queries involving representatives do not necessarily involve inferences involving the concept and property hierarchies.

We consider this chapter's running example (Figure 3.1) and the following BGP, denoted Q, which involves a multiple inheritance:

```
{?x rdf:type C}
```

The query rewriting denoted Q' corresponds to:

```
{ ?x rdf:type ?y.  filter((?y ≥ 176 && ?y<192)
 ‖ (?y ≥ 156 && ?y < 160) ‖ (?y ≥ 168 && ?y < 176))}
```

In Q', the first disjunct corresponds to the standard interval defined in Section 2.2.4 and the last two are computed using the nonRep data structures. That is, we search whether the concept C is involved in a multiple inheritance by checking its presence as a key in nonRep. If it is the case, it will return a set of concepts and for each of these concepts a disjunct is added to the FILTER clause over that variable. In our running example, nonRep(C) returns a set with concepts E and F, resp. the identifiers 156 and 168. These values correspond to the lower bounds of the intervals and upper bounds are compute as stated in LiteMat's query reformulation approach.

Based on the intervals present in this FILTER clause, a simple optimization can be performed. It has the possible effect of reducing the number of disjuncts in the filter clause of a query reformulation. The optimized queries are Q'' :

```
{?x rdf:type ?y.  filter((?y≥168 && ?y<192) ‖ (?y≥156 &&
                          ?y<160))
```

This optimized rewriting now contains two conjuncts instead of three. In more involved queries, this optimization can have an important impact on query performance.

3.3 RDFS++ extensions for LiteMat

3.3.1 Support for `owl:inverseOf`

Concerning `owl:inverseOf` properties, we propose the following simple approach. For each URI property and its inverse, denoted $\langle p, p^- \rangle$, we retain in our ABox[14], only one of the two URIs which is therefore denoted as the property representative, *i.e.*, p_r. For each pair $\langle p, p^- \rangle$ in the ABox, a representative, p_r, is selected based on the largest number of occurrences over the pair $\langle p, p^- \rangle$.

In the LiteMat property dictionary associated to the locate function[10], *i.e.*, URI to identifier key-value structure, both property URIs are associated to the same integer identifier: both p_r and p^- are associated to a unique pid value as computed in [11]. In the extract property dictionary, *i.e.*, id to URI key-value structure, only the representative property is stored since answers to queries requiring an extract operation on the property are expressed with the representative p_r.

During the ABox encoding, all triples already expressed with p_r are normally encoded using the individuals and properties dictionaries. Concerning all triples expressed with p_r^-, *e.g.*, $i1\ p_r^-\ i2$, they are transformed as follows: the subject and object of the original triples respectively become the object and subject of a new triple and the property is switched to the representative.

Example: Let $parentOf$ `owl:inverseOf` $childOf$ be a TBox axiom and $parentOf$ is selected as the representative in this property pair. Table 3.1 displays an original ABox and its resulting transformation.

Original ABox	Transformed ABox
dominique parentOf jean.	dominique parentOf jean.
dominique parentOf pierre.	dominique parentOf pierre.
marie childOf pierre.	**pierre parentOf marie.**

Table 3.1: TBox encoding facts

A similar transformation is applied to graph patterns of a SPARQL query whenever the inverse of a representative is identified in a BGP. Starting with the following query:

```
SELECT ?x ?y WHERE {?x childOf ?y}
```

The corresponding reformulation would be:

```
SELECT ?y ?x WHERE {?y parentOf ?x}.
```

This would return an answer with 3 tuples including the pair ⟨ pierre, marie ⟩.

3.3.2 Support for `owl:transitiveProperty`

Let consider a function $trans(G, p) = G'$, with G an RDF graph, p a transitive property and G' a subgraph of G solely composed of triples with the p property. Intuitively, G' is composed of, not necessarily connected, chains, trees or DAGs of individuals (see Figure 3.2 for examples of the first two structures).

In this section, we propose two encoding schemes: one for the chain and another one for tree structures that are following the logical approach of LiteMat. That is, it provides semantic-aware identifiers to ABox individuals encountered in these structures. In the current state of the LiteMat data management system, the transitive closure of DAG transitive structures is materialized.

The characteristics that we are aiming for in this encoding schemes are: (i) **compactness** since no materialization is required for the chain and tree structures, (ii) **determinism** since the identifier of each individual in a chain or tree is computed deterministically and (iii) **scalability** since all encoding tasks are performed in a distributed manner on a distributed engine, namely Apache Spark and its GraphX graph computing component.

In both the chain and tree encoding, our processing starts with the computation of $trans(G, p)$ for all transitive properties of the associated ontology, resulting in a set of subgraphs. Then, the system computes the connected components for each of these subgraphs. Intuitively, the connected components operation groups vertices into connected subgraphs. This can easily be performed in a scalable manner with Spark's GraphX component. GraphX's API comes with a connected component function that runs in a distributed manner over a cluster of machines. The resulting connected component is assigned a distinct identifier corresponding to the lowest node identifier of the connected component. The encoding of individuals in these graphs is made of a quadruple of integer values which correspond to: fid which is 0 if the transitive structure is a chain or 1 if it is a tree, pid the identifier of the transitive property, $ccid$ the connected component identifier and lid a local node identifier. This procedure can be summarized in algorithm 1.

The chain and tree structures are distinguished by the computation of their local identifiers. Figure 3.2 presents in each node the label of the individuals (*i.e.*, URIs) and its identifier. In the case of a chain structure, it is sufficient to assign integer values that define a total order over the set of lid of a given triple $\langle fid, pid, ccid \rangle$. With such an approach, the computation of all descendants (respectively ancestors) of a given individuals $\langle fid, pid, ccid, lid_i \rangle$ will amount to retrieve individuals identified by $\langle fid, pid, ccid, lid_x \rangle$ for all $lid_i < lid_x$ (respectively, $lid_i > lid_x$).

The encoding of tree structures is more involved. For instance, in Figure 3.2(b), individuals E,F and G do not belong in the transitive closure of B or D. In this case, the incremental, naive assignment of local identifier is not sufficient to efficiently detect that a node is not in the transitive closure of another node in this graph. We adopt a local node identifier approach that is inspired by our LiteMat binary approach. Intuitively, the encoding algorithm is recursively assigning binary identifiers in a top-down manner from the root of the tree. The root node starts with a single bit set to 1. Then we identify all directly linked individuals for a node. The size of this set of individuals justifies the length of the binary encoding

Algorithm 1: Compute transitive closures.

Input: Graph G, Transitive properties set P
Output: A set of transitive closures S

```
for a property p_i in P do
    Retrieve the sub-graph g_i where all the edges are p_i from G;
    for each connected component cc_j in g_i do
        if cc_j is a transitive chain then
            Transitive closure identifier id_t ← (0, id_{p_i}, id_{cc_j});
        else
            if cc_j is a transitive tree then
                Transitive closure identifier id_t ← (1, id_{p_i}, id_{cc_j});
            end
        end
        Add id_t and cc_j to S;
    end
end
return S;
```

for each of these individuals. For instance, in Figure 3.2(b), A has three directly connected individuals (namely B, C and D) so two bits are necessary to encode them. The temporary local identifier at each level starts with counter set to 1 and is incremented by 1, and each of these individuals is prefixed with the identifier of their local root. Thus the identifier of B in Figure 3.2(b) is 101 (with the left most bit inherited from A and 01 computed at this level). This computation is performed recursively until all nodes are assigned a value. A final step consists in normalizing the temporary identifier: all identifiers have to be encoding with the same binary length. In our example, F and G are the identifiers with the longest binary encoding (*i.e.*, 6 bits) so all nodes of the tree are right-completed with bits set at 0 to reach the same length. The identifiers for each node in Figure 3.2(b) are displayed in each box, the gray 0 of an identifier are the results of the normalization while the local fragment is in black.

Given this local identifier strategy, we can easily find whether a given node is in the transitive closure of another node of that same graph. This operation is based on checking whether the subtraction of two identifiers is contained in a given interval. Let consider the connected component graph of a transitive property. Due to the normalization step, all identifiers of this connected component are encoded using n bits. Moreover, we introduce a function, $localLength$, that returns the non-normalized binary encoding length of a node. For instance, in Figure 3.2(b), $localLength$ of A,C and G are respectively 1, 3 and 6. For two nodes of this graph, α and β, β is in the transitive closure of α if $\beta - \alpha$ is included in $[0, 2^{n-localLength(\alpha)}[$.

Using this approach, we can efficiently compute that G is in the transitive closure of A, B and E but not of B, D and F.

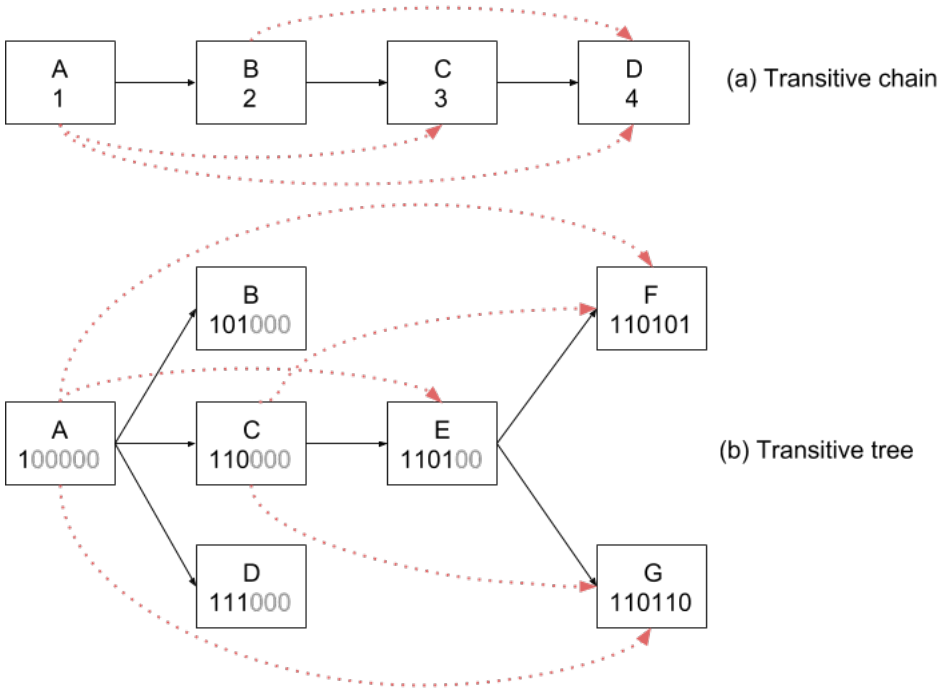

Figure 3.2: A chain and a tree of a transitive property. Dotted red arrows correspond to the transitive closure. In each node, a label and its local identifier (lid)

Since LiteMat's `owl:sameAs` encoding scheme, *i.e.*, a tuple $\langle cliqueId, localId \rangle$, does not rely on a local identification total order, it is possible to compose `owl:sameAs` identifiers with transitive ones. This means that individuals involved in a chain or tree transitive structure can be involved in a `owl:sameAs` clique by reusing their identifiers in the sameAs encoding scheme. In such a case, the $localId$ corresponds to the whole transitive identifier.

3.4 Query processing in the presence of transitive properties

We now present query processing with a BGP involving a transitive property. We consider a BGP containing a single triple pattern asking for all subjects (respectively objects) related, via a transitive property, to an object (respectively subject). A similar approach can be applied for more complex BGPs.

3.4.1 Query encoding

As with most RDF triples, the query needs to be translated to an identifier-based form which requires look-ups to several LiteMat dictionaries. Using the property dictionary, we obtain the identifier of the property and we will also get the information that this property

is transitive. Then, we search for the identifier of the object (respectively subject). In a full materialization case, this identifier corresponds to a single identifier while in the case of LiteMat, it corresponds to a 4-tuple identifier, *i.e.*, ⟨fid, pid, ccid, lid⟩.

3.4.2 Variable assignments

While the full materialization approach requires a complete scan over all triples plus an extraction from the individuals dictionary, the same query can be answered much more efficiently with LiteMat. Intuitively, due to LiteMat's semantic-aware encoding, we can rely solely on some simple computation to directly search for answers in the dictionary. In fact, we will search for all individual dictionary entries where the key is of the form $\langle fid, pid, ccid, X \rangle$ where one of the following computations is performed:

- if fid corresponds to a chain, *i.e.*, $fid = 0$, then the system retrieves all values lower then lid.
- if fid is a tree, *i.e.*, $fid = 1$, then the system retrieves all values comprised between]lid, ((lid $>>$ lid encoding length)+1)$<<$ lid encoding length)[

3.5 Related Work

Most RDF stores are encoding the triple elements, *i.e.*, URIs, blank nodes and literals, of their data sets. That is, instead of storing triple elements they are storing a triple of integer values where each value is associated to one triple element. These mappings are persisted in dictionaries and most systems store them in two distinct dictionaries: String to Id and an Id to String data structures.

For the String to Id dictionary, a B+-tree is generally used while the Id to String dictionary can efficiently be addressed by a standard array for its constant time direct access [35]. Most encoding schema used in RDF stores used an arbitrary approach. That is, no meaning is given to the dictionary identifiers.

LiteMat is not one of a kind to propose a smart encoding. In fact, the Waterfowl system [13], designed and developed by our research group, is the main influence behind LiteMat. Moreover, in [48], the authors present an approach where each entity (concept or property) in the corresponding hierarchy is assigned a numeric value according to a breadth-first visit of the hierarchy. Then, there is a guarantee that any sub-hierarchy is associated to a consecutive set of numeric values, *i.e.*, an interval. Like in LiteMat, each entity is associated to an interval covering the indexes of all its sub-entities. Considering String to Id operations, an entity will be encoded using the smallest integer of its interval, *i.e.*, the one induced by the breadth-first visit. This so called Semantic Index can be constructed in polynomial time in the size of the entity hierarchies. Using this Semantic Index, any query over entity hierarchies can now be expressed as a simple range queries.

Nevertheless, both of these smart encoding scheme are pushing their logical approaches

as far as LiteMat is currently doing. With the RDFS++ extension presented in this chapter, LiteMat is reaching an expressive level that the other two systems are not addressing.

3.6 Evaluation

3.6.1 Multiple inheritance evaluation

Our multiple inheritance experimentation consists in evaluating the performance of the database construction and query processing phases. In the database construction phase, we evaluate the duration and memory consumption dimensions. In the query processing phase, we compare LiteMat against a full materialization approach using a set of SPARQL queries. Our implementation can be accessed at this github link[1]. We will explain these two phases in detail within the following section.

Experimental setting

The experimentation was run on a MacBook pro with a 2.9 GHz Intel Core i5 and 8 GB LPDDR3 RAM.

Datasets and query workload

The purpose of this evaluation is to compare the performance of LiteMat approach with that of a full materialization (denoted FullMat) approach in the context of TBox with multiple inheritance. In this evaluation, we test some simple BGPs which enables us to use some auto-generated ontologies containing only a hierarchy of concepts as data sets. Each ontology is associated with an ABox that involves only some facts in form of $\{X$ `rdf:type` $Y\}$ where X is an instance and Y is a concept.

In order to construct our multiple inheritance datasets, we first generate a hierarchy in form of a tree where each node is a concept. Once we add a certain concept to the bottom of the tree, there will be a 30% chance that this concept is involved in a multiple inheritance. If so, we will choose a certain number of concepts from higher levels as its super concepts and register the corresponding triples. After the ontology construction, we will pick 50% of the concepts and create triples like $\{X$ `rdf:type` $Y\}$ in the ABox.

[1] https://github.com/xwq610728213/multipleInheritanceEvaluation

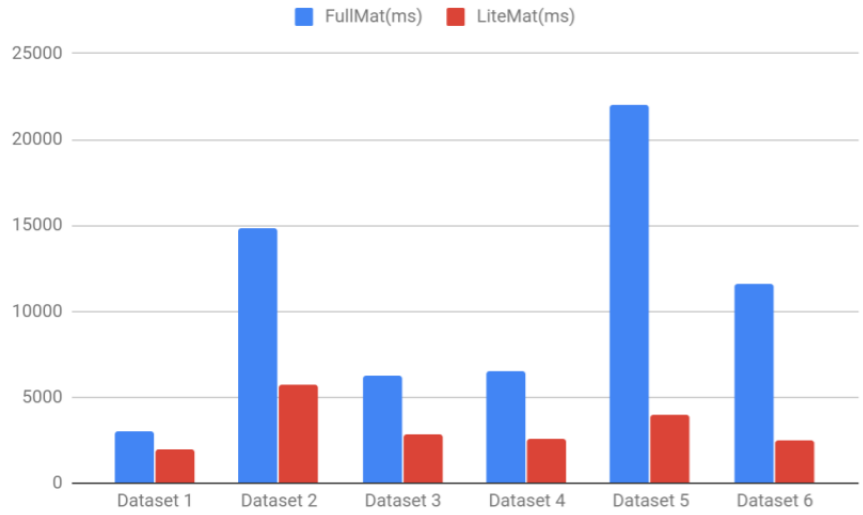

Figure 3.3: Database construction time of LiteMat vs a full materialization

DataSet	Maximum Depth	Maximum Branches	Maximum Inheritance	TboxSize (KB)	AboxSize (KB)
Data set 1	5	4	4	25	8
Data set 2	6	6	3	476	176
Data set 3	6	5	3	90	34
Data set 4	7	3	3	22	8
Data set 5	7	5	4	514	169
Data set 6	8	3	3	51	18

Table 3.2: Details of testing data sets

Table 3.2 shows details of each data set. Maximum Depth gives the depth of the deepest concept in the hierarchy. Maximum Branches indicates how many branches a node can possess in maximum. While generating sub concepts of a certain concept, we will choose a random number between 0 and Maximum Branches as the number of its sub concepts. Once a concept is chosen to be part of a multiple inheritance, Maximum Inheritance limits the number of its direct super concepts. For example, Data set 1 is a DAG structure with a maximum depth of 5, each node has $[0, 4]$ branches and a multiple inheritance concept has $[2, 4]$ super concepts. More details can be found with the link[2].

Database construction performance

In this section, we compare LiteMat encoding with the FullMat encoding in two aspects, database construction time and memory consumption.

[2]https://github.com/xwq610728213/MultiInheritanceGenerator

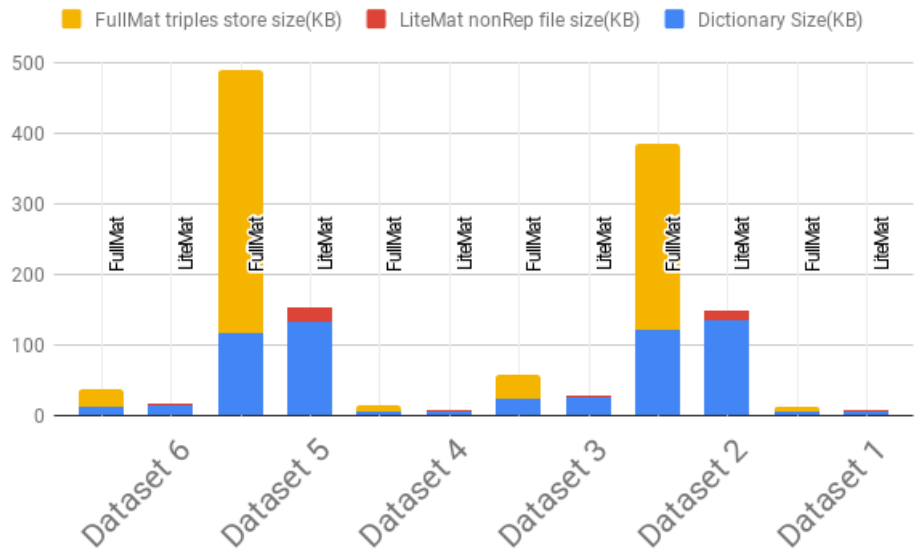

Figure 3.4: Memory space required by LiteMat vs a full materialization

Figure 3.3 shows the comparison of database construction time between LiteMat and FullMat within each data set. The database construction time includes encoding elements, constructing dictionary and other necessary data structures, *e.g.*, triple store for FullMat and nonRep list for LiteMat. Obviously, LiteMat constructs database much faster than FullMat in all tests because FullMat must deduce all possible statements, *i.e.*, direct and indirect ones, during database construction, which takes some time, while LiteMat encodes elements only by direct connections.

The comparison of memory consumption is given in figure 3.4. We can not directly compare the RAM consumption during runtime. Thus we store the in-memory data structures, *e.g.*, dictionary, triple store and nonRep structure, into files and directly compare the size of these files. Because the size of these files is proportional to the RAM consumption, we consider the comparison in size of these files reflects the relation of RAM consumption between two approaches. As we can conclude from the figure, although the dictionary of LiteMat is a little bit larger than that of FullMat, considering the nonRep list of Litemat is much smaller than the triple store of FullMat, our LiteMat approach takes much less total space than FullMat.

Query performance

As for query evaluation, we only test the query pattern in form of `{?x rdf:type C}`, which is a frequently appearing triple pattern in queries concerning concept hierarchy. This query pattern demands all the instances belonging to a concept C or its sub-concepts. We randomly retrieve 15 concepts as C from the ontology and generate 15 queries for each data sets, and compare the query processing time between LiteMat and FullMat.

Figure 3.5 shows the results. Each cross in the figure represents the answering time of a

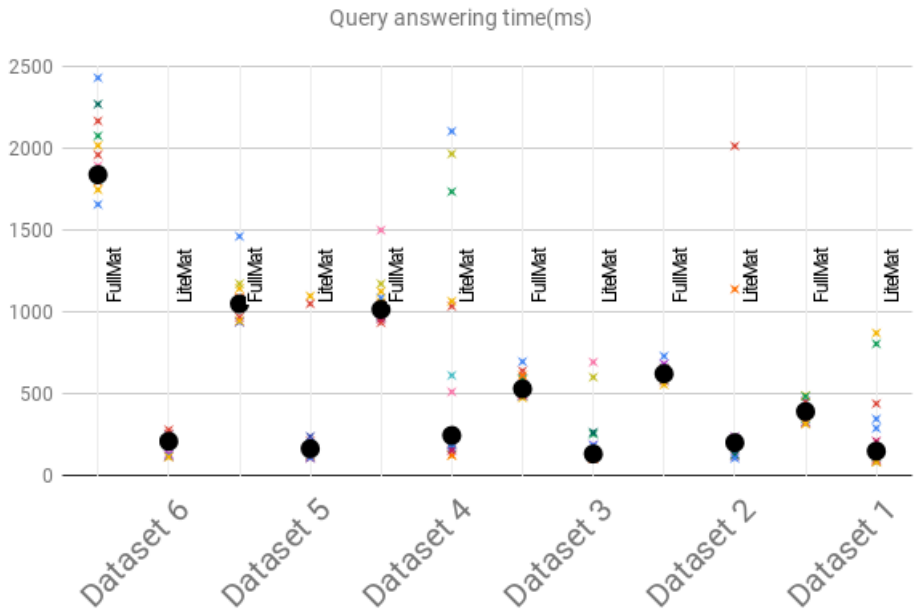

Figure 3.5: Query answering time distribution of LiteMat vs full materialization

query with different approaches. The big black point indicates the median of the 15 queries answering times. As we can see from the figure, the median of query answering time with LiteMat is always smaller than that of FullMat, thus we can say our LiteMat approach performs better in the most of queries. However, there exists some queries where LiteMat takes more time than FullMat. One reason is that in order to answer these queries, LiteMat needs to search nonRep data structure many times, because some values of a nonRep structure appear as the key (or the super concept of the key) in other nonRep structure.

3.6.2 Transitive property evaluation

Experimental setting

The evaluation was conducted on a cluster composed of three Dell PowerEdge R410 running a Debian GNU/Linux distribution with a 3.16.0-4-amd64 kernel version. Each machine has 64GB of DDR3 RAM, a 900GB 7200rpm SATA disk and two Intel Xeon E5645 processors. Each processor is constituted of 12 cores running at 2.40GHz and allowing to run two threads in parallel (hyper threading). The machines are connected via a 1GB/s Ethernet network adapter. We used Spark version 2.3.2 and implemented all experiments in Scala version 2.11.6. More details on the scripts can be found here[3]. The Spark configuration of our evaluation runs our prototype on a subset of the cluster corresponding to 36 cores and 24GB of RAM per machine.

[3]https://github.com/xwq610728213/LitematPlusPlus

Datasets and query workload

In this evaluation, we are aiming to test and stress our approaches with different sizes of transitive chains and trees. We thus resort to a synthetic benchmark solution, namely the Lehigh University BenchMark (LUBM)[17], a well-established benchmark on the university domain that contains a transitive property, named `lubm:subOrganisationOf`. Since both the `rdfs:domain` and `rdfs:range` of this property are the Organization concept, we can use it to create long chain and tree structures.

Table 3.3 presents the datasets that we are using throughout this experimentation. Intuitively, the name of each dataset describes the number of universities, *e.g.*, 5K or 10K for respectively 5.000 and 10.000 universities, a letter, *i.e.*, either 'c' or 't' which respectively stand for chain and tree structures and a number that corresponds to the maximum depth of the structure. Note that the 5K_c20 and 10_c20 correspond to large shallow trees which are supposed to mitigate the advantages provided by LiteMat. In total, 10 datasets are evaluated, 4 chains and 6 trees.

In this section, we provide an evaluation of our query processing solution. We are now considering answering a single triple pattern that retrieves either all the ancestors or descendants of a group in the transitive closure of the `lubm:subOrganisationOf` property. These two queries have been executed over the some of the 10K datasets and respectively correspond to:

`SELECT ?X WHERE {?x lubm:subOrganisationOf C}` to compute ancestors and

`SELECT ?X WHERE {C lubm:subOrganisationOf ?x}` to retrieve descendants where C is an individual involved in the queried dataset, for instance, this could be: `<http://www.Department10.University1000.edu/ResearchGroup1>`.

This limited evaluation already provides some valuable insight on the potential of LiteMat query processing with transitive properties.

Dataset name	Depth sizes [min,max]	#Branches [min,max]	#Triples	Size (MB)	#Triples materialized	Increase due to materialization
5K_c20	[10, 20]	1	1.689.907	318,4	23.579.485	x 14
5K_c100	[20, 100]	1	6.752.637	1.280	230.004.339	x 34.1
5K_t5	[10, 20]	[1, 5]	5.062.616	957.9	39.362.874	x 7.8
5K_t10	[20, 100]	[5, 10]	12.624.667	2.400	109.223.271	x 8.7
5K_t20	[2, 5]	[10, 20]	5.898.803	1.120	14.064.044	x 2.4
10K_c20	[10, 20]	1	3.376.055	636,8	48.712.856	x 14.4
10K_c100	[20, 100]	1	13.522.653	2.560	461.042.361	x 34.1
10K_t5	[10, 20]	[1, 5]	10.119.755	1.920	71.304.115	x 7.0
10K_t10	[20, 100]	[5, 10]	25.260.771	4.800	216.690.752	x 8.6
10K_t20	[2, 5]	[10, 20]	11.804.988	2.240	28.155.826	x 2.4

Table 3.3: Characteristics of evaluated datasets

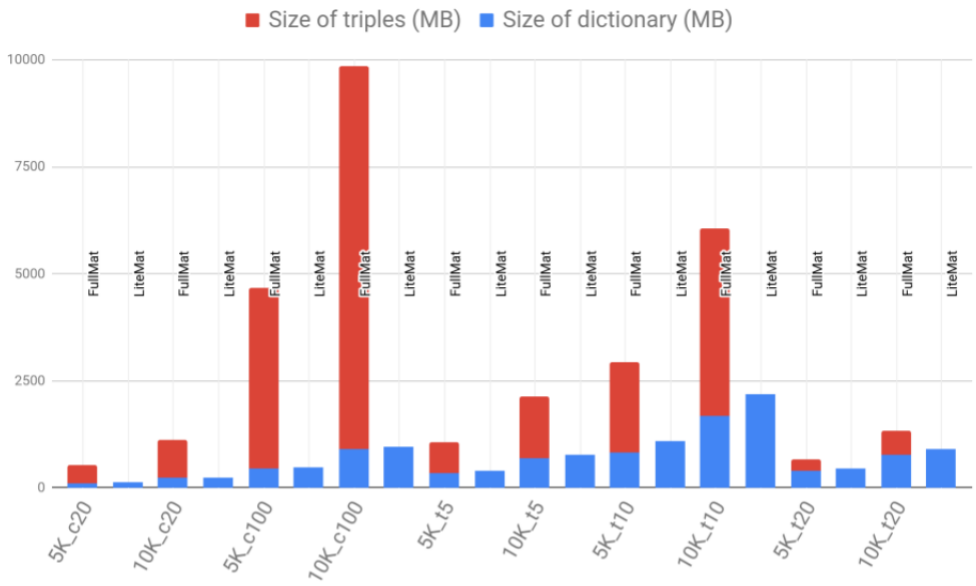

Figure 3.6: Memory space required by LiteMat vs a full materialization

Compression and encoding performance

In this section, we are mainly interested in two performance dimensions: the memory space reduction provided by LiteMat compared to a full materialization and the duration of LiteMat's encoding against the full materialization computation.

Figure 3.6 presents comparisons of the memory space required by the LiteMat approach against a full materialization. In the latter approach, both the set of materialized triples as well as the dictionaries are required to answer inference-enabled SPARQL queries while in the case of LiteMat, only the dictionaries are necessary. The figure emphasizes that, for any datasets, both dictionaries are about the same size, with the ones of FullMat being a little bit more compact for trees due to the overhead LiteMat identifiers, *i.e.*, a long value for the materialization against a 4-tuple of long values and an integer for LiteMat.

Obviously, for long chains and large trees, the amount of materialized triples can be quite important, *i.e.*, for 5K_c100 and 10K_c100, the set of materialized triples of 34 times larger than to their original triple sets. Figure 3.3 and the LiteMat approaches correspond to only 10% of their sizes. Considering 5K_c20 and 10K_c20, LiteMat's approach is still around 70% of total materialized approach.

Figure 3.7 provides some details on the duration of the different computation steps involved in both the full materialization and LiteMat approaches. The common steps of these two approaches are the loading of the dataset and the computation of the connected components. We can see that the time taken by the former is quite negligible compared to the other tasks. Unsurprisingly, the computation of the connected components takes a lot of time on all experimentation. The LiteMat encoding and full materialization share a common naïve encoding of individuals step (which is included in both times). Overall, we note that for chains of a transitive property, the difference between both approaches is not sig-

Figure 3.7: Durations (in seconds) for the full materialization and LiteMat approaches

nificant, *i.e.*, LiteMat is only between 2 and 3% faster than the full materialization. This is not true for structures taking the form of a tree. In that case, LiteMat's encoding is 45 to 50% faster when the depths of structure is relative large for transitive properties, *i.e.*, [10,20] and [20,100]. We consider that this is mainly due to the recursive parsing of the tree to compute the transitive closure. The duration difference between the two approaches is less important, *i.e.*, around 11%, when the tree structure depths lies in the [2,5] range.

Query processing

Our preliminary evaluation of the query processing consists of a cold retrieval of all descendants of a give individual and the average of five hot queries that retrieve all descendants (*i.e.*, hot1) and ancestors (*i.e.*, hot2). Figure 3.8 provides measures conducted on the largest datasets of our experimentation, *e.g.*, 10K_c100 and 10K_t20 of respectively 9.8GB and 6GB for the materialized approaches. In order to provide a complete overview of the approaches, In this figure, all measures (loading time, cold, hot1 and hot2) emphasize shorter execution times for LiteMat. Considering the loading times, LiteMat is between 6 to 10 times faster than the complete materialization. This is mainly due to the fact that LiteMat solely relies on the dictionaries and not on the triples set. This aspect impacts the cold runs where LiteMat is between 2 and 8 times faster then the full materialization. Finally, for hot runs, LiteMat is 2 to 3 times faster than a complete materialization.

3.7 Conclusion

In this chapter, we have extended the expressiveness of LiteMat to reach the level of RDFS++ with multiple inheritance. This has been achieved by pushing the original logical approach

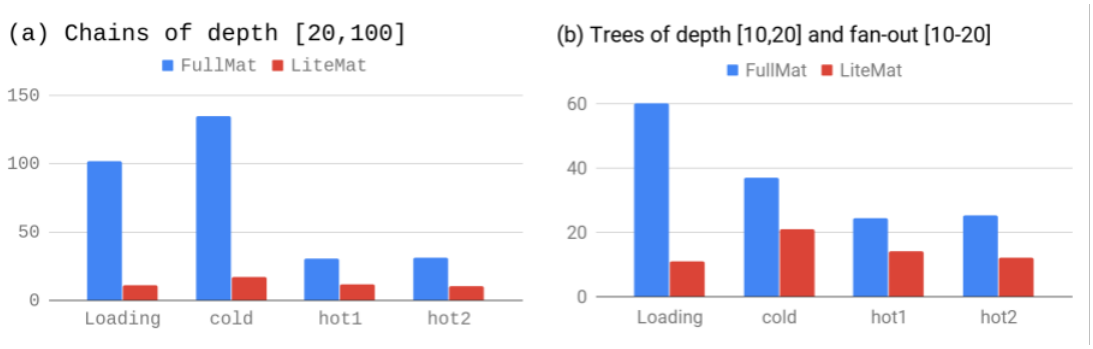

Figure 3.8: Query processing on chains and trees with full materialization and liteMat (times in seconds)

of LiteMat that consists in assigning meaningful identifiers to elements of the TBox and the ABox. The evaluation of transitive structures taking the forms of chains and trees has been demonstrated with properties such as low memory footprint, speed of encoding and efficiency of query processing.

Nevertheless, there is room for improvement in directions such as a more efficient support of graph transitive structures and the fact that certain individuals can be contained in structures of different transitive properties. The optimization of query processing, considering both RDFS++ and multiple inheritance aspects, is another direction for future work. On the implementation side, we are considering using indexed Spark abstractions and are considering algorithms such as those presented in [25] to compute connected components.

Chapter 4

SuccinctEdge

Now is no time to think of what you do not have. Think of what you can do with what there is.

– Ernest Hemingway

In this chapter, we provide details on the design and implementation of SuccinctEdge, an in-memory, self-indexed, compact and reasoning-enabled RDF store for Edge Computing. We demonstrate its efficiency on real-world and synthetic data sets.

4.1 Introduction

As said in this thesis introduction, Edge computing emerges as an innovative platform for services requiring low latency decision making. Its success partly depends on the existence of efficient data management systems. We consider that knowledge graph management systems have a key role to play in this context due to their data integration and reasoning features.

Our prototype system, SuccinctEdge[1], has been designed for Edge Computing from the get go and adopts the RDF data model. The adoption of this data model is motivated by the data integration and reasoning facilities it provides. Considering the former, the Linked Data principles[2] together with the large set of Knowledge Graphs (KGs) available via the Linked Open Data initiatives[3] ease the design of Internet of Things (IoT) applications. For instance, ontologies such as the Sensor, Observation, Sample, Actuator (SOSA [4]), Quantities, Units, Dimensions, and Types (QUDT) [5] or Smart Applicances Reference (SAREF)[6] considerably simplify the task of describing, manipulating and connecting sensors and actuators. These ontologies also serve smart measure management when reasoning services are introduced in SPARQL queries to infer implicit consequences from explicitly represented knowledge.

SuccinctEdge favors a compressed, single index storage approach to a solution based on multiple indexes that could potentially improve query execution but at the cost of a higher memory footprint. The applications we are targeting with SuccinctEdge are the processing of a flow of RDF graphs (sent from sensors or actuators) which are sharing a common topology. These graphs are continuously queried by a set of SPARQL queries. In a typical use case, these queries are searching for anomalies occurring over a network of sensors (see Section 4.2 for a motivating example). As a result, these queries are executed once per graph instance.

Our system makes an intensive use of succinct data structures (SDS)[33], a family of data structures that adopts a compression rate close to theoretical optimum, but simultaneously allows efficient decompression-free query operations on the compressed data. Together with our single index approach, SDS guarantees a low memory footprint that fits with an in-memory storage approach. The decompression-free aspects also tends to reduce the number of CPU cycles on standard queries and inferences.

SuccinctEdge's reasoning services are based on the LiteMat encoding solution[12]. This approach prevents inference materialization and reduces the cost of the SPARQL query rewriting task, the two most frequent reasoning solutions in RDF stores. As a result of encoding most triple entries with integer values, this approach improves the efficiency of graph pattern matching and compresses RDF data sets, thus limiting the memory footprint

[1] https://github.com/xwq610728213/SuccinctEdge
[2] https://www.w3.org/wiki/LinkedData
[3] https://lod-cloud.net/
[4] http://www.w3.org/TR/ns/sosa
[5] http://qudt.org/schema/qudt
[6] https://ontology.tno.nl/saref.ttl

of a given graph.

SuccinctEdge is addressing the compact storage and efficient querying of RDF data via SPARQL queries in the presence of RDFS reasoning in an Edge Computing environment. The main contributions of this chapter are to (i) present a self-index, compact, in-memory storage layout based on the bitmap and wavelet tree SDSs, (ii) propose a decompression-free (*i.e.*, the SDS compressed graph does not need any decompression step to enable query execution), efficient query processing and optimization of SPARQL BGPs which are transformed into access, rank and select SDS operations, (iii) support reasoning during query processing using a smart encoding approach and (iv) propose a simple and automatic approach to express complex queries requiring inferences by preventing end-users from learning the details of used ontologies and ontology annotations used at each sensor.

We demonstrate the efficiency of our implementation on an evaluation conducted on real-world and synthetic data sets. This chapter is organized as follows. In Section 4.2, we motivate our approach with a real-world example in an industrial setting. Section 4.3 presents the overall architecture of SuccinctEdge. The query optimizer and processor is presented in Section 4.4. Section 4.5 relates our research to existing work and Section 4.6 provides a detailed experimentation. We conclude the chapter and present directions for future work in Section 4.7.

4.2 Motivating example

In this chapter, we consider an upcoming deployment of SuccinctEdge at some of ENGIE's buildings where an IoT network is deployed. ENGIE is a multinational company operating in fields such as energy transition, generation and distribution.

Our running example focuses on data harvested from a building management system with a first focus on potable water distribution. Intuitively, a flow of measures are obtained from a network of sensors. A thorough analysis permits to detect anomalies such as leaks or other abnormal situations from, for instance, pressure and flow measurements. The measures are usually represented as text files (*e.g.*, CSV) but, thanks to some mapping assertions and dedicated digital services deployed through APIs, are transformed into a form of RDF graph (to be detailed later in this chapter) and annotated with concepts and properties of a domain ontology.

Figure 4.1 presents an extract of such a graph which concerns pressure and chemistry measures related to the water distribution management. Given such graph instances, our SuccinctEdge system executes queries that can detect some patterns such as anomalies linked to the water management system, *e.g.*, incorrect chemistry properties, network leak, etc. In a non edge computing context, each measure would transit on a computing network to a more powerful machine that could process the anomaly detection. Such an approach has several drawbacks: (i) it makes an intensive use of the computing network via large amount of message exchanges. As a consequence, the network can rapidly be overloaded, *e.g.*, devices on the edge of the network generally have low bandwidth, (ii) the high-end computing machine also risks to be overcharged and stressed from the amount of data

received (potentially from hundreds to thousands of sensors) and (iii) sending these data packets over the network is not cost-free for these sensors, *e.g.*, in terms of energy consumption.

In a context where anomalies are the exception, it makes sense to detect anomalies as close as possible to the sensors since it would require to (i) send fewer data over the computing network as that would occur only in anomaly cases, (ii) reduce decision latency and (iii) keep the high-end computing machine unstressed.

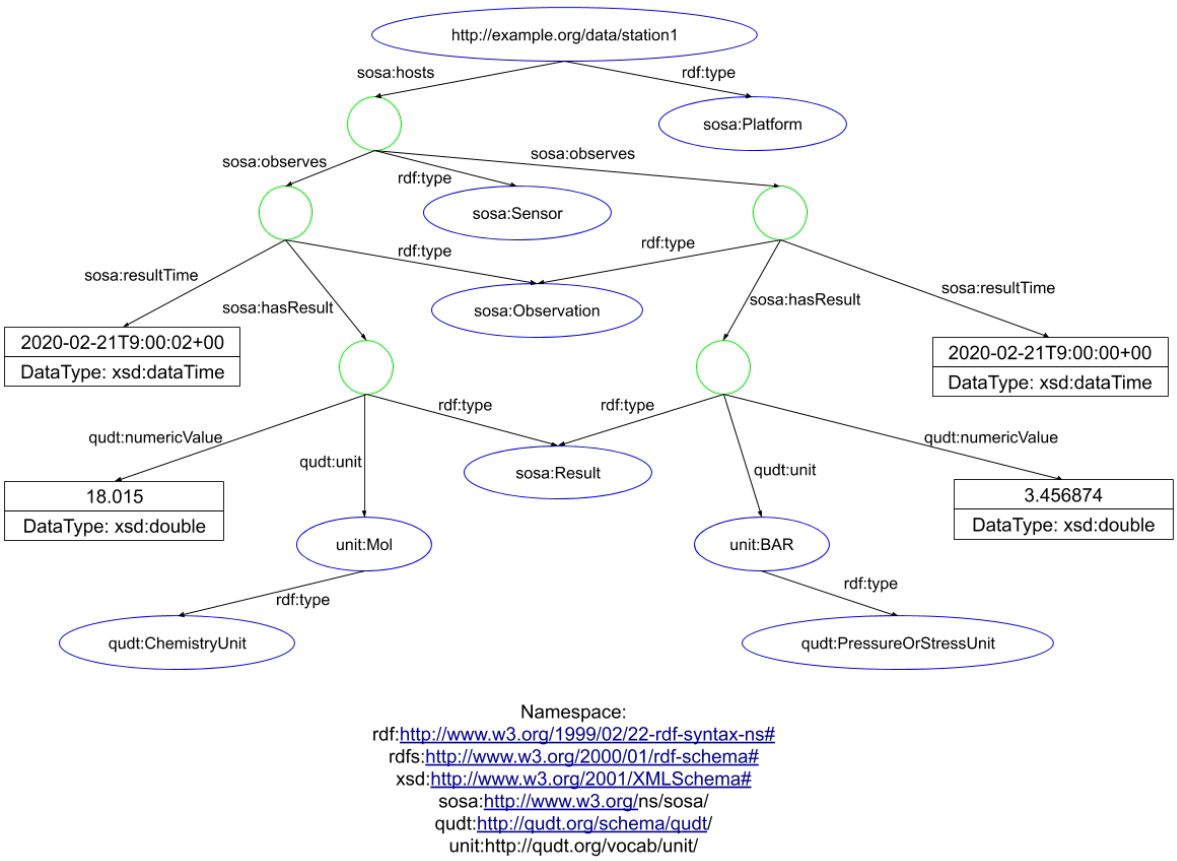

Figure 4.1: Graph extract of our use-case (green nodes are blank nodes)

In our experimentation at ENGIE, we are designing a query-based anomaly detection approach that does not require from the end-users a high level of expertise on the underlying domain ontologies and its reasoning services. Hence these users only express queries in relatively high concept terms and do not have to worry about the inferences which are handled automatically by the system. Expressing a query with abstract concepts, *i.e.*, high in the concept hierarchy, permits to write a single query that can tackle sensors performing similar measures but annotated with different concepts and possibly with different measure units. This is an important requisite for our use case where different sensor brands and types can coexist in a given network. The simplicity of this approach was expected from ENGIE for productivity reasons. In fact, it enables its sensor personnel to concentrate on their tasks and not on adapting a given query to the potentially large number of sensors

in an industrial setting. Moreover, adding new sensors with different types in a platform is easier as long as they conform to a set of standard ontologies.

For instance, in the following real-world example, 2 sensor platforms are measuring similar values, *e.g.*, pressure and chemistry-related, but each sensor annotates them with different concepts. Considering Station1 the pressure and chemistry are respectively annotated with $qudt : PressureOrStressUnit$ and $qudt : Chemistry$, while for Station2, it is resp. $qudt : Pressure$ and $qudt : AmountOfSubstanceUnit$. Moreover, the pressure value in Station1 is expressed in Bar while it is measured in HectoPascal in Station2.

Since, the QUDT ontology[7] states that:

$qudt : AmountOfSubstanceUnit \sqsubseteq qudt : Chemistry \sqsubseteq qudt : ScienceUnit$ and $qudt : PressureOrStressUnit \sqsubseteq qudt : PressureUnit \sqsubseteq qudt : MechanicsUnit$, a single SPARQL query can be written to address the peculiarities of each sensor at these 2 stations. The following query detects anomalies related to an incorrect pressure value (either expressed in Bar or HectoPascal) for sensors of stations 1 and 2:

```
SELECT ?t ?v ?u WHERE{
?x a sosa:Platform ; sosa:hosts ?s .
?s sosa:observes ?o ; a sosa:Sensor .
?o sosa:hasResult ?y ; a sosa:Observation ;
sosa:resultTime ?ts . ?y a sosa:Result ;
qudt:numericValue ?v1; qudt:unit ?u1 .
?u1 a qudt:PressureUnit. FILTER (?newV<3.00 || ?newV>4.50)
BIND(if(regex(str(?u1),"http://qudt.org/vocab/unit/BAR"),
        ?v1,
    if(regex(str(?u1),"http://qudt.org/vocab/unit/HectoPA"),
        ?v1/1000,0)
    ) as ?newV) .
}
```

Query 4.1: Anomaly detection SPARQL query example

4.3 Architecture overview

Before providing an overview of the SuccinctEdge RDF store, we describe a standard running setting at an ENGIE building. Typically, the person responsible for the building maintenance supervises a set of IoT devices from a SuccinctEdge server. From this central computer, the administrator is able to register new IoT devices installed in this set of buildings. Each IoT device typically runs a SuccinctEdge instance (client) which can execute many SPARQL queries. The administrator receives alerts from SuccinctEdge instances has abnormal sensor measures are occurring. Hence, each sensor modification (*e.g.*, a sensor is

[7] https://qudt.org/

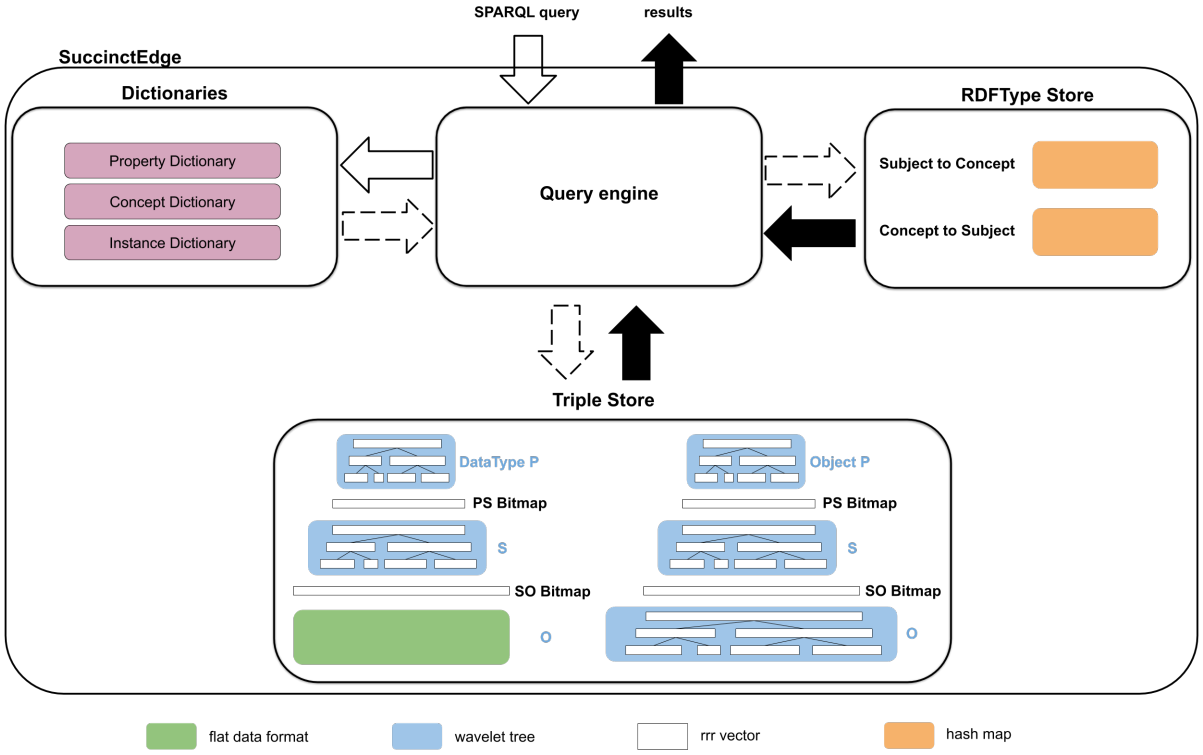

Figure 4.2: Architecture overview of SuccinctEdge

replaced due to a failure, a sensor data schema is modified) must go through an administration step which is performed on a central computer. Apart from such maintenance operations, this server also performs the pre-processing task consisting of encoding ontologies using the LiteMat scheme. In this context, and we consider in a large number of industrial settings, the ontologies are stable and thus rarely change. As explained previously, in SuccinctEdge, these ontologies take the form of a set of dictionaries (since their semantics are encoded via the use of LiteMat). These dictionaries are broadcasted to the different SuccinctEdge instances running at the edge.

An overview of SuccinctEdge's architecture is presented in Figure 4.2. Like most RDF stores, all triples are encoded according to some dictionaries. The underlying basic concept of a dictionary is to provide a bijective function mapping long terms (*e.g.*, URIs, blank nodes or literals) to short identifiers (*e.g.*, integers). More precisely, a dictionary should provide two basics operations : `string-to-id` and `id-to-string` (also referred in the literature as `locate` and `extract` operations). In a typical use of SuccinctEdge, the query engine will call the `locate` operation to rewrite the query into a list to match the data encoding, while the `extract` operation will be called to translate the result into the original format. In our case, we are obviously using LiteMAt as the encoding solution.

The Triple store component adopts a single index based on the predicate, subject, object (PSO) triple permutation. That is, the triples of the graph are sorted in ascending order over the P, S and O values of our dictionaries. The PSO order is motivated by the fact that the basic graph pattern of queries submitted to SuccinctEdge have predicates filled in with URIs (as opposed to variables). This corresponds to typical IoT use cases where queries

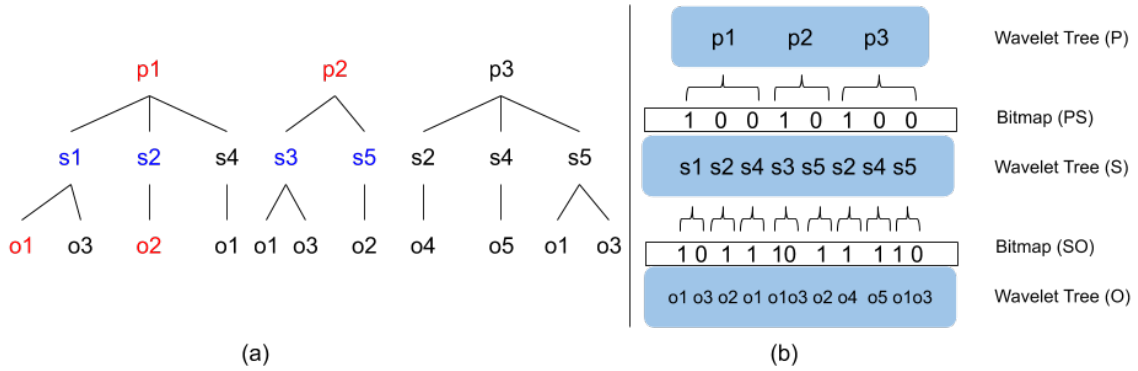

Figure 4.3: RDF graph representation: (a) as a PSO-based forest and (b) in SuccinctEdge as a combination of wavelet trees and bitmaps (only considering object properties)

are retrieving information from measures rather than serving to discover patterns in the graphs. In fact, there is no need for discovery since the graph patterns are well known in advance and are very rarely modified (*i.e.*, mostly due to sensor failure in industrial use-cases).

The Triple store component also highlights that we make a distinction between object (except for `rdf:type`) and datatype properties. In the former, objects are individuals and thus encoded with the respective instance dictionary while in the latter, objects are literals and stored using a flat data structure to store literals. This last data structure is motivated by the fact that it is not reasonable to create an entry in the instance dictionary for each new literal value. Intuitively, a sensor generally sends numerical values corresponding to physical measurement at a given time. Depending on the precision of these measures, the amount of different values to store in the instance dictionary is potentially infinite. So, we prefer to store the values as they have been sent by sensors, possibly with some redundancy, in order to prevent a complex and costly individual dictionary management.

In terms of data structures, WTs are used for the property and subject layers as well as the object layer for object properties. In order to relate a WT of one layer to another, we are using a BM. Figure 4.3(b) represents the triple set of Figure 4.3(a) where a WT corresponds to balanced tree of BMs. Intuitively the PS (respectively SO) bitmap permits to link a given P (resp. S) to several S (resp. O) values. In Figure 4.3(b), p1 is connected to s1, s2 and s4 because the PS bitmap starts with a 100 sequence: '1' states that the sequence of p1 starts with a given subject (s1) and the '00' states that 2 other subjects are linked to p1. Moreover, the 4th bit in the PS BM (*i.e.*, set to '1') starts the sequence of the second property entry in the P WT (*i.e.*, p2).

Finally, triples containing a `rdf:type` property are stored in the RDFType store layout. These triples generally represent an important proportion of the triple set in real-world RDF data sets. We simply store them in a red-black tree in order to maintain the search complexity to O(log(n)) while being fast when we insert `rdf:type` triples during database construction.

4.4 Query processing and optimization

In this section, we present the query optimization and processing solutions developed for SuccinctEdge. Their main goals are respectively to define an efficient TP join ordering, by combining heuristic and cost-based approaches, and to generate a physical plan composed of SDS operations (*i.e.*, access, rank and select).

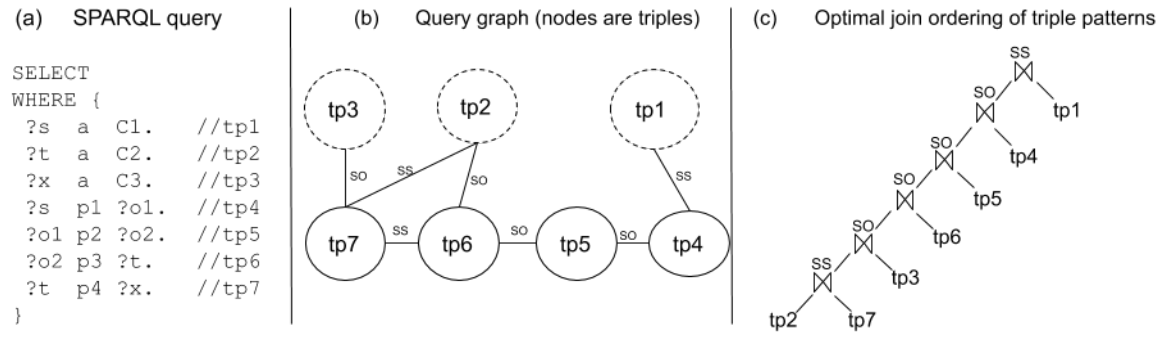

Figure 4.4: Query, query graph and join ordering

4.4.1 Query Optimization

The design of our query optimizer considers the limitations of the devices on which SuccinctEdge is running on, *i.e.*, limited memory space and computing power. Due to these constraints, our system only generates left-deep join trees which generally reduce the amount of memory used by the search process.

As stated in [36], join ordering is the most crucial issue in SPARQL query optimization. This is mainly due to the potentially high number of triple patterns and thus of join operations that one can find in BGPs. For instance, in our IoT building management experimentation, we have frequently encountered queries in the range of 10 joins.

In order to optimize a given SPARQL query, our query engine constructs a query graph where each TP of the SPARQL query corresponds to a node of the query graph. Each query graph node is also annotated to state whether its property is `rdf:type` or not. The nodes in this graph are connected if they share a common variable, hence forming a join. Moreover, the edges of this query graph are labeled with a join type, either SO or SS for respectively subject-object and subject-subject joins.

Figure 4.4(b) displays the query graph associated with the SPARQL query presented in Figure 4.4(a). This query contains 7 TPs, denoted tp1 .. tp7. The dotted nodes in the query graph correspond to `rdf:type` TPs.

Given a query graph, our optimizer uses Algorithm 2 to produce a join order. Intuitively, starting from a given TP, it invokes an overloaded $getMostSelective$ method to search for the next TP to join with. This method uses a set of static rules together with some data statistics. In terms of the former, we have been influenced by Heuristic 1 of [50]

which defines an execution order for the 8 possible TP combinations. In the context of SuccinctEdge, we do not need to consider all combinations since TPs with either zero or three variables, *i.e.*, (s, p, o) and $(?s, ?p, ?o)$, are highly unlikely to occur in a real-world IoT SPARQL query. Intuitively, this heuristic states that TPs with the fewest variables should be executed first. Our adaptation re-orders the original proposition by taking into account the fact that our access paths are limited to PSO for non `rdf:type` properties and to SO/OS paths for `rdf:type` triples. As presented in Section 4.3, the latter access path (SO/OS on `rdf:type`) is more efficient than the one based on the SDS structures. Our TP order is thus:

(s,rdf:type,?o)>(?s,rdf:type,o)>(s,p,?o)>(?s,p,o)>(?s,p,?o), where p denotes any property different from `rdf:type` and the relation tp1 > tp2 states that tp1 should be executed before tp2. The (s,p,?o)>(?s,p,o) order is due to the navigation mode in our multi-layer SDS triple representation which is PSO based, *i.e.*, it is more efficient to retrieve objects given a subject/property pair than to compute subjects from a property/object pair. The `?s rdf:type ?o` TP is not considered relevant in a practical IoT context.

Algorithm 2: Computation of a TP order

Input: query graph G
Output: ordered sequence of TPs

```
1 tpOrder ← ∅;
2 n ← getMostSelective(rdf : type);
3 tpOrder ← tpOrder + n;
4 while not all G nodes are in tpOrder do
5 |   n ← getMostSelective(tpOrder);
6 |   tpOrder ← tpOrder + n;
7 end
8 return tpOrder;
```

This first heuristic is generally not sufficient to decide which TP to execute first among a set of other TPs. Hence, we are considering a second heuristic that takes into consideration the linearity required by a left-deep join tree and examines the types of join possible between TPs. Due to the PSO self-index SDS structure used for non-`rdf:type` triples, SS joins are preferred over SO joins, *i.e.*, $S \bowtie S > S \bowtie O$. Other forms of joins, *i.e.*, SP, OP, PP have a lower priority since they are rarely encountered in the setting where SuccinctEdge is relevant.

In order to minimize intermediate results, the optimizer also relies on a set of statistics computed at dictionary creation-time. Intuitively, each dictionary persists the number of occurrences of each of its entries, *i.e.*, concept, property and non-literal individuals. Our statistic approach considers the hierarchy position of a given concept or property when computing the total number of triples it is involved in. For example, with the following concept hierarchy $C_2 \sqsubseteq C_1 \sqsubseteq C_0$ and $C_3 \sqsubseteq C_0$, the set of triples involving instances of concept C_0 will be the set of instances of type C_i with i $\in (0, 1, 2, 3)$. A similar process is applied to get the correct statistics for properties involved in a property hierarchy. Finally,

some statistics are also computed at run-time, *e.g.*, the BM and WT data structures facilitate the computation of certain statistics. For instance, Algorithm 3 computes the number of triples containing a certain property.

Algorithm 2 first starts with the identification of the most selective `rdf:type` TP with an SS join. In the case it does not find an `rdf:type` TP or finds only `rdf:type` TP connected with SO joins, it then selects a non-`rdf:type` TP to start with. In the case several TPs satisfy our constraint, the statistics permit to take a decision. That first TP is appended to our $tpOrder$ sequence. We then loop over the remaining nodes of the query graph until all TPs have been added to the sequence. At each iteration of the loop, the $getMostSelective$ method considers TPs in the $tpOrder$ sequence and searches for the next TP to append to this sequence. This search is again based on our two heuristics and the usage of statistics.

Example: The left-deep join tree displayed in Figure 4.4(c) has been defined using Algorithm 2 considering that tp2 is more selective than tp1, *i.e.*, the number of occurrences of C2 is lower than the one of C1. Once tp2 has been selected, the optimizer has the choice to join it with tp6 or tp7. tp7 is chosen since a SS join is preferred to a SO join. At this stage, the number of occurrences of concept C3, *i.e.*, tp3, can be lower than the number of already computed binding for ?x, and thus tp3 is selected. Given that tp2, tp7 and tp3 have already been considered, tp6 is the only alternative that can be considered and similarly for the remaining TPs, *i.e.*, tp5, tp4 and tp1.

Algorithm 3: Compute the number of triples corresponding to a certain predicate.

Input: Predicate p

Output: Number n

1 $id_p \leftarrow FindIdFromDictionary(p)$;
2 $index_p \leftarrow wt_p.select(1, id_p)$;
3 $index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
4 $index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
5 $index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
6 $index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
7 $n \leftarrow index_{oEnd} - index_{oBegin}$;
8 return n;

4.4.2 Query processing

Once an order is defined by SuccinctEdge's query optimizer, our system translates TPs into SDS's standard operations: `access`, `rank` and `select`. We are using an additional function, namely $rangeSearch(a, b, c)$, which finds all the occurrences of value c in the interval (a, b). It uses a binary search, *i.e.*, due to the ordering imposed on subjects for a given property, and returns the indexes of matching values. The use of this function speeds up query execution since it efficiently prunes searches by just computing the boundaries of the Subject WT, *i.e.*, first and last subject values of a given property, instead of scanning

all values of that interval. A similar optimization is used when searching objects of given property/subject pair, *i.e.*, using the boundary of Object WT.

We now present three translation examples in Algorithm 4, 5 and 6 for respectively the $(s, p, ?o)$, $(?s, p, o)$ and $(?s, p, ?o)$ TPs. Algorithm 4 shows how to retrieve an answer set with a $(s, p, ?o)$ TP. The idea is to first compute an interval of object values related to a given predicate and subject pair. This is performed by navigating through our BM and WT structures. All the objects in this interval are the results of this TP. Algorithm 5 retrieves all the subjects of a $(?s, p, o)$ TP. Unlike Algorithm 4, we can not locate all the subjects directly. So our strategy is to get the interval of all the objects corresponding to the known predicate top-down, after which we locate the object in this interval (there may be multiple appearances) and get the corresponding subjects. Algorithm 6 aims to retrieve all the subjects and object with a given predicate of a $(?s, p, ?o)$ TP. The strategy is to find the interval of subjects with the help of predicate's index. Then, for each subject in this interval, we retrieve all the corresponding objects and add these triples to result set.

Algorithm 4: Search the triple pattern $(s, p, ?o)$

Input: Subject s, predicate p

Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$id_s \leftarrow FindIdFromDictionary(s)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
for $index_s$ *in* $wt_s.rangeSearch(index_{sBegin}, index_{sEnd}, id_s)$ **do**
    $index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
    $index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
    for $index_o \leftarrow index_{oBegin}$ **to** $index_{oEnd}$ **do**
        $id_o \leftarrow wt_o[index_o]$;
        add (id_s, id_p, id_o) into res;
    end
end
return res;

In cases where reasoning services over properties are necessary to provide an exhaustive answer set, we can replace $index_p$ with a continuous interval corresponding to a LiteMat interval. This interval is efficiently computed given the order imposed on leaves of a certain WT, *e.g.*, Property WT for the property hierarchy. The larger and deeper a property hierarchy, the more efficient this optimization approach since it prevents from navigating in the complete tree of a given WT. An example of this replacement is shown in Algorithm 7, which is transformed from Algorithm 6 in case of a property hierarchy reasoning. Similar transformations can be applied to Algorithms 4 and 5.

TPs containing `rdf:type` are processed differently using the RDFType store component, where some simple structure look-ups permit to efficiently retrieve to subjects of a

Algorithm 5: Search the triple pattern $(?s, p, o)$

Input: Predicate p, object o

Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
$index_{oBegin} \leftarrow bitmap_{so}.select(index_{sBegin} + 1, 1)$;
$index_{oEnd} \leftarrow bitmap_{so}.select(index_{sEnd} + 2, 1)$;
for $index_o$ *in* $wt_o.rangeSearch(index_{oBegin}, index_{oEnd}, id_o)$ **do**
    $index_s \leftarrow bitmap_{so}.rank(index_o + 1, 1) - 1$;
    $id_s \leftarrow wt_s[index_s]$;
    add (id_s, id_p, id_o) into res;
end
return res;

given concept or the concepts of a given subject.

The next step corresponds to joining the results obtained from the execution of TPs. This occurs when different TPs share a common variable. One of our joining approach amounts to propagate variable assignments from one TP to another. Consider the triple set of Figure 4.3(a) and TPs $(?s, p1, o1)$ and $(?s, p2, ?o)$. The first TP gets the following assignments: $?s : \{s1, s2\}$ which will serve to dynamically generate $(s1, p2, ?o)$ and $(s2, p2, ?o)$ for the second triple.

During the join operation, we can benefit from a merge join (due to the original PSO value order) in certain cases when the values assigned to a joining variable to the TP are kept in order. For instance, in the case of a star-shaped BGP, *e.g.*, $(?s, p1, o1)$ and $(?s, p2, ?o)$, thanks to the facts that all the subjects connected to a certain predicate are ordered and that all the objects connected to one certain subject are also ordered, we can perform a merge join on the subject variable. Figure 4.5 provides a graph pattern (on the right side) and an RDF Graph (left side). From the first TP, we can retrieve $\{(p1, s1, o1), (p1, s2, o1)\}$ as the answer set. Clearly, since the subjects are ordered for a given predicate, the system can easily use a merge join with the 2nd TP of the query. In cases where the order is not guaranteed, we use nested loop joins.

Previous executions steps are repeated until all the TPs have been processed. Then the answer set of the query is translated using our dictionaries and presented to the end-user or application.

Algorithm 6: Search the triple pattern $(?s, p, ?o)$

Input: Predicate p
Output: Results res

$id_p \leftarrow FindIdFromDictionary(p)$;
$index_p \leftarrow wt_p.select(1, id_p)$;
$index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
$index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
for $index_s \leftarrow index_{sBegin}$ **to** $index_{sEnd}$ **do**
    $id_s \leftarrow wt_s[index_s]$;
    $index_{oBegin} \leftarrow bitmap_{so}.select(index_s + 1, 1)$;
    $index_{oEnd} \leftarrow bitmap_{so}.select(index_s + 2, 1)$;
    for $index_o \leftarrow index_{oBegin}$ **to** $index_{oEnd}$ **do**
        $id_o \leftarrow wt_o[index_o]$;
        add (id_s, id_p, id_o) into res;
    end
end
return res;

Algorithm 7: Search the triple pattern $(?s, p, ?o)$ with property hierarchy reasoning

Input: Predicate p
Output: Results res

$id_{pBegin} \leftarrow FindIdFromDictionary(p)$;
Compute id_{pEnd} with help of LiteMat;
$index_{p_begin} \leftarrow wt_p.select(1, id_{pBegin})$;
for *each* id_p *from* $index_{pBegin}$ **to** *position i where* $wt_p.access(i) > id_{pEnd}$ **do**
    $index_{sBegin} \leftarrow bitmap_{ps}.select(index_p + 1, 1)$;
    $index_{sEnd} \leftarrow bitmap_{ps}.select(index_p + 2, 1)$;
    for $index_s \leftarrow index_{sBegin}$ **to** $index_{sEnd}$ **do**
        $id_s \leftarrow wt_s[index_s]$;
        $index_{oBegin} \leftarrow bitmap_{so}.select(index_s + 1, 1)$;
        $index_{oEnd} \leftarrow bitmap_{so}.select(index_s + 2, 1)$;
        for $index_o \leftarrow index_{oBegin}$ **to** $index_{oEnd}$ **do**
            $id_o \leftarrow wt_o[index_o]$;
            add (id_s, id_p, id_o) into res;
        end
    end
end
return res;

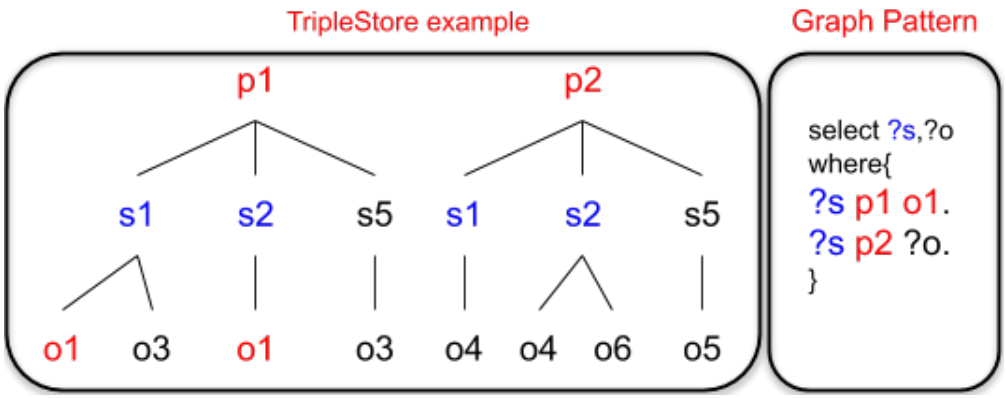

Figure 4.5: Merge join example

4.5 Related work

Existing RDF systems can be classified with two aspects, storage mode and computing mode, where storage mode refers to hard-drive based storage or RAM based storage, computing mode indicates it is designed for Cloud Computing (or a single powerful server) or Edge Computing. In the following parts of this section, we are going to list some existing RDF systems within these two aspects.

Apache Jena

Apache Jena is a very popular open-source, Java-based Semantic Web framework, it is equipped with various types of programming interfaces such as APIs to manipulate RDF(S) and OWL, an API for RDF query processing, denoted ARQ. The system supports different data management solutions, among which Jena TDB, with its in-memory and disk-based persistence, is getting the most of our interest.

The in-memory version provides RAM-based approach to store RDF data. It permits to process a query at a higher speed compared to disk-based approach. However, this RAM-based approach doesn't consider data compression aspect which could be important in an edge device. The hard-drive version persists all information on disk. The data set of this latter version consists of Node Table, Triple and Quad Indexes and Prefixes Table. Taking a closer look to Triple and Quad Indexes. This part holds multiple indexes which can gain higher query speed by sacrificing memory footprint.

RDF4J

RDF4J[8] is an open source RDF management system, it supports SPARQL 1.1 and the read/write of different RDF data file formats(RDF/XML, N-triples *etc.*).

RDF4J offers both RAM based and hard-drive based storage mode, the RAM based mode has an excellent performance for small data sets while the disk persisted mode is designed

for medium-sized data sets in the order of 100 million triples. We consider RDF4J is not designed for Edge Computing as data compression doesn't appear at the core of its design principle.

RDF-3X

RDF-3X[35][37] represents as a fast, disk-based RDF store. The system proposes to create 6 full indexes(SPO, SOP, POS, PSO, OSP, OPS), 6 aggregated indexes(SP, PS, SO, OS, PO, OP) together with 3 one-value indexes(S, P, O) for all the RDF triples. With this fully indexed RDF data structure, the system can give responses to any kind of triple patterns(TPs) with the minimum searching effort, which means no matter where the variables situate in a TP, the system can always search into the proper index where most of the potential results are neighbored.

As for query processing, RDF-3X benefits from a bottom-up dynamic programming algorithm, where for the potentially occurred star-shape joins at the bottom, the operations have an opportunity be intensively optimized with a merge join strategy thanks to all kinds of indexes in the system.

Due to these characteristics, RDF-3X targets powerful servers rather than small Edge devices.

RDFox

RDFox[34] is an RAM based, centralized RDF system designed for powerful servers. [42] presented a distributed system based on RDFox approach but it seems like the system is not maintained.

RDFox stores RDF data in a data structure called TripleTable which contains a TripleList where a number of indexes are implemented in order to accelerate query processing.

RDFox supports rule-based Datalog inferences with a materialization strategy which proposes to store all the newly generated data by inference in the database. It support also `owl:sameAs` by applying a query rewriting strategy. During our evaluation, we tried to experimented RDFox on a Raspberry Pi but we could run the system on this type of device.

WaterFowl

WaterFowl[13] was designed as a first attempt to use SDS for RDF storage and query processing. Although its compactness can be used in an edge computing setting, it lacks the different object storage implementation and query processing (including optimization) features of SuccinctEdge.

RDF4Led

RDF4Led[51] is an RDF database system designed for Edge Computing, its design considers the flash-based storage structure with an algorithm which can reduce the join memory footprint.

The system benefits from a molecule-based storage model at the physical layer together with a buffer layer to accelerate the page searching. More over, the system is implemented with a join propagation algorithm which can efficiently reduce the memory consumption during the join operation.

μRDF Store

μRDF[9] is an RDF store designed for micro-controller with very limited memory. It store RDF triples only once with a navigational queries optimized index. This reduce the need of memory as their experiment sets are of 8-64K RAM. As for query processing, considering to avoid large intermediate results in their memory-limited device, µRDF applies a greedy strategy: as soon as a binding is found for a triple pattern, the next triple pattern is processed.

However, μRDF's aiming environment is too resource-constrained to implement a sufficient-function RDF store which usually demands join optimization and reasoning services.

Wiselib TupleStore

Wiselib TupleStore[20] is an RDF store designed for light-weight OS *e.g.*, TinyOS and Contiki, it applies B+ tree based hash set which is used for dictionary and tuple container implementations. Based on this fundamental data structure, the system supports three basic operations: **insert**, **query** and **erase** to operate on its RDF data, **insert** and **erase** allow to modify the RDF data and **query** permits to extract information based on a basic triple pattern.

Although Wiselib TupleStore is a good attempt of RDF store for edge devices, its desired devices are too resource-constrained, which could be the reason why the system doesn't support queries with multiple triple patterns which demand join operations. More over, the system doesn't support reasoning services which could be useful in real case.

Header Dictionary Triples(HDT)

Finally, it makes sense to write about Header Dictionary Triples (HDT)[29] which is not a full-fledge RDF store but a popular compact data structure and binary serialization for RDF data. The Triples component of HDT requires that triples are sorted in a specific order, *e.g.*, SPO. The triples are stored in so-called Bitmap Triples which represents a forest of RDF trees, *e.g.*, each tree is rooted with a given subject value. The remaining tree layers, *e.g.*, for P and O, each correspond to a sequence of identifiers and a bit sequence which

connects layers like our BMs. Like HDT, SuccinctEdge represents RDF triples as trees but it makes an extensive use of WTs and depends on three different storage approaches, namely Object-triple-store, Datatype-triple-store and RDFType-store. Moreover, SuccinctEdge is equipped with a full-fledged query processing component and supports RDFS reasoning within SPARQL queries.

4.6 Evaluation

4.6.1 Experimental setting

Our experimentation is conducted on a Raspberry Pi 3B+ which can be considered as a typical edge computing device on which we can run some sophisticated programs. This small computer is equipped with a Cortex-A53 (ARMv7l) 32-bit SoC 1.4GHz CPU and 1GB LPDDR2 SDRAM. A SD-card, a widely used memory solution on such devices, of 8GB is used as persistent storage.

Considering the evolution of technology, it is widely accepted that edge computing devices will be more and more powerful in the near future. Hence, it is quite obvious that devices with sufficient calculation power and memory, *e.g.*, Raspberry Pis, Odroids, etc. , will be deployed at the edge of networks.

4.6.2 Datasets and queries

The experimentation uses both synthetic and real-world data sets. This duality is motivated by the current lack of large graphs emitted from sensors at our industrial partner. In fact, our real-world data sets, which correspond to the water management distribution in ENGIE's building, consist of 250 and 500 triples. They are denoted with their number of triples in this experimentation.

Due to these size limitations, it is not possible to stress SuccinctEdge in terms of graph sizes. Hence, we are also experimenting with the synthetic Lehigh University Benchmark (LUBM)[8] which can be easily configured to produce large data sets. Starting from a LUBM with one university, *i.e.*, composed of over 103.000 triples (denoted 100K), we created several triple subsets of 1.000, 5.000, 10.000, 25.000 and 50.000 triples which are respectively denoted as 1K, 5K, 10K, 25K and 50K in the remaining of this section. They are used to evaluate the behaviors of the five evaluated data management systems. Note that some of these synthetic data sets have triple set size way beyond what most sensors are currently emitting in real-world industrial use-cases. All submitted queries are detailed in Appendix A..1 and data sets are available on the system's Github page.

[8] http://swat.cse.lehigh.edu/projects/lubm/

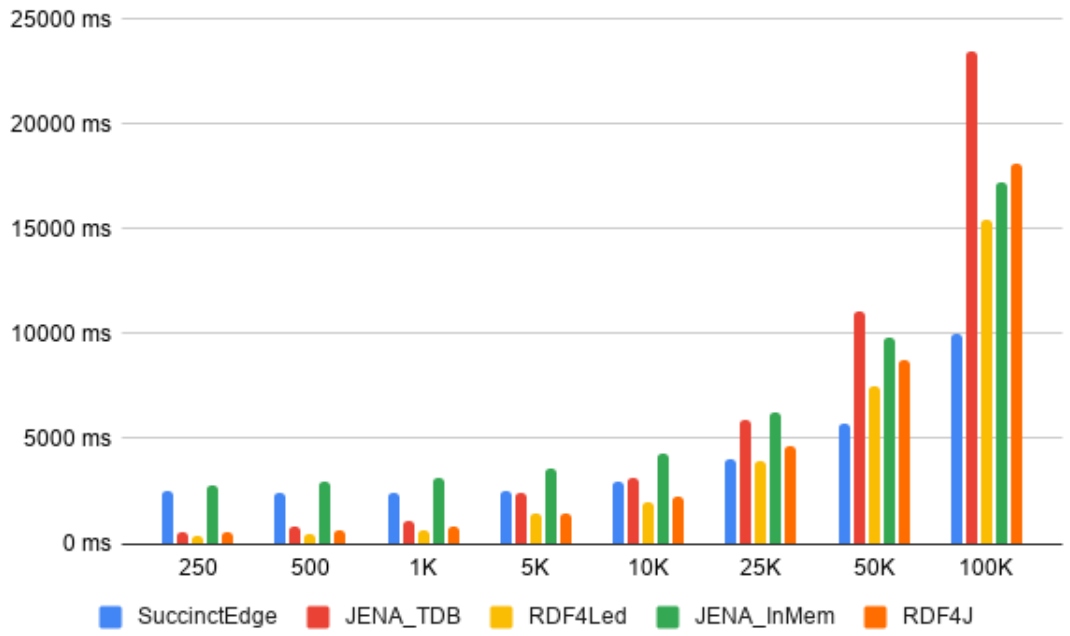

Figure 4.6: Construction time comparison

4.6.3 Experimentation results

In this section, we are aiming to compare the previously mentioned RDF stores (*i.e.*, Jena TDB, Jena in-memory, RDF4Led, RDF4J and SuccinctEdge) on the following dimensions: graph construction time, memory footprint (*i.e.*, the storage space taken by different systems with the previous data sets), query execution performances on different triple patterns and basic graph patterns. Lastly, we evaluate the performance (duration time) of queries which necessitate reasoning services to produce an exhaustive answer set.

Back-end construction time

The back-end construction time corresponds to the time taken by each system to read the data set file and to construct its proper storage layout (including indexes in the case of all systems except SuccinctEdge which is self-index) on which queries can be asked.

In order to fully evaluate the performances of all the systems, we compare the back-end construction time of these systems with data sets ranging from 250 to over 100.000 triples. Figure 4.6 provides details on this experimentation. SuccinctEdge doesn't show much advantage when data set is rather small (up to 1.000 triples). We attribute this to the fact that the SDS-Lite library which is responsible for creating SuccinctEdge's BMs and WTs has an important start-up overhead that is relatively important compared to the effective duration of the structures. We consider that this may be optimized in future work, but it is out of the scope of this thesis. However, as the data sets grow larger, our system outperforms all other systems.

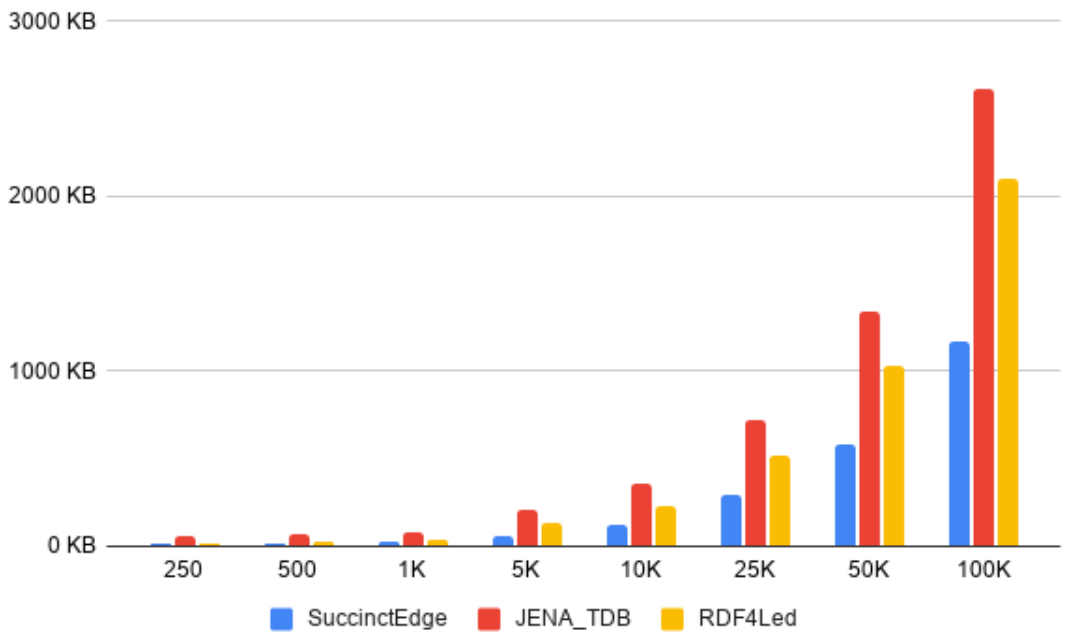

Figure 4.7: Dictionary size comparison

Storage size

As SuccinctEdge is an in-memory RDF system, it is difficult to directly compare the memory occupation against Jena TDB and RDF4Led (which are both disk-based RDF stores). We persisted all the data structures existing in SuccinctEdge to disk in order to make a fair comparison.

We separately consider the dictionary and triple storage spaces. Figure 4.7 provides the three systems' dictionary sizes for all 8 data sets. In all cases, Jena TDB requires the largest memory footprint and SuccinctEdge takes about half of the size of RDF4Led.

Considering the triple storage space, displayed in Figure 4.8, SuccinctEdge consumes much smaller space thanks to its SDS-based storage implementation and self-index approach. This enables to reach one of our goal which is to store as much data as possible in a given amount of RAM.

We are also comparing the main-memory footprint of SuccinctEdge with the in-memory systems, *i.e.*, RDF4J and Jena_InMem. In this evaluation, it is not possible to distinguish between the space used for the dictionaries and the data sets. So we provide the total space amount. Figure 4.9 yields the experiment results. We can see that as the amount of data grows, SuccinctEdge gradually shows its strength in saving memory space. We mainly attribute this to the size of the indexes stored by both RDF4J and Jena_InMem.

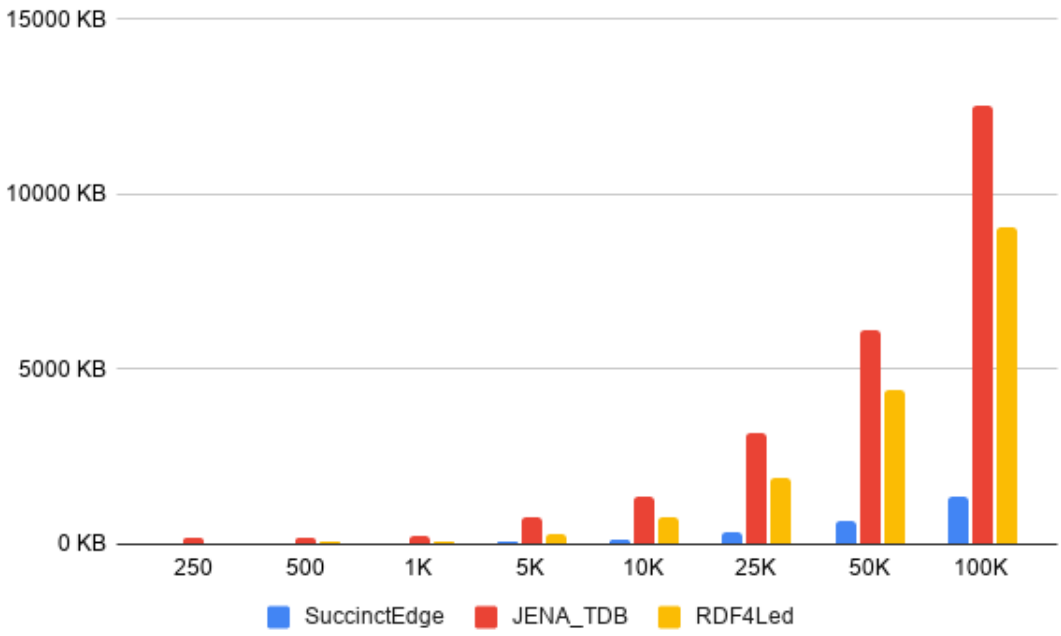

Figure 4.8: Storage size without dictionary comparison

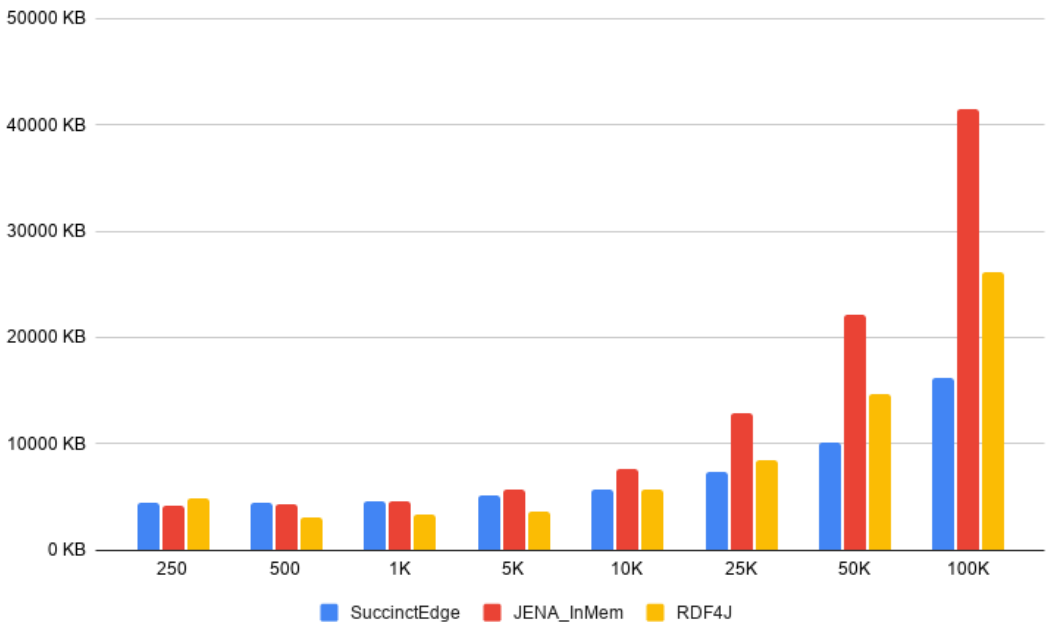

Figure 4.9: RAM footprint comparison

Table 4.1: Data retrieval for a single **S,P,?o** TP. The first row represents the number of triples in the answer set. All times in ms. Bold times are a column's most efficient.

	Query performance				
Query name	S6	S7	S8	S9	S10
Selectivity	4	66	129	257	513
SuccinctEdge	**0.3**	**3.5**	**6.2**	**10.9**	23.3
RDF4Led	12	28	33	47	84
Jena TDB	7	16	22	27	33
Jena_InMem	5	11	15	19	29
RDF4J	3	6	10	11.1	**13**

Triple pattern query

Considering query processing, we start the evaluation with single triple patterns,*i.e.*, excluding the cost of join operations, in order to directly compare the performance of data retrieval in different systems.

We first consider the two interesting triple patterns containing a single variable in the context of SuccinctEdge: **S,P,?o** (queries S1 to S5) and **?s,P,O** (queries S6 to S10) . Moreover, we consider these two triple patterns with different selectivity, *i.e.*, result sets ranging from 4 to 521 tuples. Table 4.1 and 4.2 provide the results of this experimentation for the LUBM1 dataset (over 100.000 triples).

As said previously, in an IoT setting, we are mainly interested in executing a query on the freshest data and such a query is generally execute only once per graph instance. Hence, we are only considering hot runs.

SuccinctEdge outperforms other systems on almost all query selectivity. It is only on relatively non-selective, at least considering an IoT context, that SuccinctEdge gets beaten by RDF4J (S4, S5 and S10). Considering our potable water distribution running example, the answer set of each query is clearly very selective. That is only a small set of tuples are retrieved from a specific query out of a given measure. We consider that this will be the case for many industrial situations. Thus, high selective queries is clearly the main playground for RDF stores running in Edge computing. In the case of selective queries, SuccinctEdge can be up to one order of magnitude faster than its RDF4J most direct competitor, *e.g.*, Table 4.1 S6 with a result set of size 4.

Figure 4.10 shows the results of several randomly picked **?s,P,?o** queries (triple patterns with a constant predicate and variable subject and object, denoted S11 to S15). We can see from the results that SuccinctEdge outperforms the other systems. Clearly, the conclusion obtained on single triple patterns with a single variable that the more selective, the more efficient SuccinctEdge is compared to the other systems, is confirmed. We attribute this to SuccinctEdge's in-memory approach and structure which is **?s,P,?o**-friendly due to its PSO self-index approach. Moreover Jena TDB and RDF4Led also have PSO or POS indexes but are disk-based database, for whom, loading data from disk takes a non-negligible time. The

Table 4.2: Data retrieval for a single **?s,P,O** TP. The first row represents the number of triples in the answer set. All times in ms. Bold times are a column's most efficient.

	Query performance				
Query name	S1	S2	S3	S4	S5
Selectivity	5	17	135	283	521
SuccinctEdge	**0.7**	**1.5**	**10.1**	20.7	32.0
RDF4Led	6	9	51	71	81
Jena TDB	7	8	30	32	41
Jena_InMem	7	8	15	21	27
RDF4J	3	3	11	**16**	**21**

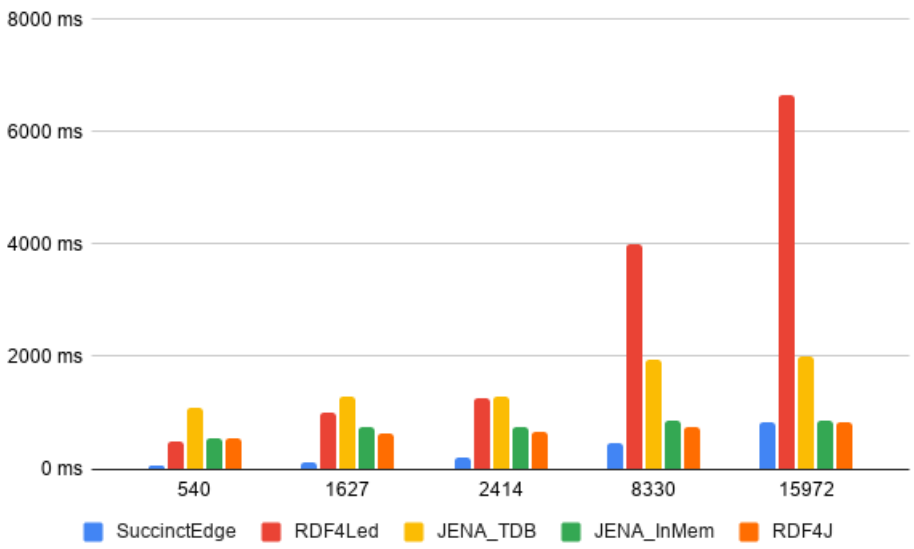

Figure 4.10: Data retrieval of queries with only one triple pattern of type **?s,P,?o**, the x-axis represents the number of triples in the answer set.

numbers of triples in the answer sets of our single variable TP experimentation are much smaller than that of the **?s,P,?o**. This leads to greater differences between the different systems. This is again due to the fact that RDF4Led and Jena TDB are loading data from disk. Nevertheless, we can consider that retrieving over 500 tuples at a time from a single sensor is already quite unusual for an IoT use case. The comparison with in-memory stores (RDF_InMem and RDF4J) highlights that SuccinctEdge is faster for answer sets lower than 10.000 tuples. At 16.000 result set tuples, The three systems behave similarly. Again, from the point of view of our experimentation partner, this is currently unusual for real-world industrial IoT use cases.

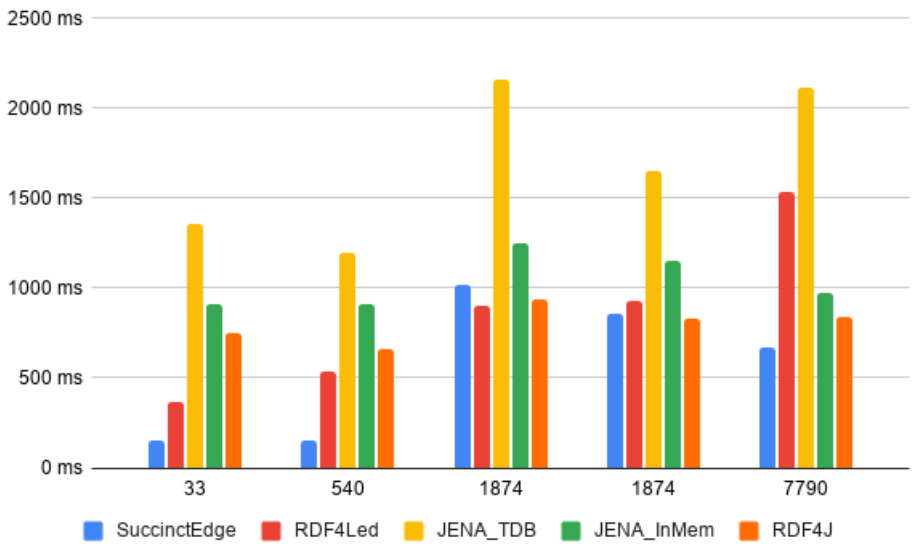

Figure 4.11: Queries with multiple triple pattern (x-axis corresponds to the number of tuples in the answer set)

Graph pattern query

We now compare performances over queries containing multiple triple patterns, *i.e.*, requiring joins. Four queries with different selectivity values (answer sets ranging from 540 to close to 8.000 tuples) have been executed. They are denoted M1 to M5 and contain up to 10 TPs in the BGP. We can see in Figure 4.11 that RDF4Led and SuccinctEdge are always outperforming Jena TDB. SuccinctEdge is either more efficient than RDF4Led or slightly less efficient that RDF4Led. This showcases that in some cases RDF4Led finds a better TP query ordering strategy than SuccinctEdge and/or benefits from its large set of available indexes. Considering the latter, it is a price we are willing to pay for a lower memory footprint. Nevertheless, the former reason emphasizes that we can improve our query optimizer.

The comparison with the in-memory RDF stores emphasizes that the three systems behave similarly except for highly selective queries where SuccinctEdge is again more efficient. The differences between the query executions depend on the patterns used in the BGP of these five queries. Overall, we are satisfied that our system, with a single index, is at least at the same level than the two other systems.

Queries with RDFS reasoning

Our final experimentation concerns queries requiring some reasoning services. We have generated six queries (denoted R1 to R6) containing a mixture of RDFS:subClassOf and RDFS:subPropertyOf inferences. These queries present different selectivity characteristics, ranging from 15 to 8.345 tuples in the answer sets and contain up to 10 TPs in the BGP.

For SuccinctEdge, the reasoning service is automatically supported by LiteMat's encod-

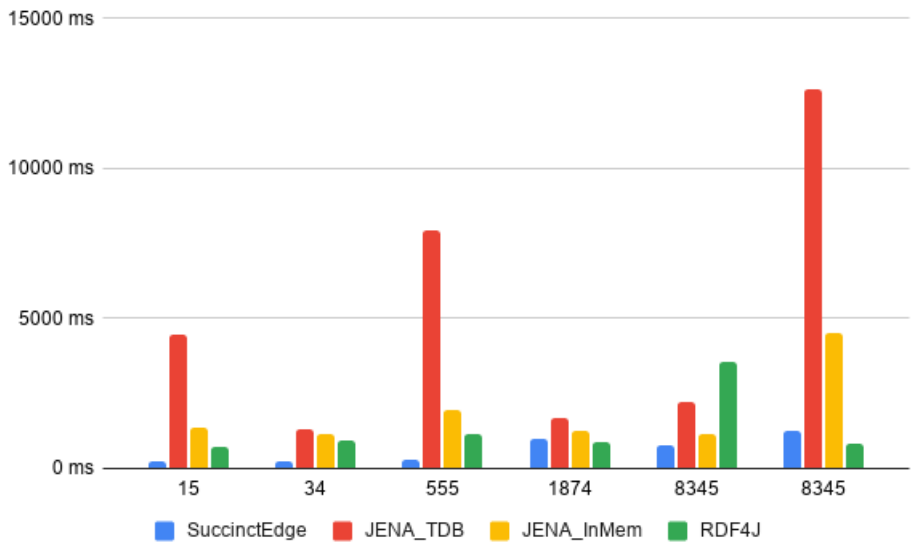

Figure 4.12: Queries with RDFS reasoning (x-axis corresponds to the number of tuples in the answer set)

ing and is hence native in the system. This is not the case for the other systems for which we have rewritten each query as the union of all the possible sub-queries. Since RDF4Led doesn't support the SPARQL UNION clause no results are presented in Figure 4.12 for this system. Obviously, SuccinctEdge is much more efficient than Jena TDB. It is quite logical that the more entailments the query requires, the more efficient SuccinctEdge is compared to a system like Jena TDB.

As for Jena_InMem, it performs better than Jena TDB while still falling behind SuccinctEdge. When compared with RDF4J, SuccinctEdge performs better or similarly depending on the complexity of the reasoning services, *i.e.*, number of SPARQL UNION clauses. Note that we provide manual query rewriting to the Jena and RDF4J systems while these systems could implement the reasoning task with their APIs. In doing so, we provide a clear advantage to these systems since they do not have to load the ontology to perform the reasoning. Moreover, the extra cost of computing the UNION rewriting is not considered in the times of the Jena and RDF4J executions.

4.7 Conclusion

We have presented the first, to the best of our knowledge, KG inference-enabled data management system designed for Edge computing that is equipped with reasoning services, in fact RDFS++. Thanks to its design characteristics, *i.e.*, unique index, compactness, in-memory approach, we have demonstrated that SuccinctEdge outperforms its direct competitors on the following dimensions: query performance on different query patterns, efficiency of reasoning services, back-end size and creation time.

The system is currently being deployed at some building facilities at ENGIE and should

help in detecting anomalies in resource, *e.g.*, gas, water, distribution and energy consumption. Due to its generic nature, SuccinctEdge is relevant for many IoT use cases such as anomaly and risk detection, supervising energy production and distribution.

In the future, we are aiming to improve the query optimizer and support queries ranging several graphs. We are also considering to design a more efficient management of objects linked to datatype properties and to increase the expressiveness of supported ontology languages, *e.g.*, OWL2RL. Moreover, we are considering the possibility of exchanging information with a larger graph portion that would reside in a database management system residing on the Cloud.

Chapter 5

Streaming SuccinctEdge

Those who flow as life flows know they need no other force.

– Lao Tzu

In this chapter, we present extensions that enable the continuous query processing of unbounded data emitted from multiple sensors.

5.1 Introduction

In this chapter, we extend SuccinctEdge with the capacity to process unbounded data, *i.e.*, data streams. This is an important features in an IoT context where anomalies and other exceptional situations must be detected as early as possible. Designing such a new version of SuccinctEdge requires the extension of some components, *e.g.*, continuous query processor, and additional software modules, *e.g.*, dealing with the continuous arrival of data streams. Considering the latter, providing guarantees on low latency (the time between the start and the completion of an event) and high throughput (the total amount of work done in a given time) is a primordial design aspect. In order to address this issue, we experimented with two existing open source messaging systems. One of our extension relies on the Eclipse Mosquitto[1] system. This lightweight message broker is suitable for Edge devices and offers a set of essential features for a stream processing platform. The other extension benefits from Apache Edgent[2], which provides analytic capability of data coming from multiple sensor devices while still saving memory footprint. Using these frameworks enables us to concentrate on tasks such as query processing and optimization, reasoning and supporting several streaming models and window strategies. To the best of our knowledge, our system, named Streaming SuccinctEdge, is the first system fulfilling the need for RDF stream processing at the Edge with reasoning capacities. In fact, the KG compliant Edge computing ecosystem either proposes RDF stores for Edge devices which are not handling data streams, *e.g.*, RDF4Led [51], or RDF Stream Processing (RSP) engines which have not been designed considering Edge computing use cases, *e.g.*, C-SPARQL [4], CQELS [27] or bigSR[46]. The only comparable system is Fed4Edge[38] but it does not support reasoning services.

Concretely, our contributions are: (i) an evolution of SuccinctEdge toward querying unbounded RDF graphs and hence providing anomaly and risk detection based on a temporal analysis of events received from sensors, (ii) the support of different stream processing models, *i.e.*, true streaming and microbatch, and window strategies, *i.e.*, tumbling and sliding, (iii) a rewrite of SuccinctEdge's query execution components: support for a continuous SPARQL extension, a new inference-enabled query optimization approach based on the distinction between static and dynamic portions of continuous queries, and (iv) a thorough evaluation of the correctness, robustness, latency and throughput dimensions in a real-world scenario.

This chapter is organized as follows. In the next section, we introduce a real-world motivating example. Section 5.3, provides an overview of our Streaming SuccinctEdge system. We present some related work in Section 5.4. We evaluate our system in Section 5.5 and conclude the chapter in Section 5.6.

[1] https://mosquitto.org/

[2] https://github.com/apache/incubator-retired-edgent

5.2 Motivating example

5.2.1 Data flow

This running example represents a frequent use case in sensor-based anomaly detection analysis and corresponds to a real world scenario encountered at our energy partner. It takes place in a building where hundreds of sensors are monitored. Data produced by these sensors are continuously ingested and analyzed in order to detect anomalies, *e.g.*, water leak, energy over-consumption.

Figure 5.1 presents the typical data flow of this IoT setting. It begins with the installation of a new sensor (step 1). The people responsible for this installation, denoted the IoT Persons, declare to platform Administrators the schema associated to the measures retrieved from this device (step 2). Note that this approach is also applied when an existing sensor is replaced. Therefore, we consider that it is not possible for a sensor to be changed without the administrators being aware of it. The Administrators then ask a team of domain experts

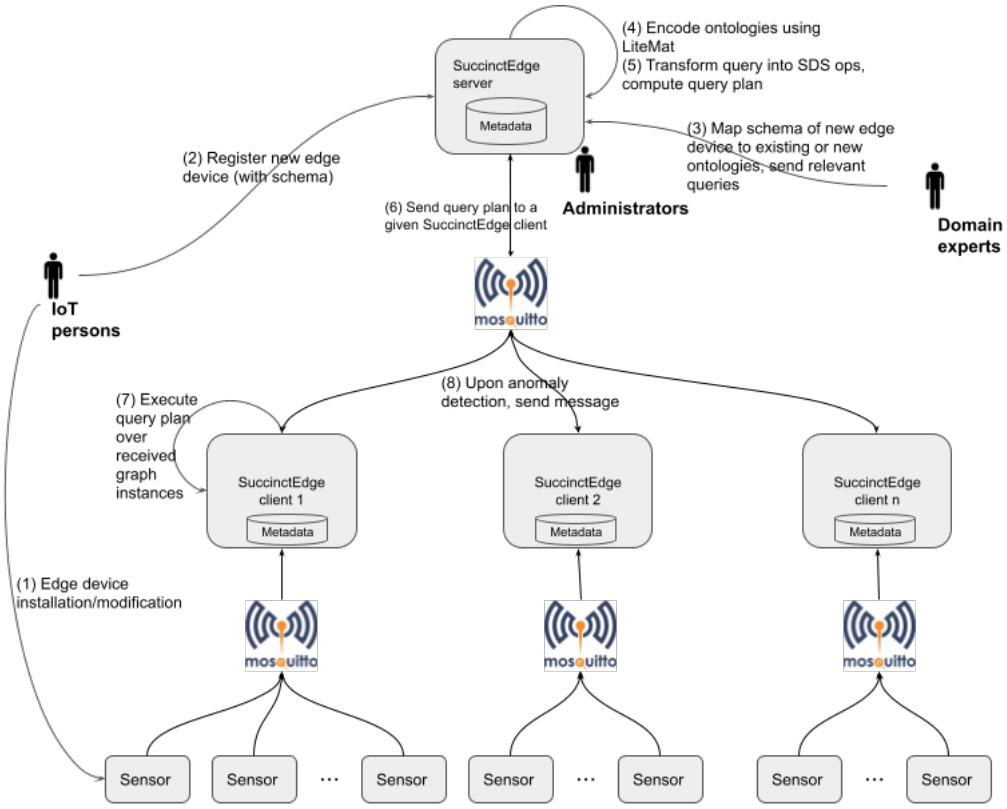

Figure 5.1: Data flow for setting a Streaming SuccinctEdge platform

to map this schema, *e.g.*, CSV, to a semantic representation, *e.g.*, OWL. A large set of ontologies are available to annotate the IoT and sensor domains, *e.g.*, Sensor, Observation, Sample, Actuator (SOSA[3]), Quantities, Units, Dimensions, and Types (QUDT[4]) or Smart Applicances

[3]http://www.w3.org/TR/ns/sosa
[4]http://qudt.org/schema/qudt

Reference (SAREF[5]). The semantic representation of these data is an important incentive to use the RDF data model in such a use case. In the context of our experimentation at ENGIE (a multinational company operating in fields such as energy transition, generation and distribution), the use of these ontologies considerably simplified the task of describing, manipulating and connecting sensors and actuators. Domain experts are also responsible for providing relevant queries, *i.e.*, those enabling anomaly and risk detection (step 3).

Administrators can then perform the encoding, using LiteMat, of ontologies required by this new graph (step 4) and transform the SPARQL queries into optimized queries expressed in terms of SDS operations (step 5). These physical query plans are expressed in a small set of operations supported by SuccinctEdge and are sent to the appropriate SuccinctEdge client (step 6). These queries are then executed continuously when receiving messages from the sensors (step 7). These queries can only be modified upon Administrators request, *e.g.*, when the connected sensor is changed. In case an anomaly is detected by a query, a message is sent from the SuccinctEdge client to the SuccinctEdge server with some context information such as device and query identifiers, abnormal data and event time of the anomaly (step 8). Moreover, both SuccinctEdge's client and server maintain some metadata, *e.g.*, query/sensor, client instance/sensor, client instance/mosquitto associations, etc.

Another version of our system extension relies on Apache Edgent framework. As shown in figure 5.2, the principal data flow is quite similar to the one based on Mosquitto. But here, the measures produced by sensors are ingested by an Apache Edgent instance which itself communicates with a SuccinctEdge client. The Edgent-SuccinctEdge connection can either be a one-to-one (device n) or one-to-many (devices 1 and 2).

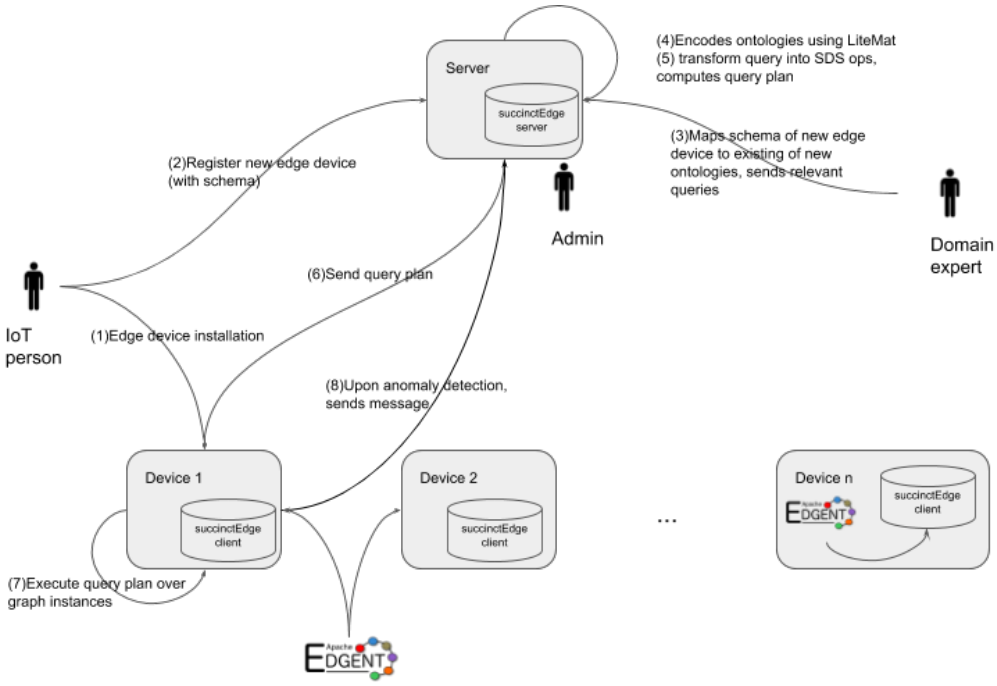

Figure 5.2: Data flow for setting a Streaming SuccinctEdge platform

[5]https://ontology.tno.nl/saref.ttl

5.2.2 Semantic consideration

Figure 5.3 presents an extract of a graph processed by a Streaming SuccinctEdge instance. It contains some measures related to the distribution of some commodities, *e.g.*, water or gas, in a building. Given such a graph, our system executes queries that can detect some anomaly patterns, *e.g.*, distribution network leaks.

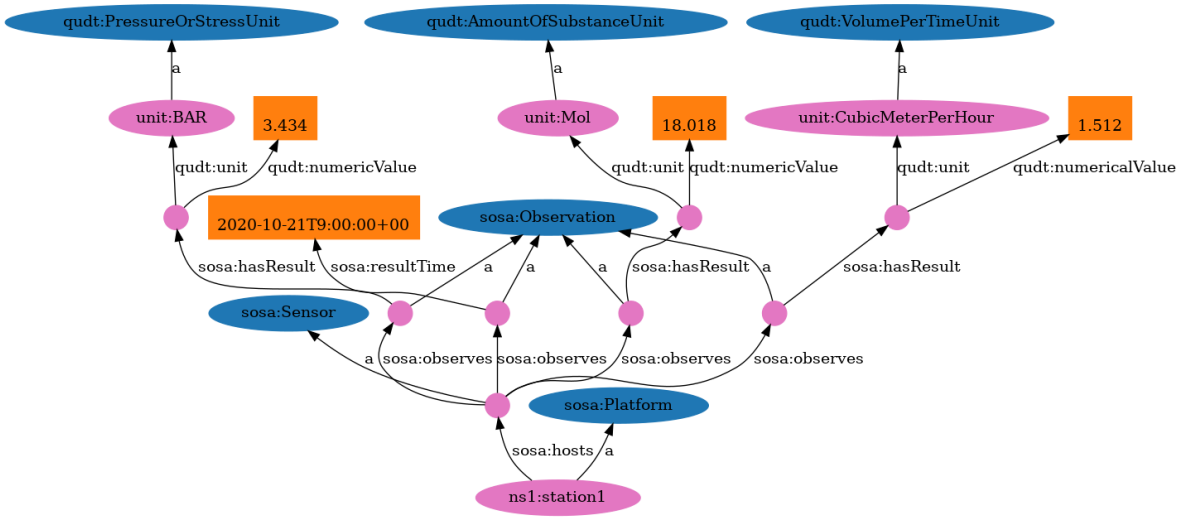

Figure 5.3: Graph extract of our use-case

In our experimentation at ENGIE and on the Waves project[6], we found out that several types and brands of sensors are frequently being used to observe similar measures, *e.g.*, pressure, flow. These sensors may also produce measures expressed in different units, *e.g.*, Bar, $Pascal$, psi, $Torr$ for pressure measures; ft^3/min, gal/min, m/min, m^3/h for volume per time unit. In this context, it is necessary to integrate all retrieved information into a single information system. The ability to associate KG concepts and properties to the measures produced by these sensors is a first step toward this semantic integration. Moreover, domain experts generally define concepts of observable properties, *e.g.*, AtmosphericTemperature, which can be organized into hierarchies and can hence be used for reasoning purposes. RDFS++ inferences are efficiently processed in SuccinctEdge, via query rewriting, thanks to the usage of LiteMat.

A second semantic integration step consists in making it easier to write SPARQL queries by automatically transforming queries to the characteristics of a sensor, *e.g.*, based on concept annotations and units being used. To support these requirements, we encourage domain experts to express queries in relatively high concept terms. Hence, they do not have to worry about the inferences which are handled automatically by the system. Expressing a query with abstract concepts, *i.e.*, high in the concept hierarchy, permits to write a single query that can tackle sensors performing similar measures but annotated with different concepts and possibly with different measure units. This is an important requisite for our use case where different sensor brands and types coexist in a given network. This approach's simplicity was highly expected by ENGIE for productivity reasons. In fact, it

[6]https://waves-rsp.org/index.html

enables its sensor staff to concentrate on their tasks and not on adapting a given query to a potentially large number of sensors in industrial settings.

Let consider two sensor platforms. The first station corresponds to the one described in Figure 5.3 where the pressure is typed as $qudt : PressureOrStressUnit$ and is expressed in the Bar unit. In the second one, a similar pressure measure is typed as $qudt : PressureUnit$ and is expressed in the HectoPA unit. Since, the QUDT ontology[7] states that: $qudt : PressureOrStressUnit \sqsubseteq qudt : PressureUnit$, a single SPARQL query (detailed in Section 5.3.2) can be written to address the specificity of each sensor at these stations.

5.3 Streaming SuccinctEdge presentation

In this section, we present the main design principles of SuccinctEdge. We also detail components dedicated to stream processing that have been introduced to our first version of SuccinctEdge.

5.3.1 Architecture

SuccinctEdge adopts a self-index approach. This means that a single copy of RDF triples is stored in the system. The predicate, subject, object (PSO) triple permutation order has been selected because the queries submitted to SuccinctEdge rarely have variables at the predicate position. In fact in such a setting, there is no need to run discovery queries, *i.e.*, queries that permit to understand what kind of information is contained in a graph, since the application's KG is stable and well-understood by domain experts.

Given our PSO indexing approach, we make a distinction between datatype properties, *i.e.*, where the object is a literal, and object properties, *i.e.*, where the object is not a literal. In most use cases we have encountered, the relationships between instances in the RDF graphs rarely change because they represent the connections between physical objects, *e.g.*, platforms, sensors. We can thus represent all triples containing an object property (except rdf:type for which a special storage is proposed) with a combination of WTs and BMs data structures. Intuitively, each predicate, subject, object set is stored as a WT (respectively WT_p, WT_s and WT_o) and two BMs respectively connect the WT_p to WT_s and WT_s to WT_o.

Meanwhile, the data generated by sensors, *i.e.*, numerical measures, which are objects of some datatype properties, change continually. A high update rate is not adapted to a WT storage for the two following reasons: (i) each object value would require a single identifier but this is not reasonable since these values are mostly numerical and thus possibly infinite, (ii) updating a WT can not be performed efficiently. Figure 5.4 shows the details of the data structures used for triples containing a datatype property. Properties and subjects are stored as in the object properties part, *i.e.*, with 2 WTs and a BM. A BM connects the subject WT

[7] https://qudt.org/

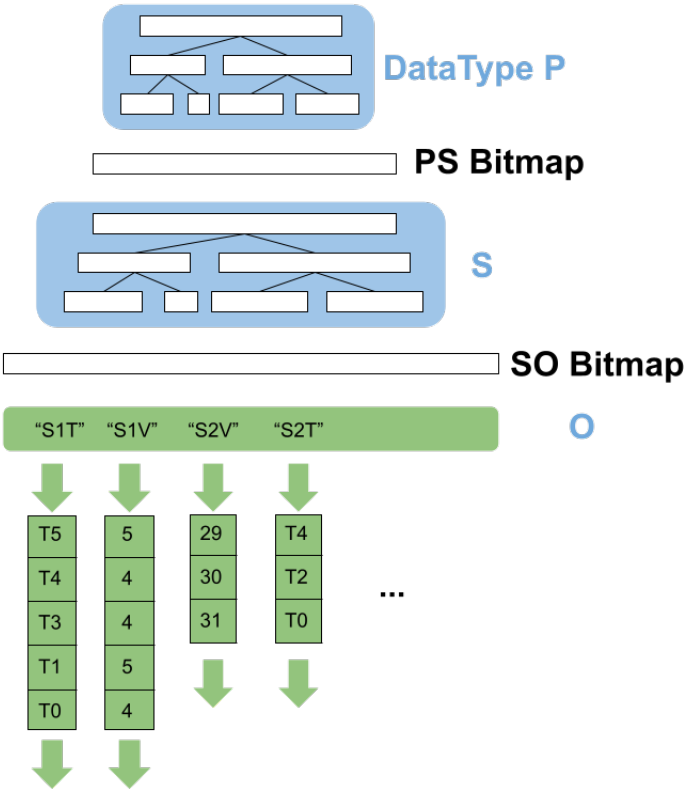

Figure 5.4: Data type property structure with two WTs and two BMs

to an object layer, denoted O). In this layer, each object represents a dynamic data which is timestamped and frequently appended. There, the system stores a pointer for each object which is pointed to a queue-like structure. When new data comes in, we push it to the front of the queue. These queue-like structures have some auxiliary functions to optimize the aggregation operations (*i.e.*, MIN, MAX, AVG, SUM, COUNT) present in a query. The corresponding function is activated on demand from the system, *e.g.*, it may depend on the streaming semantic (more on this in the next sub-section). When we execute a triple pattern (TP) with datatype property, we search the index interval of objects using the WTs and BMs of the first two layers, then for each object in this interval, we take the value in its corresponding data queue and compute the function indicated in the query.

Example 5.1: We consider the data transfer of 2 different sensors (S1 and S2) is given in Figure 5.4. We assume that each sensor measures a single value, *i.e.*, S1V and S2V. "S1T" represents the timestamp received from Sensor1 and "S1V" refers to its measure, respectively, "S2T" and "S2V" indicate the timestamp and measure of Sensor2. We can see that SuccinctEdge has distributed a queue-like structure for each data series.

Even though two sensors send their data at different frequencies, *e.g.*, Sensor 1 (resp. 2) emit a message every 200ms (resp. 300ms), SuccintEdge can still handle the situation thanks to a map structure implementation to distribute each data-type object to a set of its corresponding sensor. By using this data structure, we can easily keep all the data sequences from one sensor in the same length. Moreover, sliding and tumbling streaming windows

impose the maintenance of cursors on these structures.

5.3.2 Continuous SPARQL extension

Several projects have extended SPARQL to support the continuous querying of RDF streams. As a well-established approach, [4] has influenced our own SPARQL extension. Compared to the C-SPARQL syntax, we are currently only supporting logical windows, *i.e.*, time-based, and our RANGE description block appears in the SELECT clause. In addition to tumbling windows, we also support sliding windows using the STEP keyword. We can also associate an aggregation operation to each query variable.

Example 5.2: The following query (corresponding to our motivating example) detects anomalies related to an incorrect pressure value (either expressed in Bar or HectoPascal) for sensors at stations 1 and 2. We can see that in the RANGE clause a tumbling window of 5 seconds is required and ?v1, which is a numerical variable, is followed by a [MAX] which indicates that for each binding of ?v1 in the result set, we take the maximum in the data window. The FILTER clause detects anomalies, the BIND clause performs some data transformations.

```
SELECT ?x ?s ?ts ?v1[MAX]
[RANGE 5000 MS TUMBLING] WHERE {
?x a sosa:Platform; sosa:hosts ?s.
?s sosa:observes ?o; a sosa:Sensor.
?o sosa:hasResult ?y; a sosa:Observation;
sosa:resultTime ?ts.
?y a sosa:Result; qudt:numericValue ?v1;
qudt:unit ?u1. ?u1 a qudt:PressureUnit.
FILTER (?newV<3.00 || ?newV>4.50)
BIND(if(regex(str(?u1),
"http://qudt.org/vocab/unit/BAR"),?v1,
if(regex(str(?u1),
"http://qudt.org/vocab/unit/HectoPA"),
?v1/1000,0)) as ?newV)}
```

Query 5.1: Streaming anomaly detection SPARQL query example

5.3.3 Query processing

The query processor described in Section 4.4 has been extended with a decomposition of a query's BGP. The motivation for this new query processor is two-fold and based on an observation of real-world IoT settings. First, continuous queries analyzing streaming data are (i) generally highly selective, *i.e.*, return rather small answer sets (tens of tuples), and (ii)

retrieve static, *e.g.*, sensor and entity identifiers, and dynamic, *e.g.*, analysis of recent measures and their timestamps, information. Moreover, a sensor can produce a set of measures corresponding to different types of information, *e.g.*, pressure, flow, pH, etc.. But most of the time, a single value is produced per information type in a given sensor output.

Second, the graph associated to a sensor rarely changes except if the sensor is replaced. As stated in Section 5.2.1, the replacement of a sensor in our IoT ecosystem is necessarily notified to the team of administrators of the overall platform. In general, this forces to check or re-execute steps 3 to 6 of Figure 5.1.

Taking into account these two observations, our new query processor makes a distinction between a static subset of the BGP and a dynamic one. Intuitively, when a sensor produces its first measures, the complete BGP of the continuous query is executed and the bindings of the distinguished variables of the static part of the BGP are cached by SuccinctEdge. Note that this query execution may involve some form of (RDFS) reasoning which are handled by LiteMat's rewriting facility. Then, for successive measures produced by this same sensor, only the dynamic portion of the BGP needs to be executed and integrated in the query result set. The dynamic distinguished variables correspond to objects of datatype properties and thus do not influence a query's graph pattern matching since they correspond to graph leaves. The execution of this dynamic BGP subset may require the computation of aggregation functions and some data transformation, *e.g.*, transform a pressure from Bar to Pascal. The main design principle of our query processing component is to take advantage of this aspect and to compute a physical plan only once for a given query. This drastically improves query execution since in a continuous query processing setting, a query may be computed an undefined number of times. This approach is reminiscent to a parameterized query, aka prepared statement, where a query is pre-compiled and only needs some parameters to complete its execution. In our streaming context, the parameters correspond to the dynamic part of the BGP, *i.e.*, the objects associated to datatype properties (including timestamps or measures) or the result of applying an aggregation function over them.

In Query 5.1, the cached static part corresponds to the `?x` and `?s` variables, respectively the platform and sensor URIs, while the dynamic part corresponds to the `?ts` and ?v1[MAX] variables, respectively the event timestamp and maximum pressure value of a pressure measure whose maximum value exceeds a certain threshold.

We have seen in Figure 5.1 that the computation of the physical plan is performed at a SuccinctEdge server which has generally more resources, in terms of CPU and memory, than an Edge device where a SuccinctEdge client is running. SuccinctEdge's query optimizer mixes heuristics with a cost-based approach. The statistics of the latter are stored in the dictionaries of LiteMat which remain on the machine running the SuccinctEdge server. The remaining of the query evaluation is performed on a SuccinctEdge client.

Algorithm 8 summarizes the computation process of a continuous query.

In this algorithm, we must point out that the datatype-object's data structure in SuccinctEdge 's TripleStore is in charge of updating sensor's data and computing time-based aggregation automatically. Once we want to show out the result, the output data can be reached via the pointer stored in datatype clauses in R.

Algorithm 8: Process a continuous query over streaming data.

Input: Physical plan of a query Q
Output: Result set R

```
for each triple pattern TP in Q do
    if The predicate presents to be non-datatype then
        Compute the result set R_TP of TP;
        Compute R = R ⋈ R_TP;
    end
    else
        Compute the result set R_TP of TP where the objects are pointed to certain
          data structures in the 3rd layer of Datatype TripleStore;
        Compute R = R ⋈ R_TP;
    end
end
for each data structure pointed by a datatype object do
    Settle updating parameter (window, window mode, time-base aggregation
      mode etc. ) to the data structure pointed by the object;
end
return R;
```

5.3.4 Data stream exchange modes

Streaming SuccinctEdge supports two data stream exchange modes. In Figure 5.5, data events are represented as shapes. In the true streaming mode, each sensor is immediately sent to its SuccinctEdge client, via Mosquitto. In the microbatch mode, the sensor retains a certain amount of events, typically corresponding to the length of a temporal window. Once the boundary of this window is attained, the complete set of data is sent to the SuccinctEdge client, also via Mosquitto. This mode limits the number of data exchanged over the network. These two modes can run under the sliding and tumbling window semantics.

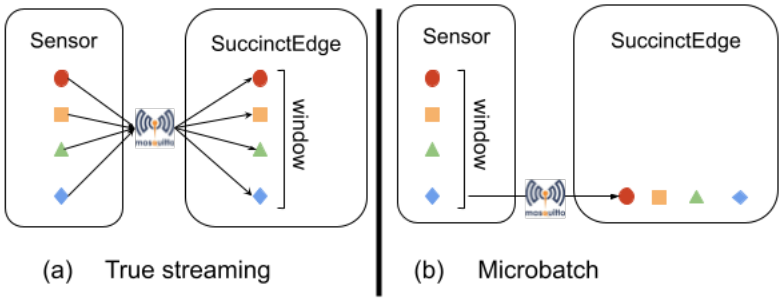

Figure 5.5: Supported exchange modes

Another version with Apache Edgent supports three modes of data exchange. The three modes are summarized in Figure 5.6. In this figure, data events are represented as shapes.

In the Event-at-a-time mode, each data stream received by Edgent is directly sent to SuccinctEdge for further processing, *e.g.*, computing aggregations within a SPARQL query. In this mode, the window management is fully handled by SuccinctEdge. In the Event-set-at-a-time mode (*aka* micro-batch), we are using Edgent capacity to retain a certain amount of events, typically corresponding to the length of a temporal window. Once the boundary of this window is attained, the complete set of data is sent to SuccinctEdge. This mode does not require SuccinctEdge to handle window management and limits the number of data exchanged over the network. Finally, the aggregate-at-a-time mode takes advantage of Edgent analytical capacity which computes aggregate functions, including User Defined Functions (UDF) on its data. Once a window boundary is reached, the result of this computation is sent to SuccinctEdge which can then integrate into its query processing.

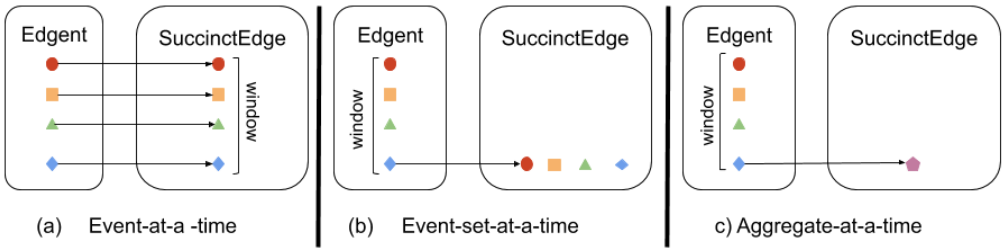

Figure 5.6: Supported exchange modes

5.4 Related works

In this section, we present systems in the messaging and continuous SPARQL querying categories.

5.4.1 Messaging systems

We consider two open-source systems which propose the mechanisms required for a messaging system and streaming broker. Note that we do not consider systems such as Apache Kafka [26] and Apache Pulsar[8] which are designed to run on the Cloud or large servers.

Apache Edgent

Apache Edgent[9] is an open source Java library designed to reduce the load on data centers by having analytic capabilities at the edge. It is capable of performing analytical operations on streams of data coming from multiple devices while leaving a small memory footprint, hence allowing it to run on edge devices. Edgent allows to transform data easily by providing windows, streams and standard operations on aggregation functions such as average,

[8]https://pulsar.apache.org/
[9]https://edgent.incubator.apache.org/

count, sum, min and max. More complex transformations are also possible via the definition of UDF.

Eclipse Mosquitto

Eclipse Mosquitto[28] is a lightweight message broker that implements MQTT (Message Queuing Telemetry Transport), a publish-subscribe message protocol that generally uses TCP/IP. It has been designed to run on devices with limited resources such as sensors, *i.e.*, handling over 1.000 clients on less than 3MB of RAM. Therefore, it is particularly well suited for the Edge computing environment.

In the context of our Streaming SuccinctEdge platform, Mosquitto supports the data exchange between SuccinctEdge's server and client as well as between SuccinctEdge client and sensors (see Figure 5.1).

5.4.2 SPARQL continuous query extensions

In this section, we present two of the main continuous query extension that have been designed for the SPARQL query language.

C-SPARQL

C-SPARQL[4] extends SPARQL towards streaming processing. It supports two types of time window mode over data streams: sliding window and tumbling window. Sliding window is moved smoothly in time series while tumbling window obeys a consume-and-drop strategy which slides the window at a time by a predefined range. Moreover, C-SPARQL supports declaration of stream origins by using **FROM STREAM** key words. **STEP** key word is used to indicate updating frequency on a sliding window and **RANGE** key word serves at declaring the window size. The aggregation of C-SPARQL presents to be an extension of SQL-style aggregation, which means it can aggregate data from a certain column of the result set. However, in our cases of sensor data, the aggregation will be applied on data coming from a certain sensor over time series, which is out of C-SPARQL's consideration.

CQELS

CQELS[27] is designed more for large static data sets, such as *LINKED DATA*, together with large streaming data set, *e.g.*, *LINKED STREAM DATA*. Different from C-SPARQL which is implemented over existing streaming data management system and triple stores, CQELS is built with its own components. It is considered that with such an approach, the system may perform and adapt better to input data's changes. Like C-SPARQL, CQELS also supports **RANGE** to indicate window size of a sliding window, while it lacks the support of tumbling window which may reduce computational pressure in real use cases. Just like C-SPARQL,

CQELS doesn't consider aggregations of data over time series from a certain sensor, which could be often required in anomaly detection cases.

5.5 Experimentation

In this section, we compare the new storage layout of SuccinctEdge against HDT and RDF stores that have been designed for Edge computing or could run on such environments. We evaluate our system correctness and robustness. Then, we study its latency and throughput characteristics. Evaluation are conducted on several scenarios, running different window strategies and exchange modes.

5.5.1 Experimental setting

Our experimentation are conducted on a Raspberry Pi 3B+. It is equipped with a Cortex-A53 (ARMv7l) 32-bit SoC 1.4GHz CPU and 1GB LPDDR2 SDRAM. A SD-card of 8GB is used if data needs to be persisted.

SuccinctEdge is implemented in C++ (version 14) and uses the SDS-lite library[10]. Installation details can be found on github[11]. Eclipse Mosquitto (version 2.0) runs using a JDK version 1.8. Data is transferred using the TCP protocol. Apache Edgent version 1.2.0-incubating has been installed and runs using a JDK version 1.8. Data is transferred between Edgent and SuccinctEdge using the TCP protocol. Considering our comparison with HDT, we used HDT-Java (version 2.1.2) for the size, construction time and queries with a single TP (Q1 to Q4) evaluations. We used HDT-Jena (version 2.1.2) for queries requiring joins, *i.e.*, Q5 to Q8 with respectively 2, 2, 2 and 3 TPs in the BGP. In the query processing evaluation, we are comparing the query processing component of Streaming SuccinctEdge against RDF4Led[51], an in-memory Apache Jena[12] (version 3.15) database implementation and RDF4J's Memory Store[13] (version 3.4.0)

The context of this evaluation is anchored in our running example (Section 5.2.1) where real-world pressure measures are analyzed. In fact, abnormal measures are those that do not belong to a 4 to 5 bars interval. We have defined five different scenarios to demonstrate the correctness of our system. They are presented in Figure 5.7 and intuitively state that no anomalies are occurring (Scenario 0), few anomalies happen due to pressure values under and above a certain threshold (Scenario 1), series of anomalies are followed by correct measures (Scenario 2), continuous anomalies after a certain time due to varying measures being above a the max threshold (Scenario 3) and continuous anomalies after some time due to a complete loss of pressure (Scenario 4).

[10] https://github.com/simongog/sdsl-lite
[11] https://github.com/xwq610728213/SuccinctEdgeWithStreaming
[12] https://jena.apache.org/
[13] https://rdf4j.org/

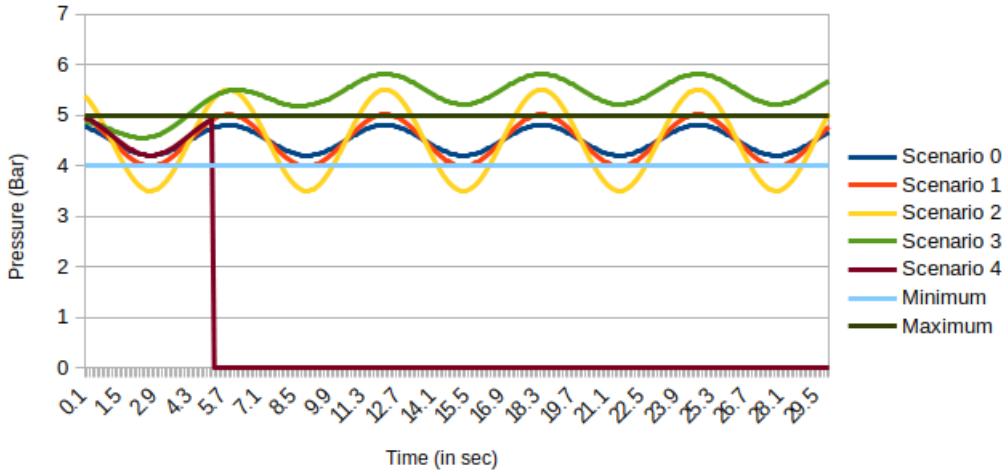

Figure 5.7: Experimentation scenarios on pressure measures

5.5.2 Comparison against HDT

In this section, we compare some aspects of HDT with the SuccinctEdge/LiteMat pair. This comparison is performed in a non-streaming setting since HDT does not natively support a streaming mode. Hence, the Mosquitto vs Edgent issue is not important, since they share the same query processing approach, in this first experimentation. Nevertheless, the evaluation has been conducted on the Mosquitto implementation.

Moreover, with this evaluation, we are mainly interested on the compactness, creation duration and query processing of these two systems. This is motivated by the fact that HDT is becoming an important serialization for RDF data.

Figure 5.8 emphasizes that HDT is more compact and takes less time for the construction of its RDF representation than SuccinctEdge. Considering the size, SuccinctEdge pays the price of its datatype representation which is currently not compressed and which represents around 50% of the LUBM data sets. The longer construction times of SuccinctEdge are explained by a complex storage layout based on layers of WTs and BMs and the queue-like structure for objects of data properties. Nevertheless, this storage design is responsible for the fast query execution that we will witeness in the next sections.

The in-memory and query processing efficiency of SuccinctEdge is demonstrated in Table 5.1. For queries with a single TP, *i.e.*, Q1 to Q4, SuccinctEdge is up to an order of magnitude more efficient than HDT for queries with a small answer set, *i.e.*, under 50 tuples for Q1 and Q2. For Q3 with around 500 tuples in the answer set, both systems are comparable. HDT is two times more efficient for large result sets (Q4). Considering queries with multiple TPs,*i.e.*, Q5 to Q8, SuccinctEdge can be up to two orders of magnitude more efficient that HDT-Jena. More information about these queries can be found in Appendix A..2. Over all queries, SuccintEdge particularly outperforms HDT in the presence of TP joins and for small result sets which are especially relevant in streaming event-driven applications.

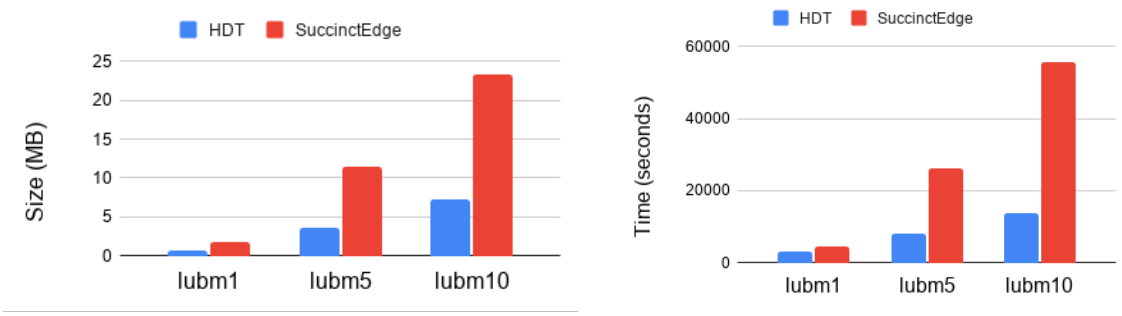

Figure 5.8: HDT and SuccinctEdge comparisons. Left : Size of data and dictionaries, Right: Construction time

Queries	Answer set	HDT	SuccinctEdge	Queries	Answer set	HDT	SuccinctEdge
Q1	3	5	0.45	Q5	3	381	0.545
Q2	41	11	1.609	Q6	41	383	6.4
Q3	512	13	14.4	Q7	540	367	36
Q4	7790	41	98	Q8	7790	426	133

Table 5.1: Query processing, answer sets in number of tuples and times in ms

5.5.3 Query processing

In 4.6, we have demonstrated that SuccinctEdge's query execution outperforms most of its direct competitors, *i.e.*, RDF4Led, Apache Jena and RDF4J. In this section, we are checking if Streaming SuccinctEdge retains this advantage when considering its novel storage layout and query processor.

First, the query optimization and generation of a physical plan is still efficient in Streaming SuccinctEdge since for queries ranging from 6 to 12 TPs, the time taken by these two tasks range between 3 and 9ms. Recall that this is performed off-line in Step 5 of Figure 5.1. So the overhead of parsing, optimizing and generating a physical plan for this SPARQL extension is rather low.

Since none of RDF4Led, Jena or RDF4J can process RDF streams, we are only evaluating them in a static setting anchored in our running example. Figure 5.9 presents the execution of 6 queries influenced from the query of Example 2 and differing by the number of TPs in the BGP: M1/R1, M2/R2 and M3/R3 have resp. 6, 9 and 12 TPs and R queries imply some reasoning. All queries are executed on a synthetic graph of 3.000 triples. We can see that the new query processor of SuccinctEdge retains its properties and outperforms its competitors.

5.5.4 SuccinctEdge-Mosquitto evaluations

The evaluation of SuccinctEdge-Mosquitto streaming architecture considers the correctness, robustness of the system as well as the system's latency and throughput. In the la-

Figure 5.9: Query performance of SuccinctEdge in static mode. 'M' queries do not require any reasoning. 'R' queries require reasoning on the concept hierarchy.

scenario	Window type	stream mode	# expected anomalies	# detected anomalies
0	tumbling sliding	ts, m	0	0
1,2,3	tumbling	ts, m	9	9
1,2,3	sliding	ts, m	26	26
4	tumbling	ts, m	5	5
4	sliding	ts, m	25	25

Table 5.2: Correctness with ts: true streaming, m: microbatch and all times in ms

tency and throughput evaluations, we distinguish the experiment into single sensor per client case and multiple sensors per client case.

Correctness and robustness

These scenarios have all been evaluated in different setting: window strategies (tumbling and sliding windows) and stream communication modes (true streaming and microbatch). Table 5.2 presents results obtained from an experimentation over 30 seconds with 5 seconds windows (similar results were obtained for 60 seconds and 10 seconds windows). The results emphasize that in all scenarios, window strategies and stream communication modes our system detects the correct number of anomalies.

The scenarios have also been tested over a setting implying several sensors communicating with a single SuccinctEdge client, *i.e.*, up to 40, and with different frequencies, *i.e.*,

with two sets of 20 sensors sending messages respectively every 200 and 300ms. The same 100% correctness has been observed. Finally, we evaluated a 40 sensors platform under sliding windows (from 5 sec to 5 min steps), true streaming (5 min to 1 hour) for over 3 days without any system failure.

Latency and throughput

When considering latency and throughput properties, we are not considering the execution of the query which retrieves and caches the static portion of the query since this is amortized by our query processing approach. Rather, we are only considering the impact of successive measure receptions, *i.e.*, computation of the aggregate functions, anomaly detection and integration of the dynamic and static parts of the query.

- **Single sensor per client:**
  This experimentation was performed on Scenarios 2 and 3 (the most demanding ones) with a single sensor connected to a Mosquitto and SuccinctEdge client. We have evaluated the latency over many situations: sensor sending at rates of 1 and 100 measures/second, true streaming and microbatch, window lengths in [5, 120) seconds, tumbling and sliding windows (with steps in [1, 30) seconds). In all cases, the average latency was around 150μs. Thus a client can support a throughput of over 6.500 measures par second.
- **Multiple sensors per client:**
  The evaluation conducted over multiple sensors confirms the query latency observed for a single sensor up to a certain number of sensors. In fact, Figure 5.10, highlights that for one measure per second, SuccinctEdge scales to at least 80 sensors. But between 50 and 60 sensors, at a sending rate of 100 measures per second, SuccinctEdge is not able to process its queries. Therefore, for a setting where each sensor sends a measure every 10ms, a new SuccinctEdge client is needed every 50 or so sensors. Considering the cost of a Raspberry Pi, this is not a limitation of our overall streaming solution.

5.5.5 SuccinctEdge-Edgent evaluations

The architecture SuccinctEdge-Edgent considers the same dimension than the Mosquitto-based implementation, *i.e.*, correctness, robustness and latency within its three data exchange modes.

Correctness, robustness and latency

Tables 5.3 to 5.7 enable the evaluation of the correctness of our different approaches and present the average latency observed in each experimentation. In all scenarios, each execution is running for 30 seconds. The window sizes are set to 5 seconds and when the sliding

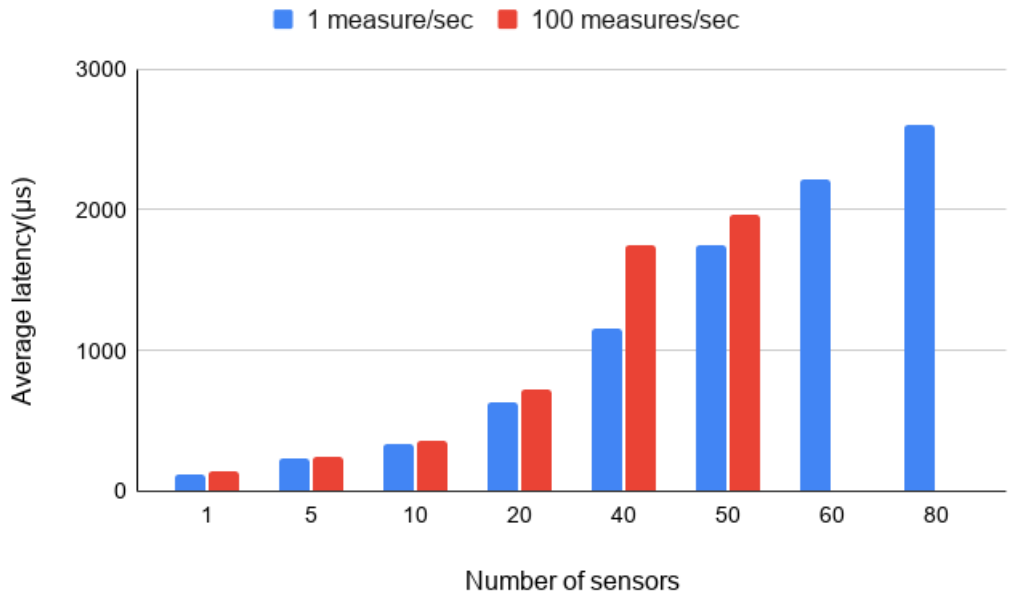

Figure 5.10: Average query processing latency (in μs) with 1 to 40 sending with a streaming, sliding (5sec steps), windows of 10 sec, Scenario 3 setting

window strategy is used, its step is defined at 1 second. The aggregation functions that are serving to detect anomalies are based on computing aggregation functions, *e.g.*, minimum, maximum and average, on values of pressure measures.

We can see that in all scenarios, our system is correct. That is, it detects the expected number of anomalies.

When considering latency and throughput properties, we are not considering the first execution of the query which retrieves and caches the static portion of the query. Rather, we are only considering the impact of successive measure receptions, *i.e.*, computation of the aggregate functions, anomaly detection and integration of the dynamic and static parts of the query. We consider latency as the time interval between receiving an event, *i.e.*, a measure, set of measures or result of an aggregation function, in SuccinctEdge and seeing the effect of processing this event in SuccinctEdge. Intuitively, it corresponds to how long it takes for an event to be processed. Moreover, we consider that throughput indicates how many events the system can process per unit of time. So it is a measure of the system's processing capacity.

We can see in Tables 5.3 to 5.7 that the latency is always under 1ms. So we can consider a throughput of 1.000 events per second.

As shown in Tables 5.3 to 5.7, in our running setting, SuccinctEdge has a latency of 1ms. Aggregating data is a trivial task for both SuccinctEdge and Edgent but the inter process communications (IPC) appear to be limiting the system.

Table 5.3: Evaluation of Scenario 0 - 5 and 10 seconds windows

Mode	Window strategy	Aggregation functions	Expected Anomalies	Detected Anomalies	Average latency (ms)
Event	Tumbling	min/max/avg	0	0	<1
Event	Sliding	min/max/avg	0	0	<1
EventSet	Tumbling	min/max/avg	0	0	<1
EventSet	Sliding	min/max/avg	0	0	<1
Agg	Tumbling	min/max/avg	0	0	<1
Agg	Sliding	min/max/avg	0	0	<1

Table 5.4: Evaluation of Scenario 1 - 5 seconds window

Mode	Window strategy	Aggregation function	Expected Anomalies	Detected Anomalies	Average latency (ms)
Event	Tumbling	min/max	9	9	<1
Event	Sliding	min/max	26	26	<1
EventSet	Tumbling	min/max	9	9	<1
EventSet	Sliding	min/max	26	26	<1
Agg	Tumbling	min/max	9	9	<1
Agg	Sliding	min/max	26	26	<1

Maximum processing capacity

We have just seen that Streaming SuccinctEdge presents interesting, at least for a standard industrial IoT context, latency and throughput characteristics considering our running setting. In this experimentation, we are searching for the maximum processing capacity on a Raspberry Pi 3B+, *i.e.*, 1 GB of RAM. This is being performed by running Scenario 3 with 1 measure every 100μseconds for certain amount of time. Table 5.8 confirms a 1ms latency until SuccinctEdge reaches a limit of 6.000.000 events in a streaming window. This limit is characterized by an "out of memory" message from the process running SuccinctEdge. We obtained a similar output when running the same experimentation with the Aggregate-at-a-time mode, *i.e.*, the aggregate function is computed on Edgent.

5.6 Conclusion

In this chapter, we have presented an original attempt to process RDF stream at the edge of a computing infrastructure. The main characteristics of our system, which is denoted as Streaming SuccinctEdge, are compactness and the ability to infer implicit consequences on-the-fly. Two architectures have been designed and implemented. They are based on two different open source messaging systems. They both presented interesting properties, expected in a sensors' streaming data context where data are produced at a high velocity. Nevertheless, in the furture we will only retain the Eclispe Mosquitto approach. This is

Table 5.5: Evaluation of Scenario 2 - 5 seconds window

Mode	Window strategy	Aggregation function	Expected Anomalies	Detected Anomalies	Average latency (ms)
Event	Tumbling	min/max	9	9	<1
Event	Sliding	min/max	26	26	<1
EventSet	Tumbling	min/max	9	9	<1
EventSet	Sliding	min/max	26	26	<1
Agg	Tumbling	min/max	9	9	<1
Agg	Sliding	min/max	26	26	<1

Table 5.6: Evaluation of Scenario 3 - 5 seconds window

Mode	Window strategy	Aggregation function	Expected Anomalies	Detected Anomalies	Average latency (ms)
Event	Tumbling	min/max	6	6	<1
Event	Sliding	min/max	26	26	<1
EventSet	Tumbling	min/max	6	6	<1
EventSet	Sliding	min/max	26	26	<1
Agg	Tumbling	min/max	6	6	<1
Agg	Sliding	min/max	26	26	<1

mainly due to fact that Apache Edgent's implementation and maintenance is abandoned. In fact, Edgent never reach Apache's top level status. When we first started on Edgent implementation, the Apache project did not seem to be in jeopardy. This is a disappointment because Edgent presented several interesting features such the ability to define analytic-oriented UDFs.

Nevertheless, Mosquitto also has its set of great features. With the Mosquitto SuccinctEdge's implementation, we conducted a thorough experimentation over the most frequently used streaming models (true streaming and microbatch) and window strategies (sliding and tumbling) emphasized the correctness, robusteness and scalability of the system. In the near future, Streaming SuccinctEdge will execute in a large industrial setting and we are eager to study its behavior. As future work, communication and cooperation across SuccinctEdge clients will be integrated.

Table 5.7: Evaluation of Scenario 4 - 5 seconds window

Mode	Window strategy	Aggregation function	Expected Anomalies	Detected Anomalies	Average latency (ms)
Event	Tumbling	min/max	5	5	<1
Event	Sliding	min/max	25	25	<1
EventSet	Tumbling	min/max	5	5	<1
EventSet	Sliding	min/max	25	25	<1
Agg	Tumbling	min/max	5	5	<1
Agg	Sliding	min/max	25	25	<1

Table 5.8: Latency for Event-at-a-time mode, average function, 1 measure per 100 μseconds

	duration (seconds)					
	15sec	30	60	150	300	600
Events per window (in thousands)	150	300	600	1.500	3.000	6.000
Window strategy						
Tumbling	<1 ms	<1ms	<1ms	<1ms	<1ms	_
Sliding	<1ms	<1ms	<1ms	<1ms	<1ms	_

Chapter 6

Conclusion

Study the past, if you would divine the future.

– Confucius

In this section, we conclude this PhD thesis and propose some future works. The contributions are split into 3 major parts, LiteMat extensions, SuccinctEdge design and streaming extension of SuccinctEdge. Future works are concentrated on pushing SuccinctEdge towards server-edge cooperation and adapting SuccinctEdge to more concrete use cases.

6.1 Summary of contributions

The contributions of this thesis can be summarized as follows.

6.1.1 LiteMat extensions

The solution of the multi-inheritance problem of LiteMat

Lite was originally not able to handle multi-inheritance of concept hierarchies. To address this problem, our solution is to choose a representative among the direct super concepts of a certain concept C and encode C based on the encoding of this representative. The relations between C and other non-representatives are stored in a key/value data structure. Together with the data structure, we design an adapted query processing approach. We compared our solution with full materialization in the aspects of database construction and query answering performances. The results prove that our solution performs better than full materialization in most cases.

Extension of LiteMat towards transitive property

The efficient handling of transitive property is sometimes required in real-world queries. This is the main motivation for our extension of LiteMat towards this kind of ontology property. We propose two different encoding schemes for elements in chain and tree transitive structure. Both satisfy the requirement of compactness, determinism and scalability. We also propose the scenario of generating the encoding of each element with the adapted query processing to do fast reasoning. We prove that our solution performs better than the full materialization strategy both in memory occupation and query speed by evaluating two strategy with data sets ranging from 1.7 million triples to 11.8 million triples. As for future research directions, we point out that the strategy could be extended towards the case that an individual may be contained in multiple transitive structures.

6.1.2 SuccinctEdge - an RDF store for edge devices

An RDF store dedicated to edge devices

With the big bang of the deployment of small devices such as sensors, the management of these sensors' data becomes a problem. Hence a dedicated RDF store is needed. Currently, state of the art RDF stores are mostly designed for large servers. So they do not meet the characteristics of an Edge environment. In fact, very few RDF stores have been designed for Edge Computing. This motivated us to design and implement our own system. SuccinctEdge makes full use of LiteMat to accelerate reasoning services and to store RDF data. Considering this section we highlight four components: a query engine which supports important optimizations such as join order deduction, a dictionary which stores the mapping

between RDF elements and identifiers, a RDFtype store which store $rdf : type$ property, and a triple store which stores all the other triples. SuccinctEdge can be considered as the first in-memory RDF store for edge device with support of fast RDFS reasoning.

A triple store based on SDS

An edge device often has limited resources such as memory space and computational power. Thus, an RDF store for such devices must have a small memory footprint without a heavy decompression need to retrieve information. To satisfy these requirements, we make intensive use of SDS to compress RDF data into one index, this allows us to get a compact RDF store. Note that LiteMat is highly relevant in this context due to the binary aspects of SDS. We designed some adapted algorithms based on the three basic SDS operations to retrieve information from our data structure without decompression, these algorithms ensure that SuccinctEdge can still maintain a high querying speed. During the evaluation, we demonstrated that SuccinctEdge has a great data compression ability and a good query processing speed compared with Jena TDB, RDF4Led and RDF4J.

A query optimizer making compromise between calculation complexity and query efficiency

One of the most costly operation in a database system during query processing happens to be join ordering. That is the reason why different join orders of a query can cause great difference in query speed. However, calculating the best join order often takes much calculation and requires much extra meta-data to be stored in the database system. Aiming to reduce calculation complexity while still getting an efficient enough join order, we designed a heuristic-based join optimizer. This optimizer generates left-deep join plans with a light weight algorithm. The join plan will be then utilised in the result-searching phase of query processing. Thanks to the query optimizer, SuccinctEdge presents satisfying query performances in most cases.

6.1.3 Extensions towards streaming processing

A streaming extension of SuccinctEdge aiming at processing streaming numeric data

Data collected from sensors frequently come continuously. Thus some special data structure is required to store and process sensors' data. We split the triple store of SuccinctEdge into two part according to the type of triple's predicate (data-type and non-data-type). For the data-type part, WT is no longer suitable for objects layer as updating WT frequently can not be performed efficiently. In this case, we link each data-type object to a queue-like structure to store and aggregate dynamic data. This helps SuccinctEdge to support queries on dynamic graph and to efficiently process time-based data-aggregation operations. Thanks to our newly designed data structure, the system can handle data coming from different

sensors at different frequencies. As for query processing, our streaming approach split a query into dynamic and static parts, the temporary result of the static part is maintained during the continuous query execution, while the dynamic part is calculated continuously. The final result of a continuous query is generated by merging the newly produced dynamic result to the maintained static result. This approach has a great effect on accelerating query processing for the reason that it can reduce repeating query processing of the static part of a query.

A SPARQL extension that permits to query dynamic RDF graph

To query from the streaming data stored in non-data-type triple store, an extension of SPARQL is required as SPARQL can not handle continuous RDF querying. We designed our SPARQL extension inspired from C-SPARQL syntax which is one of the well-designed approaches for continuous querying. This extension supports logical window aggregation of streaming data where the window size can be indicated by RANGE keyword. It also supports sliding and tumbling window modes which could satisfy different cases. These extensions can deal with most of our use cases in querying the dynamic RDF graph on an edge devices.

Two different architectures of Streaming SuccinctEdge

We designed two different architectures of Streaming SuccinctEdge as our prototype systems to process streaming sensor data. One is based on Apache Edgent, the other is based on Eclipse Mosquitto. The two prototypes are evaluated under different data-exchange modes which are designed according to frameworks' characteristics. We evaluated the two architectures in the aspects of correctness, robustness, throughput and latency with different streaming modes, different window sizes and different sensors' numbers and prove that both approaches are of low latency, high throughput and good scalability.

6.2 Future Work

Now that we've defined an architecture for an RDF store that runs on Edge Computing, it's time to integrate some collaborations between these nodes. Hence, it does not come at a surprise that future work on SuccinctEdge will mainly concentrate on distribution aspects. In the following, we present the research that will soon be conducted on SuccinctEdge.

6.2.1 Distributed query processing

Intraquery parallelism

As presented in Figure 5.1, a SuccinctEdge server receives SPARQL queries from a domain experts. These queries are translated and sent to SuccinctEdge clients. Currently, our sys-

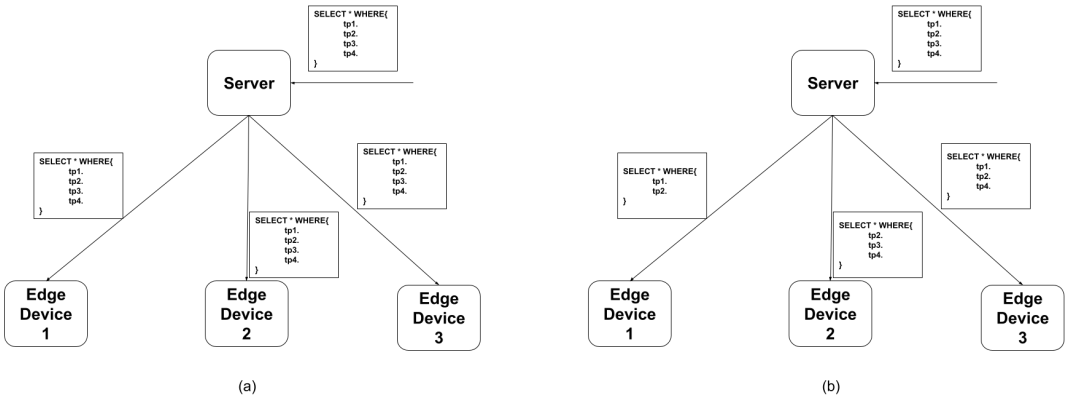

Figure 6.1: Graph pattern distribution strategies

tem is dealing with cases corresponding to Figure 6.1(a), *i.e.*, a query is broadcasted to clients that can execute a query in its entirety. In some situations, a SPARQL query could only be answered using the data retrieved from different SuccinctEdge clients. This corresponds to Figure 6.1(b). In that case, the query processor at a SuccinctEdge server will break up a single query into a number of subqueries. Each subquery will be executed at a different SuccinctEdge client. This approach corresponds to intraquery parallelism encountered in distributed database systems[40]. This extension is also related to the research conducted in triple pattern fragments[52] and its decomposition of a BGP into a set of distinct TPs.

We consider that the current state of metadata stored at a SuccinctEdge server can enable intraquery parallelism in our Edge computing device.

Combining query answer sets

At the moment, a SuccinctEdge client can query different sensors with a single or different queries. But it is not currently able to query a set of sensors with the same query. This scenario is nonetheless quite relevant in use cases where one needs to confirm an anomaly detection from a set of sensors. For instance, consider a building where sensors are measuring the temperature in each office. The building is, for example, equipped with a SuccinctEdge client at each floor of the building. Probably, detecting an anomaly at the floor level requires that a certain amount of temperature anomalies are discovered in each (or a majority) of the offices at that level. In order to support such a feature, we require to combine the answer of queries executed over the data emitted by different sensors into a single query.

Another future work consists in combining the result sets computed at different SuccintcEdge clients at a SuccinctEdge server or client. These two solutions are respectively depicted in Figure 6.2(a) and (b).

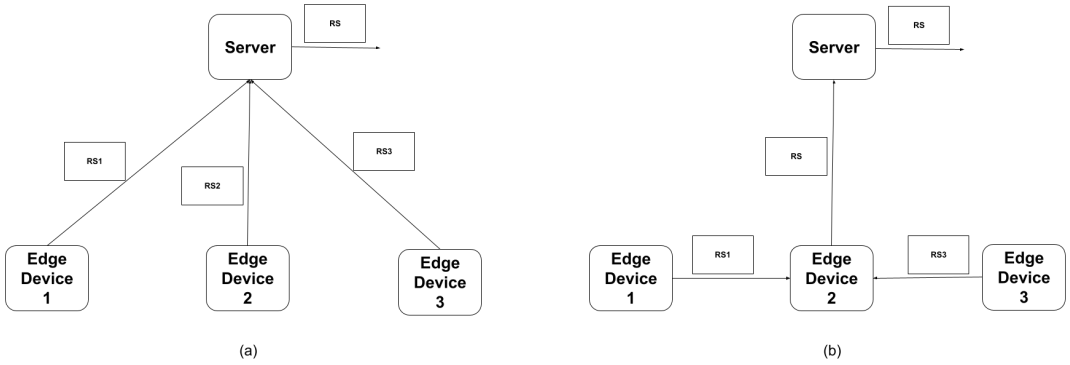

Figure 6.2: Result set collection strategies

6.2.2 Producing a summary of sensor data

The efficient management of edge device's memory is an important aspect of an RDF store running at the Edge. With the continuous reception of sensor data, we know that at some point the memory at the SuccinctEdge client will be saturated. So currently our approach is to flush the main memory to leave some space for the next measures. This may not be acceptable because we may miss some interesting information by doing so. Such information could be submitted to an analytical database system (running on on the SuccinctEdge server machine, which is powerful and equipped with large disks, or the Cloud) to discover, for instance, valuable anomaly patterns. Nevertheless, one of our priorities is to preserve to network bandwidth by being as frugal as possible when sending messages over the network. Hence, one of the solution is to create summaries of data collected at SuccinctEdge clients. An efficient approach is needed to generate, compress and later easily process these summaries.

Bibliography

[1] Daniel J. Abadi et al. "Scalable Semantic Web Data Management Using Vertical Partitioning". In: *Proceedings of the 33rd International Conference on Very Large Data Bases 2007*. 2007, pp. 411–422. URL: http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf.

[2] Yuan Ai, Mugen Peng, and Kecheng Zhang. "Edge computing technologies for Internet of Things: a primer". In: *Digital Communications and Networks* 4.2 (2018), pp. 77 –86. ISSN: 2352-8648. DOI: https://doi.org/10.1016/j.dcan.2017.07.001. URL: http://www.sciencedirect.com/science/article/pii/S2352864817301335.

[3] Franz Baader et al., eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. ISBN: 0-521-78176-0.

[4] Davide Francesco Barbieri et al. "C-SPARQL: SPARQL for continuous querying". In: *Proceedings of the 18th international conference on World wide web*. 2009, pp. 1061–1062.

[5] Tim Berners-Lee, Mark Fischetti, and Michael L. Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. HarperInformation, 2000. ISBN: 006251587X.

[6] Tim Berners-Lee, James A. Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific American* 284.5 (May 2001), pp. 34–43. ISSN: 0036-8733. URL: http://www.scientificamerican.com/article.cfm?id=the-semantic-web.

[7] Andre Bolles, Marco Grawunder, and Jonas Jacobi. "Streaming SPARQL - Extending SPARQL to Process Data Streams". In: *The Semantic Web: Research and Applications*. Ed. by Sean Bechhofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 448–462. ISBN: 978-3-540-68234-9.

[8] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. "Sesame: A generic architecture for storing and querying rdf and rdf schema". In: *International semantic web conference*. Springer. 2002, pp. 54–68.

[9] Victor Charpenay, Sebastian Käbisch, and Harald Kosch. "$$\mu$$$RDFStore : TowardsExtendingtheSemanticWebtoEmbeddedDevices". In: *The Semantic Web: ESWC 2017 Satellite Events*. Ed. by Eva Blomqvist et al. Cham: Springer International Publishing, 2017, pp. 76–80. ISBN: 978-3-319-70407-4.

[10] Olivier Curé and Guillaume Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing*. Morgan Kaufmann, 2015. ISBN: 978-0-12-799957-9. DOI: 10.1016/C2013-0-14009-3. URL: https://doi.org/10.1016/C2013-0-14009-3.

[11] Olivier Curé et al. “LiteMat: A scalable, cost-efficient inference encoding scheme for large RDF graphs”. In: *2015 IEEE International Conference on Big Data, Santa Clara, CA, USA*. 2015, pp. 1823–1830. DOI: 10.1109/BigData.2015.7363955. URL: https://doi.org/10.1109/BigData.2015.7363955.

[12] Olivier Curé et al. “LiteMat, an Encoding Scheme with RDFS++ and Multiple Inheritance Support”. In: *The Semantic Web: ESWC 2019 Satellite Events - Revised Selected Papers*. 2019, pp. 269–284. DOI: 10.1007/978-3-030-32327-1_47. URL: https://doi.org/10.1007/978-3-030-32327-1_47.

[13] Olivier Curé et al. “WaterFowl: A Compact, Self-indexed and Inference-Enabled Immutable RDF Store”. In: *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*. 2014, pp. 302–316. DOI: 10.1007/978-3-319-07443-6_21. URL: https://doi.org/10.1007/978-3-319-07443-6_21.

[14] Giuseppe De Giacomo and Maurizio Lenzerini. “TBox and ABox reasoning in expressive description logics.” In: *KR* 96.316-327 (1996), p. 10.

[15] S. Decker et al. “The Semantic Web: the roles of XML and RDF”. In: *IEEE Internet Computing* 4.5 (2000), pp. 63–73. DOI: 10.1109/4236.877487.

[16] Dimitrios Georgakopoulos and Prem Prakash Jayaraman. “Internet of things: from internet scale sensing to smart services”. In: *Computing* 98.10 (2016), pp. 1041–1058.

[17] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems.” In: *J. Web Sem.* 3.2-3 (July 4, 2006), pp. 158–182. URL: http://dblp.uni-trier.de/db/journals/ws/ws3.html#GuoPH05.

[18] Eric Hamilton. “What is Edge Computing: The Network Edge Explained”. In: *cloudwards.net* (2018). URL: https://www.cloudwards.net/what-is-edge-computing.

[19] Stephen Harris and Nicholas Gibbins. “3store: Efficient Bulk RDF Storage”. In: *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*. Ed. by Raphael Volz, Stefan Decker, and Isabel F. Cruz. Vol. 89. CEUR Workshop Proceedings. CEUR-WS.org, 2003. URL: http://ceur-ws.org/Vol-89/harris-et-al.pdf.

[20] Henning Hasemann, Alexander Kröller, and Max Pagel. “The wiselib tuplestore: a modular RDF database for the internet of things”. In: *arXiv preprint arXiv:1402.7228* (2014).

[21] Pascal Hitzler, Markus Krtzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. 1st. Chapman amp; Hall/CRC, 2009. ISBN: 142009050X.

[22] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edition)*. URL: https://www.w3.org/TR/2012/REC-owl2-primer-20121211. (accessed: 01.09.2016).

[23] Arne Holst. *IoT connected devices worldwide 2019-2030*. Jan. 2021. URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/#:~:text=ThenumberofInternetof,Chinawith3.17billiondevices..

[24] Herman J. ter Horst. "Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary". In: *J. Web Semant.* 3.2-3 (2005), pp. 79–115. DOI: 10.1016/j.websem.2005.06.001. URL: https://doi.org/10.1016/j.websem.2005.06.001.

[25] Raimondas Kiveris et al. "Connected Components in MapReduce and Beyond". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 18:1–18:13. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670997. URL: http://doi.acm.org/10.1145/2670979.2670997.

[26] Martin Kleppmann and Jay Kreps. "Kafka, Samza and the Unix Philosophy of Distributed Data". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 4–14. URL: http://sites.computer.org/debull/A15dec/p4.pdf.

[27] Danh Le-Phuoc et al. "A native and adaptive approach for unified processing of linked streams and linked data". In: *International Semantic Web Conference*. Springer. 2011, pp. 370–388.

[28] Roger A Light. "Mosquitto: server and client implementation of the MQTT protocol". In: *The Journal of Open Source Software* 2.13 (2017), p. 265. DOI: 10.21105/joss.00265. URL: https://app.dimensions.ai/details/publication/pub.1085668591andhttps://doi.org/10.21105/joss.00265.

[29] Miguel A. Martínez-Prieto, Mario Arias Gallego, and Javier D. Fernández. "Exchange and Consumption of Huge RDF Data". In: *The Semantic Web: Research and Applications - 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings*. Ed. by Elena Simperl et al. Vol. 7295. Lecture Notes in Computer Science. Springer, 2012, pp. 437–452. DOI: 10.1007/978-3-642-30284-8_36. URL: https://doi.org/10.1007/978-3-642-30284-8_36.

[30] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. "Simple and Efficient Minimal RDFS". In: *Web Semant.* 7.3 (Sept. 2009), pp. 220–234. ISSN: 1570-8268. DOI: 10.1016/j.websem.2009.07.003. URL: http://dx.doi.org/10.1016/j.websem.2009.07.003.

[31] Sergio Muñoz, Jorge Pérez, and Claudio Gutiérrez. "Simple and Efficient Minimal RDFS". In: *J. Web Semant.* 7.3 (2009), pp. 220–234. DOI: 10.1016/j.websem.2009.07.003. URL: https://doi.org/10.1016/j.websem.2009.07.003.

[32] J. Ian Munro. "Tables". In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by V. Chandru and V. Vinay. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 37–42. ISBN: 978-3-540-49631-1.

[33] Gonzalo Navarro. "Wavelet trees for all". In: *J. Discrete Algorithms* 25 (2014), pp. 2–20. DOI: 10.1016/j.jda.2013.07.004. URL: https://doi.org/10.1016/j.jda.2013.07.004.

[34] Yavor Nenov et al. "RDFox: A Highly-Scalable RDF Store". In: Oct. 2015, pp. 3–20. ISBN: 978-3-319-25009-0. DOI: 10.1007/978-3-319-25010-6_1.

[35] Thomas Neumann and Gerhard Weikum. "RDF-3X: A RISC-Style Engine for RDF". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), 647–659. ISSN: 2150-8097. DOI: 10.14778/1453856.1453927. URL: https://doi.org/10.14778/1453856.1453927.

[36] Thomas Neumann and Gerhard Weikum. "Scalable join processing on very large RDF graphs". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009.* Ed. by Ugur Çetintemel et al. ACM, 2009, pp. 627–640. DOI: 10.1145/1559845.1559911. URL: https://doi.org/10.1145/1559845.1559911.

[37] Thomas Neumann and Gerhard Weikum. "The RDF3X engine for scalable management of RDF data". In: *The Vldb Journal - VLDB* 19 (Feb. 2010), pp. 91–113. DOI: 10.1007/s00778-009-0165-y.

[38] Manh Nguyen-Duc et al. "Autonomous RDF stream processing for IoT edge devices". In: *Joint International Semantic Technology Conference*. Springer. 2019, pp. 304–319.

[39] Ralph Swick Ora Lassila. *Resource Description Framework (RDF) Model and Syntax.* Oct. 1997. URL: https://www.w3.org/TR/WD-rdf-syntax-971002/.

[40] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition.* Springer, 2020. ISBN: 978-3-030-26252-5. DOI: 10.1007/978-3-030-26253-2. URL: https://doi.org/10.1007/978-3-030-26253-2.

[41] Charith Perera, Chi Harold Liu, and Srimal Jayawardena. "The emerging internet of things marketplace from an industrial perspective: A survey". In: *IEEE Transactions on Emerging Topics in Computing* 3.4 (2015), pp. 585–598.

[42] Anthony Potter et al. "Distributed RDF Query Answering with Dynamic Data Exchange". In: *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I.* Ed. by Paul Groth et al. Vol. 9981. Lecture Notes in Computer Science. 2016, pp. 480–497. DOI: 10.1007/978-3-319-46523-4_29. URL: https://doi.org/10.1007/978-3-319-46523-4_29.

[43] Eric Prud'hommeaux and And Seaborne. 2008. URL: https://www.w3.org/TR/rdf-sparql-query/#bgpExtend.

[44] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. "Succinct Indexable Dictionaries with Applications to Encoding K-Ary Trees, Prefix Sums and Multisets". In: *ACM Trans. Algorithms* 3.4 (Nov. 2007), 43–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290680. URL: https://doi.org/10.1145/1290672.1290680.

[45] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. "Succinct Indexable Dictionaries with Applications to Encoding k-Ary Trees, Prefix Sums and Multisets". In: *ACM Trans. Algorithms* 3.4 (Nov. 2007), 43–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290680. URL: https://doi.org/10.1145/1290672.1290680.

[46] Xiangnan Ren et al. "BigSR: real-time expressive RDF stream reasoning on modern Big Data platforms". In: *IEEE International Conference on Big Data*. IEEE, 2018, pp. 811–820. DOI: 10.1109/BigData.2018.8621947.

[47] Xiangnan Ren et al. "Strider^{R}: Massive and distributed RDF graph stream reasoning". In: *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. 2017, pp. 3358–3367. DOI: 10.1109/BigData.2017.8258321.

[48] Mariano Rodríguez-Muro and Diego Calvanese. "High Performance Query Answering over DL-Lite Ontologies". In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. KR'12. Rome, Italy: AAAI Press, pp. 308–318. ISBN: 978-1-57735-560-1. URL: http://dl.acm.org/citation.cfm?id=3031843.3031879.

[49] Mariano Rodriguez-Muro and Diego Calvanese. "Quest, an OWL 2 QL Reasoner for Ontology-based Data Access". In: *Proceedings of OWL: Experiences and Directions Workshop 2012, Heraklion, Crete, Greece, May 27-28, 2012*. Ed. by Pavel Klinov and Matthew Horridge. Vol. 849. CEUR Workshop Proceedings. CEUR-WS.org, 2012. URL: http://ceur-ws.org/Vol-849/paper_20.pdf.

[50] Petros Tsialiamanis et al. "Heuristics-based query optimisation for SPARQL". In: *15th International Conference on Extending Database Technology, EDBT , Proceedings*. 2012, pp. 324–335. DOI: 10.1145/2247596.2247635. URL: https://doi.org/10.1145/2247596.2247635.

[51] Anh Lê Tuán et al. "RDF4Led: an RDF engine for lightweight edge devices". In: *Proceedings of the 8th International Conference on the Internet of Things, IOT 2018*. 2018, 2:1–2:8. DOI: 10.1145/3277593.3277600. URL: https://doi.org/10.1145/3277593.3277600.

[52] Ruben Verborgh et al. "Triple Pattern Fragments: A low-cost knowledge graph interface for the Web". In: *Journal of Web Semantics* 37-38 (2016), pp. 184–206. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2016.03.003. URL: https://www.sciencedirect.com/science/article/pii/S1570826816000214.

[53] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: sextuple indexing for semantic web data management". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 1008–1019.

[54] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. "Hexastore: sextuple indexing for semantic web data management". In: *Proc. VLDB Endow.* 1.1 (2008), pp. 1008–1019. DOI: 10.14778/1453856.1453965. URL: http://www.vldb.org/pvldb/vol1/1453965.pdf.

Appendix A.

Queries

A..1 Queries for SuccinctEdge evaluation

A total of 26 queries have been evaluated over a LUBM data set consisting of over 100.000 triples. They can be dispatched into 2 groups: whether their contain a single triple pattern or multiple ones. In this section, we list only the most prominent queries and provide templates for the other ones. Moreover, we present their main characteristics. The interested reader can access all of them on the paper companion GitHub page[1]. The following prefixes apply to all queries: lubm <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>, rdf <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

A..1.1 Single triple pattern queries

This first set of queries contain a single triple pattern in the WHERE clause. We distinguish between queries with a single variable, either at the object (denoted sp?) or subject (denoted ?po) position, from queries with two variables (denoted ?p?). As explained in the paper, we do not consider that variables at the property position make sense in SuccinctEdge's use cases.

SP?o queries

The identification of these 5 queries range from S1 to S5. We used the following query template:

```
SELECT ?X WHERE {X1 P1 ?X}
```

For S1, P1 binds to the lubm:takesCourse property and X1 is an undergraduate student

[1] https://github.com/xwq610728213/SuccinctEdge

Table A..1: Query summary with the following notations: 'SS' and 'OS' respectively correspond to subject, subject and object,subject joins; 'Co' for concept hierarchy inferences, 'Pr' for property hierarchy inferences

	Query performance													
Systems	S1-5	S6-10	S11-15	M1	M2	M3	M4	M5	R1	R2	R3	R4	R5	R6
TP number	1	1	1	2	3	5	3	11	5	5	3	6	3	11
TP type(s)	sp?	?po	?p?	?p?	?p? ?po	?p? ?po	?p? ?po sp?o	?p? ?po	?p? ?po	?p? ?spo	?p? ?po	?p? ?po	?p?	?p? ?spo sp?
Join type	-	-	-	SS	SS	SS,OS	OS	SS,OS OO	SS,OS	SS,OS	SS	SS,OS	OS	SS,OS OO
Join number	0	0	0	1	2	4	4	10	4	2	2	5	2	10
Path length	1	1	1	1	1	3	3	4	3	3	1	3	3	4
Selectivity	[4,513]	[5,521]	[540,15972]	540	1874	1874	7790	33	15	555	1874	1874	8345	34
Derived triples	0	0	0	0	0	0	0	0	15	540	1874	1874	555	1
Reasoning type	-	-	-	-	-	-	-	-	Co	Co Pr	Co Pr	Co Pr	Pr	Pr

constant. For queries S2 to S5, P1 binds to lubm:publicationAuthor and the X1 bind to different publication instances. The selectivity of these queries are in Table 4.1.

?sPO queries

These queries are identified from S6 to S10 and correspond to the following query template:

```
SELECT ?X WHERE { ?X P1 O1 }
```

P1 and O1 correspond to property and individual constants which for S6 to S10 respectively take the values (all properties are in the lubm namespace) : advisor/assistant professor constant, takesCourse/ course constant, worksFor/department constant, name/ publication constant, memberOf/ department constant.

?sP?o queries

```
S11: SELECT ?X ?Y ?Z WHERE { ?X lubm:worksFor ?Z }
S12: SELECT ?X ?Y ?Z WHERE { ?X lubm:teacherOf ?Y}
S13: SELECT ?X ?Y ?Z WHERE {
     ?X lubm:undergraduateDegreeFrom ?Y .}
S14: SELECT ?X ?Y ?Z WHERE { ?X lubm:emailAddress ?Y }
S15: SELECT ?X ?Y ?Z WHERE { ?X lubm:name ?Y }
```

A..1.2 Multiple triple patterns queries

In this set of queries, the BGP is composed of several triple patterns. The 11 queries in this category can be decomposed into those requiring or not some reasoning services (either

based on concept or property hierarchies).

Non-inference queries

All prefixed with 'M'.

```
M1: SELECT ?X ?Y ?Z WHERE { ?X lubm:worksFor ?Z .
 ?X lubm:name ?Y .}
M2: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?X rdf:type lubm:GraduateStudent .
 ?X lubm:undergraduateDegreeFrom ?Y .}
M3:  SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?X rdf:type lubm:GraduateStudent .
 ?Z rdf:type lubm:Department .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
M4: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University }
M5: SELECT * WHERE {
 <http://www.Department0...Publication14>
 lubm:publicationAuthor ?p. ?st lubm:memberOf ?o2.
 ?p a lubm:AssociateProfessor. ?p lubm:worksFor ?o.
 ?o a lubm:department. ?o lubm:subOrganizationOf ?u.
 ?u a lubm:University. ?p lubm:teacherOf ?te.
 ?te a lubm:Course. ?st lubm:takesCourse ?te.
 ?st a lubm:UndergraduateStudent. }
```

Inference queries

The identifier of these queries is prefixed with an 'R' since they involve a form of reasoning.

```
R1: SELECT ?X ?Y ?Z WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department . ?X lubm:headOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
R2: SELECT ?X ?Y ?Z WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department .  ?X lubm:worksFor ?Z .
 ?Z lubm:subOrganizationOf ?Y .
 ?Y rdf:type lubm:University .}
R3: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
```

```
 ?X rdf:type lubm:Student .
 ?X lubm:undergraduateDegreeFrom ?Y .}
R4: SELECT ?X ?Y ?Z ?N WHERE { ?X rdf:type lubm:Person .
 ?Z rdf:type lubm:Department . ?X lubm:memberOf ?Z .
 ?Z lubm:subOrganizationOf ?Y .  ?Y lubm:name ?N.
 ?Y rdf:type lubm:University . }
R5: identical to M4 but computes inferences  over the
memberOf property
R6: identical to M5 but computes inferences over the
memberOf and worksFor properties.
```

A..2 Queries for Streaming SuccinctEdge evaluation

This section contains all the queries Evaluated over streaming SuccinctEdge.

A..2.1 Queries for comparison against HDT

In this section, queries Q1-Q8 correspond to all the queries tested in Section 5.5.2.

```
Q1: SELECT ?X WHERE {
 <http://www.Department0.University0.edu/FullProfessor0>
    lubm:teacherOf ?X .
}
Q2: SELECT ?X WHERE {
 ?X lubm:worksFor <http://www.Department0.University0.edu> .
}
Q3:  SELECT ?X WHERE {
 ?X lubm:memberOf <http://www.Department1.University0.edu> .
}
Q4: SELECT ?X ?Z WHERE { ?X lubm:memberOf ?Z .}
Q5: SELECT ?X ?Y WHERE {
 <http://www.Department0.University0.edu/FullProfessor0>
    lubm:teacherOf ?X .
 ?X rdf:type ?Y .}
Q6: SELECT ?X WHERE {
 ?X lubm:worksFor <http://www.Department0.University0.edu> .
 ?X lubm:name ?Y .}
Q7: SELECT ?X ?Y ?Z WHERE { ?X lubm:worksFor ?Z .
 ?X lubm:name ?Y .}
Q8: SELECT ?X ?Y ?Z WHERE { ?X lubm:memberOf ?Z .
```

```
?Z lubm:subOrganizationOf ?Y .
?Y rdf:type lubm:University .}
```